

**MATLAB®**

Function Reference

**R2012b**

**MATLAB®**

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto: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 1984–2012 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](http://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](http://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)





## Function Reference

**1**

<b>Desktop Tools and Development Environment</b> . . . . .	<b>1-3</b>
Startup and Shutdown . . . . .	1-3
Command Window and History . . . . .	1-4
Help for Using MATLAB . . . . .	1-5
Workspace . . . . .	1-5
Managing Files . . . . .	1-6
Programming Tools . . . . .	1-7
System . . . . .	1-10
<b>Data Import and Export</b> . . . . .	<b>1-12</b>
File Opening, Loading, and Saving . . . . .	1-13
Text Files . . . . .	1-14
Spreadsheets . . . . .	1-15
Low-Level File I/O . . . . .	1-16
Images . . . . .	1-16
Scientific Data . . . . .	1-17
Audio and Video . . . . .	1-27
XML Documents . . . . .	1-29
Memory Mapping . . . . .	1-29
File Name Construction . . . . .	1-30
File Compression . . . . .	1-30
Internet File Access . . . . .	1-30
<b>Mathematics</b> . . . . .	<b>1-31</b>
Arrays and Matrices . . . . .	1-32
Linear Algebra . . . . .	1-38
Elementary Math . . . . .	1-42
Polynomials . . . . .	1-46
Interpolation and Computational Geometry . . . . .	1-47
Cartesian Coordinate System Conversion . . . . .	1-49
Nonlinear Numerical Methods . . . . .	1-50
Special Functions . . . . .	1-54
Sparse Matrices . . . . .	1-55
Math Constants . . . . .	1-59

<b>Data Analysis</b> .....	<b>1-60</b>
Basic Operations .....	1-60
Descriptive Statistics .....	1-60
Filtering and Convolution .....	1-61
Interpolation and Regression .....	1-61
Fourier Transforms .....	1-62
Derivatives and Integrals .....	1-62
Time Series Objects .....	1-63
Time Series Collections .....	1-66
<b>Programming and Data Types</b> .....	<b>1-68</b>
Data Types .....	1-68
Data Type Conversion .....	1-76
Operators and Special Characters .....	1-79
Strings .....	1-81
Bit-Wise Operations .....	1-84
Logical Operations .....	1-84
Relational Operations .....	1-85
Set Operations .....	1-86
Date and Time Operations .....	1-86
Programming in MATLAB .....	1-87
<b>Object-Oriented Programming</b> .....	<b>1-94</b>
Classes and Objects .....	1-94
Handle Classes .....	1-95
Events and Listeners .....	1-96
Meta-Classes .....	1-96
Enumerations .....	1-97
Heterogeneous Arrays .....	1-97
<b>Graphics</b> .....	<b>1-99</b>
Basic Plots and Graphs .....	1-99
Plotting Tools .....	1-100
Annotating Plots .....	1-100
Specialized Plotting .....	1-101
Bit-Mapped Images .....	1-105
Printing .....	1-106
Handle Graphics .....	1-106
<b>3-D Visualization</b> .....	<b>1-111</b>
Surface and Mesh Plots .....	1-111
View Control .....	1-113

Lighting .....	1-115
Transparency .....	1-115
Volume Visualization .....	1-116
<b>GUI Development .....</b>	<b>1-118</b>
Predefined Dialog Boxes .....	1-118
User Interface Deployment .....	1-119
User Interface Development .....	1-120
User Interface Objects .....	1-120
Objects from Callbacks .....	1-121
GUI Utilities .....	1-121
Program Execution .....	1-122
<b>External Interfaces .....</b>	<b>1-124</b>
Shared Libraries .....	1-124
Java .....	1-124
.NET .....	1-125
Component Object Model and ActiveX .....	1-127
Web Services .....	1-129
Serial Port Devices .....	1-130

## Alphabetical List

**2**

**Index**



# Function Reference

---

Desktop Tools and Development Environment (p. 1-3)

Startup, Command Window, help, editing and debugging, tuning, other general functions

Data Import and Export (p. 1-12)

General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images

Mathematics (p. 1-31)

Arrays and matrices, linear algebra, other areas of mathematics

Data Analysis (p. 1-60)

Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis

Programming and Data Types (p. 1-68)

Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers

Object-Oriented Programming (p. 1-94)

Functions for working with classes and objects

Graphics (p. 1-99)

Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics

3-D Visualization (p. 1-111)

Surface and mesh plots, view control, lighting and transparency, volume visualization

GUI Development (p. 1-118)

GUIDE, programming graphical user interfaces

External Interfaces (p. 1-124)

Interfaces to shared libraries, Java, .NET, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines

## Desktop Tools and Development Environment

Startup and Shutdown (p. 1-3)	Startup and shutdown options, preferences
Command Window and History (p. 1-4)	Control Command Window and History, enter statements and run functions
Help for Using MATLAB (p. 1-5)	Command line help, online documentation in the Help browser, demos
Workspace (p. 1-5)	Manage variables
Managing Files (p. 1-6)	Work with files, MATLAB search path, manage variables
Programming Tools (p. 1-7)	Edit and debug MATLAB code , improve performance, source control, publish results
System (p. 1-10)	Identify current computer, license, product version, and more

### Startup and Shutdown

exit	Terminate MATLAB® program (same as quit)
finish	Termination file for MATLAB program
matlab (UNIX)	Start MATLAB program (UNIX® platforms)
matlab (Windows)	Start MATLAB program (Windows® platforms)
matlabrc	Startup file for MATLAB program
prefdir	Folder containing preferences, history, and layout files
preferences	Open Preferences dialog box

quit	Terminate MATLAB program
startup	Startup file for user-defined options
userpath	View or change user portion of search path

## **Command Window and History**

clc	Clear Command Window
commandhistory	Open Command History window, or select it if already open
commandwindow	Open Command Window, or select it if already open
diary	Save Command Window text to file
dos	Execute DOS command and return result
format	Set display format for output
home	Send cursor home
more	Control paged output for Command Window
perl	Call Perl script using appropriate operating system executable
system	Execute operating system command and return result
unix	Execute UNIX command and return result



## Help for Using MATLAB

builddocsearchdb	Build searchable documentation database
demo	Access product examples in Help browser
doc	Reference page in Help browser
docsearch	Help browser search
echodemo	Run example script step-by-step in Command Window
help	Help for functions in Command Window
lookfor	Search for keyword in all help entries
web	Open Web page or file in browser

## Workspace

clearvars	Clear variables from memory
delete	Remove files or graphics objects
openvar	Open workspace variable in Variables editor or other graphical editing tool
pack	Consolidate workspace memory
uiimport	Import data interactively
which	Locate functions and files
who	List variables in workspace
whos	List variables in workspace, with sizes and types
workspace	Open Workspace browser to manage workspace

## Managing Files

Search Path (p. 1-6)	View and change MATLAB search path
File Operations (p. 1-6)	View and change files and folders

## Search Path

addpath	Add folders to search path
genpath	Generate path string
path	View or change search path
path2rc	Save current search path to pathdef.m file
pathsep	Search path separator for current platform
pathtool	Open Set Path dialog box to view and change search path
restoredefaultpath	Restore default search path
rmpath	Remove folders from search path
savepath	Save current search path
userpath	View or change user portion of search path
which	Locate functions and files

## File Operations

See also “Data Import and Export” on page 1-12 functions.

cd	Change current folder
copyfile	Copy file or folder
delete	Remove files or graphics objects
dir	List folder contents

fileattrib	Set or get attributes of file or folder
filebrowser	Open Current Folder browser, or select it if already open
isdir	Determine whether input is folder
lookfor	Search for keyword in all help entries
ls	List folder contents
matlabroot	Root folder
mkdir	Make new folder
movefile	Move file or folder
pwd	Identify current folder
recycle	Set option to move deleted files to recycle folder
rmdir	Remove folder
tempdir	Name of system's temporary folder
toolboxdir	Root folder for specified toolbox
type	Display contents of file
visdiff	Compare two text files, MAT-Files, binary files, or folders
what	List MATLAB files in folder

## Programming Tools

Editing Files (p. 1-8)	Edit files
Debugging Programs (p. 1-8)	Debug MATLAB program files
Code Performance (p. 1-8)	Improve performance and find potential problems in MATLAB code
Source Control (p. 1-9)	Interface MATLAB with source control system
Publishing (p. 1-9)	Publish MATLAB code and results



profsave	Save profile report in HTML format
tic	Start clock to measure performance
toc	Stop clock to measure performance

## Source Control

checkin	Check files into source control system (UNIX platforms)
checkout	Check files out of source control system (UNIX platforms)
cmopts	Name of source control system
customverctrl	Allow custom source control system (UNIX platforms)
undocheckout	Undo previous checkout from source control system (UNIX platforms)
verctrl	Source control actions (Windows platforms)

## Publishing

grabcode	Extract MATLAB code from file published to HTML
notebook	Open MATLAB Notebook in Microsoft® Word software (on Microsoft Windows platforms)
publish	Generate view of MATLAB file in specified format
snapnow	Force snapshot of image for inclusion in published document

## System

Operating System Interface (p. 1-10)	Exchange operating system information and commands with MATLAB
MATLAB Version and License (p. 1-11)	Information about MATLAB version and license

## Operating System Interface

clipboard	Copy and paste strings to and from system clipboard
computer	Information about computer on which MATLAB software is running
dos	Execute DOS command and return result
getenv	Environment variable
hostid	Server host identification number
perl	Call Perl script using appropriate operating system executable
setenv	Set environment variable
system	Execute operating system command and return result
unix	Execute UNIX command and return result
winqueryreg	Item from Windows registry

## **MATLAB Version and License**

<code>ismac</code>	Determine if version is for Mac OS X platform
<code>ispc</code>	Determine if version is for Windows (PC) platform
<code>isstudent</code>	Determine if version is Student Version
<code>isunix</code>	Determine if version is for UNIX platform
<code>javachk</code>	Generate error message based on Java™ feature support
<code>license</code>	Return license number or perform licensing task
<code>prefdir</code>	Folder containing preferences, history, and layout files
<code>usejava</code>	Determine whether Java feature is supported in MATLAB software
<code>ver</code>	Version information for MathWorks® products
<code>verLessThan</code>	Compare toolbox version to specified version string
<code>version</code>	Version number for MATLAB and libraries

## Data Import and Export

File Opening, Loading, and Saving (p. 1-13)	Open files; transfer data between files and MATLAB workspace
Text Files (p. 1-14)	Delimited or formatted I/O to text files
Spreadsheets (p. 1-15)	Excel and Lotus 1-2-3 files
Low-Level File I/O (p. 1-16)	Read and write operations at the byte or character level
Images (p. 1-16)	Graphics files
Scientific Data (p. 1-17)	CDF, FITS, HDF formats
Audio and Video (p. 1-27)	Read and write audio and video, record and play audio
XML Documents (p. 1-29)	Documents written in Extensible Markup Language
Memory Mapping (p. 1-29)	Access file data via memory map using MATLAB array indexing
File Name Construction (p. 1-30)	Get path, directory, filename information; construct filenames
File Compression (p. 1-30)	Extract from tar or zip file; compress and decompress files
Internet File Access (p. 1-30)	Send email, download from URL, or connect to FTP server

For a list of file formats that you can read with MATLAB, see “Supported File Formats”.

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	File separator for current platform
fullfile	Build full file name from parts



tempdir	Name of system's temporary folder
tempname	Unique name for temporary file

## File Opening, Loading, and Saving

daqread	Read Data Acquisition Toolbox™ (.daq) file
importdata	Load data from file
load	Load data from MAT-file into workspace
matfile	Load and save parts of variables in MAT-files
open	Open file in appropriate application
save	Save workspace variables to file
uigetdir	Open standard dialog box for selecting directory
uigetfile	Open standard dialog box for retrieving files
uiimport	Import data interactively
uiputfile	Open standard dialog box for saving files
uisave	Interactively save workspace variables to MAT-file
winopen	Open file in appropriate application (Windows)
fclose	Close one or all open files
feof	Test for end-of-file
ferror	Information about file I/O errors
fgetl	Read line from file, removing newline characters

<code>fgets</code>	Read line from file, keeping newline characters
<code>fopen</code>	Open file, or obtain information about open files
<code>fprintf</code>	Write data to text file
<code>fread</code>	Read data from binary file
<code>frewind</code>	Move file position indicator to beginning of open file
<code>fscanf</code>	Read data from text file
<code>fseek</code>	Move to specified position in file
<code>ftell</code>	Position in open file
<code>fwrite</code>	Write data to binary file

## **Text Files**

<code>csvread</code>	Read comma-separated value file
<code>csvwrite</code>	Write comma-separated value file
<code>dlmread</code>	Read ASCII-delimited file of numeric data into matrix
<code>dlmwrite</code>	Write matrix to ASCII-delimited file
<code>fileread</code>	Read contents of file into string
<code>textread</code>	Read data from text file; write to multiple outputs
<code>textscan</code>	Read formatted data from text file or string
<code>type</code>	Display contents of file

xmlread	Read XML document and return Document Object Model node
xmlwrite	Write XML Document Object Model node
xslt	Transform XML document using XSLT engine

## Spreadsheets

Microsoft Excel (p. 1-15)	Read and write Microsoft Excel spreadsheet
Lotus 1-2-3 (p. 1-15)	Read and write Lotus WK1 spreadsheet

### Microsoft Excel

xlsinfo	Determine whether file contains Microsoft Excel® spreadsheet
xlsread	Read Microsoft Excel spreadsheet file
xlswrite	Write Microsoft Excel spreadsheet file

### Lotus 1-2-3

wk1info	Determine whether file contains 1-2-3 WK1 worksheet
wk1read	Read Lotus 1-2-3 WK1 spreadsheet file into matrix
wk1write	Write matrix to Lotus 1-2-3 WK1 spreadsheet file

## Low-Level File I/O

<code>fclose</code>	Close one or all open files
<code>feof</code>	Test for end-of-file
<code>ferror</code>	Information about file I/O errors
<code>fgetl</code>	Read line from file, removing newline characters
<code>fgets</code>	Read line from file, keeping newline characters
<code>fopen</code>	Open file, or obtain information about open files
<code>fprintf</code>	Write data to text file
<code>fread</code>	Read data from binary file
<code>frewind</code>	Move file position indicator to beginning of open file
<code>fscanf</code>	Read data from text file
<code>fseek</code>	Move to specified position in file
<code>ftell</code>	Position in open file
<code>fwrite</code>	Write data to binary file

## Images

<code>exifread</code>	Read EXIF information from JPEG and TIFF image files
<code>im2java</code>	Convert image to Java image
<code>imfinfo</code>	Information about graphics file
<code>imread</code>	Read image from graphics file
<code>imwrite</code>	Write image to graphics file
<code>Tiff</code>	MATLAB Gateway to LibTIFF library routines

## Scientific Data

Common Data Format (p. 1-17)	Work with CDF files
Network Common Data Form (p. 1-22)	Work with netCDF files
Flexible Image Transport System (p. 1-26)	Work with FITS files
Hierarchical Data Format (p. 1-26)	Work with HDF files
Band-Interleaved Data (p. 1-27)	Work with band-interleaved files

## Common Data Format

### High-level I/O Functions

<code>cdfepoch</code>	Convert MATLAB formatted dates to CDF formatted dates
<code>cdfinfo</code>	Information about Common Data Format (CDF) file
<code>cdfread</code>	Read data from Common Data Format (CDF) file
<code>cdfwrite</code>	Write data to Common Data Format (CDF) file
<code>todatenum</code>	Convert CDF epoch object to MATLAB datenum

### Library Information

<code>cdflib</code>	Summary of Common Data Format (CDF) capabilities
<code>cdflib.getConstantNames</code>	Names of Common Data Format (CDF) library constants
<code>cdflib.getConstantValue</code>	Numeric value corresponding to Common Data Format (CDF) library constant

<code>cdflib.getFileBackward</code>	Return current backward compatibility mode setting
<code>cdflib.getLibraryCopyright</code>	Copyright notice of Common Data Format (CDF) library
<code>cdflib.getLibraryVersion</code>	Library version and release information
<code>cdflib.getValidate</code>	Library validation mode
<code>cdflib.setFileBackward</code>	Set backward compatibility mode
<code>cdflib.setValidate</code>	Specify library validation mode

### **File Operations**

<code>cdflib.close</code>	Close Common Data Format (CDF) file
<code>cdflib.create</code>	Create Common Data Format (CDF) file
<code>cdflib.delete</code>	Delete existing Common Data Format (CDF) file
<code>cdflib.getCacheSize</code>	Number of cache buffers used
<code>cdflib.getChecksum</code>	Checksum mode
<code>cdflib.getCompression</code>	Compression settings
<code>cdflib.getCompressionCacheSize</code>	Number of compression cache buffers
<code>cdflib.getCopyright</code>	Copyright notice in Common Data Format (CDF) file
<code>cdflib.getFormat</code>	Format of Common Data Format (CDF) file
<code>cdflib.getMajority</code>	Majority of variables
<code>cdflib.getName</code>	Name of Common Data Format (CDF) file
<code>cdflib.getReadOnlyMode</code>	Read-only mode
<code>cdflib.getStageCacheSize</code>	Number of cache buffers for staging

<code>cdflib.getVersion</code>	Common Data Format (CDF) library version and release information
<code>cdflib.inquire</code>	Basic characteristics of Common Data Format (CDF) file
<code>cdflib.open</code>	Open existing Common Data Format (CDF) file
<code>cdflib.setCacheSize</code>	Specify number of dotCDF cache buffers
<code>cdflib.setChecksum</code>	Specify checksum mode
<code>cdflib.setCompression</code>	Specify compression settings
<code>cdflib.setCompressionCacheSize</code>	Specify number of compression cache buffers
<code>cdflib.setFormat</code>	Specify format of Common Data Format (CDF) file
<code>cdflib.setMajority</code>	Specify majority of variables
<code>cdflib.setReadOnlyMode</code>	Specify read-only mode
<code>cdflib.setStageCacheSize</code>	Specify number of staging cache buffers for Common Data Format (CDF) file

## Variables

<code>cdflib.closeVar</code>	Close specified variable from multifile format Common Data Format (CDF) file
<code>cdflib.createVar</code>	Create new variable
<code>cdflib.deleteVar</code>	Delete variable
<code>cdflib.deleteVarRecords</code>	Delete range of records from variable
<code>cdflib.getVarAllocRecords</code>	Number of records allocated for variable
<code>cdflib.getVarBlockingFactor</code>	Blocking factor for variable
<code>cdflib.getVarCacheSize</code>	Number of multifile cache buffers

<code>cdflib.getVarCompression</code>	Information about compression used by variable
<code>cdflib.getVarData</code>	Single value from record in variable
<code>cdflib.getVarMaxAllocRecNum</code>	Maximum allocated record number for variable
<code>cdflib.getVarMaxWrittenRecNum</code>	Maximum written record number for variable
<code>cdflib.getVarName</code>	Variable name, given variable number
<code>cdflib.getVarNum</code>	Variable number, given variable name
<code>cdflib.getVarNumRecsWritten</code>	Number of records written to variable
<code>cdflib.getVarPadValue</code>	Pad value for variable
<code>cdflib.getVarRecordData</code>	Entire record for variable
<code>cdflib.getVarReservePercent</code>	Compression reserve percentage for variable
<code>cdflib.getVarsMaxWrittenRecNum</code>	Maximum written record number for CDF file
<code>cdflib.getVarSparseRecords</code>	Information about how variable handles sparse records
<code>cdflib.hyperGetVarData</code>	Read hyperslab of data from variable
<code>cdflib.hyperPutVarData</code>	Write hyperslab of data to variable
<code>cdflib.inquireVar</code>	Information about variable
<code>cdflib.putVarData</code>	Write single value to variable
<code>cdflib.putVarRecordData</code>	Write entire record to variable
<code>cdflib.renameVar</code>	Rename existing variable
<code>cdflib.setVarAllocBlockRecords</code>	Specify range of records to be allocated for variable
<code>cdflib.setVarBlockingFactor</code>	Specify blocking factor for variable



<code>cdflib.setVarCacheSize</code>	Specify number of multi-file cache buffers for variable
<code>cdflib.setVarCompression</code>	Specify compression settings used with variable
<code>cdflib.setVarInitialRecs</code>	Specify initial number of records written to variable
<code>cdflib.setVarPadValue</code>	Specify pad value used with variable
<code>cdflib.SetVarReservePercent</code>	Specify reserve percentage for variable
<code>cdflib.setVarsCacheSize</code>	Specify number of cache buffers used for all variables
<code>cdflib.setVarSparseRecords</code>	Specify how variable handles sparse records

### **Attributes and Entries**

<code>cdflib.createAttr</code>	Create attribute
<code>cdflib.deleteAttr</code>	Delete attribute
<code>cdflib.deleteAttrEntry</code>	Delete attribute entry
<code>cdflib.deleteAttrgEntry</code>	Delete entry in global attribute
<code>cdflib.getAttrEntry</code>	Value of entry in attribute with variable scope
<code>cdflib.getAttrgEntry</code>	Value of entry in global attribute
<code>cdflib.getAttrMaxEntry</code>	Number of last entry for variable attribute
<code>cdflib.getAttrMaxgEntry</code>	Number of last entry for global attribute
<code>cdflib.getAttrName</code>	Name of attribute, given attribute number
<code>cdflib.getAttrNum</code>	Attribute number, given attribute name
<code>cdflib.getAttrScope</code>	Scope of attribute

<code>cdflib.getNumAttrEntries</code>	Number of entries for attribute with variable scope
<code>cdflib.getNumAttrgEntries</code>	Number of entries for attribute with global scope
<code>cdflib.getNumAttributes</code>	Number of attributes with variable scope
<code>cdflib.getNumgAttributes</code>	Number of attributes with global scope
<code>cdflib.inquireAttr</code>	Information about attribute
<code>cdflib.inquireAttrEntry</code>	Information about entry in attribute with variable scope
<code>cdflib.inquireAttrgEntry</code>	Information about entry in attribute with global scope
<code>cdflib.putAttrEntry</code>	Write value to entry in attribute with variable scope
<code>cdflib.putAttrgEntry</code>	Write value to entry in attribute with global scope
<code>cdflib.renameAttr</code>	Rename existing attribute

### Utilities

<code>cdflib.computeEpoch</code>	Convert time value to CDF_EPOCH value
<code>cdflib.computeEpoch16</code>	Convert time value to CDF_EPOCH16 value
<code>cdflib.epoch16Breakdown</code>	Convert CDF_EPOCH16 value to time value
<code>cdflib.epochBreakdown</code>	Convert CDF_EPOCH value into time value

## Network Common Data Form

### High-level I/O Functions

<code>nccreate</code>	Create variable in NetCDF file
<code>ncdisp</code>	Display contents of NetCDF data source in Command Window
<code>ncinfo</code>	Return information about NetCDF data source
<code>ncread</code>	Read data from variable in NetCDF data source
<code>ncreadatt</code>	Read attribute value from NetCDF data source
<code>ncwrite</code>	Write data to NetCDF file
<code>ncwriteatt</code>	Write attribute to NetCDF file
<code>ncwritescema</code>	Add NetCDF schema definitions to NetCDF file

### **Library Functions**

<code>netcdf</code>	Summary of MATLAB Network Common Data Form (NetCDF) capabilities
<code>netcdf.getChunkCache</code>	Retrieve chunk cache settings for NetCDF library
<code>netcdf.inqLibVers</code>	Return NetCDF library version information
<code>netcdf.setChunkCache</code>	Set default chunk cache settings for NetCDF library
<code>netcdf.setDefaultFormat</code>	Change default netCDF file format

### **File Operations**

<code>netcdf.abort</code>	Revert recent netCDF file definitions
<code>netcdf.close</code>	Close netCDF file
<code>netcdf.create</code>	Create new NetCDF dataset

<code>netcdf.endDef</code>	End netCDF file define mode
<code>netcdf.inq</code>	Return information about netCDF file
<code>netcdf.inqFormat</code>	Determine format of NetCDF file
<code>netcdf.inqGrps</code>	Retrieve array of child group IDs
<code>netcdf.inqUnlimDims</code>	Return list of unlimited dimensions in group
<code>netcdf.open</code>	Open NetCDF data source
<code>netcdf.reDef</code>	Put open netCDF file into define mode
<code>netcdf.setFill</code>	Set netCDF fill mode
<code>netcdf.sync</code>	Synchronize netCDF file to disk

### **Dimensions**

<code>netcdf.defDim</code>	Create netCDF dimension
<code>netcdf.inqDim</code>	Return netCDF dimension name and length
<code>netcdf.inqDimID</code>	Return dimension ID
<code>netcdf.renameDim</code>	Change name of netCDF dimension

### **Groups**

<code>netcdf.defGrp</code>	Create group in NetCDF file
<code>netcdf.inqDimIDs</code>	Retrieve list of dimension identifiers in group
<code>netcdf.inqGrpName</code>	Retrieve name of group
<code>netcdf.inqGrpNameFull</code>	Complete pathname of group
<code>netcdf.inqGrpParent</code>	Retrieve ID of parent group.
<code>netcdf.inqNcid</code>	Return ID of named group
<code>netcdf.inqVarIDs</code>	IDs of all variables in group

**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

### Flexible Image Transport System

fitsdisp	Display FITS metadata
fitsinfo	Information about FITS file
fitsread	Read data from FITS file
fitswrite	Write image to FITS file

### Hierarchical Data Format

h5create	Create HDF5 data set
h5disp	Display contents of HDF5 file
h5info	Return information about HDF5 file
h5read	Read data from HDF5 data set
h5readatt	Read attribute from HDF5 file
h5write	Write to HDF5 data set
h5writeatt	Write HDF5 attribute

hdf	Summary of MATLAB HDF4 capabilities
hdf5	Summary of MATLAB HDF5 capabilities
hdf5info	Information about HDF5 file
hdf5read	Read HDF5 file
hdf5write	Write data to file in HDF5 format
hdfinfo	Information about HDF4 or HDF-EOS file
hdfread	Read data from HDF4 or HDF-EOS file
hdftool	Browse and import data from HDF4 or HDF-EOS files

### **Band-Interleaved Data**

multibandread	Read band-interleaved data from binary file
multibandwrite	Write band-interleaved data to file

### **Audio and Video**

Reading and Writing Files (p. 1-28)	Input/output data to audio and video file formats
Recording and Playback (p. 1-28)	Record and listen to audio
Utilities (p. 1-29)	Convert audio signal

## Reading and Writing Files

audioinfo	Information about audio file
aufinfo	Information about NeXT/SUN (.au) sound file
auread	Read NeXT/SUN (.au) sound file
auwrite	Write NeXT/SUN (.au) sound file
mmfileinfo	Information about multimedia file
movie2avi	Create Audio/Video Interleaved (AVI) file from MATLAB movie
VideoReader	Read video files
VideoWriter	Write video files
wavinfo	Information about WAVE (.wav) sound file
wavread	Read WAVE (.wav) sound file
wavwrite	Write WAVE (.wav) sound file

## Recording and Playback

audiodevinfo	Information about audio device
audioplayer	Create object for playing audio
audiorecorder	Create object for recording audio
sound	Convert matrix of signal data to sound
soundsc	Scale data and play as sound



## Utilities

beep	Produce beep sound
lin2mu	Convert linear audio signal to mu-law
mu2lin	Convert mu-law audio signal to linear
exifread	Read EXIF information from JPEG and TIFF image files
im2java	Convert image to Java image
imfinfo	Information about graphics file
imread	Read image from graphics file
imwrite	Write image to graphics file
Tiff	MATLAB Gateway to LibTIFF library routines

## XML Documents

xmlread	Read XML document and return Document Object Model node
xmlwrite	Write XML Document Object Model node
xslt	Transform XML document using XSLT engine

## Memory Mapping

disp (memmapfile)	Information about memmapfile object
get (memmapfile)	Memmapfile object properties
memmapfile	Construct memmapfile object

## File Name Construction

filemarker	Character to separate file name and internal function name
fileparts	Parts of file name and path
filesep	File separator for current platform
fullfile	Build full file name from parts
tempdir	Name of system's temporary folder
tempname	Unique name for temporary file

## File Compression

gunzip	Uncompress GNU zip files
gzip	Compress files into GNU zip files
tar	Compress files into tar file
untar	Extract contents of tar file
unzip	Extract contents of zip file
zip	Compress files into zip file

## Internet File Access

FTP	Connect to FTP server
sendmail	Send email message to address list
urlread	Download URL content to MATLAB string
urlwrite	Download URL content and save to file

# Mathematics

Arrays and Matrices (p. 1-32)	Basic array operators and operations, creation of elementary and specialized arrays and matrices
Linear Algebra (p. 1-38)	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
Elementary Math (p. 1-42)	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
Polynomials (p. 1-46)	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
Interpolation and Computational Geometry (p. 1-47)	Interpolation, Delaunay triangulation, convex hulls, Voronoi diagrams, domain generation
Cartesian Coordinate System Conversion (p. 1-49)	Conversions between Cartesian and polar or spherical coordinates
Nonlinear Numerical Methods (p. 1-50)	Differential equations, optimization, integration
Special Functions (p. 1-54)	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions
Sparse Matrices (p. 1-55)	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
Math Constants (p. 1-59)	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

## Arrays and Matrices

Basic Information (p. 1-32)

Display array contents, get array information, determine array type

Operators (p. 1-33)

Arithmetic operators

Elementary Matrices and Arrays (p. 1-34)

Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.

Array Operations (p. 1-35)

Operate on array content, apply function to each array element, find cumulative product or sum, etc.

Array Manipulation (p. 1-36)

Create, sort, rotate, permute, reshape, and shift array contents

Specialized Matrices (p. 1-37)

Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

## Basic Information

disp

Display text or array

iscolumn

Determine whether input is column vector

isempty

Determine whether array is empty

isequal

Test arrays for equality

isequaln

Test arrays for equality, treating NaN values as equal

isfinite

Array elements that are finite

isfloat

Determine if input is floating-point array

isinf

Array elements that are infinite

isinteger

Determine if input is integer array

islogical

Determine if input is logical array

ismatrix	Determine whether input is matrix
isnan	Array elements that are NaN
isnumeric	Determine if input is numeric array
isrow	Determine whether input is row vector
isscalar	Determine whether input is scalar
issparse	Determine whether input is sparse
isvector	Determine whether input is vector
length	Length of vector or largest array dimension
max	Largest elements in array
min	Smallest elements in array
ndims	Number of array dimensions
numel	Number of elements in array or subscripted array expression
size	Array dimensions

## Operators

Arithmetic Operators + - * / \ ^ ' ,	Matrix and array arithmetic
ctranspose	Complex conjugate transpose
minus	Minus
mpower	Matrix power
plus	Plus
times	Array multiply
transpose	Transpose
uminus	Unary minus
uplus	Unary plus

+	Addition
+	Unary plus
-	Subtraction
-	Unary minus
*	Matrix multiplication
^	Matrix power
\	Backslash or left matrix divide
/	Slash or right matrix divide
'	Transpose
.'	Nonconjugated transpose
.*	Array multiplication (element-wise)
.^	Array power (element-wise)
.\	Left array divide (element-wise)
./	Right array divide (element-wise)

## **Elementary Matrices and Arrays**

blkdiag	Construct block diagonal matrix from input arguments
colon (:)	Create vectors, array subscripting, and for-loop iterators
diag	Diagonal matrices and diagonals of matrix
eye	Identity matrix
freqspace	Frequency spacing for frequency response
ind2sub	Subscripts from linear index
linspace	Generate linearly spaced vectors

logspace	Generate logarithmically spaced vectors
ndgrid	Rectangular grid in N-D space
ones	Create array of all ones
rand	Uniformly distributed pseudorandom numbers
randi	Uniformly distributed pseudorandom integers
randn	Normally distributed pseudorandom numbers
RandStream	Random number stream
rng	Control random number generation
sub2ind	Convert subscripts to linear indices
zeros	Create array of all zeros

## Array Operations

See “Linear Algebra” on page 1-38 and “Elementary Math” on page 1-42 for other array operations.

accumarray	Construct array with accumulation
arrayfun	Apply function to each element of array
bsxfun	Apply element-by-element binary operation to two arrays with singleton expansion enabled
cast	Cast variable to different data type
cross	Vector cross product
cumprod	Cumulative product
cumsum	Cumulative sum
dot	Vector dot product

<code>idivide</code>	Integer division with rounding option
<code>kron</code>	Kronecker tensor product
<code>prod</code>	Product of array elements
<code>sum</code>	Sum of array elements
<code>surfnorm</code>	Compute and display 3-D surface normals
<code>tril</code>	Lower triangular part of matrix
<code>triu</code>	Upper triangular part of matrix

## **Array Manipulation**

<code>blkdiag</code>	Construct block diagonal matrix from input arguments
<code>cat</code>	Concatenate arrays along specified dimension
<code>circshift</code>	Shift array circularly
<code>diag</code>	Diagonal matrices and diagonals of matrix
<code>end</code>	Terminate block of code, or indicate last array index
<code>flipdim</code>	Flip array along specified dimension
<code>fliplr</code>	Flip matrix left to right
<code>flipud</code>	Flip matrix up to down
<code>horzcat</code>	Concatenate arrays horizontally
<code>inline</code>	Construct inline object
<code>ipermute</code>	Inverse permute dimensions of N-D array
<code>permute</code>	Rearrange dimensions of N-D array
<code>repmat</code>	Replicate and tile array



reshape	Reshape array
rot90	Rotate matrix 90 degrees
shiftdim	Shift dimensions
sort	Sort array elements in ascending or descending order
sortrows	Sort rows in ascending order
squeeze	Remove singleton dimensions
vectorize	Vectorize expression
vertcat	Concatenate arrays vertically

### **Specialized Matrices**

compan	Companion matrix
gallery	Test matrices
hadamard	Hadamard matrix
hankel	Hankel matrix
hilb	Hilbert matrix
invhilb	Inverse of Hilbert matrix
magic	Magic square
pascal	Pascal matrix
rosser	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

## Linear Algebra

Matrix Analysis (p. 1-38)	Compute norm, rank, determinant, condition number, etc.
Linear Equations (p. 1-39)	Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.
Eigenvalues and Singular Values (p. 1-40)	Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.
Matrix Logarithms and Exponentials (p. 1-41)	Matrix logarithms, exponentials, square root
Factorization (p. 1-41)	Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

## Matrix Analysis

cond	Condition number with respect to inversion
condeig	Condition number with respect to eigenvalues
det	Matrix determinant
norm	Vector and matrix norms
normest	2-norm estimate
null	Null space
orth	Range space of matrix
rank	Rank of matrix
rcond	Matrix reciprocal condition number estimate
rref	Reduced row echelon form

subspace	Angle between two subspaces
trace	Sum of diagonal elements

## Linear Equations

chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cond	Condition number with respect to inversion
condest	1-norm condition number estimate
funm	Evaluate general matrix function
ichol	Incomplete Cholesky factorization
ilu	Sparse incomplete LU factorization
inv	Matrix inverse
ldl	Block LDL' factorization for Hermitian indefinite matrices
linsolve	Solve linear system of equations
lscov	Least-squares solution in presence of known covariance
lsqnonneg	Solve nonnegative least-squares constraints problem
lu	LU matrix factorization
luinc	Sparse incomplete LU factorization
pinv	Moore-Penrose pseudoinverse of matrix
qr	Orthogonal-triangular decomposition
rcond	Matrix reciprocal condition number estimate

## Eigenvalues and Singular Values

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Eigenvalues and eigenvectors
eigs	Largest eigenvalues and eigenvectors of matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
sqrtn	Matrix square root
ss2tf	Convert state-space filter parameters to transfer function form
svd	Singular value decomposition
svds	Find singular values and vectors

## Matrix Logarithms and Exponentials

expm	Matrix exponential
logm	Matrix logarithm
sqrtm	Matrix square root

## Factorization

balance	Diagonal scaling to improve eigenvalue accuracy
cdf2rdf	Convert complex diagonal form to real block diagonal form
chol	Cholesky factorization
cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
ichol	Incomplete Cholesky factorization
ilu	Sparse incomplete LU factorization
ldl	Block LDL' factorization for Hermitian indefinite matrices
lu	LU matrix factorization
luinc	Sparse incomplete LU factorization
planerot	Givens plane rotation
qr	Orthogonal-triangular decomposition
qrdelete	Remove column or row from QR factorization

qrinsert	Insert column or row into QR factorization
qrupdate	Rank 1 update to QR factorization
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
svd	Singular value decomposition

## Elementary Math

Trigonometric (p. 1-42)	Trigonometric functions with results in radians or degrees
Exponential (p. 1-44)	Exponential, logarithm, power, and root functions
Complex (p. 1-45)	Numbers with real and imaginary components, phase angles
Rounding and Remainder (p. 1-45)	Rounding, modulus, and remainder
Discrete Math (p. 1-46)	Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

## Trigonometric

acos	Inverse cosine; result in radians
acosd	Inverse cosine; result in degrees
acosh	Inverse hyperbolic cosine
acot	Inverse cotangent; result in radians
acotd	Inverse cotangent; result in degrees
acoth	Inverse hyperbolic cotangent

acsc	Inverse cosecant; result in radians
acscd	Inverse cosecant; result in degrees
acsch	Inverse hyperbolic cosecant
asec	Inverse secant; result in radians
asecd	Inverse secant; result in degrees
asech	Inverse hyperbolic secant
asin	Inverse sine; result in radians
asind	Inverse sine; result in degrees
asinh	Inverse hyperbolic sine
atan	Inverse tangent; result in radians
atan2	Four-quadrant inverse tangent
atan2d	Four-quadrant inverse tangent; result in degrees
atand	Inverse tangent; result in degrees
atanh	Inverse hyperbolic tangent
cos	Cosine of argument in radians
cosd	Cosine of argument in degrees
cosh	Hyperbolic cosine
cot	Cotangent of argument in radians
cotd	Cotangent of argument in degrees
coth	Hyperbolic cotangent
csc	Cosecant of argument in radians
cscd	Cosecant of argument in degrees
csch	Hyperbolic cosecant
hypot	Square root of sum of squares
sec	Secant of argument in radians
secd	Secant of argument in degrees
sech	Hyperbolic secant

sin	Sine of argument in radians
sind	Sine of argument in degrees
sinh	Hyperbolic sine of argument in radians
tan	Tangent of argument in radians
tand	Tangent of argument in degrees
tanh	Hyperbolic tangent

## **Exponential**

exp	Exponential
expm1	Compute $\exp(x)-1$ accurately for small values of $x$
log	Natural logarithm
log10	Common (base 10) logarithm
log1p	Compute $\log(1+x)$ accurately for small values of $x$
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
nextpow2	Next higher power of 2
nthroot	Real $n$ th root of real numbers
pow2	Base 2 power and scale floating-point numbers
power	Array power
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output



realsqrt	Square root for nonnegative real arrays
sqrt	Square root

## Complex

abs	Absolute value and complex magnitude
angle	Phase angle
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cplxpair	Sort complex numbers into complex conjugate pairs
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Check if input is real array
j	Imaginary unit
real	Real part of complex number
sign	Signum function
unwrap	Correct phase angles to produce smoother phase plots

## Rounding and Remainder

ceil	Round toward positive infinity
fix	Round toward zero
floor	Round toward negative infinity
idivide	Integer division with rounding option

mod	Modulus after division
rem	Remainder after division
round	Round to nearest integer

## **Discrete Math**

factor	Prime factors
factorial	Factorial function
gcd	Greatest common divisor
isprime	Array elements that are prime numbers
lcm	Least common multiple
nchoosek	Binomial coefficient or all combinations
perms	All possible permutations
primes	Generate list of prime numbers
rat	Rational fraction approximation
rats	Rational fraction approximation

## **Polynomials**

conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
poly	Polynomial with specified roots
polyder	Polynomial derivative
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting

polyint	Integrate polynomial analytically
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
residue	Convert between partial fraction expansion and polynomial coefficients
roots	Polynomial roots

## Interpolation and Computational Geometry

Interpolation (p. 1-47)	Data interpolation, data gridding, polynomial evaluation, nearest point search
Delaunay Triangulation (p. 1-48)	Delaunay triangulation, triangular surface and mesh plots
Convex Hull (p. 1-48)	Plot convex hull, plotting functions
Voronoi Diagrams (p. 1-49)	Plot Voronoi diagram, patch graphics object, plotting functions
Domain Generation (p. 1-49)	Generate arrays for 3-D plots, or for N-D functions and interpolation

## Interpolation

dsearchn	N-D nearest point search
griddata	Interpolate scattered data
griddatan	Data gridding and hypersurface fitting (dimension $\geq 2$ )
griddedInterpolant	Interpolant for gridded data
interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)

interpft	1-D interpolation using FFT method
interp	N-D data interpolation (table lookup)
meshgrid	Rectangular grid in 2-D and 3-D space
mkpp	Make piecewise polynomial
ndgrid	Rectangular grid in N-D space
padcoef	Padé approximation of time delays
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
ppval	Evaluate piecewise polynomial
spline	Cubic spline data interpolation
TriScatteredInterp	Interpolate scattered data
tsearchn	N-D closest simplex search
unmkpp	Piecewise polynomial details

## **Delaunay Triangulation**

delaunay	Delaunay triangulation
delaunayn	N-D Delaunay triangulation
size (TriRep)	Size of triangulation matrix
trimesh	Triangular mesh plot
triplot	2-D triangular plot
TriRep	Triangulation representation
trisurf	Triangular surface plot

## **Convex Hull**

convexHull (DelaunayTri)	Convex hull
convhull	Convex hull

<code>convhulln</code>	N-D convex hull
<code>patch</code>	Create one or more filled polygons
<code>trisurf</code>	Triangular surface plot

## Voronoi Diagrams

<code>patch</code>	Create one or more filled polygons
<code>voronoi</code>	Voronoi diagram
<code>voronoiDiagram (DelaunayTri)</code>	Voronoi diagram
<code>voronoin</code>	N-D Voronoi diagram

## Domain Generation

<code>meshgrid</code>	Rectangular grid in 2-D and 3-D space
<code>ndgrid</code>	Rectangular grid in N-D space

## Cartesian Coordinate System Conversion

<code>cart2pol</code>	Transform Cartesian coordinates to polar or cylindrical
<code>cart2sph</code>	Transform Cartesian coordinates to spherical
<code>pol2cart</code>	Transform polar or cylindrical coordinates to Cartesian
<code>sph2cart</code>	Transform spherical coordinates to Cartesian

## Nonlinear Numerical Methods

Ordinary Differential Equations (p. 1-50)	Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations (p. 1-51)	Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Boundary Value Problems (p. 1-52)	Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution
Partial Differential Equations (p. 1-52)	Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-53)	Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature) (p. 1-53)	Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

## Ordinary Differential Equations

decic	Compute consistent initial conditions for ode15i
deval	Evaluate solution of differential equation problem
ode113	Solve nonstiff differential equations; variable order method
ode15i	Solve fully implicit differential equations, variable order method
ode15s	Solve stiff differential equations and DAEs; variable order method

ode23	Solve nonstiff differential equations; low order method
ode23s	Solve stiff differential equations; low order method
ode23t	Solve moderately stiff ODEs and DAEs; trapezoidal rule
ode23tb	Solve stiff differential equations; low order method
ode45	Solve nonstiff differential equations; medium order method
odeget	Ordinary differential equation options parameters
odeset	Create or alter options structure for ordinary differential equation solvers
odextend	Extend solution of initial value problem for ordinary differential equation

## **Delay Differential Equations**

dde23	Solve delay differential equations (DDEs) with constant delays
ddeget	Extract properties from delay differential equations options structure
ddensd	Solve delay differential equations (DDEs) of neutral type
ddesd	Solve delay differential equations (DDEs) with general delays

<code>ddeset</code>	Create or alter delay differential equations options structure
<code>deval</code>	Evaluate solution of differential equation problem

### **Boundary Value Problems**

<code>bvp4c</code>	Solve boundary value problems for ordinary differential equations
<code>bvp5c</code>	Solve boundary value problems for ordinary differential equations
<code>bvpget</code>	Extract properties from options structure created with <code>bvpset</code>
<code>bvpinit</code>	Form initial guess for BVP solvers
<code>bvpset</code>	Create or alter options structure of boundary value problem
<code>bvpxtend</code>	Form guess structure for extending boundary value solutions
<code>deval</code>	Evaluate solution of differential equation problem

### **Partial Differential Equations**

<code>pdepe</code>	Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
<code>pdeval</code>	Evaluate numerical solution of PDE using output of <code>pdepe</code>



## Optimization

fminbnd	Find minimum of single-variable function on fixed interval
fminsearch	Find minimum of unconstrained multivariable function using derivative-free method
fzero	Find root of continuous function of one variable
lsqnonneg	Solve nonnegative least-squares constraints problem
optimget	Optimization options values
optimset	Create or edit optimization options structure

## Numerical Integration (Quadrature)

dblquad	Numerically evaluate double integral over rectangle
integral	Numerically evaluate integral
integral2	Numerically evaluate double integral
integral3	Numerically evaluate triple integral
quad	Numerically evaluate integral, adaptive Simpson quadrature
quad2d	Numerically evaluate double integral over planar region
quadgk	Numerically evaluate integral, adaptive Gauss-Kronrod quadrature
quadl	Numerically evaluate integral, adaptive Lobatto quadrature

quadv	Vectorized quadrature
triplequad	Numerically evaluate triple integral

## Special Functions

airy	Airy functions
besselh	Bessel function of third kind (Hankel function)
besseli	Modified Bessel function of first kind
besselj	Bessel function of first kind
besselk	Modified Bessel function of second kind
bessely	Bessel function of second kind
beta	Beta function
betainc	Incomplete beta function
betaincinv	Beta inverse cumulative distribution function
betaln	Logarithm of beta function
ellipj	Jacobi elliptic functions
ellipke	Complete elliptic integrals of first and second kind
erf	Error function
erfc	Complementary error function
erfcinv	Inverse complementary error function
erfcx	Scaled complementary error function
erfinv	Inverse error function
expint	Exponential integral
gamma	Gamma function

gammainc	Incomplete gamma function
gammaincinv	Inverse incomplete gamma function
gammaaln	Logarithm of gamma function
legendre	Associated Legendre functions

## Sparse Matrices

Elementary Sparse Matrices (p. 1-55)	Create random and nonrandom sparse matrices
Full to Sparse Conversion (p. 1-56)	Convert full matrix to sparse, sparse matrix to full
Sparse Matrix Manipulation (p. 1-56)	Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern
Reordering Algorithms (p. 1-57)	Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
Linear Algebra (p. 1-57)	Compute norms, eigenvalues, factorizations, least squares, structural rank
Linear Equations (Iterative Methods) (p. 1-58)	Methods for conjugate and biconjugate gradients, residuals, lower quartile
Tree Operations (p. 1-58)	Elimination trees, tree plotting, factorization analysis

## Elementary Sparse Matrices

spdiags	Extract and create sparse band and diagonal matrices
speye	Sparse identity matrix

sprand	Sparse uniformly distributed random matrix
sprandn	Sparse normally distributed random matrix
sprandsym	Sparse symmetric random matrix

### **Full to Sparse Conversion**

find	Find indices and values of nonzero elements
full	Convert sparse matrix to full matrix
sparse	Create sparse matrix
sconvert	Import matrix from sparse matrix external format

### **Sparse Matrix Manipulation**

issparse	Determine whether input is sparse
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
nzmax	Amount of storage allocated for nonzero matrix elements
spalloc	Allocate space for sparse matrix
spfun	Apply function to nonzero sparse matrix elements
spones	Replace nonzero sparse matrix elements with ones
spparms	Set parameters for sparse matrix routines
spy	Visualize sparsity pattern

## Reordering Algorithms

amd	Approximate minimum degree permutation
colamd	Column approximate minimum degree permutation
colperm	Sparse column permutation based on nonzero count
dmperm	Dulmage-Mendelsohn decomposition
ldl	Block LDL' factorization for Hermitian indefinite matrices
randperm	Random permutation
symamd	Symmetric approximate minimum degree permutation
symrcm	Sparse reverse Cuthill-McKee ordering

## Linear Algebra

cholinc	Sparse incomplete Cholesky and Cholesky-Infinity factorizations
condest	1-norm condition number estimate
eigs	Largest eigenvalues and eigenvectors of matrix
ichol	Incomplete Cholesky factorization
ilu	Sparse incomplete LU factorization
luinc	Sparse incomplete LU factorization
normest	2-norm estimate
spaugment	Form least squares augmented system

sprank	Structural rank
svds	Find singular values and vectors

### **Linear Equations (Iterative Methods)**

bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method
bicgstabl	Biconjugate gradients stabilized (l) method
cg	Conjugate gradients squared method
gmres	Generalized minimum residual method (with restarts)
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
symmlq	Symmetric LQ method
tfqmr	Transpose-free quasi-minimal residual method

### **Tree Operations**

etree	Elimination tree
etreeplot	Plot elimination tree
gplot	Plot nodes and links representing adjacency matrix
symbfact	Symbolic factorization analysis
treelayout	Lay out tree or forest

---

treeplot	Plot picture of tree
unmesh	Convert edge matrix to coordinate and Laplacian matrices

## Math Constants

eps	Floating-point relative accuracy
i	Imaginary unit
Inf	Infinity
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
j	Imaginary unit
NaN	Not-a-Number
pi	Ratio of circle's circumference to its diameter
realmax	Largest positive floating-point number
realmin	Smallest positive normalized floating-point number

## Data Analysis

Basic Operations (p. 1-60)	Sums, products, sorting
Descriptive Statistics (p. 1-60)	Statistical summaries of data
Filtering and Convolution (p. 1-61)	Data preprocessing
Interpolation and Regression (p. 1-61)	Data fitting
Fourier Transforms (p. 1-62)	Frequency content of data
Derivatives and Integrals (p. 1-62)	Data rates and accumulations
Time Series Objects (p. 1-63)	Methods for <code>timeseries</code> objects
Time Series Collections (p. 1-66)	Methods for <code>tscollection</code> objects

## Basic Operations

<code>brush</code>	Interactively mark, delete, modify, and save observations in graphs
<code>cumprod</code>	Cumulative product
<code>cumsum</code>	Cumulative sum
<code>linkdata</code>	Automatically update graphs when variables change
<code>prod</code>	Product of array elements
<code>sort</code>	Sort array elements in ascending or descending order
<code>sortrows</code>	Sort rows in ascending order
<code>sum</code>	Sum of array elements

## Descriptive Statistics

<code>corrcoef</code>	Correlation coefficients
<code>cov</code>	Covariance matrix



max	Largest elements in array
mean	Average or mean value of array
median	Median value of array
min	Smallest elements in array
mode	Most frequent values in array
std	Standard deviation
var	Variance

## Filtering and Convolution

conv	Convolution and polynomial multiplication
conv2	2-D convolution
convn	N-D convolution
deconv	Deconvolution and polynomial division
detrend	Remove linear trends
filter	1-D digital filter
filter2	2-D digital filter

## Interpolation and Regression

interp1	1-D data interpolation (table lookup)
interp2	2-D data interpolation (table lookup)
interp3	3-D data interpolation (table lookup)
interpn	N-D data interpolation (table lookup)
polyfit	Polynomial curve fitting
polyval	Polynomial evaluation

## Fourier Transforms

abs	Absolute value and complex magnitude
angle	Phase angle
cplxpair	Sort complex numbers into complex conjugate pairs
fft	Fast Fourier transform
fft2	2-D fast Fourier transform
fftn	N-D fast Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fftw	Interface to FFTW library run-time algorithm tuning control
ifft	Inverse fast Fourier transform
ifft2	2-D inverse fast Fourier transform
ifftn	N-D inverse fast Fourier transform
ifftshift	Inverse FFT shift
nextpow2	Next higher power of 2
unwrap	Correct phase angles to produce smoother phase plots

## Derivatives and Integrals

cumtrapz	Cumulative trapezoidal numerical integration
del2	Discrete Laplacian
diff	Differences and approximate derivatives
gradient	Numerical gradient
polyder	Polynomial derivative

polyint  
trapz

Integrate polynomial analytically  
Trapezoidal numerical integration

## Time Series Objects

Utilities (p. 1-63)

Combine `timeseries` objects, query and set `timeseries` object properties, plot `timeseries` objects

Data Manipulation (p. 1-64)

Add or delete data, manipulate `timeseries` objects

Event Data (p. 1-65)

Add or delete events, create new `timeseries` objects based on event data

Descriptive Statistics (p. 1-65)

Descriptive statistics for `timeseries` objects

## Utilities

`append (timeseries)`

Concatenate time series objects in time dimension

`get (timeseries)`

Query `timeseries` object property values

`getdatasamples (timeseries)`

Returns subset of time series samples using subscripted index array

`getdatasamplesize (timeseries)`

Size of data sample in `timeseries` object

`getqualitydesc (timeseries)`

Data quality descriptions

`getsamples (timeseries)`

Subset of time series samples using subscripted index array

`plot (timeseries)`

Plot time series

`set (timeseries)`

Set properties of `timeseries` object

<code>timeseries</code>	Create <code>timeseries</code> object
<code>tsdata.event</code>	Construct event object for <code>timeseries</code> object
<code>tstool</code>	Open Time Series Tools GUI

## **Data Manipulation**

<code>addsample (timeseries)</code>	Add data sample to <code>timeseries</code> object
<code>ctranspose (timeseries)</code>	Transpose <code>timeseries</code> object
<code>delsample (timeseries)</code>	Remove sample from <code>timeseries</code> object
<code>detrend (timeseries)</code>	Subtract mean or best-fit line and all NaNs from <code>timeseries</code> object
<code>filter (timeseries)</code>	Shape frequency content of time-series
<code>getabstime (timeseries)</code>	Extract date-string time vector into cell array
<code>getinterpmethod (timeseries)</code>	Interpolation method for <code>timeseries</code> object
<code>getsampleusingtime (timeseries)</code>	Extract data samples into new <code>timeseries</code> object
<code>idealfilter (timeseries)</code>	Apply ideal (noncausal) filter to <code>timeseries</code> object
<code>resample (timeseries)</code>	Select or interpolate <code>timeseries</code> data using new time vector
<code>setabstime (timeseries)</code>	Set times of <code>timeseries</code> object as date strings
<code>setinterpmethod (timeseries)</code>	Set default interpolation method for <code>timeseries</code> object
<code>setuniformtime (timeseries)</code>	Modify uniform time vector of <code>timeseries</code> object

---

<code>synchronize (timeseries)</code>	Synchronize and resample two <code>timeseries</code> objects using common time vector
<code>transpose (timeseries)</code>	Transpose <code>timeseries</code> object

## Event Data

<code>addevent</code>	Add event to <code>timeseries</code> object
<code>delevent</code>	Remove <code>tsdata.event</code> objects from <code>timeseries</code> object
<code>gettsafteratevent</code>	New <code>timeseries</code> object with samples occurring at or after event
<code>gettsafterevent</code>	New <code>timeseries</code> object with samples occurring after event
<code>gettsatevent</code>	New <code>timeseries</code> object with samples occurring at event
<code>gettsbeforeatevent</code>	New <code>timeseries</code> object with samples occurring before or at event
<code>gettsbeforeevent</code>	New <code>timeseries</code> object with samples occurring before event
<code>gettsbetweenevents</code>	New <code>timeseries</code> object with samples occurring between events

## Descriptive Statistics

<code>iqr (timeseries)</code>	Interquartile range of <code>timeseries</code> data
<code>max (timeseries)</code>	Maximum value of <code>timeseries</code> data
<code>mean (timeseries)</code>	Mean value of <code>timeseries</code> data
<code>median (timeseries)</code>	Median value of <code>timeseries</code> data
<code>min (timeseries)</code>	Minimum value of <code>timeseries</code> data

std (timeseries)	Standard deviation of timeseries data
sum (timeseries)	Sum of timeseries data
var (timeseries)	Variance of timeseries data

## **Time Series Collections**

Utilities (p. 1-66)	Query and set tscollection object properties, plot tscollection objects
Data Manipulation (p. 1-67)	Add or delete data, manipulate tscollection objects

## **Utilities**

get (tscollection)	Query tscollection object property values
isempty (tscollection)	Determine whether tscollection object is empty
length (tscollection)	Length of time vector
plot (timeseries)	Plot time series
set (tscollection)	Set properties of tscollection object
size (tscollection)	Size of tscollection object
tscollection	Create tscollection object
tstool	Open Time Series Tools GUI

## Data Manipulation

<code>addsampletocollection</code>	Add sample to <code>tscollection</code> object
<code>addts</code>	Add <code>timeseries</code> object to <code>tscollection</code> object
<code>delsamplefromcollection</code>	Remove sample from <code>tscollection</code> object
<code>getabstime (tscollection)</code>	Extract date-string time vector into cell array
<code>getsamplingsusingtime (tscollection)</code>	Extract data samples into new <code>tscollection</code> object
<code>gettimeseriesnames</code>	Cell array of names of <code>timeseries</code> objects in <code>tscollection</code> object
<code>horzcat (tscollection)</code>	Horizontal concatenation for <code>tscollection</code> objects
<code>removets</code>	Remove <code>timeseries</code> objects from <code>tscollection</code> object
<code>resample (tscollection)</code>	Select or interpolate data in <code>tscollection</code> using new time vector
<code>setabstime (tscollection)</code>	Set times of <code>tscollection</code> object as date strings
<code>settimeseriesnames</code>	Change name of <code>timeseries</code> object in <code>tscollection</code>
<code>vertcat (tscollection)</code>	Vertical concatenation for <code>tscollection</code> objects

## Programming and Data Types

Data Types (p. 1-68)	Numeric, character, structures, cell arrays, and data type conversion
Data Type Conversion (p. 1-76)	Convert one numeric type to another, numeric to string, string to numeric, structure to cell array, etc.
Operators and Special Characters (p. 1-79)	Arithmetic, relational, and logical operators, and special characters
Strings (p. 1-81)	Create, identify, manipulate, parse, evaluate, and compare strings
Bit-Wise Operations (p. 1-84)	Perform set, shift, and, or, compare, etc. on specific bit fields
Logical Operations (p. 1-84)	Evaluate conditions, testing for true or false
Relational Operations (p. 1-85)	Compare values for equality, greater than, less than, etc.
Set Operations (p. 1-86)	Find set members, unions, intersections, etc.
Date and Time Operations (p. 1-86)	Obtain information about dates and times
Programming in MATLAB (p. 1-87)	Function/expression evaluation, timed execution, memory, program control, error handling, MEX programming

### Data Types

Numeric Types (p. 1-69)	Integer and floating-point data
Characters and Strings (p. 1-70)	Characters and arrays of characters
Structures (p. 1-71)	Data of varying types and sizes stored in fields of a structure



Cell Arrays (p. 1-72)	Data of varying types and sizes stored in cells of array
Map Container Objects (p. 1-73)	Select elements of Map container using indices of various data types
Function Handles (p. 1-73)	Invoke a function indirectly via handle
Java Classes and Objects (p. 1-74)	Access Java classes through MATLAB interface
Data Type Identification (p. 1-75)	Determine data type of a variable

## **Numeric Types**

arrayfun	Apply function to each element of array
cast	Cast variable to different data type
cat	Concatenate arrays along specified dimension
class	Determine class of object
find	Find indices and values of nonzero elements
intmax	Largest value of specified integer type
intmin	Smallest value of specified integer type
ipermute	Inverse permute dimensions of N-D array
isa	Determine if input is object of specified class
isequal	Test arrays for equality
isequaln	Test arrays for equality, treating NaN values as equal

<code>isfinite</code>	Array elements that are finite
<code>isinf</code>	Array elements that are infinite
<code>isnan</code>	Array elements that are NaN
<code>isnumeric</code>	Determine if input is numeric array
<code>isreal</code>	Check if input is real array
<code>permute</code>	Rearrange dimensions of N-D array
<code>realmax</code>	Largest positive floating-point number
<code>realmin</code>	Smallest positive normalized floating-point number
<code>reshape</code>	Reshape array
<code>squeeze</code>	Remove singleton dimensions
<code>zeros</code>	Create array of all zeros

## **Characters and Strings**

See “Strings” on page 1-81 for all string-related functions.

<code>cellstr</code>	Create cell array of strings from character array
<code>char</code>	Convert to character array (string)
<code>eval</code>	Execute MATLAB expression in text string
<code>findstr</code>	Find string within another, longer string
<code>isstr</code>	Determine whether input is character array
<code>sprintf</code>	Format data into string
<code>sscanf</code>	Read formatted data from string
<code>strcat</code>	Concatenate strings horizontally

<code>strfind</code>	Find one string within another
<code>strings</code>	String handling
<code>strjust</code>	Justify character array
<code>strmatch</code>	Find possible matches for string
<code>strread</code>	Read formatted data from string
<code>strrep</code>	Find and replace substring
<code>strtrim</code>	Remove leading and trailing white space from string
<code>strvcat</code>	Concatenate strings vertically

## Structures

<code>arrayfun</code>	Apply function to each element of array
<code>cell2struct</code>	Convert cell array to structure array
<code>class</code>	Determine class of object
<code>deal</code>	Distribute inputs to outputs
<code>fieldnames</code>	Field names of structure, or public fields of object
<code>getfield</code>	Field of structure array
<code>isa</code>	Determine if input is object of specified class
<code>isequal</code>	Test arrays for equality
<code>isfield</code>	Determine whether input is structure array field
<code>isstruct</code>	Determine whether input is structure array
<code>orderfields</code>	Order fields of structure array
<code>rmfield</code>	Remove fields from structure
<code>setfield</code>	Assign values to structure array field

struct	Create structure array
struct2cell	Convert structure to cell array
structfun	Apply function to each field of scalar structure

## **Cell Arrays**

cell	Create cell array
cell2mat	Convert cell array to numeric array
cell2struct	Convert cell array to structure array
celldisp	Cell array contents
cellfun	Apply function to each cell in cell array
cellplot	Graphically display structure of cell array
cellstr	Create cell array of strings from character array
class	Determine class of object
deal	Distribute inputs to outputs
isa	Determine if input is object of specified class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
isequal	Test arrays for equality
mat2cell	Convert array to cell array with potentially different sized cells
struct2cell	Convert structure to cell array

## Map Container Objects

<code>containers.Map</code>	Map values to unique keys
<code>isKey (containers.Map)</code>	Determine if <code>containers.Map</code> object contains key
<code>keys (containers.Map)</code>	Identify keys of <code>containers.Map</code> object
<code>length (containers.Map)</code>	Length of <code>containers.Map</code> object
<code>remove (containers.Map)</code>	Remove key-value pairs from <code>containers.Map</code> object
<code>size (containers.Map)</code>	Size of <code>containers.Map</code> object
<code>values (containers.Map)</code>	Identify values in <code>containers.Map</code> object

## Function Handles

<code>class</code>	Determine class of object
<code>feval</code>	Evaluate function
<code>func2str</code>	Construct function name string from function handle
<code>functions</code>	Information about function handle
<code>function_handle (@)</code>	Handle used in calling functions indirectly
<code>isa</code>	Determine if input is object of specified class
<code>isequal</code>	Test arrays for equality
<code>str2func</code>	Construct function handle from function name string

## Java Classes and Objects

cell	Create cell array
class	Determine class of object
clear	Remove items from workspace, freeing up system memory
depfun	List dependencies of function or P-file
exist	Check existence of variable, function, folder, or class
fieldnames	Field names of structure, or public fields of object
im2java	Convert image to Java image
import	Add package or class to current import list
inmem	Names of functions, MEX-files, Java classes in memory
isa	Determine if input is object of specified class
isjava	Determine whether input is Java object
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Get and set Java class path
javaMethod	Invoke Java method
javaMethodEDT	Invoke Java method from Event Dispatch Thread (EDT)
javaObject	Invoke Java constructor

javaObjectEDT	Java object constructor on Event Dispatch Thread (EDT)
javarmppath	Remove entries from dynamic Java class path
methods	Class method names
methodsview	View class methods
usejava	Determine whether Java feature is supported in MATLAB software
which	Locate functions and files

### Data Type Identification

is*	Detect state
isa	Determine if input is object of specified class
iscell	Determine whether input is cell array
iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine if input is floating-point array
ishghandle	True for Handle Graphics® object handles
isinteger	Determine if input is integer array
isjava	Determine whether input is Java object
islogical	Determine if input is logical array

isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
isreal	Check if input is real array
isstr	Determine whether input is character array
isstruct	Determine whether input is structure array
validateattributes	Check validity of array
who	List variables in workspace
whos	List variables in workspace, with sizes and types

## Data Type Conversion

Numeric (p. 1-76)	Convert data of one numeric type to another numeric type
String to Numeric (p. 1-77)	Convert characters to numeric equivalent
Numeric to String (p. 1-77)	Convert numeric to character equivalent
Other Conversions (p. 1-78)	Convert to structure, cell array, function handle, etc.

## Numeric

cast	Cast variable to different data type
double	Convert to double precision
int16	Convert to 16-bit signed integer
int32	Convert to 32-bit signed integer
int64	Convert to 64-bit signed integer



<code>int8</code>	Convert to 8-bit signed integer
<code>single</code>	Convert to single precision
<code>typecast</code>	Convert data types without changing underlying data
<code>uint16</code>	Convert to 16-bit unsigned integer
<code>uint32</code>	Convert to 32-bit unsigned integer
<code>uint64</code>	Convert to 64-bit unsigned integer
<code>uint8</code>	Convert to 8-bit unsigned integer

### **String to Numeric**

<code>base2dec</code>	Convert base N number string to decimal number
<code>bin2dec</code>	Convert binary number string to decimal number
<code>cast</code>	Cast variable to different data type
<code>hex2dec</code>	Convert hexadecimal number string to decimal number
<code>hex2num</code>	Convert hexadecimal number string to double-precision number
<code>str2double</code>	Convert string to double-precision value
<code>str2num</code>	Convert string to number
<code>unicode2native</code>	Convert Unicode® characters to numeric bytes

### **Numeric to String**

<code>cast</code>	Cast variable to different data type
<code>char</code>	Convert to character array (string)

dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
int2str	Convert integer to string
mat2str	Convert matrix to string
native2unicode	Convert numeric bytes to Unicode characters
num2str	Convert number to string

### **Other Conversions**

cell2mat	Convert cell array to numeric array
cell2struct	Convert cell array to structure array
datestr	Convert date and time to string format
func2str	Construct function name string from function handle
logical	Convert numeric values to logical
mat2cell	Convert array to cell array with potentially different sized cells
num2hex	Convert singles and doubles to IEEE® hexadecimal strings
str2func	Construct function handle from function name string
str2mat	Form blank-padded character matrix from strings
struct2cell	Convert structure to cell array

## Operators and Special Characters

Arithmetic Operators (p. 1-79)	Plus, minus, power, left and right divide, transpose, etc.
Relational Operators (p. 1-80)	Equal to, greater than, less than or equal to, etc.
Logical Operators (p. 1-80)	Element-wise and short circuit and, or, not
Special Characters (p. 1-80)	Array constructors, line continuation, comments, etc.

### Arithmetic Operators

Arithmetic Operators + - * / \ ^ ' .*	Matrix and array arithmetic
+	Plus
-	Minus
.	Decimal point
=	Assignment
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power
'	Matrix transpose
.*	Array multiplication (element-wise)
./	Array right division (element-wise)
.\	Array left division (element-wise)
.^	Array power (element-wise)
.'	Array transpose

## Relational Operators

Relational Operators < > <= >= == Relational operations

~=

< Less than

<= Less than or equal to

> Greater than

>= Greater than or equal to

== Equal to

~= Not equal to

## Logical Operators

See also “Logical Operations” on page 1-84 for functions like xor, all, any, etc.

&& Logical AND

|| Logical OR

& Logical AND for arrays

| Logical OR for arrays

~ Logical NOT

## Special Characters

Special Characters [] () {} = ' . ... , ; Special characters

: % ! @

: Create vectors, subscript arrays, specify for-loop iterations

() Pass function arguments, prioritize operators

[] Construct array, concatenate elements, specify multiple outputs from function

{ }	Construct cell array, index into cell array
.	Insert decimal point, define structure field, reference methods of object
.( )	Reference dynamic field of structure
..	Reference parent directory
...	Continue statement to next line
,	Separate rows of array, separate function input/output arguments, separate commands
;	Separate columns of array, suppress output from current command
%	Insert comment line into code
%{ %}	Insert block of comments into code
!	Issue command to operating system
' '	Construct character array
@	Construct function handle, reference class directory

## Strings

Description of Strings in MATLAB (p. 1-82)	Basics of string handling in MATLAB
String Creation (p. 1-82)	Create strings, cell arrays of strings, concatenate strings together
String Identification (p. 1-82)	Identify characteristics of strings
String Manipulation (p. 1-83)	Convert case, strip blanks, replace characters
String Parsing (p. 1-83)	Formatted read, regular expressions, locate substrings
String Evaluation (p. 1-84)	Evaluate stated expression in string
String Comparison (p. 1-84)	Compare contents of strings

## Description of Strings in MATLAB

strings String handling

### String Creation

blanks Create string of blank characters  
cellstr Create cell array of strings from character array  
char Convert to character array (string)  
sprintf Format data into string  
strcat Concatenate strings horizontally  
strvcat Concatenate strings vertically

### String Identification

isa Determine if input is object of specified class  
iscellstr Determine whether input is cell array of strings  
ischar Determine whether item is character array  
isletter Array elements that are alphabetic letters  
isspace Array elements that are space characters  
isstrprop Determine whether string is of specified category  
validatestring Check validity of text string

## String Manipulation

<code>deblank</code>	Strip trailing blanks from end of string
<code>lower</code>	Convert string to lowercase
<code>strjust</code>	Justify character array
<code>strrep</code>	Find and replace substring
<code>strtrim</code>	Remove leading and trailing white space from string
<code>upper</code>	Convert string to uppercase

## String Parsing

<code>findstr</code>	Find string within another, longer string
<code>regexp</code>	Match regular expression (case sensitive)
<code>regexpi</code>	Match regular expression (case insensitive)
<code>regexprep</code>	Replace string using regular expression
<code>regexptranslate</code>	Translate string into regular expression
<code>sscanf</code>	Read formatted data from string
<code>strfind</code>	Find one string within another
<code>strread</code>	Read formatted data from string
<code>strtok</code>	Selected parts of string

## String Evaluation

<code>eval</code>	Execute MATLAB expression in text string
<code>evalc</code>	Evaluate MATLAB expression with capture
<code>evalin</code>	Execute MATLAB expression in specified workspace

## String Comparison

<code>strcmp</code>	Compare strings (case sensitive)
<code>strcmpi</code>	Compare strings (case insensitive)
<code>strmatch</code>	Find possible matches for string
<code>strncmp</code>	Compare first n characters of strings (case sensitive)
<code>strncmpi</code>	Compare first n characters of strings (case insensitive)

## Bit-Wise Operations

<code>bitmax</code>	Maximum double-precision floating-point integer
<code>swapbytes</code>	Swap byte ordering

## Logical Operations

<code>all</code>	Determine whether all array elements are nonzero or true
<code>and</code>	Find logical AND of array or scalar inputs



any	Determine whether any array elements are nonzero
false	Logical 0 (false)
find	Find indices and values of nonzero elements
isa	Determine if input is object of specified class
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
logical	Convert numeric values to logical
not	Find logical NOT of array or scalar input
or	Find logical OR of array or scalar inputs
true	Logical 1 (true)
xor	Logical exclusive-OR

See “Operators and Special Characters” on page 1-79 for logical operators.

## Relational Operations

eq	Test for equality
ge	Test for greater than or equal to
gt	Test for greater than
le	Test for less than or equal to
lt	Test for less than
ne	Test for inequality

See “Operators and Special Characters” on page 1-79 for relational operators.

## **Set Operations**

<code>intersect</code>	Find set intersection of two arrays
<code>ismember</code>	Array elements that are members of set
<code>issorted</code>	Determine whether set elements are in sorted order
<code>setdiff</code>	Find set difference of two arrays
<code>setxor</code>	Find set exclusive OR of two arrays
<code>union</code>	Find set union of two arrays
<code>unique</code>	Find unique values in array

## **Date and Time Operations**

<code>addtodate</code>	Modify date number by field
<code>calendar</code>	Calendar for specified month
<code>clock</code>	Current time as date vector
<code>cputime</code>	Elapsed CPU time
<code>date</code>	Current date string
<code>datenum</code>	Convert date and time to serial date number
<code>datestr</code>	Convert date and time to string format
<code>datevec</code>	Convert date and time to vector of components
<code>eomday</code>	Last day of month
<code>etime</code>	Time elapsed between date vectors

now	Current date and time
weekday	Day of week

## Programming in MATLAB

Functions and Scripts (p. 1-87)	Write and execute program code, interact with caller, check input and output values, dependencies
Evaluation (p. 1-88)	Evaluate expression in string, apply function to array, run script file, etc.
Timer (p. 1-89)	Schedule execution of MATLAB commands
Variables and Functions in Memory (p. 1-90)	List, lock, or clear functions in memory, construct variable names, consolidate workspaces, refresh caches
Control Flow (p. 1-91)	Conditional control, loop control, error control, program termination
Error Handling (p. 1-92)	Generate warnings and errors, test for and catch errors, capture data on cause of error, warning control
MEX Programming (p. 1-93)	Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

## Functions and Scripts

depsdir	List dependent folders for function or P-file
depsfun	List dependencies of function or P-file
echo	Display statements during function execution

end	Terminate block of code, or indicate last array index
function	Declare function name, inputs, and outputs
input	Request user input
inputname	Variable name of function input
inputParser	Parse function inputs
mfilename	File name of currently running function
namelengthmax	Maximum identifier length
nargchk	Validate number of input arguments
nargin	Number of function input arguments
narginchk	Validate number of input arguments
nargout	Number of function output arguments
nargoutchk	Validate number of output arguments
pcode	Create protected function file
script	Sequence of MATLAB statements in file
syntax	Two ways to call MATLAB functions
varargin	Variable-length input argument list
varargout	Variable-length output argument list

## **Evaluation**

ans	Most recent answer
arrayfun	Apply function to each element of array

assert	Generate error when condition is violated
cellfun	Apply function to each cell in cell array
echo	Display statements during function execution
eval	Execute MATLAB expression in text string
evalc	Evaluate MATLAB expression with capture
evalin	Execute MATLAB expression in specified workspace
feval	Evaluate function
iskeyword	Determine whether input is MATLAB keyword
isvarname	Determine whether input is valid variable name
pause	Halt execution temporarily
script	Sequence of MATLAB statements in file
structfun	Apply function to each field of scalar structure
symvar	Determine symbolic variables in expression
tic	Start clock to measure performance
toc	Stop clock to measure performance

## Timer

delete (timer)	Remove timer object from memory
disp (timer)	Information about timer object

get (timer)	Timer object properties
isvalid (timer)	Determine whether timer object is valid
set (timer)	Configure or display timer object properties
start	Start timer(s) running
startat	Start timer(s) running at specified time
stop	Stop timer(s)
timer	Construct timer object
timerfind	Find timer objects
timerfindall	Find timer objects, including invisible objects
wait	Wait until timer stops running

## **Variables and Functions in Memory**

ans	Most recent answer
assignin	Assign value to variable in specified workspace
clearvars	Clear variables from memory
datatipinfo	Produce short description of input variable
genvarname	Construct valid variable name from string
global	Declare global variables
inmem	Names of functions, MEX-files, Java classes in memory
isglobal	Determine whether input is global variable
memory	Display memory information

<code>mislocked</code>	Determine if function is locked in memory
<code>mlock</code>	Prevent clearing function from memory
<code>munlock</code>	Allow clearing functions from memory
<code>namelengthmax</code>	Maximum identifier length
<code>pack</code>	Consolidate workspace memory
<code>persistent</code>	Define persistent variable
<code>rehash</code>	Refresh function and file system path caches

## **Control Flow**

<code>break</code>	Terminate execution of <code>for</code> or <code>while</code> loop
<code>continue</code>	Pass control to next iteration of <code>for</code> or <code>while</code> loop
<code>end</code>	Terminate block of code, or indicate last array index
<code>error</code>	Display message and abort function
<code>for</code>	Execute statements specified number of times
<code>if/elseif/else</code>	Execute statements if condition is true
<code>parfor</code>	Parallel for loop
<code>return</code>	Return to invoking function
<code>switch/case/otherwise</code>	Switch among several cases based on expression

try/catch	Execute statements and catch resulting errors
while	Repeatedly execute statements while condition is true

## **Error Handling**

addCause (MException)	Record additional causes of exception
assert	Generate error when condition is violated
disp (MException)	Display MException object
eq (MException)	Compare scalar MException objects for equality
error	Display message and abort function
errorf	Information about file I/O errors
getErrorReport (MException)	Get error message for exception
isequal (MException)	Compare scalar MException objects for equality
last (MException)	Last uncaught exception
lastwarn	Last warning message
MException	Capture error information
ne (MException)	Compare scalar MException objects for inequality
rethrow	Reissue error
rethrow (MException)	Reissue existing exception
throw (MException)	Issue exception and terminate function
throwAsCaller (MException)	Throw exception as if from calling function



<code>try/catch</code>	Execute statements and catch resulting errors
<code>warning</code>	Warning message

## **MEX Programming**

<code>dbmex</code>	Enable MEX-file debugging (on UNIX platforms)
<code>inmem</code>	Names of functions, MEX-files, Java classes in memory
<code>mex</code>	Compile MEX-function from C/C++ or Fortran source code
<code>mex.getCompilerConfigurations</code>	Get compiler configuration information for building MEX-files
<code>mexext</code>	Binary MEX-file name extension

## Object-Oriented Programming

Classes and Objects (p. 1-94)	Get information about classes and objects
Handle Classes (p. 1-95)	Define and use handle classes
Events and Listeners (p. 1-96)	Define and use events and listeners
Meta-Classes (p. 1-96)	Access information about classes without requiring instances
Enumerations (p. 1-97)	Information about enumeration classes
Heterogeneous Arrays (p. 1-97)	

### Classes and Objects

class	Determine class of object
classdef	Class definition keywords
exist	Check existence of variable, function, folder, or class
inferiorto	Specify inferior class relationship
ismethod	Determine if method of object
isobject	Determine if input is MATLAB object
isprop	Determine if property of object
loadobj	Modify load process for object
methods	Class method names
methodsview	View class methods
properties	Class property names
saveobj	Modify save process for object
subsasgn	Subscripted assignment
subsindex	Subscript indexing with object

<code>suboref</code>	Redefine subscripted reference for objects
<code>substruct</code>	Create structure argument for <code>subsasgn</code> or <code>suboref</code>
<code>superclasses</code>	Superclass names
<code>superiorto</code>	Establish superior class relationship

## Handle Classes

<code>addlistener</code> (handle)	Create event listener
<code>addprop</code> (dynamicprops)	Add dynamic property
<code>copy</code> (matlab.mixin.Copyable)	Copy array of handle objects
<code>delete</code> (handle)	Handle object destructor
<code>dynamicprops</code>	Abstract class used to derive handle class with dynamic properties
<code>findobj</code> (handle)	Find handle objects matching specified conditions
<code>findprop</code> (handle)	Find <code>meta.property</code> object associated with property name
<code>get</code> (hgsetget)	Query property values of handle objects derived from <code>hgsetget</code> class
<code>getdisp</code> (hgsetget)	Override to change command window display
<code>handle</code>	Abstract class for deriving handle classes
<code>hgsetget</code>	Abstract class used to derive handle class with set and get methods
<code>isvalid</code> (handle)	Is object valid handle class object
<code>matlab.mixin.Copyable</code>	Superclass providing copy functionality for handle objects

<code>notify (handle)</code>	Notify listeners that event is occurring
<code>relationaloperators (handle)</code>	Equality and sorting of handle objects
<code>set (hgsetget)</code>	Assign property values to handle objects derived from <code>hgsetget</code> class
<code>setdisp (hgsetget)</code>	Override to change command window display

## Events and Listeners

<code>addlistener (handle)</code>	Create event listener
<code>event.EventData</code>	Base class for all data objects passed to event listeners
<code>event.listener</code>	Class defining listener objects
<code>event.PropertyEvent</code>	Listener for property events
<code>event.proplistener</code>	Define listener object for property events
<code>events</code>	Event names
<code>notify (handle)</code>	Notify listeners that event is occurring

## Meta-Classes

<code>meta.abstractDetails</code>	Find abstract methods and properties
<code>meta.class</code>	<code>meta.class</code> class describes MATLAB classes
<code>meta.class.fromName</code>	Return <code>meta.class</code> object associated with named class

<code>meta.DynamicProperty</code>	<code>meta.DynamicProperty</code> class describes dynamic property of MATLAB object
<code>meta.EnumeratedValue</code>	Describes enumeration members of MATLAB class
<code>meta.event</code>	<code>meta.event</code> class describes MATLAB class events
<code>meta.MetaData</code>	Superclass for MATLAB object metadata
<code>meta.method</code>	<code>meta.method</code> class describes MATLAB class methods
<code>meta.package</code>	<code>meta.package</code> class describes MATLAB packages
<code>meta.package.fromName</code>	Return <code>meta.package</code> object for specified package
<code>meta.package.getAllPackages</code>	Get all top-level packages
<code>meta.property</code>	<code>meta.property</code> class describes MATLAB class properties
<code>metaclass</code>	Obtain <code>meta.class</code> object

## Enumerations

<code>enumeration</code>	Display class enumeration members and names
--------------------------	---------------------------------------------

## Heterogeneous Arrays

<code>cat (matlab.mixin.Heterogeneous)</code>	Concatenation for heterogeneous arrays
<code>getDefaultScalarElement (matlab.mixin.Heterogeneous)</code>	Return default object for heterogeneous array operations

horzcat  
(matlab.mixin.Heterogeneous)

Horizontal concatenation for  
heterogeneous arrays

matlab.mixin.Heterogeneous

Superclass for heterogeneous array  
formation

vertcat  
(matlab.mixin.Heterogeneous)

Vertical concatenation for  
heterogeneous arrays

# Graphics

Basic Plots and Graphs (p. 1-99)	Linear line plots, log and semilog plots
Plotting Tools (p. 1-100)	GUIs for interacting with plots
Annotating Plots (p. 1-100)	Functions for and properties of titles, axes labels, legends, mathematical symbols
Specialized Plotting (p. 1-101)	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images (p. 1-105)	Display image object, read and write graphics file, convert to movie frames
Printing (p. 1-106)	Printing and exporting figures to standard formats
Handle Graphics (p. 1-106)	Creating graphics objects, setting properties, finding handles

## Basic Plots and Graphs

box	Axes border
errorbar	Plot error bars along curve
hold	Retain current graph when adding new graphs
line	Create line object
LineStyle (Line Specification)	Line specification string syntax
loglog	Log-log scale plot
plot	2-D line plot
plot3	3-D line plot
plotyy	2-D line plots with y-axes on both left and right side

polar	Polar coordinate plot
semilogx	Semilogarithmic plot
semilogy	Semilogarithmic plot
subplot	Create axes in tiled positions
xlim	Set or query $x$ -axis limits
ylim	Set or query $y$ -axis limits
zlim	Set or query $z$ -axis limits

## Plotting Tools

figurepalette	Show or hide
pan	Pan view of graph interactively
plotbrowser	Show or hide figure
plotedit	Interactively edit and annotate plots
plottools	Show or hide plot tools
propertyeditor	Show or hide
rotate3d	Rotate 3-D view using mouse
showplottool	Show or hide figure plot tool
zoom	Turn zooming on or off or magnify by factor

## Annotating Plots

annotation	Create annotation objects
clabel	Contour plot elevation labels
datacursormode	Enable, disable, and manage interactive data cursor mode
datetick	Date formatted tick labels
gtext	Mouse placement of text in 2-D view



legend	Graph legend for lines and patches
rectangle	Create 2-D rectangle object
textlabel	Produce TeX format from character string
title	Add title to current axes
xlabel	Label $x$ -axis
ylabel	Label $y$ -axis
zlabel	Label $z$ -axis

## Specialized Plotting

Area, Bar, and Pie Plots (p. 1-102)	1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-102)	Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-102)	Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-103)	Stair, step, and stem plots
Function Plots (p. 1-103)	Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-103)	Plots for showing distributions of data
Polygons and Surfaces (p. 1-104)	Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-104)	Plots of point distributions
Animation (p. 1-104)	Functions to create and play movies of plots

**Area, Bar, and Pie Plots**

area	Filled area 2-D plot
bar3	Plot 3-D bar graph
bar3h	Plot horizontal 3-D bar graph
barh	Plot bar graph horizontally
pareto	Pareto chart
pie	Pie chart
pie3	3-D pie chart

**Contour Plots**

contour	Contour plot of matrix
contour3	3-D contour plot
contourc	Low-level contour plot computation
contourf	Filled 2-D contour plot
ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter

**Direction and Velocity Plots**

comet	2-D comet plot
comet3	3-D comet plot
compass	Plot arrows emanating from origin
feather	Plot velocity vectors
quiver	Quiver or velocity plot
quiver3	3-D quiver or velocity plot

## Discrete Data Plots

stairs	Stairstep graph
stem	Plot discrete sequence data
stem3	Plot 3-D discrete sequence data

## Function Plots

ezcontour	Easy-to-use contour plotter
ezcontourf	Easy-to-use filled contour plotter
ezmesh	Easy-to-use 3-D mesh plotter
ezmeshc	Easy-to-use combination mesh/contour plotter
ezplot	Easy-to-use function plotter
ezplot3	Easy-to-use 3-D parametric curve plotter
ezpolar	Easy-to-use polar coordinate plotter
ezsurf	Easy-to-use 3-D colored surface plotter
ezsurfc	Easy-to-use combination surface/contour plotter
fplot	Plot function between specified limits

## Histograms

hist	Histogram plot
histc	Histogram count
rose	Angle histogram plot

## **Polygons and Surfaces**

cylinder	Generate cylinder
delaunay	Delaunay triangulation
delaunayn	N-D Delaunay triangulation
ellipsoid	Generate ellipsoid
fill	Filled 2-D polygons
fill3	Filled 3-D polygons
inpolygon	Points inside polygonal region
pcolor	Pseudocolor (checkerboard) plot
polyarea	Area of polygon
rectint	Rectangle intersection area
ribbon	Ribbon plot
slice	Volumetric slice plot
sphere	Generate sphere
waterfall	Waterfall plot

## **Scatter/Bubble Plots**

plotmatrix	Scatter plot matrix
scatter	Scatter plot
scatter3	3-D scatter plot

## **Animation**

frame2im	Return image data associated with movie frame
getframe	Capture movie frame
im2frame	Convert image to movie frame

movie	Play recorded movie frames
noanimate	Change EraseMode of all objects to normal

## Bit-Mapped Images

cmpermute	Rearrange colors in colormap
cmunique	Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image
dither	Convert image, increasing apparent color resolution by dithering
frame2im	Return image data associated with movie frame
im2frame	Convert image to movie frame
im2java	Convert image to Java image
image	Display image object
imagesc	Scale data and display image object
imapprox	Approximate indexed image by reducing number of colors
imfinfo	Information about graphics file
imformats	Manage image file format registry
imread	Read image from graphics file
imwrite	Write image to graphics file
ind2rgb	Convert indexed image to RGB image
rgb2ind	Convert RGB image to indexed image

## Printing

hgexport	Export figure
orient	Hardcopy paper orientation
print	Print figure or save to file
printdlg	Print dialog box
printopt	Configure printer defaults
printpreview	Preview figure to print
saveas	Save figure or Simulink block diagram using specified format

## Handle Graphics

Graphics Object Identification (p. 1-106)	Find and manipulate graphics objects via their handles
Object Creation (p. 1-107)	Create core graphics objects
Annotation Objects (p. 1-108)	Property descriptions for annotation objects
Plot Objects (p. 1-108)	Property descriptions for plot objects
Figure Windows (p. 1-109)	Control and save figures
Axes Operations (p. 1-109)	Operate on axes objects
Object Property Operations (p. 1-110)	Query, set, and link object properties

## Graphics Object Identification

allchild	Find all children of specified objects
ancestor	Ancestor of graphics object
copyobj	Copy graphics objects and their descendants
delete	Remove files or graphics objects

findall	Find all graphics objects
findfigs	Find visible offscreen figures
findobj	Locate graphics objects with specific properties
gca	Current axes handle
gcbf	Handle of figure containing object whose callback is executing
gcbo	Handle of object whose callback is executing
gco	Handle of current object
get	Query Handle Graphics object properties
ishandle	Test for valid graphics object handle
propedit	Open Property Editor
set	Set Handle Graphics object properties

## Object Creation

axes	Create axes graphics object
figure	Create figure graphics object
hgggroup	Create hgggroup object
hgtransform	Create hgtransform graphics object
image	Display image object
light	Create light object
line	Create line object
patch	Create one or more filled polygons
rectangle	Create 2-D rectangle object
root object	Root

surface	Create surface object
text	Create text object in current axes
uicontextmenu	Create context menu

## **Annotation Objects**

Annotation Arrow Properties	Define annotation arrow properties
Annotation Doublearrow Properties	Define annotation doublearrow properties
Annotation Ellipse Properties	Define annotation ellipse properties
Annotation Line Properties	Define annotation line properties
Annotation Rectangle Properties	Define annotation rectangle properties
Annotation Textarrow Properties	Define annotation textarrow properties
Annotation Textbox Properties	Define annotation textbox properties

## **Plot Objects**

Areaseries Properties	Define areaseries properties
Barseries Properties	Define barseries properties
Contourgroup Properties	Define contourgroup properties
Errorbarseries Properties	Description of errorbarseries properties
Lineseries Properties	Description of lineseries properties
Quivergroup Properties	Define quivergroup properties
Scattergroup Properties	Define scattergroup properties
Stairseries Properties	Define stairseries properties
Stemseries Properties	Define stemseries properties
Surfaceplot Properties	Define surfaceplot properties



## Figure Windows

clf	Clear current figure window
close	Remove specified figure
closereq	Default figure close request function
drawnow	Flush event queue and update figure window
gcf	Current figure handle
hgload	Load Handle Graphics object hierarchy from file
hgsave	Save Handle Graphics object hierarchy to file
newplot	Determine where to draw graphics objects
opengl	Control OpenGL® rendering
refresh	Redraw current figure
saveas	Save figure or Simulink block diagram using specified format
shg	Show most recent graph window

## Axes Operations

axis	Axis scaling and appearance
box	Axes border
cla	Clear current axes
gca	Current axes handle
grid	Grid lines for 2-D and 3-D plots
ishold	Current hold state
makehgtform	Create 4-by-4 transform matrix

## **Object Property Operations**

get	Query Handle Graphics object properties
linkaxes	Synchronize limits of specified 2-D axes
linkprop	Keep same value for corresponding properties of graphics objects
refreshdata	Refresh data in graph when data source is specified
set	Set Handle Graphics object properties
waitfor	Block execution and wait for event or condition

## 3-D Visualization

Surface and Mesh Plots (p. 1-111)	Plot matrices, visualize functions of two variables, specify colormap
View Control (p. 1-113)	Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits
Lighting (p. 1-115)	Add and control scene lighting
Transparency (p. 1-115)	Specify and control object transparency
Volume Visualization (p. 1-116)	Visualize gridded volume data

### Surface and Mesh Plots

Surface and Mesh Creation (p. 1-111)	Visualizing gridded and triangulated data as lines and surfaces
Domain Generation (p. 1-112)	Gridding data and creating arrays
Color Operations (p. 1-112)	Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds

### Surface and Mesh Creation

hidden	Remove hidden lines from mesh plot
mesh	Mesh plot
meshc	Plot a contour graph under mesh graph
meshz	Plot a curtain around mesh plot
peaks	Example function of two variables
surf	3-D shaded surface plot
surface	Create surface object

surf	Contour plot under a 3-D shaded surface plot
surf	Surface plot with colormap-based lighting
tetramesh	Tetrahedron mesh plot
trimesh	Triangular mesh plot
triplot	2-D triangular plot
trisurf	Triangular surface plot

### **Domain Generation**

meshgrid	Rectangular grid in 2-D and 3-D space
----------	---------------------------------------

### **Color Operations**

brighten	Brighten or darken colormap
caxis	Color axis scaling
colorbar	Colorbar showing color scale
colordef	Set default property values to display different color schemes
colormap	Set and get current colormap
colormapeditor	Open colormap editor
ColorSpec (Color Specification)	Color specification
contrast	Grayscale colormap for contrast enhancement
graymon	Set default figure properties for grayscale monitors
hsv2rgb	Convert HSV colormap to RGB colormap

rgb2hsv	Convert RGB colormap to HSV colormap
rgbplot	Plot colormap
shading	Set color shading properties
spinmap	Spin colormap
surfnorm	Compute and display 3-D surface normals
whitebg	Change axes background color

## View Control

Camera Viewpoint (p. 1-113)	Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Aspect Ratio and Axis Limits (p. 1-114)	Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-114)	Panning, rotating, and zooming views
Region of Interest (p. 1-115)	Interactively identifying rectangular regions

## Camera Viewpoint

camdolly	Move camera position and target
cameratoolbar	Control camera toolbar programmatically
camlookat	Position camera to view object or group of objects
camorbit	Rotate camera position around camera target
campan	Rotate camera target around camera position

campos	Set or query camera position
camproj	Set or query projection type
camroll	Rotate camera about view axis
camtarget	Set or query location of camera target
camup	Set or query camera up vector
camva	Set or query camera view angle
camzoom	Zoom in and out on scene
makehgtform	Create 4-by-4 transform matrix
view	Viewpoint specification
viewmtx	View transformation matrices

### **Aspect Ratio and Axis Limits**

daspect	Set or query axes data aspect ratio
pbaspect	Set or query plot box aspect ratio
xlim	Set or query <i>x</i> -axis limits
ylim	Set or query <i>y</i> -axis limits
zlim	Set or query <i>z</i> -axis limits

### **Object Manipulation**

pan	Pan view of graph interactively
reset	Reset graphics object properties to their defaults
rotate	Rotate object about specified origin and direction
rotate3d	Rotate 3-D view using mouse

selectmoveresize	Select, move, resize, or copy axes and uicontrol graphics objects
zoom	Turn zooming on or off or magnify by factor

## Region of Interest

dragrect	Drag rectangles with mouse
rbbox	Create rubberband box for area selection

## Lighting

camlight	Create or move light object in camera coordinates
diffuse	Calculate diffuse reflectance
light	Create light object
lightangle	Create or position light object in spherical coordinates
lighting	Specify lighting algorithm
material	Control reflectance properties of surfaces and patches
specular	Calculate specular reflectance

## Transparency

alim	Set or query axes alpha limits
alpha	Set transparency properties for objects in current axes
alphamap	Specify figure alphamap (transparency)

## Volume Visualization

coneplot	Plot velocity vectors as cones in 3-D vector field
contourslice	Draw contours in volume slice planes
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
flow	Simple function of three variables
interpstreamspeed	Interpolate stream-line vertices from flow speed
isocaps	Compute isosurface end-cap geometry
isocolors	Calculate isosurface and patch colors
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
reducepatch	Reduce number of patch faces
reducevolume	Reduce number of elements in volume data set
shrinkfaces	Reduce size of patch faces
slice	Volumetric slice plot
smooth3	Smooth 3-D data
stream2	Compute 2-D streamline data
stream3	Compute 3-D streamline data
streamline	Plot streamlines from 2-D or 3-D vector data
streamparticles	Plot stream particles
streamribbon	3-D stream ribbon plot from vector volume data



---

streamslice	Plot streamlines in slice planes
streamtube	Create 3-D stream tube plot
subvolume	Extract subset of volume data set
surf2patch	Convert surface data to patch data
volumebounds	Coordinate and color limits for volume data

## GUI Development

Predefined Dialog Boxes (p. 1-118)	Dialog boxes for error, user input, waiting, etc.
User Interface Deployment (p. 1-119)	Open GUIs, create the handles structure
User Interface Development (p. 1-120)	Start GUIDE, manage application data, get user input
User Interface Objects (p. 1-120)	Create GUI components
Objects from Callbacks (p. 1-121)	Find object handles from within callbacks functions
GUI Utilities (p. 1-121)	Move objects, wrap text
Program Execution (p. 1-122)	Wait and resume based on user input

### Predefined Dialog Boxes

<code>dialog</code>	Create and display empty dialog box
<code>errordlg</code>	Create and open error dialog box
<code>export2wsdlg</code>	Export variables to workspace
<code>helpdlg</code>	Create and open help dialog box
<code>inputdlg</code>	Create and open input dialog box
<code>listdlg</code>	Create and open list-selection dialog box
<code>msgbox</code>	Create and open message box
<code>printdlg</code>	Print dialog box
<code>printpreview</code>	Preview figure to print
<code>questdlg</code>	Create and open question dialog box
<code>uigetdir</code>	Open standard dialog box for selecting directory

<code>uigetfile</code>	Open standard dialog box for retrieving files
<code>uigetpref</code>	Specify and conditionally open dialog box according to user preference
<code>uiopen</code>	Interactively select file to open and load data
<code>uiputfile</code>	Open standard dialog box for saving files
<code>uisave</code>	Interactively save workspace variables to MAT-file
<code>uisetcolor</code>	Open standard dialog box for setting object's <code>ColorSpec</code>
<code>uisetfont</code>	Open standard dialog box for setting object's font characteristics
<code>waitbar</code>	Open or update wait bar dialog box
<code>warndlg</code>	Open warning dialog box

## User Interface Deployment

<code>guidata</code>	Store or retrieve GUI data
<code>guihandles</code>	Create structure of handles
<code>movegui</code>	Move GUI figure to specified location on screen
<code>openfig</code>	Open new copy or raise existing copy of saved figure
<code>setpixelposition</code>	Set component position in pixels

## User Interface Development

<code>getappdata</code>	Value of application-defined data
<code>ginput</code>	Graphical input from mouse or cursor
<code>guidata</code>	Store or retrieve GUI data
<code>guide</code>	Open GUI Layout Editor
<code>inspect</code>	Open Property Inspector
<code>isappdata</code>	True if application-defined data exists
<code>rmapdata</code>	Remove application-defined data
<code>setappdata</code>	Specify application-defined data
<code>uigetpref</code>	Specify and conditionally open dialog box according to user preference
<code>uisetpref</code>	Manage preferences used in <code>uigetpref</code>
<code>waitfor</code>	Block execution and wait for event or condition
<code>waitforbuttonpress</code>	Wait for key press or mouse-button click

## User Interface Objects

<code>matlab.apputil.create</code>	Create or modify app project file for packaging app into <code>.mlappinstall</code> file using interactive dialog box
<code>matlab.apputil.package</code>	Package app files into <code>.mlappinstall</code> file
<code>menu</code>	Generate menu of choices for user input

<code>uibuttongroup</code>	Create container object to exclusively manage radio buttons and toggle buttons
<code>uicontextmenu</code>	Create context menu
<code>uicontrol</code>	Create user interface control object
<code>uimenu</code>	Create menus and menu items on figure windows
<code>uipanel</code>	Create panel container object
<code>uipushtool</code>	Create push button on toolbar
<code>uitable</code>	Create 2-D graphic table GUI component
<code>uitoggletool</code>	Create toggle button on toolbar
<code>uitoolbar</code>	Create toolbar on figure

## Objects from Callbacks

<code>findall</code>	Find all graphics objects
<code>findfigs</code>	Find visible offscreen figures
<code>findobj</code>	Locate graphics objects with specific properties
<code>gcbf</code>	Handle of figure containing object whose callback is executing
<code>gcbo</code>	Handle of object whose callback is executing

## GUI Utilities

<code>align</code>	Align user interface controls ( <code>uicontrols</code> ) and axes
<code>getpixelposition</code>	Get component position in pixels
<code>listfonts</code>	List available system fonts

<code>movegui</code>	Move GUI figure to specified location on screen
<code>selectmoveresize</code>	Select, move, resize, or copy axes and uicontrol graphics objects
<code>setpixelposition</code>	Set component position in pixels
<code>textwrap</code>	Wrapped string matrix for given uicontrol
<code>uisetcolor</code>	Open standard dialog box for setting object's <code>ColorSpec</code>
<code>uisetfont</code>	Open standard dialog box for setting object's font characteristics
<code>uistack</code>	Reorder visual stacking order of objects

## **Program Execution**

<code>addpref</code>	Add preference
<code>getpref</code>	Preference
<code>ispref</code>	Test for existence of preference
<code>rmpref</code>	Remove preference
<code>setpref</code>	Set preference
<code>uigetpref</code>	Specify and conditionally open dialog box according to user preference
<code>uiresume</code>	Resume execution of blocked program
<code>uisetpref</code>	Manage preferences used in <code>uigetpref</code>
<code>uiwait</code>	Block program execution and wait to resume

`waitfor`

Block execution and wait for event  
or condition

`waitforbuttonpress`

Wait for key press or mouse-button  
click

## External Interfaces

Shared Libraries (p. 1-124)	Access functions stored in external shared library files
Java (p. 1-124)	Work with objects constructed from Java API and third-party class packages
.NET (p. 1-125)	Work with objects constructed from .NET assemblies
Component Object Model and ActiveX (p. 1-127)	Integrate COM components into your application
Web Services (p. 1-129)	Communicate between applications over a network using SOAP and WSDL
Serial Port Devices (p. 1-130)	Read and write to devices connected to your computer's serial port

See also *MATLAB C/C++ and Fortran API Reference* for functions you can use in external routines that interact with MATLAB programs and the data in MATLAB workspaces.

## Shared Libraries

libfunctions	Return information on functions in shared library
libfunctionsview	Display shared library function signatures in window

## Java

class	Determine class of object
fieldnames	Field names of structure, or public fields of object



import	Add package or class to current import list
inspect	Open Property Inspector
isa	Determine if input is object of specified class
isjava	Determine whether input is Java object
javaaddpath	Add entries to dynamic Java class path
javaArray	Construct Java array
javachk	Generate error message based on Java feature support
javaclasspath	Get and set Java class path
javaMethod	Invoke Java method
javaMethodEDT	Invoke Java method from Event Dispatch Thread (EDT)
javaObject	Invoke Java constructor
javaObjectEDT	Java object constructor on Event Dispatch Thread (EDT)
javarmpath	Remove entries from dynamic Java class path
methods	Class method names
methodsview	View class methods
usejava	Determine whether Java feature is supported in MATLAB software

## **.NET**

BeginInvoke	Initiate asynchronous .NET delegate call
bitnot	Bit-wise NOT

<code>cell</code>	Create cell array
<code>Combine</code>	Convenience function for static <code>.NET System.Delegate Combine</code> method
<code>enableNETfromNetworkDrive</code>	Enable access to <code>.NET</code> commands from network drive
<code>EndInvoke</code>	Retrieve result of asynchronous call initiated by <code>.NET System.Delegate BeginInvoke</code> method
<code>NET.addAssembly</code>	Make <code>.NET</code> assembly visible to MATLAB
<code>NET.Assembly</code>	Members of <code>.NET</code> assembly
<code>NET.convertArray</code>	Convert numeric MATLAB array to <code>.NET</code> array
<code>NET.createArray</code>	Array for nonprimitive <code>.NET</code> types
<code>NET.createGeneric</code>	Create instance of specialized <code>.NET</code> generic type
<code>NET.disableAutoRelease</code>	Lock <code>.NET</code> object representing a <code>RunTime Callable Wrapper (COM Wrapper)</code> so that MATLAB does not release COM object
<code>NET.enableAutoRelease</code>	Unlock <code>.NET</code> object representing a <code>RunTime Callable Wrapper (COM Wrapper)</code> so that MATLAB releases COM object
<code>NET.GenericClass</code>	Represent parameterized generic type definitions
<code>NET.invokeGenericMethod</code>	Invoke generic method of object
<code>NET.NetException</code>	Capture error information for <code>.NET</code> exception
<code>NET.setStaticProperty</code>	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

## Component Object Model and ActiveX

actxcontrol	Create Microsoft ActiveX <sup>®</sup> control in figure window
actxcontrollist	List currently installed Microsoft ActiveX controls
actxcontrolselect	Create Microsoft ActiveX control from GUI
actxGetRunningServer	Handle to running instance of Automation server
actxserver	Create COM server
addproperty	Add custom property to COM object
delete (COM)	Remove COM control or server
deleteproperty	Remove custom property from COM object
enableservice	Enable, disable, or report status of MATLAB Automation server
eventlisteners	List event handler functions associated with COM object events
events (COM)	List of events COM object can trigger
Execute	Execute MATLAB command in Automation server
Feval (COM)	Evaluate MATLAB function in Automation server
fieldnames	Field names of structure, or public fields of object

get (COM)	Get property value from interface, or display properties
GetCharArray	Character array from Automation server
GetFullMatrix	Matrix from Automation server workspace
GetVariable	Data from variable in Automation server workspace
GetWorkspaceData	Data from Automation server workspace
inspect	Open Property Inspector
interfaces	List custom interfaces exposed by COM server object
invoke	Invoke method on COM object or interface, or display methods
isa	Determine if input is object of specified class
iscom	Determine whether input is COM or ActiveX object
isevent	Determine whether input is COM object event
isinterface	Determine whether input is COM interface
ismethod (COM)	Determine whether input is COM object method
isprop (COM)	Determine whether input is COM object property
load (COM)	Initialize control object from file
MaximizeCommandWindow	Open Automation server window
methods	Class method names
methodsview	View class methods

MinimizeCommandWindow	Minimize size of Automation server window
move	Move or resize control in parent window
propedit (COM)	Open built-in property page for control
PutCharArray	Store character array in Automation server
PutFullMatrix	Matrix in Automation server workspace
PutWorkspaceData	Data in Automation server workspace
Quit (COM)	Terminate MATLAB Automation server
registerevent	Associate event handler for COM object event at run time
release	Release COM interface
save (COM)	Serialize control object to file
set (COM)	Set object or interface property to specified value
unregisterallevents	Unregister all event handlers associated with COM object events at run time
unregisterevent	Unregister event handler associated with COM object event at run time

## Web Services

callSoapService	Send SOAP message to endpoint
createClassFromWsd	Create MATLAB class based on WSDL document

<code>createSoapMessage</code>	Create SOAP message to send to server
<code>parseSoapResponse</code>	Convert response string from SOAP server into MATLAB types

## **Serial Port Devices**

<code>clear (serial)</code>	Remove serial port object from MATLAB workspace
<code>delete (serial)</code>	Remove serial port object from memory
<code>disp (serial)</code>	Serial port object summary information
<code>fclose (serial)</code>	Disconnect serial port object from device
<code>fgetl (serial)</code>	Read line of text from device and discard terminator
<code>fgets (serial)</code>	Read line of text from device and include terminator
<code>fopen (serial)</code>	Connect serial port object to device
<code>fprintf (serial)</code>	Write text to device
<code>fread (serial)</code>	Read binary data from device
<code>fscanf (serial)</code>	Read data from device, and format as text
<code>fwrite (serial)</code>	Write binary data to device
<code>get (serial)</code>	Serial port object properties
<code>instrcallback</code>	Event information when event occurs
<code>instrfind</code>	Read serial port objects from memory to MATLAB workspace

---

<code>instrfindall</code>	Find visible and hidden serial port objects
<code>isvalid (serial)</code>	Determine whether serial port objects are valid
<code>length (serial)</code>	Length of serial port object array
<code>load (serial)</code>	Load serial port objects and variables into MATLAB workspace
<code>readasync</code>	Read data asynchronously from device
<code>record</code>	Record data and event information to file
<code>save (serial)</code>	Save serial port objects and variables to file
<code>serial</code>	Create serial port object
<code>serialbreak</code>	Send break to device connected to serial port
<code>set (serial)</code>	Configure or display serial port object properties
<code>size (serial)</code>	Size of serial port object array
<code>stopasync</code>	Stop asynchronous read and write operations





# 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.getNumAttributes  
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  
cirshift  
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  
createClassFromWsd  
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  
delaunay3  
delaunayn  
delete  
delete (COM)  
delete  
delete (handle)  
delete (serial)  
delete (timer)  
deleteproperty  
delevent  
delsamplefromcollection  
demo  
depdir  
depfun  
det  
detrnd  
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  
dsearch  
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  
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  
gammanc  
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)  
getsamplingtime (tscollection)  
getTag  
getTagNames  
gettimeseriesnames  
gettsafteratevent  
gettsafterevent  
gettsatevent  
gettsbeforeatevent  
gettsbeforeevent  
gettsbetweenevents  
GetVariable  
getVersion  
GetWorkspaceData  
ginput  
global

---

gmres  
gplot  
grabcode  
gradient  
graymon  
grid  
griddata  
griddata3  
griddatan  
griddedInterpolant  
gsvd  
gt  
gtext  
guidata  
guide  
guihandles  
gunzip  
gzip  
h5create  
h5disp  
h5info  
h5read  
h5readatt  
h5write  
h5writeatt  
hadamard  
handle  
hankel  
hdf  
hdf5  
hdf5info  
hdf5read  
hdf5write  
hdfinfo  
hdfread  
hdftool  
help

helpbrowser  
helpdesk  
helpdlg  
helpwin  
hess  
matlab.mixin.Heterogeneous  
cat  
getDefaultScalarElement  
horzcat  
vertcat  
hex2dec  
hex2num  
hgexport  
hggroup  
Hgroup Properties  
hload  
hgsave  
hgsetget  
hgtransform  
Hgtransform Properties  
hidden  
hilb  
hist  
histe  
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  
interpfn  
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  
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.NetException  
NET.setStaticProperty  
ncreate  
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  
padcoef  
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  
plottedit  
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  
Remove  
RemoveAll  
timeseries  
addsample  
append  
ctranspose  
delsample  
detrend  
filter  
get  
getabstime  
getdatasamples  
getdatasamplesize  
getinterpmethod  
getqualitydesc  
getsamples  
getsamplesusingtime  
idealfilter  
iqr  
max  
mean  
median  
min  
plot  
resample  
set  
setabstime  
setinterpmethod

setuniformtime  
synchronize  
transpose  
std  
sum  
var  
real  
realloc  
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  
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  
strjust  
strmatch  
strncmp  
strncmpi  
stread  
strrep  
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  
sybmfact  
symmlq  
symrcm  
symvar  
syntax  
system  
tan  
tand  
tanh  
tar  
tempdir  
tempname  
tetramesh  
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  
tsearch  
tsearchn  
tstool  
type  
typecast  
uibbuttongroup  
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  
wavinfo

wavplay  
wavread  
wavrecord  
wavwrite  
web  
weekday  
what  
whatsnew  
which  
while  
whitebg  
who  
who  
whos  
whos  
wilkinson  
winopen  
winqueryreg  
wk1info  
wk1read  
wk1write  
workspace  
write  
writeDirectory  
writeEncodedStrip  
writeEncodedTile  
writeVideo  
xlabel  
ylabel  
xlabel  
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,  $B/A = (A' \setminus B')$ '. 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,  $A \setminus B$  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  $X = A \setminus B$  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.

# Arithmetic Operators + - \* / \ ^ '

---

If  $A$  is an  $m$ -by- $n$  matrix with  $m \approx n$  and  $B$  is a column vector with  $m$  components, or a matrix with several such columns, then  $X = A \backslash B$  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  $\text{pinv}(A) * B$ , which is the least squares solution with the smallest norm  $\|X\|$ .

- . \ Array left division.  $A \backslash B$  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  $[V, D] = \text{eig}(X)$ , 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
```

# Arithmetic Operators + - \* / \ ^ '

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)

# Arithmetic Operators + - \* / \ ^ '

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

# Arithmetic Operators + - \* / \ ^ /

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
$x \setminus y$	16/7	$x. \setminus y$	4 5/2 2
$2 \setminus x$	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

# Arithmetic Operators + - \* / \ ^ ' ,

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

## 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



**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:

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

---

```
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
```

# Logical Operators: Elementwise & | ~

---

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

## Logical Operators: Elementwise & | ~

---

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.

# Logical Operators: Elementwise & | ~

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 than 0.9:

```
rng(0, 'twister');
R=rand(5,7)
R =
    0.8147    0.0975    0.1576    0.1419    0.6557    0.7577    0.7061
```

# Logical Operators: Elementwise & | ~

---

```
    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.  $[6.9 \ 9.64 \ \text{sqrt}(-1)]$  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.

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

`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 nth 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.

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

---

- ' Matrix transpose.  $X'$  is the complex conjugate transpose of  $X$ .  $X \cdot '$  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.

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

- % 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...]	<code>horzcat(A,B,C...)</code>
Vertical concatenation	[A;B;C...]	<code>vertcat(A,B,C...)</code>
Subscript reference	A(i,j,k...)	<code>subsref(A,S)</code> . See help <code>subsref</code> .
Subscript assignment	A(i,j,k...) B	<code>subsasgn(A,S,B)</code> . See help <code>subsasgn</code> .

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

---

---

**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  $[j, j+1, \dots, k]$ , or empty when  $j > k$ .

$j:i:k$  is the same as  $[j, j+i, j+2i, \dots, j+m*i]$ , where  $m = \text{fix}((k-j)/i)$ , 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  $A(j), A(j+1), \dots, A(k)$ .

$A(:, j:k)$  is  $A(:, j), A(:, j+1), \dots, A(:, k)$ .

$A(:, :, k)$  is the  $k$ th page of three-dimensional array  $A$ .

## colon (:)

---

`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](#)

# abs

---

**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  $M \geq \text{MAX}(\text{subs})$ . 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

## accumarray

---

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)
```

# accumarray

---

```
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; result in radians

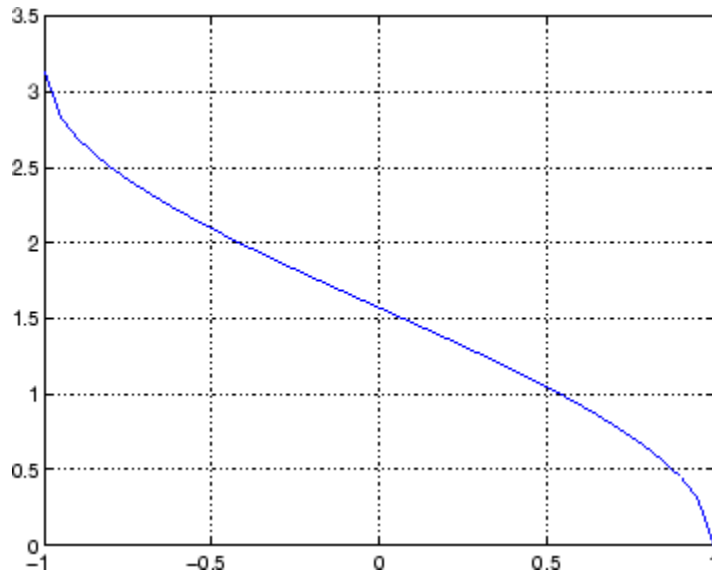
**Syntax**  $Y = \text{acos}(X)$

**Description**  $Y = \text{acos}(X)$  returns the inverse cosine (arccosine) for each element of  $X$ . For real elements of  $X$  in the domain  $[-1, 1]$ ,  $\text{acos}(X)$  is real and in the range  $[0, \pi]$ . For real elements of  $X$  outside the domain  $[-1, 1]$ ,  $\text{acos}(X)$  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
```



**Definitions**

The inverse cosine can be defined as

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

**See Also**

[acosd](#) | [acosh](#) | [cos](#)

# acosd

---

**Purpose** Inverse cosine; result in degrees

**Syntax**  $Y = \text{acosd}(X)$

**Description**  $Y = \text{acosd}(X)$  is the inverse cosine of the elements of  $X$ . The result,  $Y$ , is expressed 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`

# acosh

---

**Purpose** Inverse hyperbolic cosine

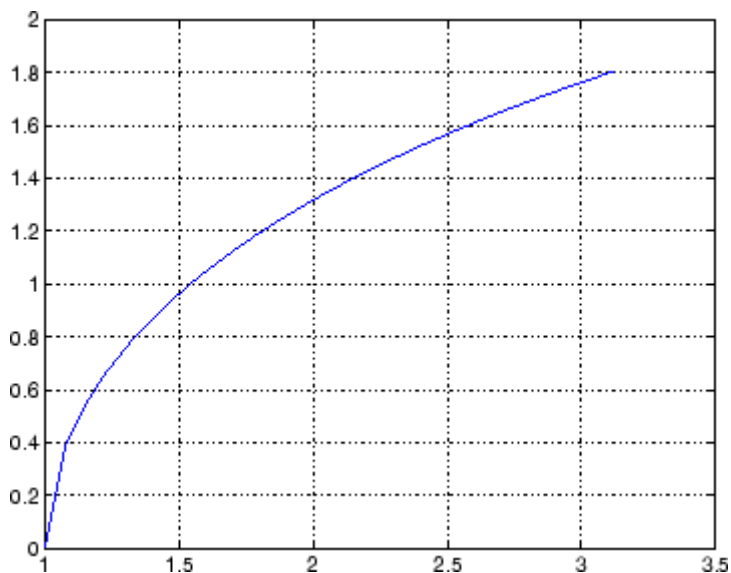
**Syntax**  $Y = \text{acosh}(X)$

**Description**  $Y = \text{acosh}(X)$  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`



**Purpose** Inverse cotangent; result in radians

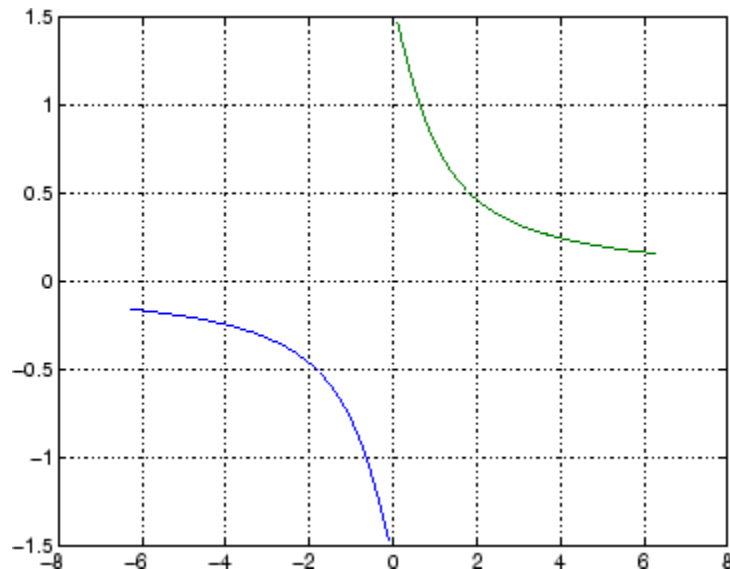
**Syntax**  $Y = \text{acot}(X)$

**Description**  $Y = \text{acot}(X)$  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
```



**Definitions** The inverse cotangent can be defined as

## acot

---

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

### See Also

cot | acotd | acoth

<b>Purpose</b>	Inverse cotangent; result in degrees
<b>Syntax</b>	$Y = \text{acot}(X)$
<b>Description</b>	$Y = \text{acot}(X)$ is the inverse cotangent of the elements of $X$ . The result, $Y$ , is expressed in degrees.
<b>Input Arguments</b>	<p><b>X - Cotangent of angle</b>  scalar value   vector   matrix   N-D array</p> <p>Cotangent of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>acotd</code> operation is element-wise when <math>X</math> is nonscalar.</p> <p><b>Data Types</b>  single   double</p> <p><b>Complex Number Support:</b> Yes</p>
<b>Output Arguments</b>	<p><b>Y - Angle in degrees</b>  scalar value   vector   matrix   N-D array</p> <p>Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as <math>X</math>.</p>
<b>Examples</b>	<p><b>Inverse Cotangent of Vector</b></p> <pre>x = [0 20 Inf]; y = acotd(x)  y =      90.0000    2.8624     0</pre> <p>The <code>acotd</code> operation is element-wise when you pass a vector, matrix, or N-D array.</p> <p><b>Inverse Cotangent of Complex Value</b></p> <pre>acotd(1+i)</pre>

# acotd

---

ans =

31.7175 -23.0535i

## See Also

cotd | cot | acot

**Purpose** Inverse hyperbolic cotangent

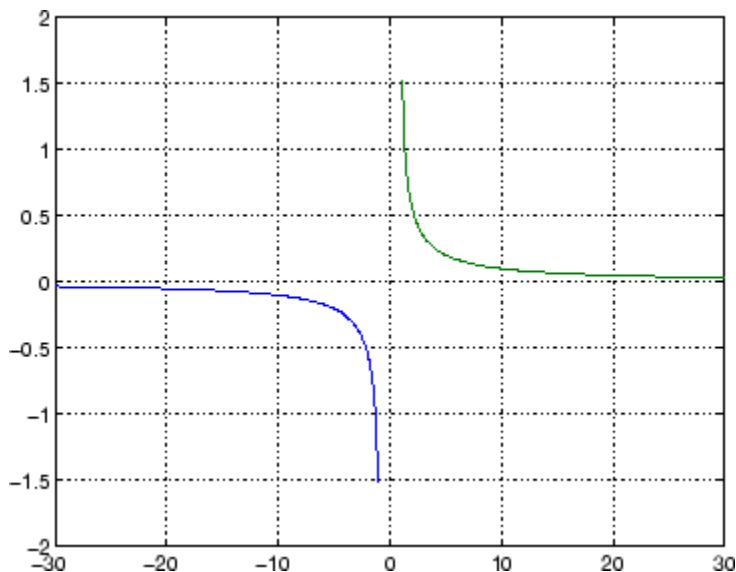
**Syntax**  $Y = \operatorname{acoth}(X)$

**Description**  $Y = \operatorname{acoth}(X)$  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  $-30 \leq x < -1$  and  $1 < x \leq 30$ .

```
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; result in radians

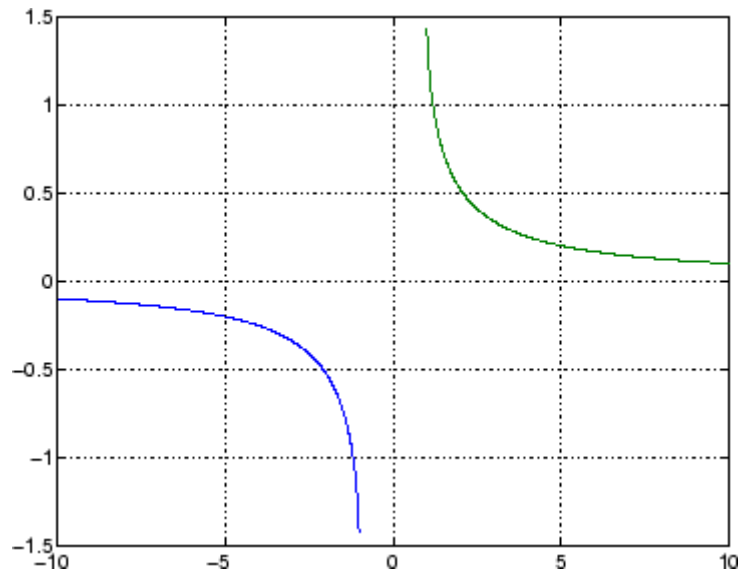
**Syntax**  $Y = \text{acsc}(X)$

**Description**  $Y = \text{acsc}(X)$  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  $-10 \leq x < -1$  and  $1 < x \leq 10$ .

```
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`

<b>Purpose</b>	Inverse cosecant; result in degrees
<b>Syntax</b>	$Y = \text{acscd}(X)$
<b>Description</b>	$Y = \text{acscd}(X)$ is the inverse cosecant of the elements of $X$ . The result, $Y$ , is expressed in degrees.
<b>Input Arguments</b>	<p><b>X - Cosecant of angle</b>  scalar value   vector   matrix   N-D array</p> <p>Cosecant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>acscd</code> operation is element-wise when <math>X</math> is nonscalar.</p> <p><b>Data Types</b>  single   double</p> <p><b>Complex Number Support:</b> Yes</p>
<b>Output Arguments</b>	<p><b>Y - Angle in degrees</b>  scalar value   vector   matrix   N-D array</p> <p>Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as <math>X</math>.</p>
<b>Examples</b>	<p><b>Inverse Cosecant of Vector</b></p> <pre>x = [20 10 Inf]; y = acscd(x)  y =      2.8660    5.7392     0</pre> <p>The <code>acscd</code> operation is element-wise when you pass a vector, matrix, or N-D array.</p> <p><b>Inverse Cosecant of Complex Value</b></p> <pre>acscd(1+i)</pre>

# acscd

---

ans =

25.9136 -30.4033i

## See Also

[cscd](#) | [csc](#) | [acsc](#)



**Purpose** Inverse hyperbolic cosecant

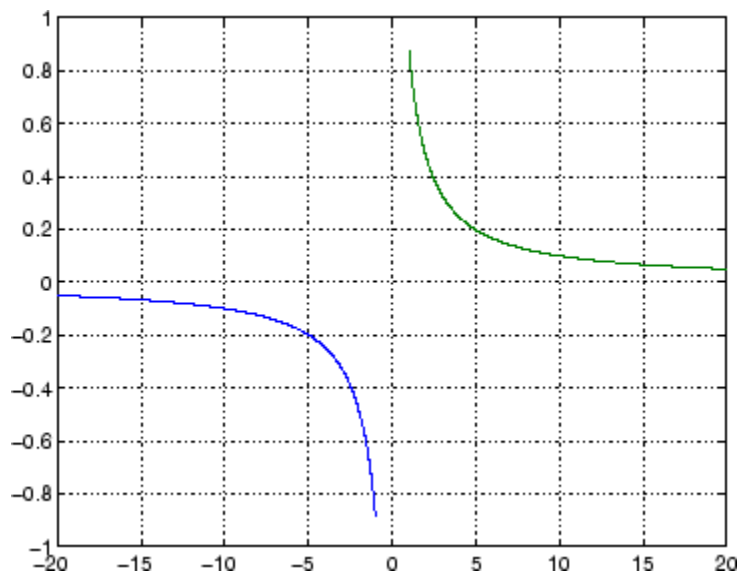
**Syntax**  $Y = \operatorname{acsch}(X)$

**Description**  $Y = \operatorname{acsch}(X)$  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  $-20 \leq x \leq -1$  and  $1 \leq x \leq 20$ .

```
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`

**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';...  
  'DbtClick' 'myDbtClickHandler';...  
  '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 2-100.

```
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

### Example 1 – Basic Control Methods

Start by creating a figure window to contain the control. Then create a control to run a Microsoft Calendar application in the window. Position the control at a [0 0] x-y offset from the bottom left of the figure window, and make it the same size (600 x 500 pixels) as the figure window.

```
f = figure('position', [300 300 600 500]);
cal = actxcontrol('mscal.calendar', [0 0 600 500], f);
```

Call the `get` method on `cal` to list all properties of the calendar, including today's date:

```
cal.get
```

For example, MATLAB displays (in part):

```
BackColor: 2.1475e+009
Day: 23
DayFont: [1x1 Interface.Standard_OLE_Types.Font]
Value: '8/20/2001'
.
.
.
```

Read today's date:

```
date = cal.Value
```

MATLAB displays a date similar to:

```
date =
```

```
8/20/2001
```

Set the Day property to a new value:

```
cal.Day = 5;  
date = cal.Value
```

MATLAB displays a date similar to:

```
date =  
    8/5/2001
```

Call `invoke` to list all available methods:

```
meth = cal.invoke
```

MATLAB displays (in part):

```
meth =  
    NextDay: 'HRESULT NextDay(handle)'  
    NextMonth: 'HRESULT NextMonth(handle)'  
    NextWeek: 'HRESULT NextWeek(handle)'  
    NextYear: 'HRESULT NextYear(handle)'  
          .  
          .  
          .
```

Invoke the `NextWeek` method to advance the current date by one week:

```
cal.NextWeek;  
date = cal.Value
```

MATLAB displays a date similar to:

```
date =  
    8/12/2001
```

Call `events` to list all calendar events that can be triggered:

cal.events

MATLAB displays:

```
Click = void Click()
DblClick = void DblClick()
KeyDown = void KeyDown(int16 KeyCode, int16 Shift)
KeyPress = void KeyPress(int16 KeyAscii)
KeyUp = void KeyUp(int16 KeyCode, int16 Shift)
BeforeUpdate = void BeforeUpdate(int16 Cancel)
AfterUpdate = void AfterUpdate()
NewMonth = void NewMonth()
NewYear = void NewYear()
```

## Example 2 – Event Handling

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.

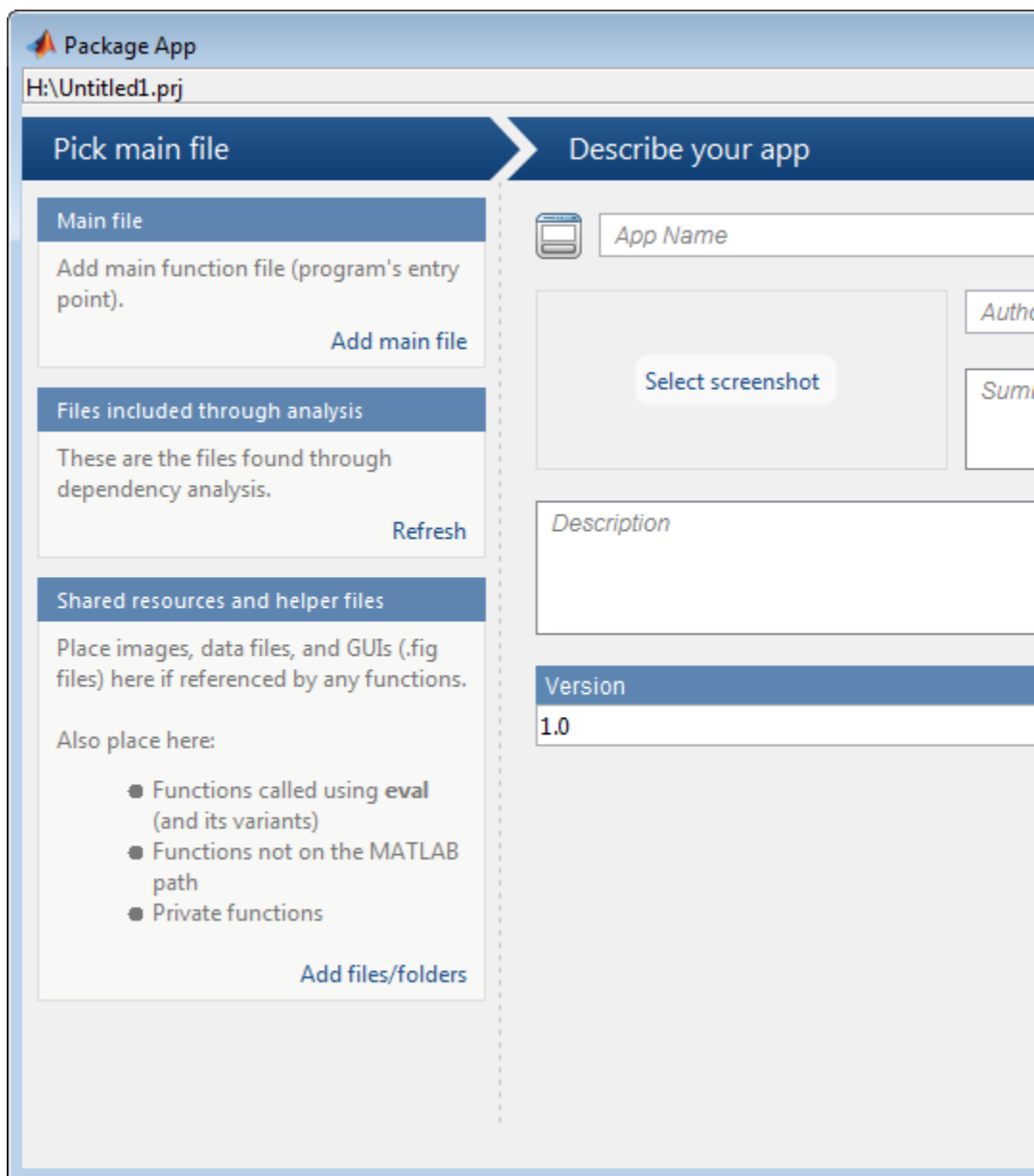
**Input Arguments** **prjfile - Full or partial path to the .prj file**  
string

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



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 opens populated with the data you previously specified for `myapp`.

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

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
```

# matlab.apputil.getInstalledAppInfo

---

```
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”

<b>Purpose</b>	Install app from a <code>.mlappinstall</code> file
<b>Syntax</b>	<code>appinfo = matlab.apputil.install(appfile)</code>
<b>Description</b>	<code>appinfo = matlab.apputil.install(appfile)</code> installs the specified app file and returns information about the app.
<b>Input Arguments</b>	<p><b>appfile - Full or partial path to <code>.mlappinstall</code> file</b> string</p> <p>Full or partial path of the app file you want to install, specified as a string.</p> <p><b>Example:</b> <code>'C:\myguis\myapps\myapp.mlappinstall'</code></p>
<b>Output Arguments</b>	<p><b>appinfo - Information about installed app</b> structure</p> <p>Information about the installed app, returned as a structure with the fields:</p> <p><b>status - Installation status</b> 'installed'   'updated'</p> <p>Installation status, returned as a string:</p> <ul style="list-style-type: none"><li>• 'installed' — New app is installed.</li><li>• 'updated' — Previously installed app is updated.</li></ul> <p><b>id - Unique identifier</b> string</p> <p>Unique identifier for the installed app, returned as a string</p> <p>The ID is for use when running or uninstalling the app programmatically.</p> <p><b>location - Folder where app is installed</b> string</p>

# matlab.apputil.install

---

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 uninstall 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”



<b>Purpose</b>	Package app files into .mlappinstall file
<b>Syntax</b>	<code>matlab.apputil.package(prjfile)</code>
<b>Description</b>	<code>matlab.apputil.package(prjfile)</code> creates a .mlappinstall file based on the information in the specified <code>prjfile</code> .
<b>Input Arguments</b>	<p><b>prjfile</b> - Full or partial path to app project (.prj) file</p> <p>string</p> <p>Full or partial path to app project (.prj) file, specified as a string.</p> <p><b>Example:</b> 'plotdata.prj'</p>
<b>Examples</b>	<p><b>Create mlappinstall File for Previously Created Project File</b></p> <p>Assume you previously created <code>myprjfile.prj</code> using <code>matlab.apputil.create</code>. The following command creates the corresponding .mlappinstall file.</p> <pre>matlab.apputil.package(`myprjfile.prj')</pre>
<b>Tips</b>	<ul style="list-style-type: none"><li>• To create a .prj file, use <code>matlab.apputil.create</code>.</li></ul>
<b>See Also</b>	<code>matlab.apputil.create</code>   <code>matlab.apputil.install</code>   <code>matlab.apputil.run</code>

# 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]}
```

## addCause (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)
```



<b>Purpose</b>	Add event to timeseries object
<b>Syntax</b>	<pre>ts = addevent(ts,e) ts = addevent(ts,Name,Time)</pre>
<b>Description</b>	<p><code>ts = addevent(ts,e)</code> adds one or more <code>tsdata.event</code> objects, <code>e</code>, to the <code>timeseries</code> object <code>ts</code>. <code>e</code> is either a single <code>tsdata.event</code> object or an array of <code>tsdata.event</code> objects.</p> <p><code>ts = addevent(ts,Name,Time)</code> constructs one or more <code>tsdata.event</code> objects and adds them to the <code>Events</code> property of <code>ts</code>. <code>Name</code> is a cell array of event name strings. <code>Time</code> is a cell array of event times.</p>
<b>Examples</b>	<p>Create a time-series object and add an event to this object.</p> <pre>%% 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';</pre> <p>View the properties (<code>EventData</code>, <code>Name</code>, <code>Time</code>, <code>Units</code>, and <code>StartDate</code>) of the event object.</p> <pre>get(e1)</pre> <p>MATLAB software responds with</p> <pre>EventData: []</pre>

# addevent

---

```
        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

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.

# audioinfo

---

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 <code>y</code>	Data Range of <code>y</code>
WAVE (.wav)	8	uint8	$0 \leq y \leq 255$
	16	int16	$-32768 \leq y \leq +32767$
	24	int32	$-2^{32} \leq y \leq 2^{32}-1$
	32	int32	$-2^{32} \leq y \leq 2^{32}-1$
	32	single	$-1.0 \leq y \leq +1.0$
	64	double	$-1.0 \leq y \leq +1.0$
FLAC (.flac)	8	uint8	$0 \leq y \leq 255$
	16	int16	$-32768 \leq y \leq +32767$
	24	int32	$-2^{32} \leq y \leq 2^{32}-1$
MP3 (.mp3), MPEG-4 AAC (.m4a, .mp4), and OGG (.ogg)	N/A	single	$-1.0 \leq y \leq +1.0$

---

**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	$0 \leq y \leq 255$
int16	$-32768 \leq y \leq +32767$
int32	$-2^{32} \leq y \leq 2^{32}-1$
single	$-1.0 \leq y \leq +1.0$
double	$-1.0 \leq y \leq +1.0$

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, 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*.

**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`.

# audiowrite

---

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

## addframe (avifile)

---

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

## addlistener (handle)

---

- 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](#)

# inputParser.addOptional

---

## 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.

# inputParser.addParamValue

---

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

**Alternatives** As an alternative to the addpath function, use the Set Path dialog box.

**Syntax**

```
addpath('folderName1','folderName2','folderName3' ...)  
addpath('folderName1','folderName2','folderName3' ... flag)  
addpath folderName1 folderName2 folderName3 ... -flag  
previous_path = addpath(...)
```

**Description**

addpath('folderName1','folderName2','folderName3' ...) adds the specified folders to the top of the search path. Use the full path name for each folder. Use genpath with addpath to add all subfolders of folderName. Use addpath statements in a startup.m file to modify the search path programmatically at startup.

addpath('folderName1','folderName2','folderName3' ... *flag*) adds the specified folders to either the top or bottom of the search path, or disables folder change detection on Windows, depending on the value of position.

Value of Flag	Result
'-begin'	Add specified folders to the top of the search path.
'-end'	Add specified folders to the bottom of the search path.
'-frozen'	Disables change detection for folders you add to the path, which conserves Windows change notification resources (Windows only). Type <code>help changenotification</code> in the Command Window for more information.

addpath folderName1 folderName2 folderName3 ... -*flag* is the command syntax.



`previous_path = addpath(...)` configures the path using the inputs and returns the previous path setting.

## 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.

## Examples

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

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

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

```
addpath c:/matlab/myfiles -end
```

Add `myfiles` and its subfolders to the search path:

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

On Windows, Add `myfiles` to the top of the search path, disable folder change notification, and display the search path before adding `myfiles`:

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

## See Also

`genpath` | `path` | `pathsep` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath`

## How To

- “Path Names in MATLAB”
- “Using the MATLAB Search Path”
- “Making Files and Folders Accessible to MATLAB”
- “Specifying Startup Options in the MATLAB Startup File”

# addpref

---

**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](#)

## addprop (dynamicprops)

---

**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`

**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`:

# addproperty

---

```
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”

## 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);
```

# inputParser.addRequired

---

## 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”



<b>Purpose</b>	Add sample to <code>tscollection</code> object
<b>Syntax</b>	<pre>tsc = addsampletocollection(tsc, 'time', Time, TS1Name, TS1Data,     TSnName, TSnData)</pre>
<b>Description</b>	<p><code>tsc = addsampletocollection(tsc, 'time', Time, TS1Name, TS1Data, TSnName, TSnData)</code> adds data samples <code>TSnData</code> to the collection member <code>TSnName</code> in the <code>tscollection</code> object <code>tsc</code> at one or more <code>Time</code> values. Here, <code>TSnName</code> is the string that represents the name of a time series in <code>tsc</code>, and <code>TSnData</code> is an array containing data samples.</p>
<b>Tips</b>	<p>If you do not specify data samples for a time-series member in <code>tsc</code>, that time-series member will contain missing data at the times given by <code>Time</code> (for numerical time-series data), NaN values, or (for logical time-series data) <code>false</code> values.</p> <p>When a time-series member requires <code>Quality</code> values, you can specify data quality codes together with the data samples by using the following syntax:</p> <pre>tsc = addsampletocollection(tsc, 'time', time, TS1Name, ...     ts1cellarray, TS2Name, ts2cellarray, ...)</pre> <p>Specify data in the first cell array element and <code>Quality</code> in the second cell array element.</p> <hr/> <p><b>Note</b> If a time-series member already has <code>Quality</code> values but you only provide data samples, 0s are added to the existing <code>Quality</code> array at the times given by <code>Time</code>.</p> <hr/>
<b>Examples</b>	<p>The following example shows how to create a <code>tscollection</code> that consists of two <code>timeseries</code> objects, where one <code>timeseries</code> does not have quality codes and the other does. The final step of the example adds a sample to the <code>tscollection</code>.</p>

# addsampletocollection

---

- 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 must be a double scalar whole number, and can be either positive or negative. 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 =
```

# addtodate

---

```
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](#)

<b>Purpose</b>	Add timeseries object to tscollection object
<b>Syntax</b>	<pre>tsc = addts(tsc,ts) tsc = addts(tsc,ts) tsc = addts(tsc,ts,Name) tsc = addts(tsc,Data,Name)</pre>
<b>Description</b>	<p><code>tsc = addts(tsc,ts)</code> adds the timeseries object <code>ts</code> to tscollection object <code>tsc</code>.</p> <p><code>tsc = addts(tsc,ts)</code> adds a cell array of timeseries objects <code>ts</code> to the tscollection <code>tsc</code>.</p> <p><code>tsc = addts(tsc,ts,Name)</code> adds a cell array of timeseries objects <code>ts</code> to tscollection <code>tsc</code>. <code>Name</code> is a cell array of strings that gives the names of the timeseries objects in <code>ts</code>.</p> <p><code>tsc = addts(tsc,Data,Name)</code> creates a new timeseries object from <code>Data</code> with the name <code>Name</code> and adds it to the tscollection object <code>tsc</code>. <code>Data</code> is a numerical array and <code>Name</code> is a string.</p>
<b>Tips</b>	<p>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.</p> <p>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 <code>StartDate</code> property. For more information about properties, see the timeseries reference page.</p>
<b>Examples</b>	<p>The following example shows how to add a time series to a time-series collection:</p> <ol style="list-style-type: none"><li>1 Create two timeseries objects, <code>ts1</code> and <code>ts2</code>.</li></ol> <pre>ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...                 'name','acceleration');</pre>

# addts

---

```
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

---

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

- `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 non-scalar and empty inputs.
- 

**Syntax**

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

**Definitions** 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

$$Ai(Z) = \left[ \frac{1}{\pi} \sqrt{\frac{Z}{3}} \right] K_{1/3}(\zeta)$$

$$Bi(Z) = \sqrt{\frac{Z}{3}} [I_{-1/3}(\zeta) + I_{1/3}(\zeta)],$$

where

$$\zeta = \frac{2}{3} Z^{3/2}.$$

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

# airy

$W = \text{airy}(k, Z)$  returns different results depending on the value of  $k$ .

<b>k</b>	<b>Returns</b>
0	The same result as $\text{airy}(Z)$
1	The derivative, $Ai'(Z)$
2	The Airy function of the second kind, $Bi(Z)$
3	The derivative, $Bi'(Z)$

$[W, \text{ierr}] = \text{airy}(k, Z)$  also returns completion flags in an array the same size as  $W$ .

<b>ierr</b>	<b>Description</b>
0	$\text{airy}$ successfully computed the Airy function for this element.
1	Illegal arguments.
2	Overflow. Returns Inf.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, $Z$ too large.
5	No convergence. Returns NaN.

## References

[1] Amos, D. E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[2] Amos, D. E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

## See Also

besseli | besselj | bessellk | bessely



**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`.

<b>HorizontalAlignment</b>	<b>Definition</b>
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 <i>x</i> -values of the group
Right	Shifts the objects' right edges to that of the first object selected
Distribute	Equalizes <i>x</i> -distances between all objects within the span of the extreme <i>x</i> -values
Fixed	Spaces objects to have a specified number of points between them in the <i>y</i> -direction

<b>VerticalAlignment</b>	<b>Definition</b>
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 <i>y</i> -values of the group
Bottom	Shifts the objects' bottom edges to that of the first object selected
Distribute	Equalizes <i>y</i> -distances between all objects within the span of the extreme <i>y</i> -values
Fixed	Spaces objects to have a specified number of points between them in the <i>x</i> -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.

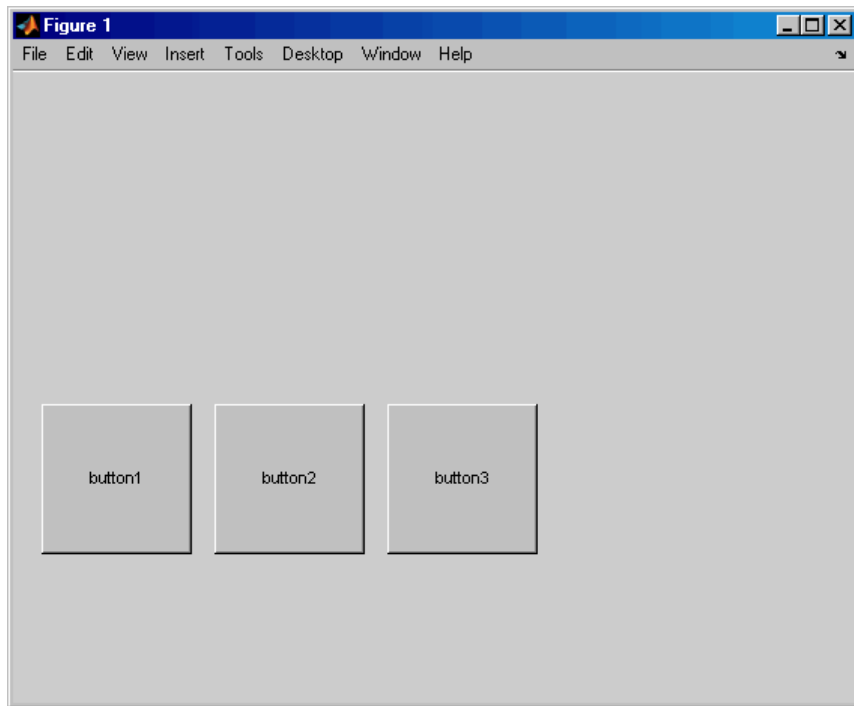
## Examples

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');
```

# align

---



**Alternatives** See “Alignment Tool — Aligning and Distributing Objects” for the GUI alternative.

**See Also** `uicontrol` | `uistack`

**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.

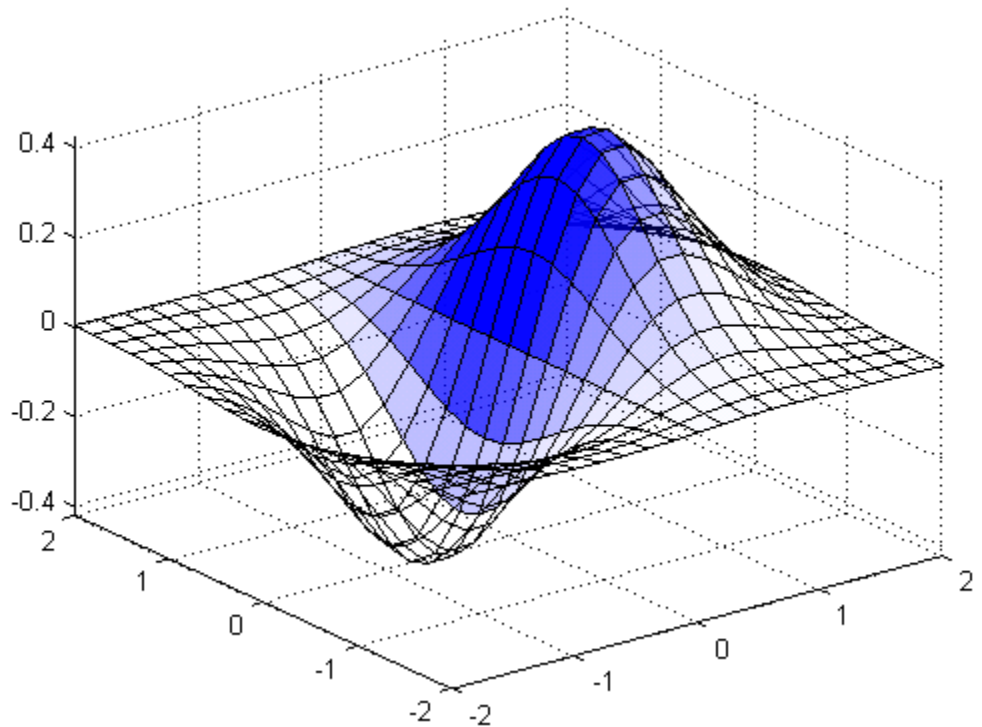
`alim(axes_handle,...)` operates on the specified axes.

## 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”

**Purpose**

Determine whether all array elements are nonzero or true

**Syntax**

$B = \text{all}(A)$   
 $B = \text{all}(A, \text{dim})$

**Description**

$B = \text{all}(A)$  tests whether *all* the elements along various dimensions of an array are nonzero or logical 1 (true).

If  $A$  is empty,  $\text{all}(A)$  returns logical 1 (true).

If  $A$  is a vector,  $\text{all}(A)$  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 matrix,  $\text{all}(A)$  treats the columns of  $A$  as vectors, returning a row vector of logical 1's and 0's.

If  $A$  is a multidimensional array,  $\text{all}(A)$  treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

$B = \text{all}(A, \text{dim})$  tests along the dimension of  $A$  specified by scalar  $\text{dim}$ .

1	1	1
1	1	0

$A$

1	1	0
---	---	---

$\text{all}(A,1)$

1
0

$\text{all}(A,2)$

**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

# all

---

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.

Applying the `all` function twice to a matrix, as in `all(all(A))`, always reduces it to a scalar condition.

```
all(all(eye(3)))
ans =
    0
```

## See Also

[any](#) | [colon \(:\)](#) | [max](#) | [mean](#) | [median](#) | [min](#) | [prod](#) | [std](#) | [sum](#)  
| [trapz](#) | [Logical Operators: Elementwise](#) | [Logical Operators:  
Short-circuit](#) | [Relational Operators](#)



**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`

# alpha

---

**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

# alpha

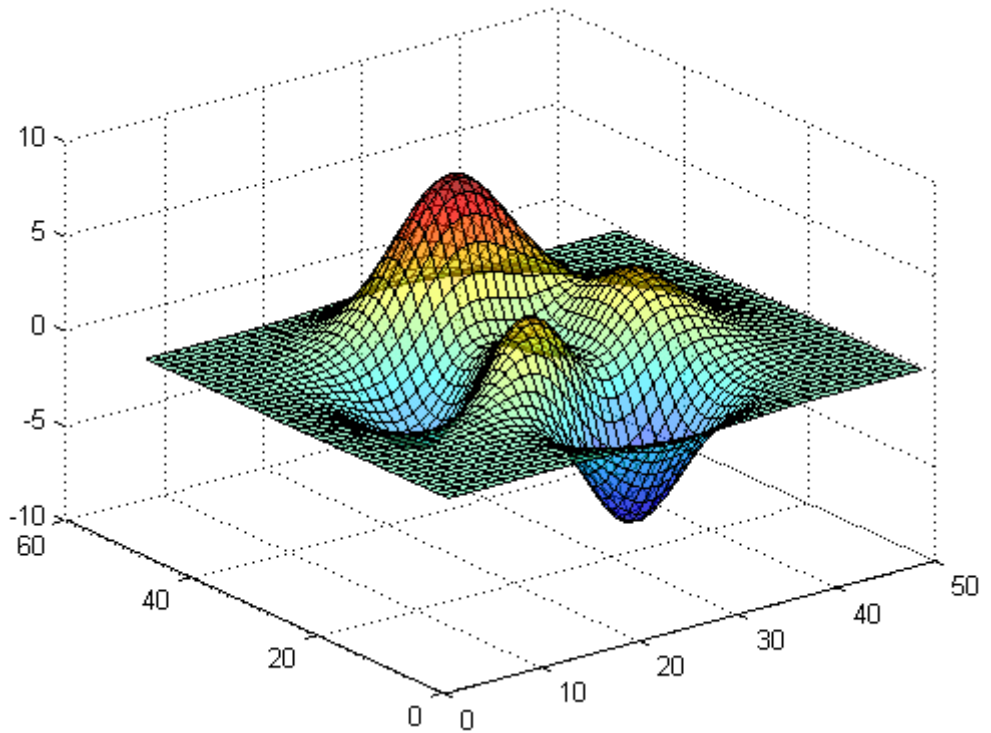
---

- '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”

# alphamap

---

**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.

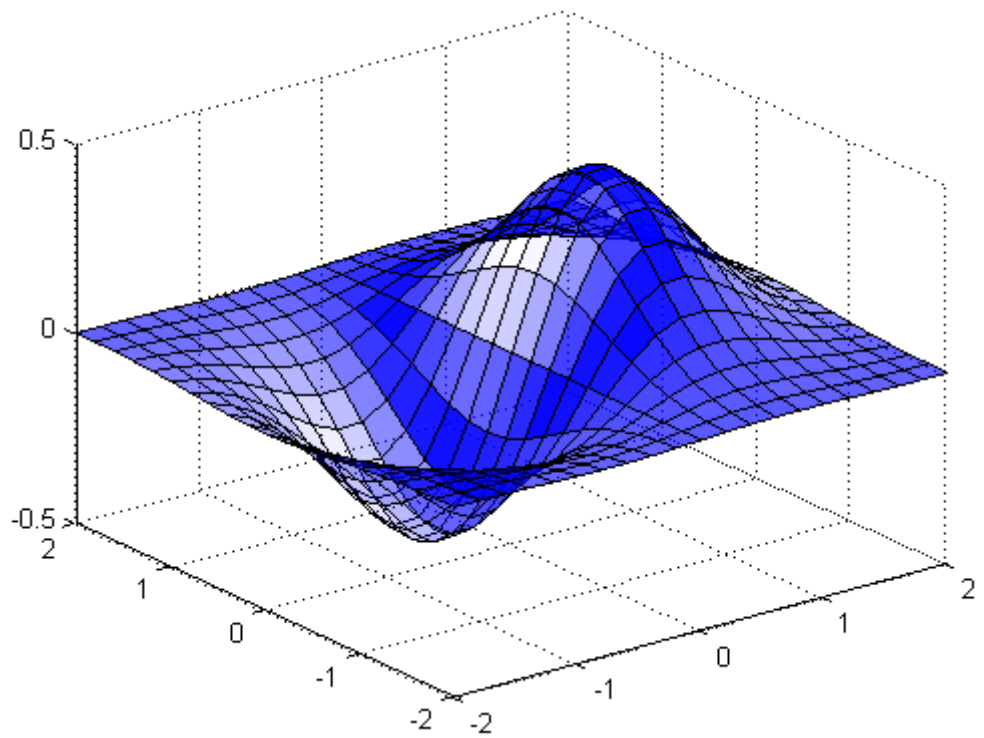
`alpha_map = alphamap(figure_handle)` returns the current alphamap from the figure identified by `figure_handle`.

`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  $\max(16, (\text{dense} * \sqrt{n}))$  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 is 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,  $C = \text{spones}(S) + \text{spones}(S')$ .

## 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 /

**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','hgroup','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 hgroup object.
hgg = hgroup;
hgl = line(randn(5),randn(5),'Parent',hgg);
% Now get the ancestor of the lines:
p = ancestor(hgg,{'figure','axes','hgroup'});
get(p,'Type')
% Now get the top-level ancestor:
ptop=ancestor(hgg,{'figure','axes','hgroup'],'toplevel');
get(ptop,'type')
```

## ancestor

---

### 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

# and

---

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](#)

<b>Purpose</b>	Phase angle
<b>Syntax</b>	$P = \text{angle}(Z)$
<b>Description</b>	<p><math>P = \text{angle}(Z)</math> returns the phase angles, in radians, for each element of complex array <math>Z</math>. The angles lie between <math>\pm\pi</math>.</p> <p>For complex <math>Z</math>, the magnitude <math>R</math> and phase angle <math>\theta</math> are given by</p> $R = \text{abs}(Z)$ $\theta = \text{angle}(Z)$ <p>and the statement</p> $Z = R \cdot \exp(i \cdot \theta)$ <p>converts back to the original complex <math>Z</math>.</p>
<b>Examples</b>	$Z = \begin{bmatrix} 1 - 1i & 2 + 1i & 3 - 1i & 4 + 1i \\ 1 + 2i & 2 - 2i & 3 + 2i & 4 - 2i \\ 1 - 3i & 2 + 3i & 3 - 3i & 4 + 3i \\ 1 + 4i & 2 - 4i & 3 + 4i & 4 - 4i \end{bmatrix}$ $P = \text{angle}(Z)$ $P = \begin{bmatrix} -0.7854 & 0.4636 & -0.3218 & 0.2450 \\ 1.1071 & -0.7854 & 0.5880 & -0.4636 \\ -1.2490 & 0.9828 & -0.7854 & 0.6435 \\ 1.3258 & -1.1071 & 0.9273 & -0.7854 \end{bmatrix}$
<b>Algorithms</b>	The angle function can be expressed as $\text{angle}(z) = \text{imag}(\log(z)) = \text{atan2}(\text{imag}(z), \text{real}(z))$ .
<b>See Also</b>	<code>abs</code>   <code>atan2</code>   <code>unwrap</code>

# annotation

---

**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”.

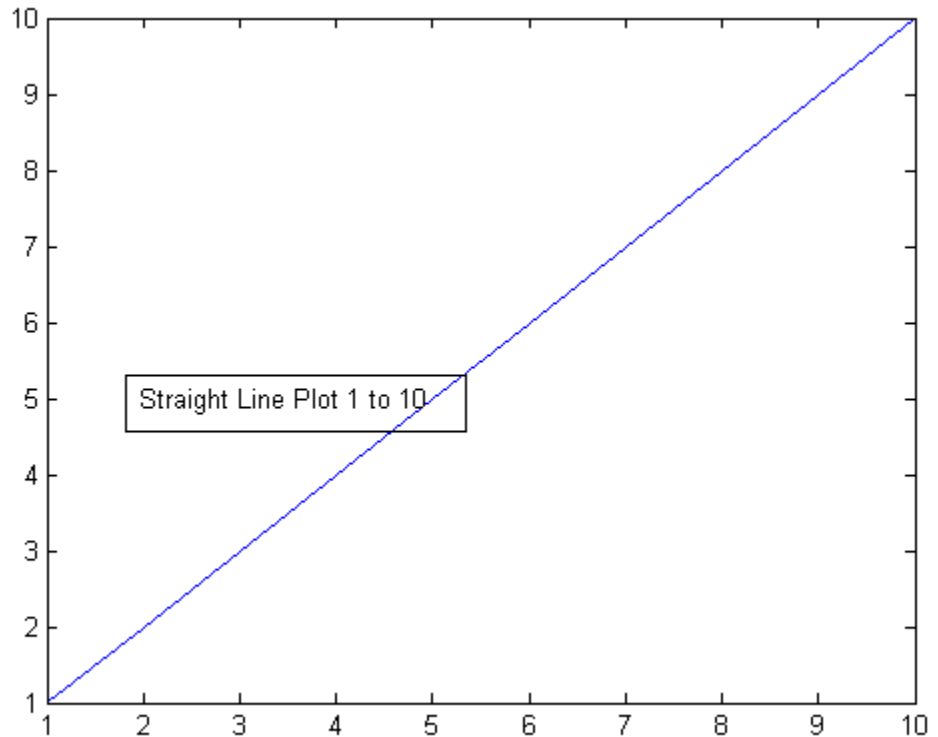
## Examples

Annotate a graph with a text box.

```
plot(1:10);
annotation('textbox', [.2 .4 .1 .1], 'String', 'Straight Line Plot 1 t
```

## annotation

---



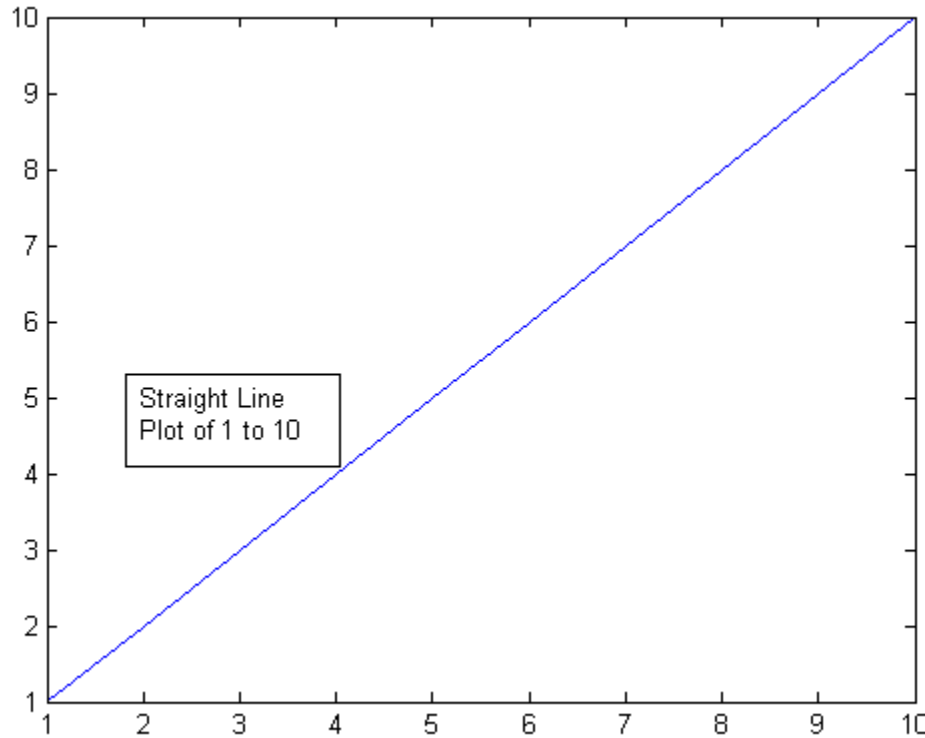
Notice that the locations specified are normalized in the figure coordinates. Length of the string automatically adjusts the width of the text box, as, `FitBoxToText` property of the annotation object is on by default.

---

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);
```



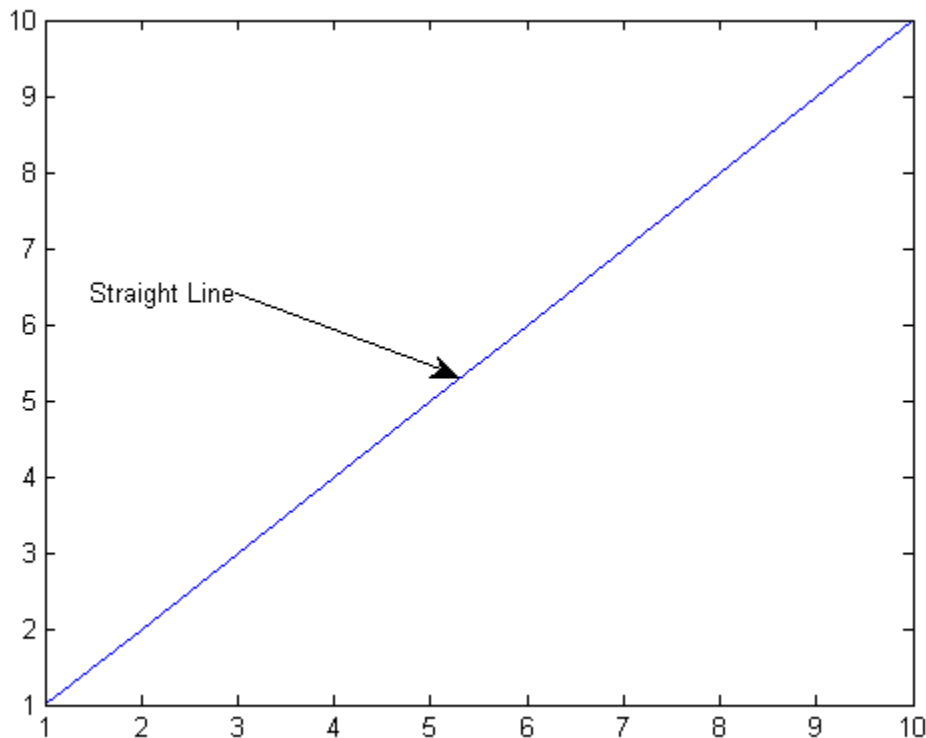
---

Insert a text arrow in a graph.

```
plot(1:10);  
a = annotation('textarrow', [.3 .5], [.6 .5], 'String', 'Straight Line');
```

# annotation

---



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”

# Annotation Arrow Properties

---

**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 =  $\frac{1}{72}$  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.

# Annotation Arrow Properties

**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 =  $\frac{1}{72}$  inch. The default value is 10. See also HeadLength.

# Annotation Arrow Properties

---

## 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 =  $\frac{1}{72}$  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  $[X_{\text{begin}} \ X_{\text{end}}]$

*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  $[Y_{\text{begin}} \ Y_{\text{end}}]$

*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 Doublearrow Properties

---

## 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

# Annotation Doublearrow Properties

*Length of second arrowhead.* Specify in points. 1 point =  $\frac{1}{72}$  inch. The default value is 10.












The first arrowhead is located at the end defined by the point  $x(\text{end})$ ,  $y(\text{end})$ .

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	

# Annotation Doublearrow Properties

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 =  $\frac{1}{72}$  inch.  
See also Head1Length.

Head2Width  
size in points

*Width of second arrowhead.* Specify in points. 1 point =  $\frac{1}{72}$  inch.  
See also Head2Length.

LineStyle  
{-} | -- | : | -. | none

*Style of arrow stem.*

# Annotation Doublearrow Properties

## 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.

# Annotation Doublearrow Properties

---

- **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 [ $X_{\text{begin}}$   $X_{\text{end}}$ ]

*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 [ $Y_{\text{begin}}$   $Y_{\text{end}}$ ]

*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 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

# Annotation Ellipse Properties

---

*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 =  $\frac{1}{72}$  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 =  $\frac{1}{72}$  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”

# Annotation Line Properties

---

**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

# Annotation Line Properties

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

# Annotation Line Properties

---

of the letter x, and the height of one characters unit is the distance between the baselines of two lines of text.

X

vector  $[X_{\text{begin}} X_{\text{end}}]$

*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  $[Y_{\text{begin}} Y_{\text{end}}]$

*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 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.*

# Annotation Rectangle Properties

---

- **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  $dy$ ,

# Annotation Rectangle Properties

---

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 =  $\frac{1}{72}$  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`

# Annotation Textarrow Properties

---

## 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')`



# Annotation Textarrow Properties

---

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 =  $\frac{1}{72}$  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 =  $\frac{1}{72}$  inch.

## FontWeight

`{normal} | bold | light | demi`

# Annotation Textarrow Properties

---

*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 =  $\frac{1}{72}$  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	

# Annotation Textarrow Properties

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.

# Annotation Textarrow Properties

---

- 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

# Annotation Textarrow Properties

*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
<code>\alpha</code>	$\alpha$	<code>\upsilon</code>	$\upsilon$	<code>\sim</code>	$\sim$
<code>\angle</code>	$\angle$	<code>\phi</code>	$\Phi$	<code>\leq</code>	$\leq$
<code>\ast</code>	$*$	<code>\chi</code>	$\chi$	<code>\infty</code>	$\infty$
<code>\beta</code>	$\beta$	<code>\psi</code>	$\psi$	<code>\clubsuit</code>	$\clubsuit$
<code>\gamma</code>	$\gamma$	<code>\omega</code>	$\omega$	<code>\diamondsuit</code>	$\diamondsuit$
<code>\delta</code>	$\delta$	<code>\Gamma</code>	$\Gamma$	<code>\heartsuit</code>	$\heartsuit$
<code>\epsilon</code>	$\epsilon$	<code>\Delta</code>	$\Delta$	<code>\spadesuit</code>	$\spadesuit$

# Annotation Textarrow Properties

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\zeta</code>	$\zeta$	<code>\Theta</code>	$\Theta$	<code>\leftrightharrow</code>	$\leftrightarrow$
<code>\eta</code>	$\eta$	<code>\Lambda</code>	$\Lambda$	<code>\leftarrow</code>	$\leftarrow$
<code>\theta</code>	$\theta$	<code>\Xi</code>	$\Xi$	<code>\Leftarrow</code>	$\Leftarrow$
<code>\vartheta</code>	$\vartheta$	<code>\Pi</code>	$\Pi$	<code>\uparrow</code>	$\uparrow$
<code>\iota</code>	$\iota$	<code>\Sigma</code>	$\Sigma$	<code>\rightarrow</code>	$\rightarrow$
<code>\kappa</code>	$\kappa$	<code>\Upsilon</code>	$\Upsilon$	<code>\Rightarrow</code>	$\Rightarrow$
<code>\lambda</code>	$\lambda$	<code>\Phi</code>	$\Phi$	<code>\downarrow</code>	$\downarrow$
<code>\mu</code>	$\mu$	<code>\Psi</code>	$\Psi$	<code>\circ</code>	$\circ$
<code>\nu</code>	$\nu$	<code>\Omega</code>	$\Omega$	<code>\pm</code>	$\pm$
<code>\xi</code>	$\xi$	<code>\forall</code>	$\forall$	<code>\geq</code>	$\geq$
<code>\pi</code>	$\pi$	<code>\exists</code>	$\exists$	<code>\propto</code>	$\propto$
<code>\rho</code>	$\rho$	<code>\ni</code>	$\ni$	<code>\partial</code>	$\partial$
<code>\sigma</code>	$\sigma$	<code>\cong</code>	$\cong$	<code>\bullet</code>	$\bullet$
<code>\varsigma</code>	$\varsigma$	<code>\approx</code>	$\approx$	<code>\div</code>	$\div$
<code>\tau</code>	$\tau$	<code>\Re</code>	$\Re$	<code>\neq</code>	$\neq$
<code>\equiv</code>	$\equiv$	<code>\oplus</code>	$\oplus$	<code>\aleph</code>	$\aleph$
<code>\Im</code>	$\Im$	<code>\cup</code>	$\cup$	<code>\wp</code>	$\wp$
<code>\otimes</code>	$\otimes$	<code>\subseteq</code>	$\subseteq$	<code>\oslash</code>	$\oslash$
<code>\cap</code>	$\cap$	<code>\in</code>	$\in$	<code>\supseteq</code>	$\supseteq$
<code>\supset</code>	$\supset$	<code>\lceil</code>	$\lceil$	<code>\subset</code>	$\subset$
<code>\int</code>	$\int$	<code>\cdot</code>	$\cdot$	<code>\o</code>	$\o$

# Annotation Textarrow Properties

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\rfloor</code>	⌋	<code>\neg</code>	¬	<code>\nabla</code>	∇
<code>\lfloor</code>	⌊	<code>\times</code>	×	<code>\ldots</code>	...
<code>\perp</code>	⊥	<code>\surd</code>	√	<code>\prime</code>	′
<code>\wedge</code>	^	<code>\varpi</code>	ϖ	<code>\0</code>	∅
<code>\rceil</code>	⌈	<code>\rangle</code>	⟩	<code>\mid</code>	
<code>\vee</code>	∨			<code>\copyright</code>	©
<code>\langle</code>	⟨				

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).

# Annotation Textarrow Properties

---

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 =  $\frac{1}{72}$  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 [ $X_{\text{begin}}$   $X_{\text{end}}$ ]

*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 [ $Y_{\text{begin}}$   $Y_{\text{end}}$ ]

## Annotation Textarrow Properties

---

*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

## 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.

# Annotation Textbox Properties

---

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

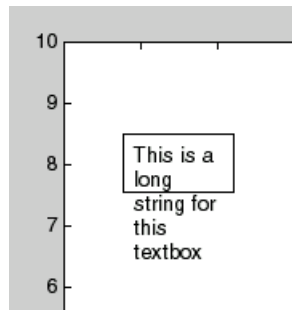
*Automatically adjust text box width and height to fit text.* MATLAB automatically wraps text strings to fit the width of the text box. However, if the text string is long enough, it can extend beyond the bottom of the text box.

# Annotation Textbox Properties

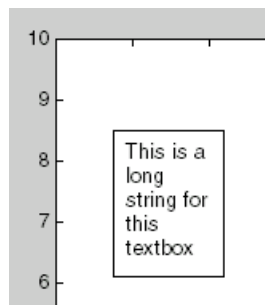
---

**Note** The property is obsolete. To control line wrapping behavior in textboxes, use `FitBoxToText` instead.

---



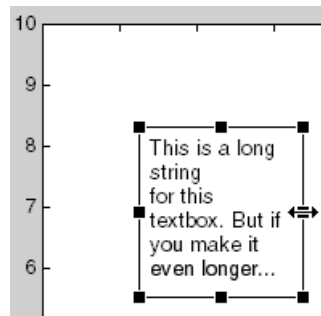
When you set this mode to on, MATLAB automatically adjusts the height of the text box to accommodate the string, doing so as you create or edit the string.



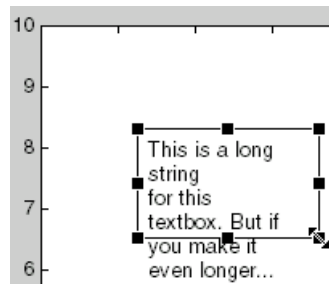
The fit-size-to-text behavior turns off if you resize the text box programmatically or manually in plot edit mode.

# Annotation Textbox Properties

---



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 =  $\frac{1}{72}$  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 =  $\frac{1}{72}$  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:

# Annotation Textbox Properties

---

- `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.

# Annotation Textbox Properties

TeX Character Sequence Table

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\alpha</code>	$\alpha$	<code>\upsilon</code>	$\upsilon$	<code>\sim</code>	$\sim$
<code>\angle</code>	$\angle$	<code>\phi</code>	$\Phi$	<code>\leq</code>	$\leq$
<code>\ast</code>	$*$	<code>\chi</code>	$\chi$	<code>\infty</code>	$\infty$
<code>\beta</code>	$\beta$	<code>\psi</code>	$\Psi$	<code>\clubsuit</code>	$\clubsuit$
<code>\gamma</code>	$\gamma$	<code>\omega</code>	$\omega$	<code>\diamondsuit</code>	$\diamondsuit$
<code>\delta</code>	$\delta$	<code>\Gamma</code>	$\Gamma$	<code>\heartsuit</code>	$\heartsuit$
<code>\epsilon</code>	$\epsilon$	<code>\Delta</code>	$\Delta$	<code>\spadesuit</code>	$\spadesuit$
<code>\zeta</code>	$\zeta$	<code>\Theta</code>	$\Theta$	<code>\leftrightarrow</code>	$\leftrightarrow$
<code>\eta</code>	$\eta$	<code>\Lambda</code>	$\Lambda$	<code>\leftarrow</code>	$\leftarrow$
<code>\theta</code>	$\theta$	<code>\Xi</code>	$\Xi$	<code>\Leftarrow</code>	$\Leftarrow$
<code>\vartheta</code>		<code>\Pi</code>	$\Pi$	<code>\uparrow</code>	$\uparrow$
<code>\iota</code>	$\iota$	<code>\Sigma</code>	$\Sigma$	<code>\rightarrow</code>	$\rightarrow$
<code>\kappa</code>	$\kappa$	<code>\Upsilon</code>		<code>\Rightarrow</code>	$\Rightarrow$
<code>\lambda</code>	$\lambda$	<code>\Phi</code>	$\Phi$	<code>\downarrow</code>	$\downarrow$
<code>\mu</code>	$\mu$	<code>\Psi</code>	$\Psi$	<code>\circ</code>	$\circ$
<code>\nu</code>	$\nu$	<code>\Omega</code>	$\Omega$	<code>\pm</code>	$\pm$
<code>\xi</code>	$\xi$	<code>\forall</code>	$\forall$	<code>\geq</code>	$\geq$
<code>\pi</code>	$\pi$	<code>\exists</code>	$\exists$	<code>\propto</code>	$\propto$
<code>\rho</code>	$\rho$	<code>\ni</code>	$\ni$	<code>\partial</code>	$\partial$

## Annotation Textbox Properties

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
<code>\sigma</code>	$\sigma$	<code>\cong</code>	$\cong$	<code>\bullet</code>	$\bullet$
<code>\varsigma</code>	$\varsigma$	<code>\approx</code>	$\approx$	<code>\div</code>	$\div$
<code>\tau</code>	$\tau$	<code>\Re</code>	$\Re$	<code>\neq</code>	$\neq$
<code>\equiv</code>	$\equiv$	<code>\oplus</code>	$\oplus$	<code>\aleph</code>	
<code>\Im</code>	$\Im$	<code>\cup</code>	$\cup$	<code>\wp</code>	$\wp$
<code>\otimes</code>	$\otimes$	<code>\subseteq</code>	$\subseteq$	<code>\oslash</code>	$\oslash$
<code>\cap</code>	$\cap$	<code>\in</code>		<code>\supseteq</code>	$\supseteq$
<code>\supset</code>	$\supset$	<code>\lceil</code>	$\lceil$	<code>\subset</code>	$\subset$
<code>\int</code>	$\int$	<code>\cdot</code>	$\cdot$	<code>\o</code>	$\o$
<code>\rfloor</code>	$\rfloor$	<code>\neg</code>	$\neg$	<code>\nabla</code>	$\nabla$
<code>\lfloor</code>	$\lfloor$	<code>\times</code>	$\times$	<code>\ldots</code>	$\dots$
<code>\perp</code>	$\perp$	<code>\surd</code>	$\surd$	<code>\prime</code>	$\prime$
<code>\wedge</code>	$\wedge$	<code>\varpi</code>	$\varpi$	<code>\0</code>	$\emptyset$
<code>\rceil</code>	$\rceil$	<code>\rangle</code>	$\rangle$	<code>\mid</code>	$ $
<code>\vee</code>	$\vee$			<code>\copyright</code>	$\copyright$
<code>\langle</code>	$\langle$				

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.

# Annotation Textbox Properties

---

## 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 =  $\frac{1}{72}$  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

---

<b>Purpose</b>	Most recent answer
<b>Syntax</b>	ans
<b>Description</b>	The MATLAB software creates the ans variable automatically when you specify no output argument.
<b>Examples</b>	<p>The statement</p> $2+2$ <p>is the same as</p> $\text{ans} = 2+2$
<b>See Also</b>	display

# any

---

**Purpose** Determine whether any array elements are nonzero

**Syntax**  
 $B = \text{any}(A)$   
 $B = \text{any}(A, dim)$

**Description**  $B = \text{any}(A)$  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 empty, any(A) returns logical 0 (false).

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 matrix, any(A) treats the columns of A as vectors, returning a row vector of logical 1's and 0's.

If A is a multidimensional array, any(A) treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

$B = \text{any}(A, dim)$  tests along the dimension of A specified by scalar *dim*.

1	0	1
0	0	0

A

1	0	1
---	---	---

any(A,1)

1
0

any(A,2)

## 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

Applying the any function twice to a matrix, as in any(any(A)), always reduces it to a scalar condition.

```
any(any(eye(3)))
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
```

# any

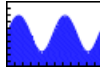
---

## 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 `areaserie`s 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.

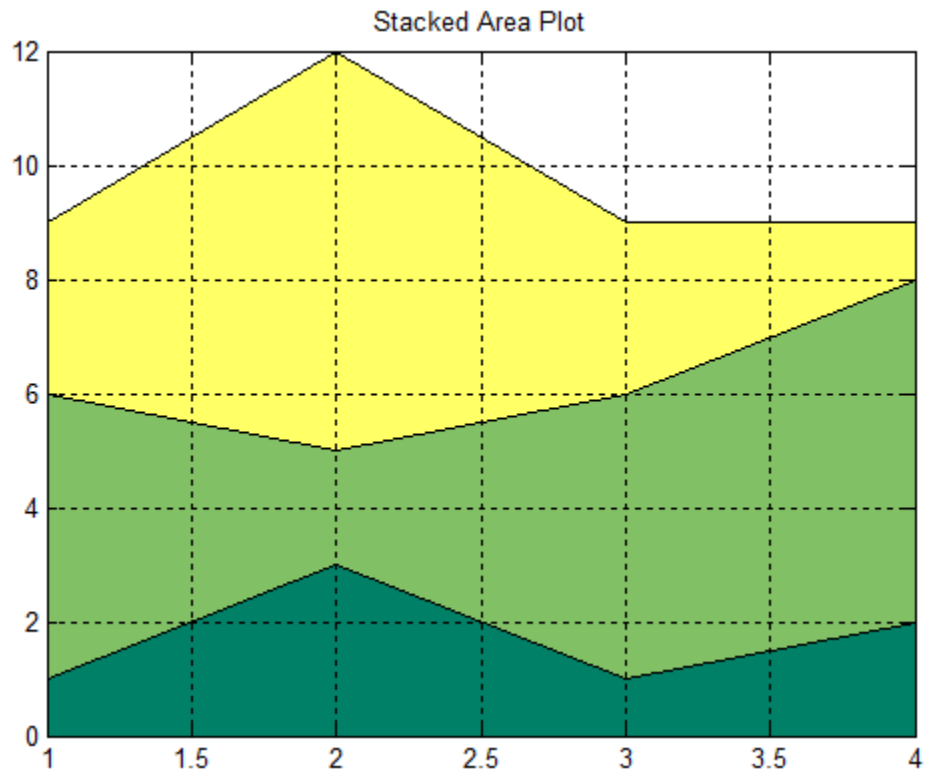
Some areaseries object properties that you set on an individual areaseries object set the values for all areaseries objects in the graph. See the property descriptions for information on specific properties.

## 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'
```



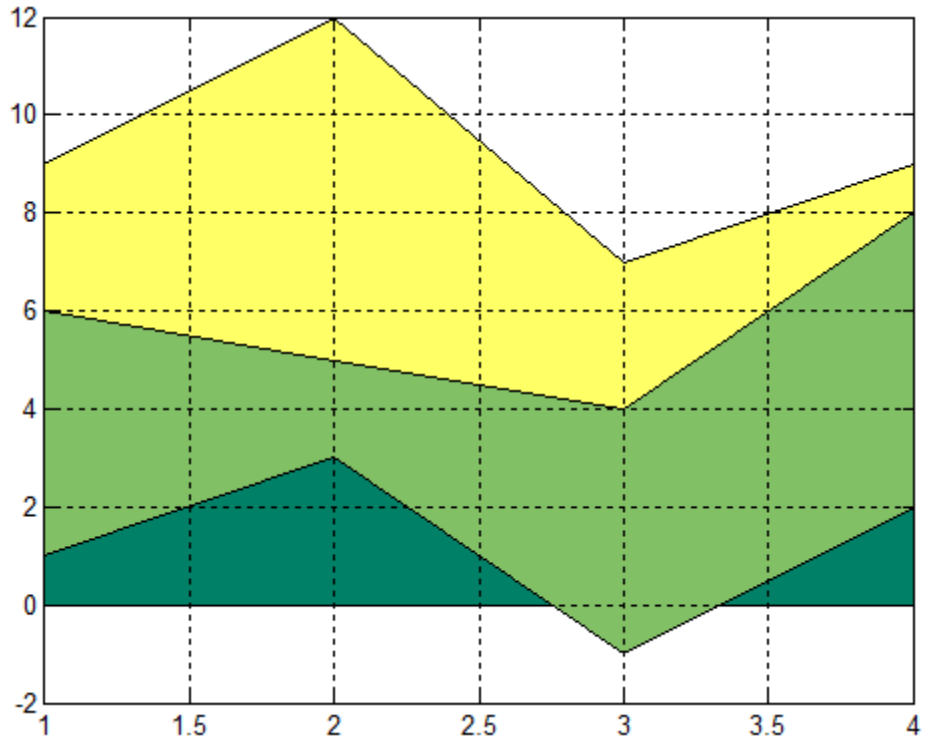
### Adjusting the Base Value

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
```

## area

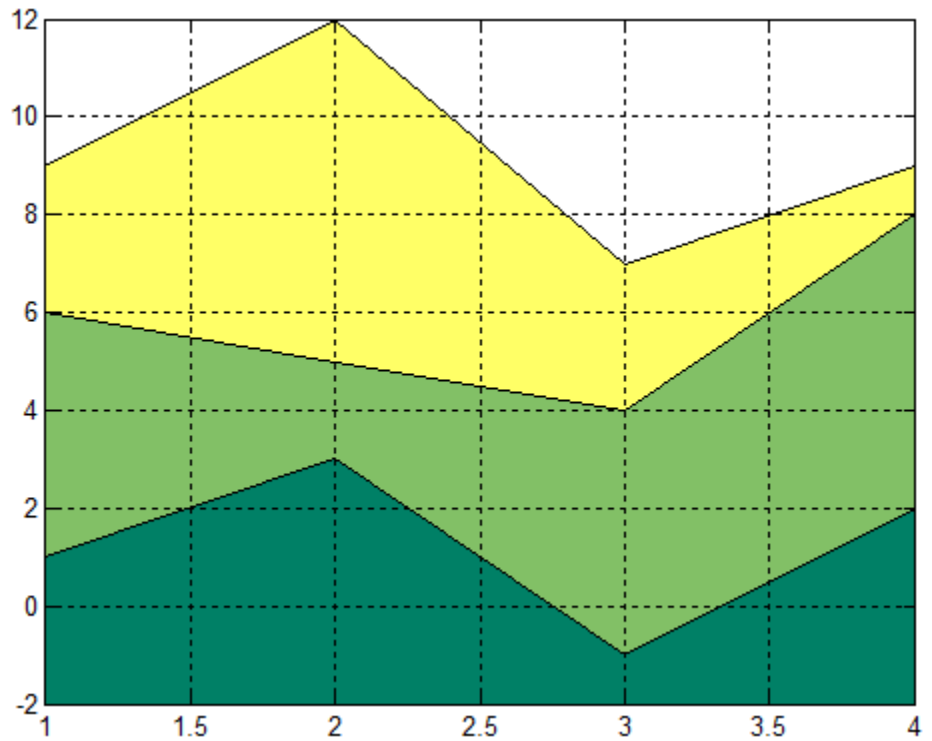
The area graph now looks like this:



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.



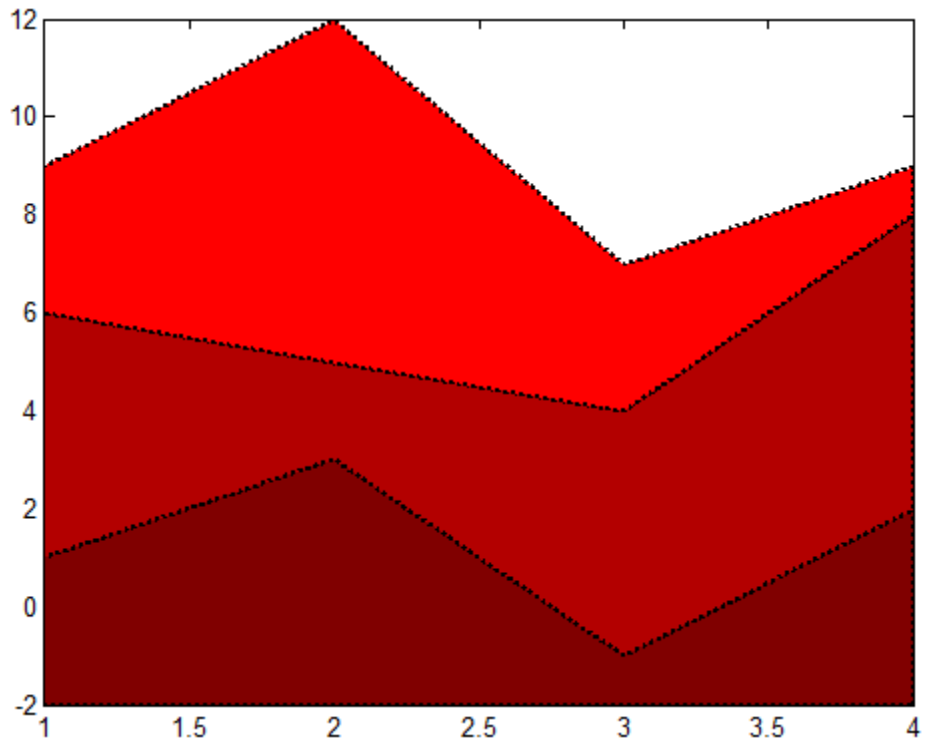
### Specifying Colors and Line Styles

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
```

# area

---



## See Also

[bar](#) | [plot](#) | [sort](#) | [“Pie Charts, Bar Plots, and Histograms”](#) | [Areaseries Properties](#)

## How To

- [“Area Graphs”](#)

## 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.

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
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

# Areaseries Properties

---

## 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.

## Areaseries Properties

---

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

## Areaseries Properties

---

*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.

# Areaseries Properties

---

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`

## Areaseries Properties

---

*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 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 edges of filled areas.*

# Areaseries Properties

---

**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 `'hgroup'`.

# Areaseries Properties

---

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`.

# Areaseries Properties

---

---

**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”

# arrayfun

---

## 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, 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`.

### **UniformOutput**

Logical value, as follows:

- |                        |                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>true</code> (1)  | Indicates that for all inputs, each output from function <code>func</code> is a cell array or a scalar value that is always of the same type. The <code>arrayfun</code> function combines the outputs in arrays <code>B1, ..., Bm</code> , where <code>m</code> is the number of function outputs. Each output array is of the same type as the individual function outputs. |
| <code>false</code> (0) | Requests that the <code>arrayfun</code> function combine the outputs into cell arrays <code>B1, ..., Bm</code> . The outputs of function <code>func</code> 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
```

# arrayfun

---

## See Also

`structfun` | `cellfun` | `spfun` | `function_handle` | `cell2mat`

## Tutorials

- “Anonymous Functions”

<b>Purpose</b>	Set FTP transfer type to ASCII
<b>Syntax</b>	<code>ascii(ftpobj)</code>
<b>Description</b>	<code>ascii(ftpobj)</code> 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.
<b>Input Arguments</b>	<b>ftpobj</b> FTP object created by <code>ftp</code> .
<b>Examples</b>	Connect to the MathWorks FTP server, and switch from binary (default) to ASCII mode:  <pre>mw=ftp('ftp.mathworks.com'); ascii(mw)</pre>
<b>See Also</b>	<code>binary</code>   <code>ftp</code>

**Purpose** Inverse secant; result in radians

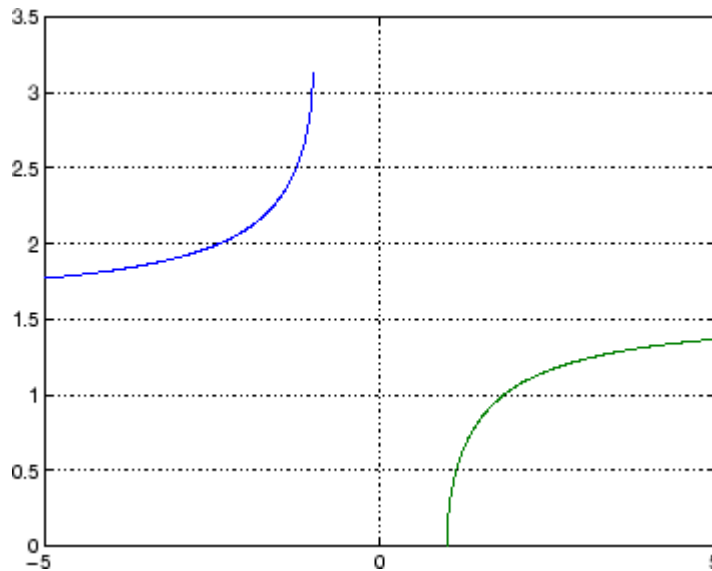
**Syntax**  $Y = \text{asec}(X)$

**Description**  $Y = \text{asec}(X)$  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  $1 \leq x \leq 5$  and  $-5 \leq x \leq -1$ .

```
x1 = -5:0.01:-1;  
x2 = 1:0.01:5;  
plot(x1,asec(x1),x2,asec(x2)), grid on
```



**See Also** `asecd` | `asech` | `sec`

<b>Purpose</b>	Inverse secant; result in degrees
<b>Syntax</b>	$Y = \text{asecd}(X)$
<b>Description</b>	$Y = \text{asecd}(X)$ is the inverse secant of the elements of $X$ . The result, $Y$ , is expressed in degrees.
<b>Input Arguments</b>	<p><b>X - Secant of angle</b>  scalar value   vector   matrix   N-D array</p> <p>Secant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>asecd</code> operation is element-wise when <math>X</math> is nonscalar.</p> <p><b>Data Types</b>  single   double</p> <p><b>Complex Number Support:</b> Yes</p>
<b>Output Arguments</b>	<p><b>Y - Angle in degrees</b>  scalar value   vector   matrix   N-D array</p> <p>Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as <math>X</math>.</p>
<b>Examples</b>	<p><b>Inverse Secant of Vector</b></p> <pre>x = [10 1 Inf]; y = asecd(x)  y =      84.2608         0    90.0000</pre> <p>The <code>asecd</code> operation is element-wise when you pass a vector, matrix, or N-D array.</p> <p><b>Inverse Secant of Complex Value</b></p> <pre>asecd(1+i)</pre>

# asecd

---

ans =

64.0864 +30.4033i

## See Also

secd | asec | sec



**Purpose** Inverse hyperbolic secant

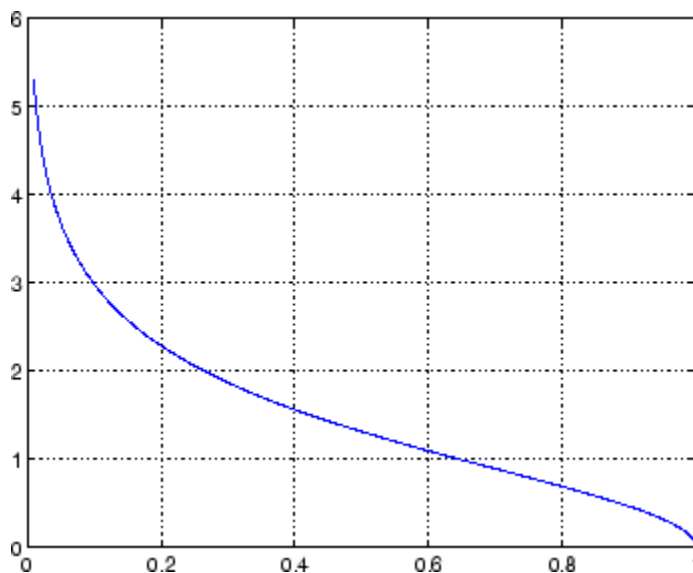
**Syntax**  $Y = \operatorname{asech}(X)$

**Description**  $Y = \operatorname{asech}(X)$  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  $0.01 \leq x \leq 1$ .

```
x = 0.01:0.001:1;  
plot(x,asech(x)), grid on
```



**See Also** `asec` | `sech` | `asinh` | `acosh`

# asin

---

**Purpose** Inverse sine; result in radians

**Syntax** `Y = asin(X)`

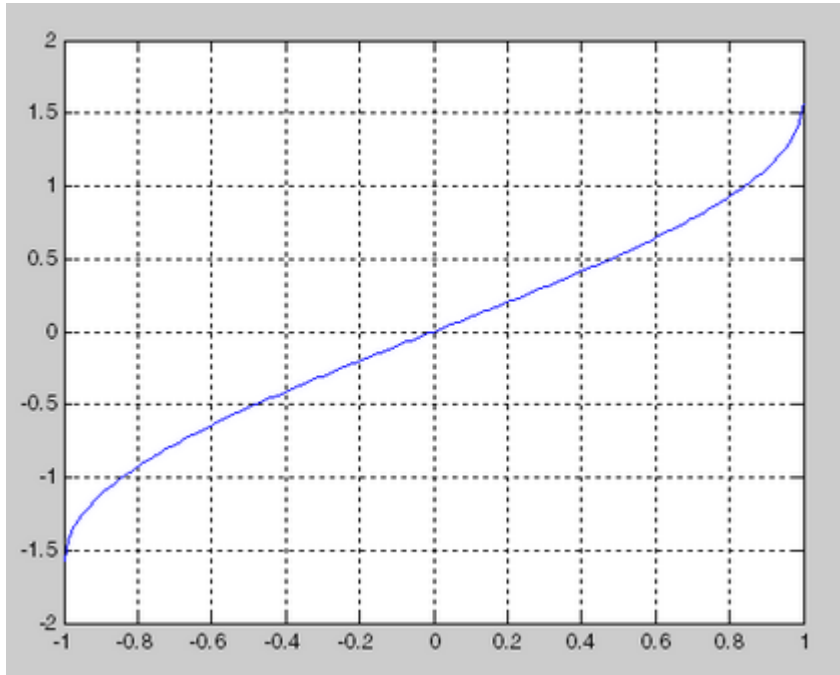
**Description** `Y = asin(X)` returns the inverse sine (arcsine) for each element of `X`. The `asin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians. For real elements of `X` in the domain `[-1, 1]`, `asin(X)` is in the range

$$\left[-\frac{\pi}{2}, \frac{\pi}{2}\right].$$

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

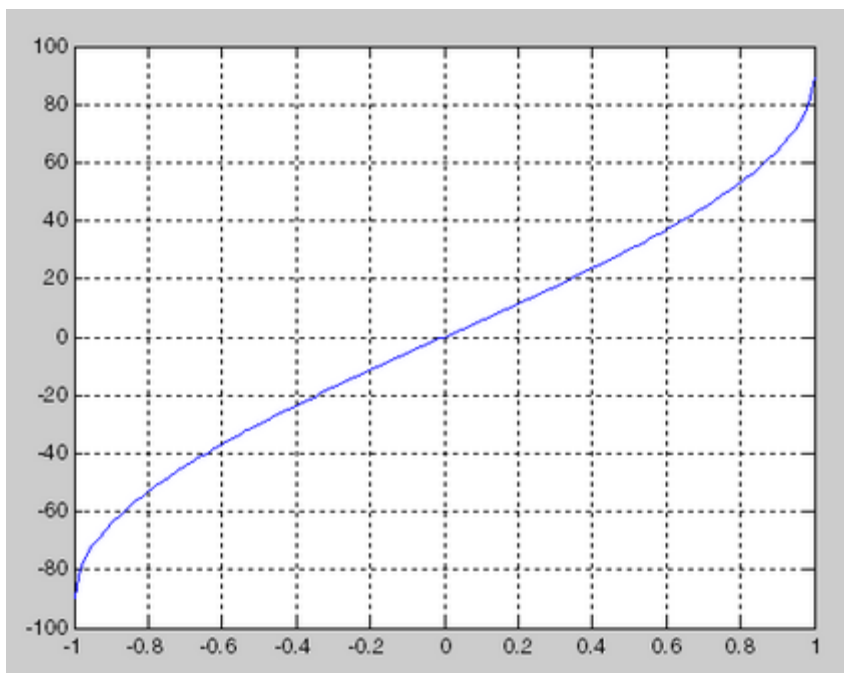
---

<b>Purpose</b>	Inverse sine; result in degrees
<b>Syntax</b>	$Y = \text{asind}(X)$
<b>Description</b>	$Y = \text{asind}(X)$ is the inverse sine of the elements of $X$ . The result, $Y$ , is expressed in degrees.
<b>Input Arguments</b>	<b>X - Sine of angle</b> 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 <code>asind</code> operation is element-wise when $X$ is nonscalar.  <b>Data Types</b> single   double <b>Complex Number Support:</b> Yes
<b>Output Arguments</b>	<b>Y - Angle in degrees</b> 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 $X$ .
<b>Examples</b>	<b>Inverse sine of one</b> Show that the inverse sine of 1 is exactly 90°.  <pre>asind(1)  ans =      90</pre> <b>Round Trip Calculation for Complex Angles</b> Show that inverse sine, followed by sine, returns the original values of $X$ .  <pre>sind(asind([2 3]))</pre>

```
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](#)

# asinh

---

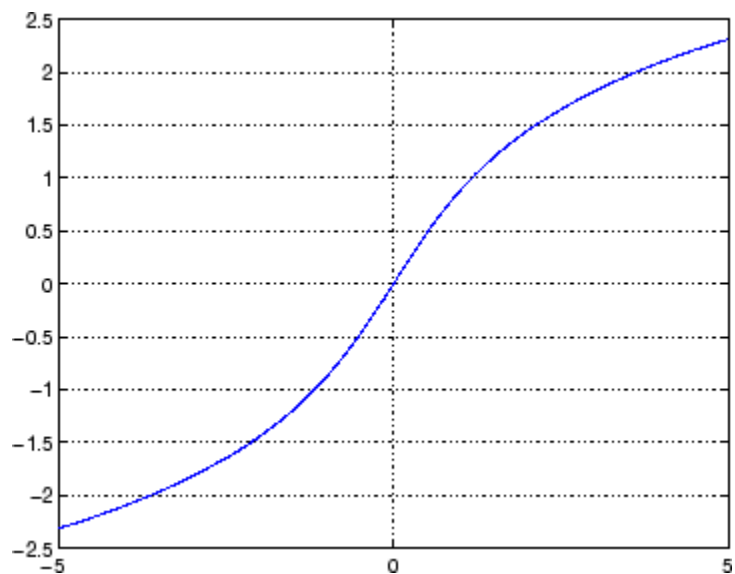
**Purpose** Inverse hyperbolic sine

**Syntax**  $Y = \operatorname{asinh}(X)$

**Description**  $Y = \operatorname{asinh}(X)$  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`

**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)
```

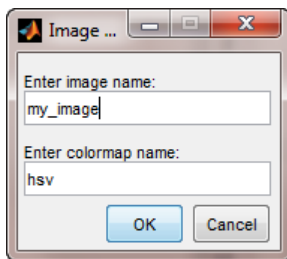


<b>Purpose</b>	Assign value to variable in specified workspace
<b>Syntax</b>	<code>assignin(ws, 'var', val)</code>
<b>Description</b>	<p><code>assignin(ws, 'var', val)</code> assigns the value <code>val</code> to the variable <code>var</code> in the workspace <code>ws</code>. The <code>var</code> input must be the array name only; it cannot contain array indices. If <code>var</code> does not exist in the specified workspace, <code>assignin</code> creates it. <code>ws</code> can have a value of 'base' or 'caller' to denote the MATLAB base workspace or the workspace of the caller function.</p> <p>The <code>assignin</code> function is particularly useful for these tasks:</p> <ul style="list-style-type: none"><li>• Exporting data from a function to the MATLAB workspace</li><li>• 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)</li></ul>
<b>Tips</b>	<p>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”.</p>
<b>Examples</b>	<p><b>Example 1</b></p> <p>This example creates a dialog box for the image display function, prompting a user for an image name and a colormap name. The <code>assignin</code> function is used to export the user-entered values to the MATLAB workspace variables <code>imfile</code> and <code>cmap</code>.</p> <pre>prompt = {'Enter image name:', 'Enter colormap name:'}; title = 'Image display - assignin example'; lines = 1; def = {'my_image', 'hsv'}; answer = inputdlg(prompt, title, lines, def);</pre>

# assignin

---

```
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; result in radians

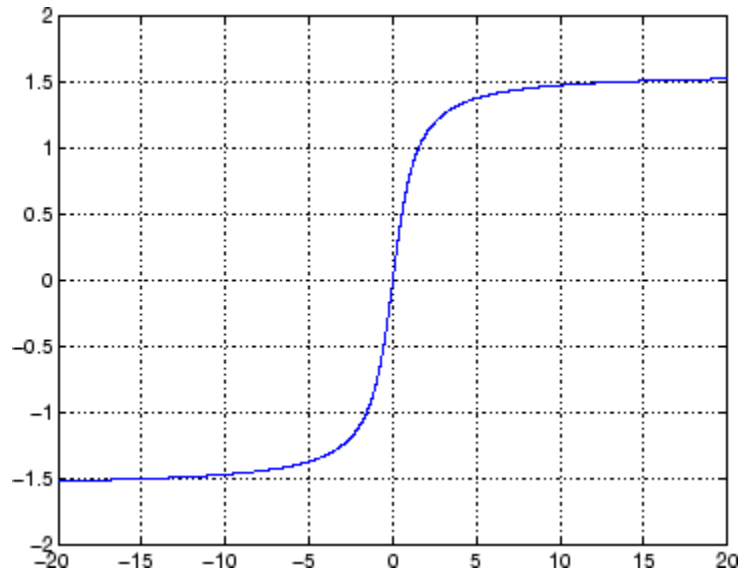
**Syntax**  $Y = \text{atan}(X)$

**Description**  $Y = \text{atan}(X)$  returns the inverse tangent (arctangent) for each element of  $X$ . For real elements of  $X$ ,  $\text{atan}(X)$  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  $-20 \leq x \leq 20$ .

```
x = -20:0.01:20;  
plot(x,atan(x)), grid on
```



**See Also** `atan2` | `tan` | `atand` | `atanh`

# atan2

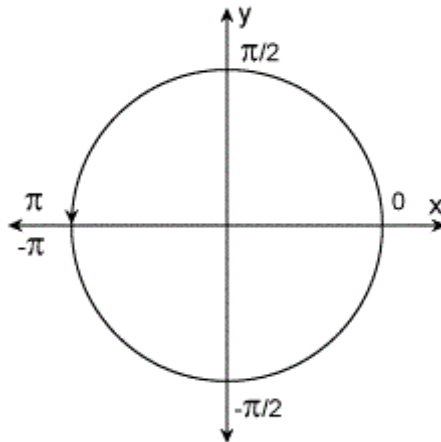
---

**Purpose** Four-quadrant inverse tangent

**Syntax**  $P = \text{atan2}(Y,X)$

**Description**  $P = \text{atan2}(Y,X)$  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  $\pi$ .  $\text{atan}$  uses  $\text{sign}(Y)$  and  $\text{sign}(X)$  to determine the specific quadrant.



$\text{atan2}(Y,X)$  contrasts with  $\text{atan}(Y/X)$ , 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`

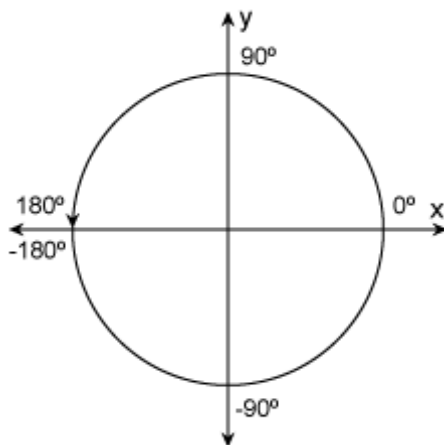
# atan2d

---

**Purpose** Four-quadrant inverse tangent; result in degrees

**Syntax**  $D = \text{atan2d}(Y,X)$

**Description**  $D = \text{atan2d}(Y,X)$  computes 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 `atan2d(Y/X)` for the inverse tangent with results on the interval  $[-90, 90]$ .

**See Also**

atan2 | atan | atan2d | tan | tand

# atand

---

**Purpose** Inverse tangent; result in degrees

**Syntax**  $Y = \text{atand}(X)$

**Description**  $Y = \text{atand}(X)$  is the inverse tangent of the elements of  $X$ . The result,  $Y$ , is expressed 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

# atanh

---

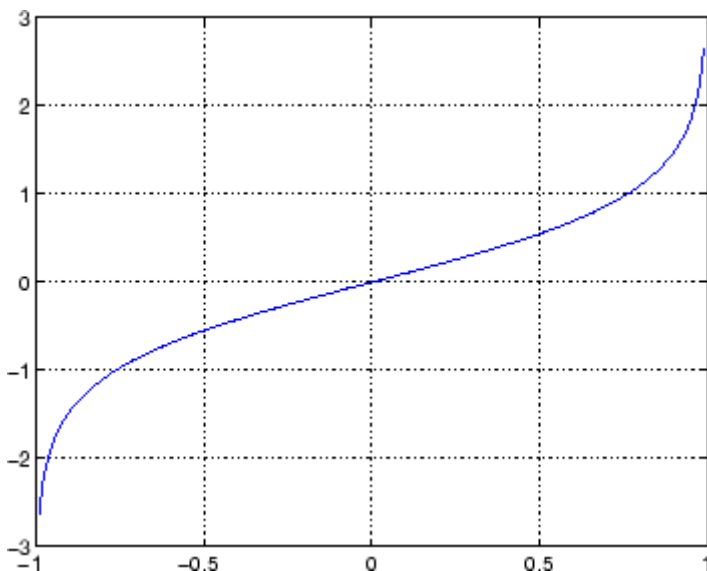
**Purpose** Inverse hyperbolic tangent

**Syntax**  $Y = \operatorname{atanh}(X)$

**Description** The `atanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.  $Y = \operatorname{atanh}(X)$  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.

# audiodevinfo

---

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

**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
<code>int8</code>	-128 to 127
<code>uint8</code>	0 to 255
<code>int16</code>	-32768 to 32767

# audioplayer

---

Data Type	Sample Value Range
single	-1 to 1
double	-1 to 1

## **F<sub>s</sub>**

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)`.

---

<code>get</code>	Query properties of <code>audioplayer</code> object.
<code>isplaying</code>	Query whether playback is in progress: returns true or false.

<code>pause</code>	Pause playback.
<code>play</code>	Play audio from beginning to end.
<code>playblocking</code>	Play, and do not return control until playback completes.
<code>resume</code>	Restart playback from paused position.
<code>set</code>	Set properties of <code>audioplayer</code> object.
<code>stop</code>	Stop playback.

See the reference pages for `get`, `play`, `playblocking`, and `set` for additional syntax options.

## Properties

<code>BitsPerSample</code>	Number of bits per sample. (Read-only)
<code>CurrentSample</code>	Current sample that the audio output device is playing. If the device is not playing, <code>CurrentSample</code> is the next sample to play with <code>play</code> or <code>resume</code> . (Read-only)
<code>DeviceID</code>	Identifier for audio device. (Read-only)
<code>NumberOfChannels</code>	Number of audio channels. (Read-only)
<code>Running</code>	Status of the audio player: 'on' or 'off'. (Read-only)
<code>SampleRate</code>	Sampling frequency in Hz.
<code>TotalSamples</code>	Total length of the audio data in samples. (Read-only)
<code>Tag</code>	String that labels the object.
<code>Type</code>	Name of the class: 'audioplayer'. (Read-only)
<code>UserData</code>	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.

# audioplayer

---

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 <code>TimerPeriod</code> property.
TimerPeriod	Time in seconds between <code>TimerFcn</code> callbacks. Default: <code>.05</code>

## 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

# audiorecorder

---

## 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)`.

---

<code>get</code>	Query properties of <code>audiorecorder</code> object.
<code>getaudiodata</code>	Create an array that stores the recorded signal values.
<code>getplayer</code>	Create an <code>audioplayer</code> object.
<code>isrecording</code>	Query whether recording is in progress: returns <code>true</code> or <code>false</code> .
<code>pause</code>	Pause recording.
<code>play</code>	Play recorded audio. This method returns an <code>audioplayer</code> object.
<code>record</code>	Start recording.
<code>recordblocking</code>	Record, and do not return control until recording completes. This method requires a second input for the length of the recording in seconds: <code>recordblocking(recorder, length)</code>
<code>resume</code>	Restart recording from paused position.
<code>set</code>	Set properties of <code>audiorecorder</code> object.
<code>stop</code>	Stop recording.

See the reference pages for `get`, `getaudiodata`, `play`, `record`, `recordblocking`, and `set` for additional syntax options.

## Properties

<code>BitsPerSample</code>	Number of bits per sample. (Read-only)
<code>CurrentSample</code>	Current sample that the audio input device is recording. If the device is not recording, <code>CurrentSample</code> is the next sample to record with <code>record</code> or <code>resume</code> . (Read-only)
<code>DeviceID</code>	Identifier for audio device. (Read-only)
<code>NumberOfChannels</code>	Number of audio channels. (Read-only)
<code>Running</code>	Status of the audio recorder: 'on' or 'off'. (Read-only)
<code>SampleRate</code>	Sampling frequency in Hz. (Read-only)
<code>TotalSamples</code>	Total length of the audio data in samples. (Read-only)
<code>Tag</code>	String that labels the object.
<code>Type</code>	Name of the class: 'audiorecorder'. (Read-only)
<code>UserData</code>	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.

<code>StartFcn</code>	Function to execute one time when recording starts.
<code>StopFcn</code>	Function to execute one time when recording stops.

# audiorecorder

---

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”

# auinfo

---

**Purpose** Information about NeXT/SUN (.au) sound file

**Syntax** `[m d] = auinfo(aufile)`

**Description** `[m d] = auinfo(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`

**Purpose**

Read NeXT/SUN (.au) sound file

**Syntax**

```
y = auread(afile)
[y,Fs] = auread(afile)
[y,Fs,nbits] = auread(afile)
[ ___ ] = auread(afile,N)
[ ___ ] = auread(afile,[N1 N2])
siz = auread(afile, 'size')
```

**Description**

`y = auread(afile)` loads a sound file specified by the string *afile*, 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(afile)` returns the sample rate (*Fs*) in Hertz used to encode the data in the file.

`[y,Fs,nbits] = auread(afile)` returns the number of bits per sample (*nbits*).

`[ ___ ] = auread(afile,N)` returns only the first *N* samples from each channel in the file.

`[ ___ ] = auread(afile,[N1 N2])` returns only samples *N1* through *N2* from each channel in the file.

`siz = auread(afile, '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
```

# auread

---

```
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](#)



---

<b>Purpose</b>	Write NeXT/SUN (.au) sound file
<b>Syntax</b>	<code>auwrite(y, aufile)</code> <code>auwrite(y, Fs, aufile)</code> <code>auwrite(y, Fs, N, aufile)</code> <code>auwrite(y, Fs, N, method, aufile)</code>
<b>Description</b>	<p><code>auwrite(y, aufile)</code> writes a sound file specified by the string <i>aufile</i>. The data should be arranged with one channel per column. Amplitude values outside the range [-1, +1] are clipped prior to writing. <code>auwrite</code> supports multichannel data for 8-bit mu-law and 8- and 16-bit linear formats.</p> <p><code>auwrite(y, Fs, aufile)</code> specifies the sample rate of the data in Hertz.</p> <p><code>auwrite(y, Fs, N, aufile)</code> selects the number of bits in the encoder. Allowable settings are <math>N = 8</math> and <math>N = 16</math>.</p> <p><code>auwrite(y, Fs, N, method, aufile)</code> allows selection of the encoding method, which can be either 'mu' or 'linear'. Note that mu-law files must be 8-bit. By default, <i>method</i> = 'mu'.</p>
<b>See Also</b>	<code>audioread</code>   <code>audiowrite</code>

# avifile

## 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
	<p>On UNIX operating systems, the only valid value is 'None'.</p> <p>On Windows systems, valid values include:</p> <ul style="list-style-type: none"> <li>• 'MSVC'</li> <li>• 'RLE'</li> <li>• 'Cinepak' on 32-bit systems.</li> <li>• 'Indeo3' or 'Indeo5' on 32-bit Windows XP systems.</li> </ul> <p>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.</p>	'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'	<p>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.</p> <p>Valid only for compressed movies.</p>	75
'videoname'	A descriptive name for the video stream, no more than 64 characters.	<i>filename</i>

## 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])  
end
```

```
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

# aviread

---

## 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, ...)
```

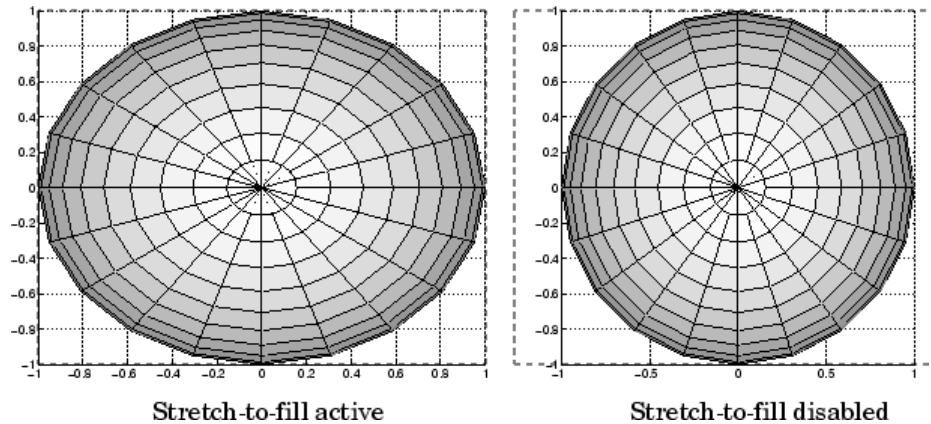
*PropertyName* is the name of the axes property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access axes properties.

### 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.



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`:

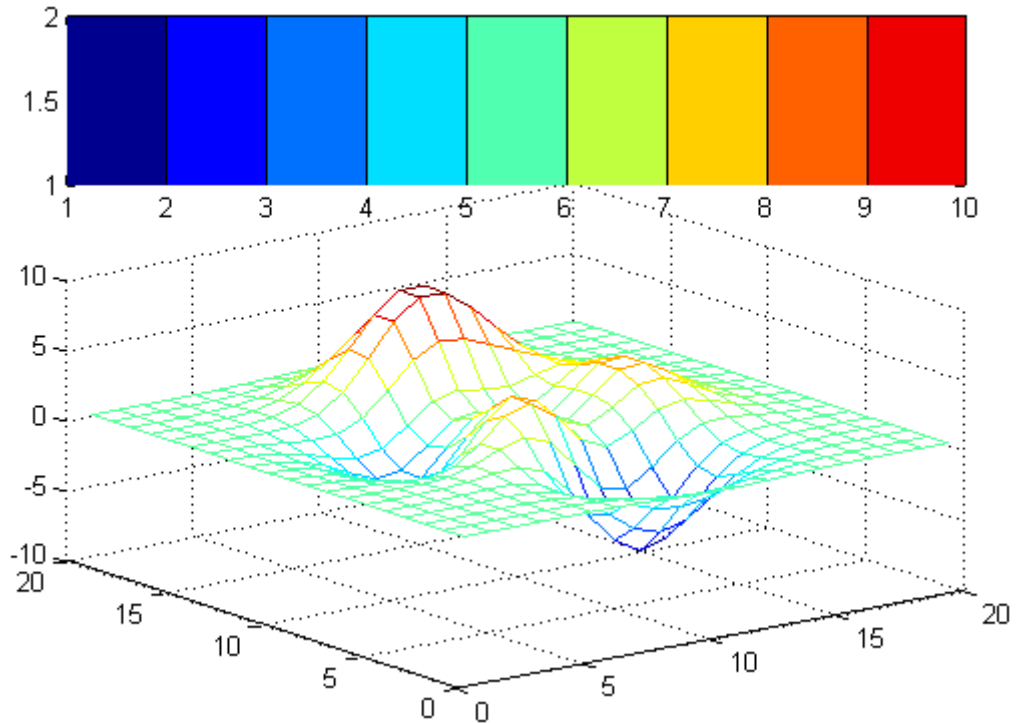
```
sphere
set(gca, 'CameraViewAngle', get(gca, 'CameraViewAngle')-5)
set(gca, 'CameraViewAngle', get(gca, 'CameraViewAngle')+5)
```

---

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”

# Axes Properties

---

## 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

# Axes Properties

---

*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).

# Axes Properties

---

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

## Axes Properties

---

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`:

## Axes Properties

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.

# Axes Properties

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 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`

`{none} | 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. Note that while the default value is `none`, the `matlabrc.m` startup file might set the axes `Color` to a specific color.

# Axes Properties

---

## 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:

$$\begin{bmatrix} x_{front} & y_{front} & z_{front} \\ x_{back} & y_{back} & z_{back} \end{bmatrix}$$

## Axes Properties

---

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

<b>X-, Y-, Z-LimitModes</b>	<b>DataAspectRatio</b>	<b>PlotBoxAspectRatio</b>	<b>Behavior</b>
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 <code>DataAspectRatio</code>

# Axes Properties

X-, Y-, Z-LimitModes	DataAspectRatio	PlotBoxAspectRatio	Behavior
			to achieve the requested <code>PlotBoxAspectRatio</code> within the limits the software selected.
auto	manual	auto	Limits chosen to span data range in all dimensions. MATLAB modifies <code>PlotBoxAspectRatio</code> to achieve the requested <code>DataAspectRatio</code> within the limits the software selected.
auto	manual	manual	Limits chosen to completely fit and center the plot within the requested <code>PlotBoxAspectRatio</code> given the requested <code>DataAspectRatio</code> (this might produce empty space around 2 of the 3 dimensions).
manual	auto	auto	MATLAB honors limits and modifies the <code>DataAspectRatio</code> and <code>PlotBoxAspectRatio</code> as necessary.

<b>X-, Y-, Z-LimitModes</b>	<b>DataAspectRatio</b>	<b>PlotBoxAspectRatio</b>	<b>Behavior</b>
manual	auto	manual	MATLAB honors limits and <code>PlotBoxAspectRatio</code> and modifies <code>DataAspectRatio</code> as necessary.
manual	manual	auto	MATLAB honors limits and <code>DataAspectRatio</code> and modifies the <code>PlotBoxAspectRatio</code> 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 <code>DataAspectRatio</code> while ignoring <code>PlotBoxAspectRatio</code> .

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

# Axes Properties

---

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

# Axes Properties

---

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

## Axes Properties

---

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.

# Axes Properties

---

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

LineStyleSpec {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.

# Axes Properties

---

- `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`

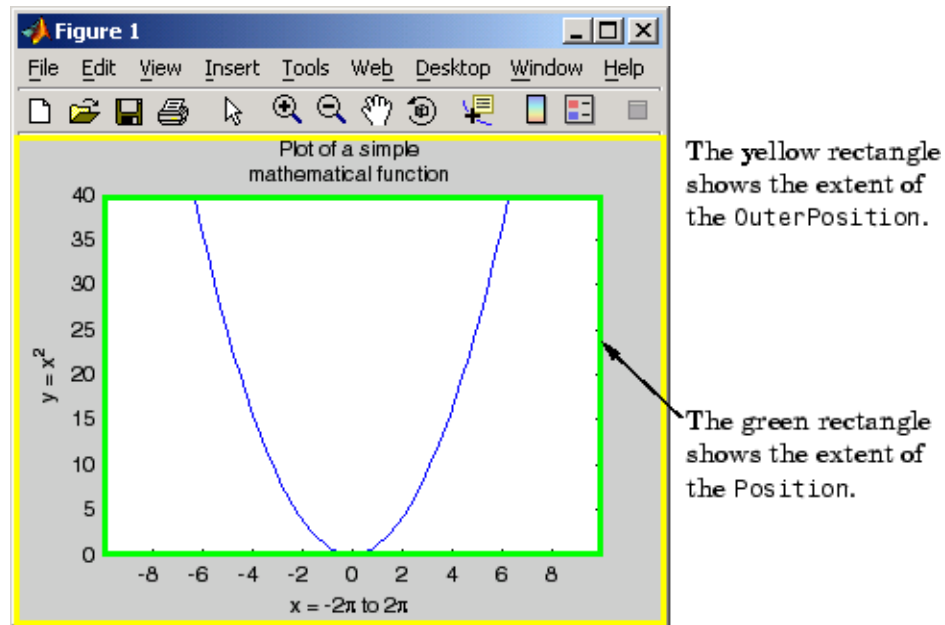
four-element vector

*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

# Axes Properties

---

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}

## Axes Properties

---

*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

## Axes Properties

---

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 =  $\frac{1}{72}$  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.

# Axes Properties

---

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')
```

## Axes Properties

---

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.

# Axes Properties

---

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

# Axes Properties

---

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.

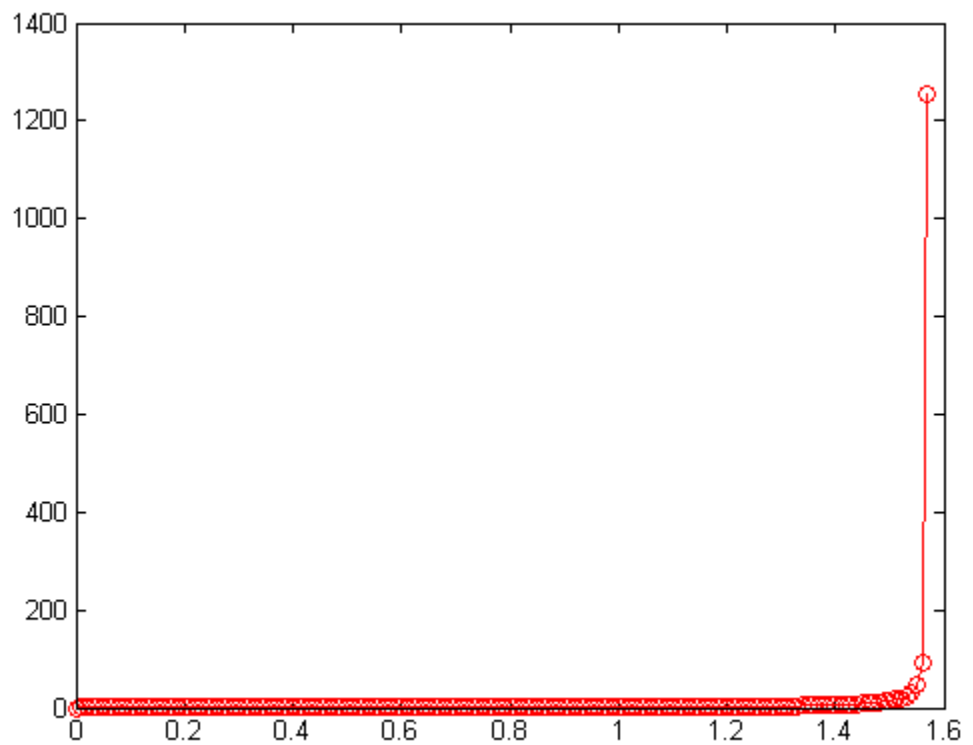
# axis

## Examples

The statements

```
x = 0:.01:pi/2;  
plot(x,tan(x),'-ro')
```

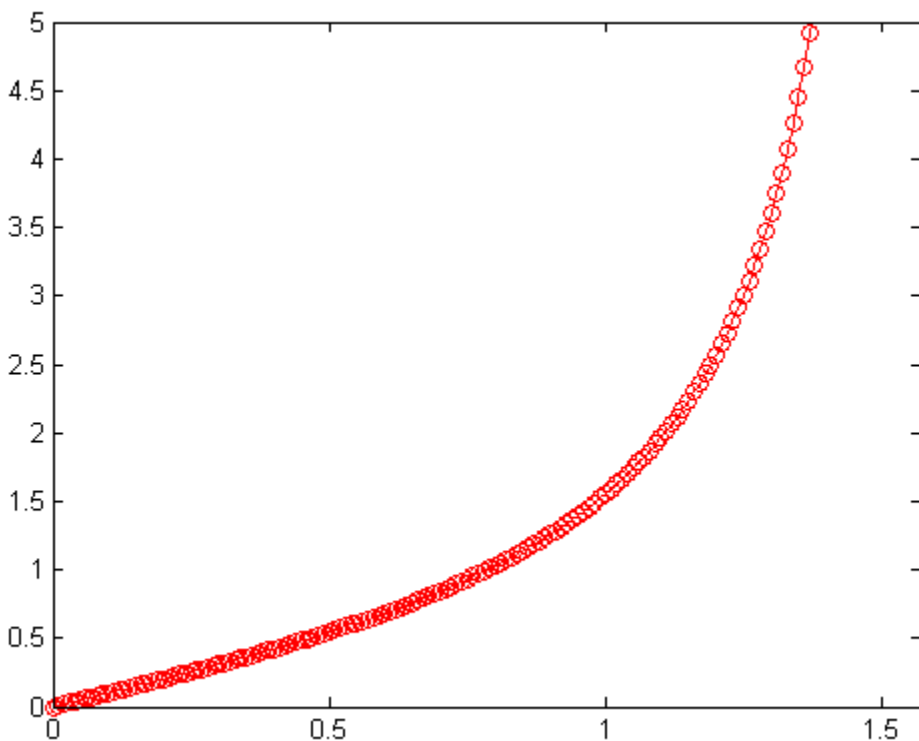
use the automatic scaling of the  $y$ -axis based on  $y_{\max} = \tan(1.57)$ , which is well over 1000:



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

---

`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`.

<b>Axes Property or Behavior</b>	<b>axis equal</b>	<b>axis normal</b>	<b>axis square</b>	<b>axis image</b>
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
<i>Stretch-to-fill</i> behavior;	disabled	active	disabled	disabled

## See Also

`axes` | `grid` | `subplot` | `xlim` | `ylim` | `zlim` | axes properties

**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  $B = T \backslash A * T$ , 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  $T(:,P) = \text{diag}(S)$  and  $B(P,P) = \text{diag}(1./S) * A * \text{diag}(S)$ .

`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,

$$\text{cond}(V) = \text{norm}(V) * \text{norm}(\text{inv}(V))$$

where

$$[V,T] = \text{eig}(A)$$

# balance

---

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  $\text{cond}(V)$  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  $\text{cond}(V)$  is 1.4421. The ill conditioning is concentrated in the scaling matrix;  $\text{cond}(T)$  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

# bar

---

**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];`

# bar

---

## 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 $Y$ . <ul style="list-style-type: none"><li>• If <math>Y</math> is a vector, bar ignores the 'stacked' option.</li><li>• If <math>Y</math> is a matrix, the bar height is the sum of the elements in the</li></ul>



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 <i>x</i> -ticks, rather than making bars span <i>x</i> -ticks as the <code>histc</code> 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, 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`.

**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 'BaseWidth' 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.

# bar

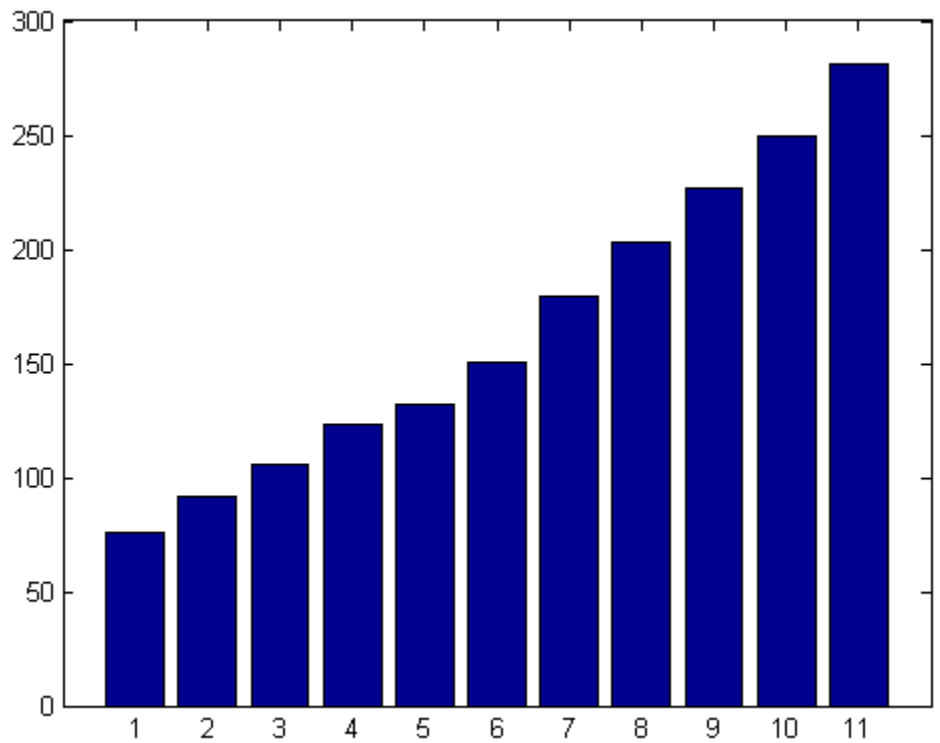
---

## Examples

### Single Data Series

Use the bar function to plot vector data.

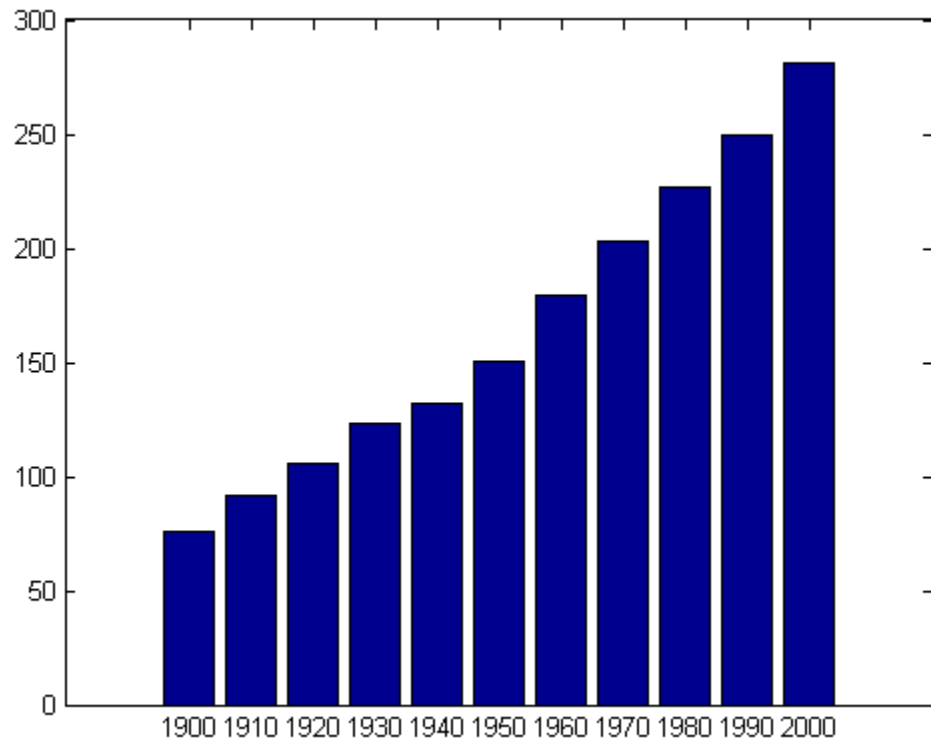
```
y = [75.995,91.972,105.711,123.203,131.669,...  
     150.697,179.323,203.212,226.505,249.633,281.422];  
figure; bar(y);
```



### 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  $\text{length}(x)$  and  $\text{length}(y)$  have to be same.

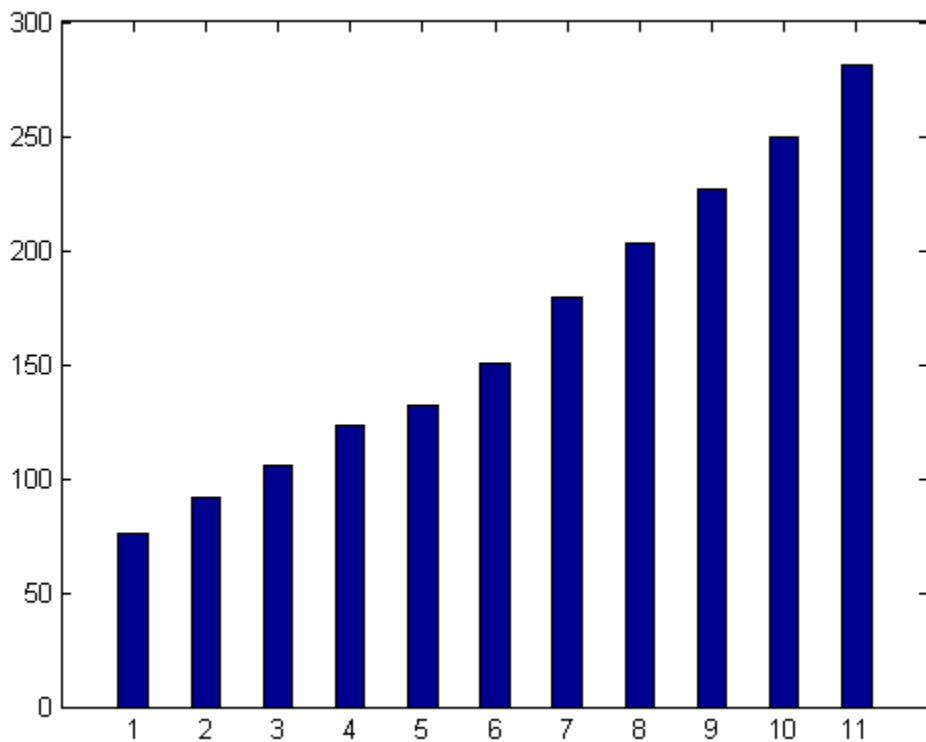
# bar

---

## Bar Width

Set width of each bar to 40 percent of the total space available for each bar.

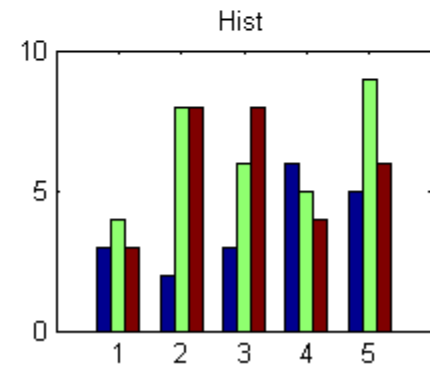
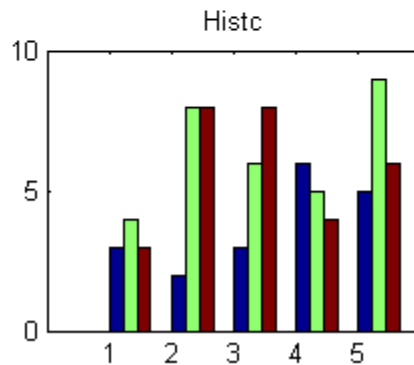
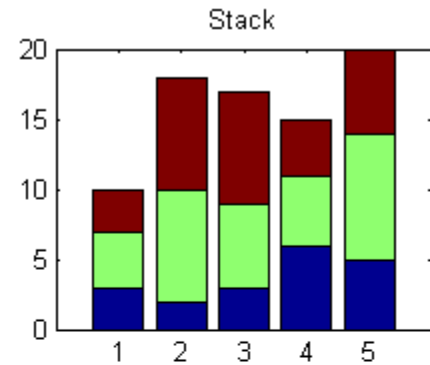
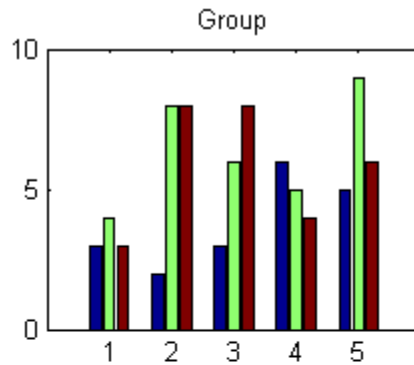
```
y = [75.995,91.972,105.711,123.203,131.669,...  
     150.697,179.323,203.212,226.505,249.633,281.422];  
figure; bar(y,0.4);
```



## 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')
```

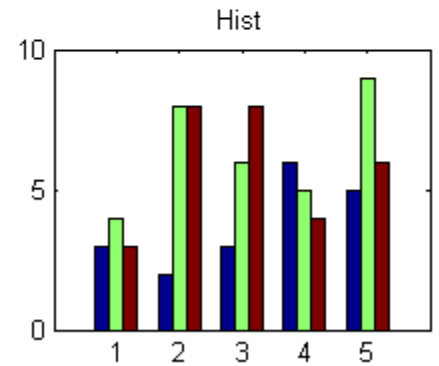
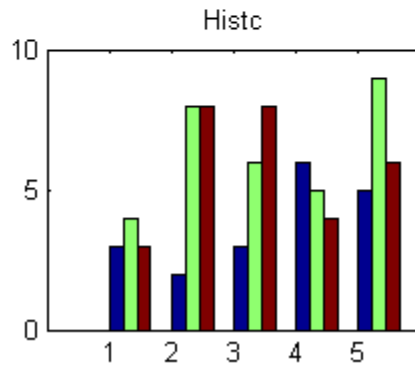
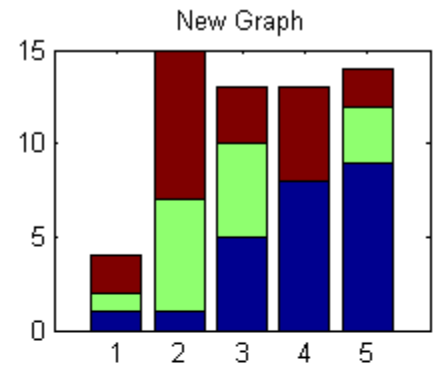
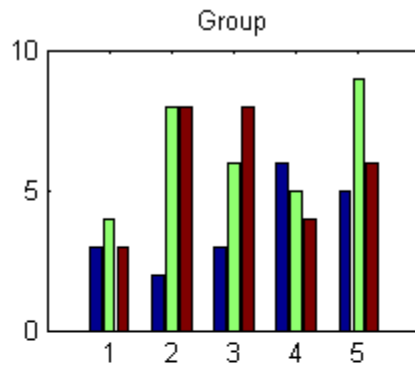


# bar

## 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')
```

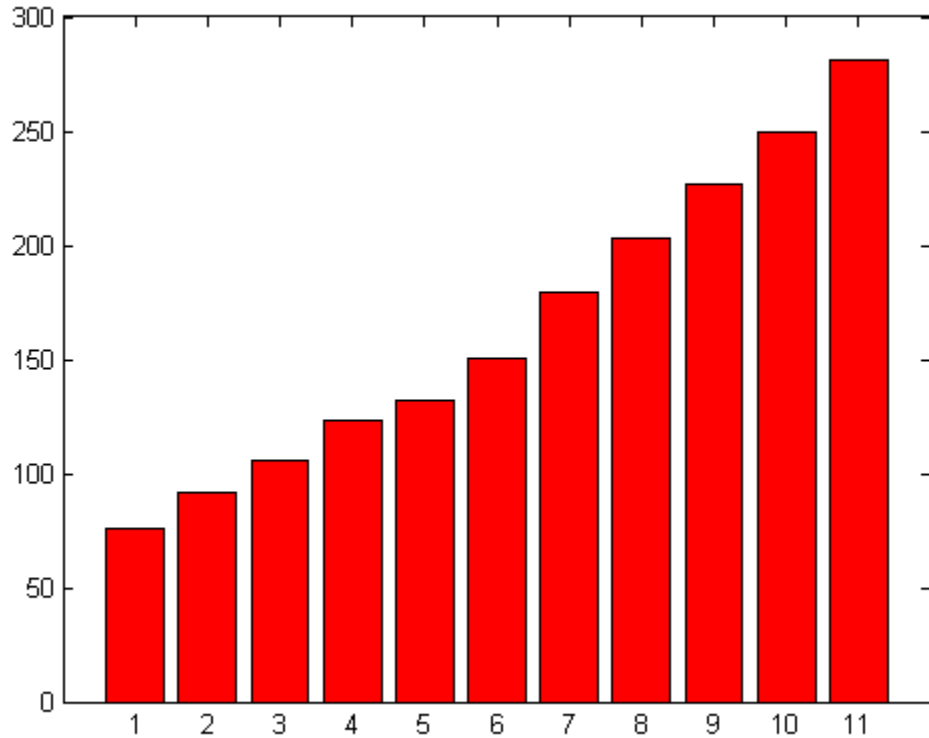


## Bar Color

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

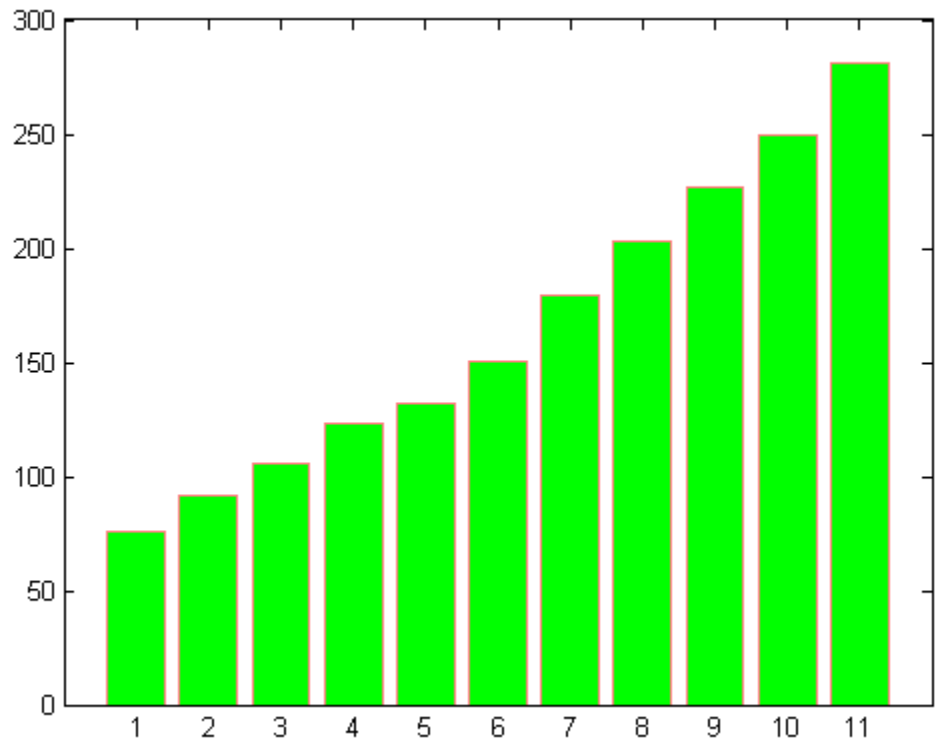
Specify green bars and use an RGB color specification for the bar edge color.

```
y = [75.995,91.972,105.711,123.203,131.669,...  
      150.697,179.323,203.212,226.505,249.633,281.422];
```

# bar

---

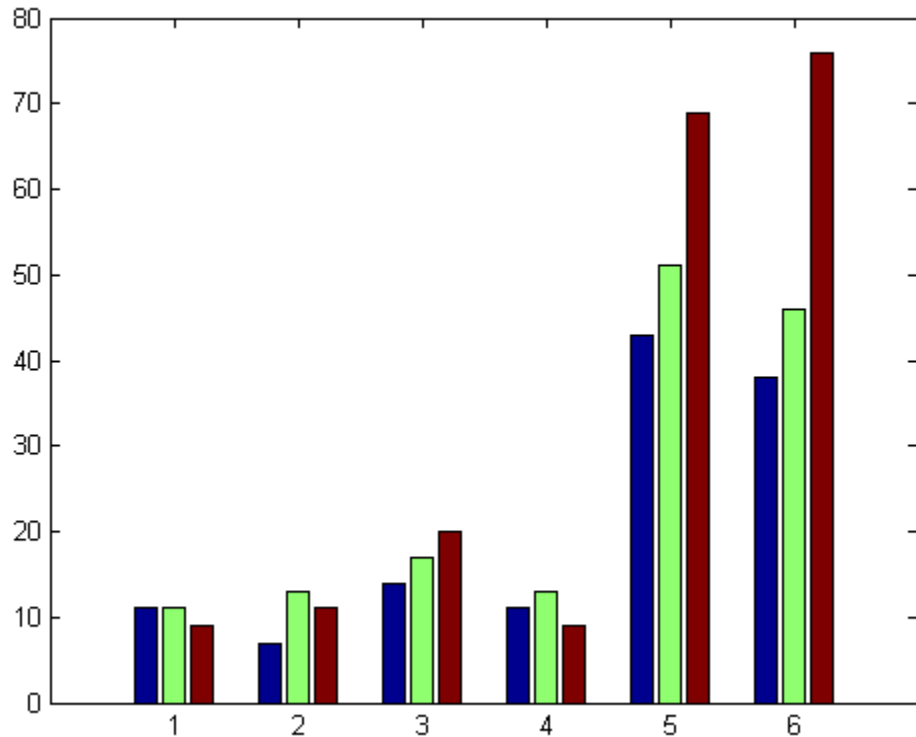
```
figure; bar(y,'g','EdgeColor',[1,0.5,0.5]);
```



## 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

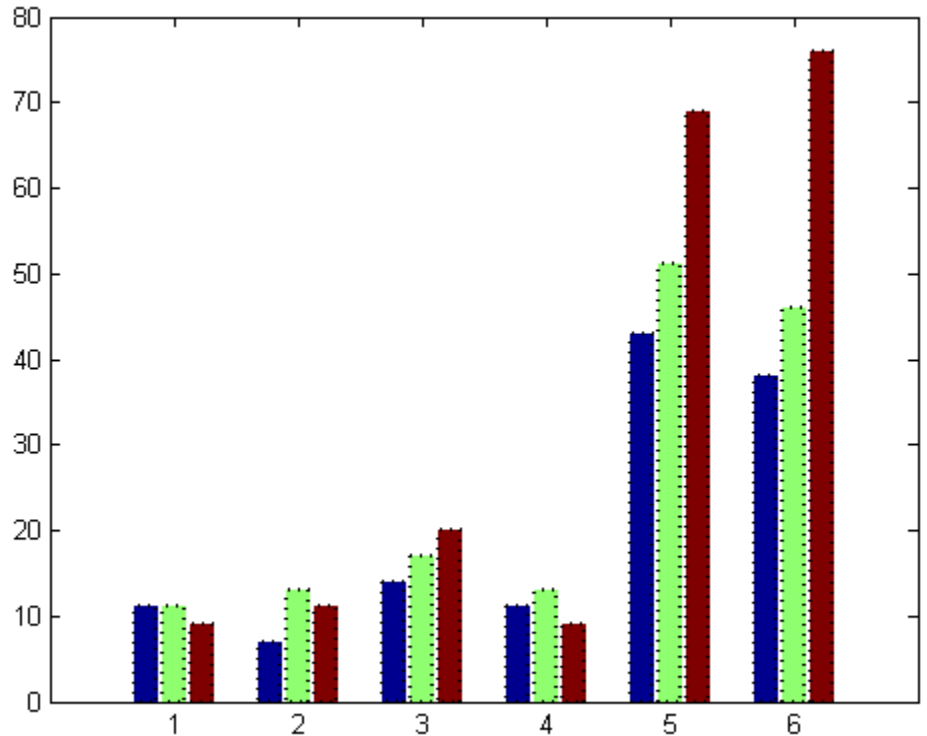
Set appearance of one column of data.

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);
```

# bar

```
set(hArray(2),'LineWidth',2,'EdgeColor','red');
```

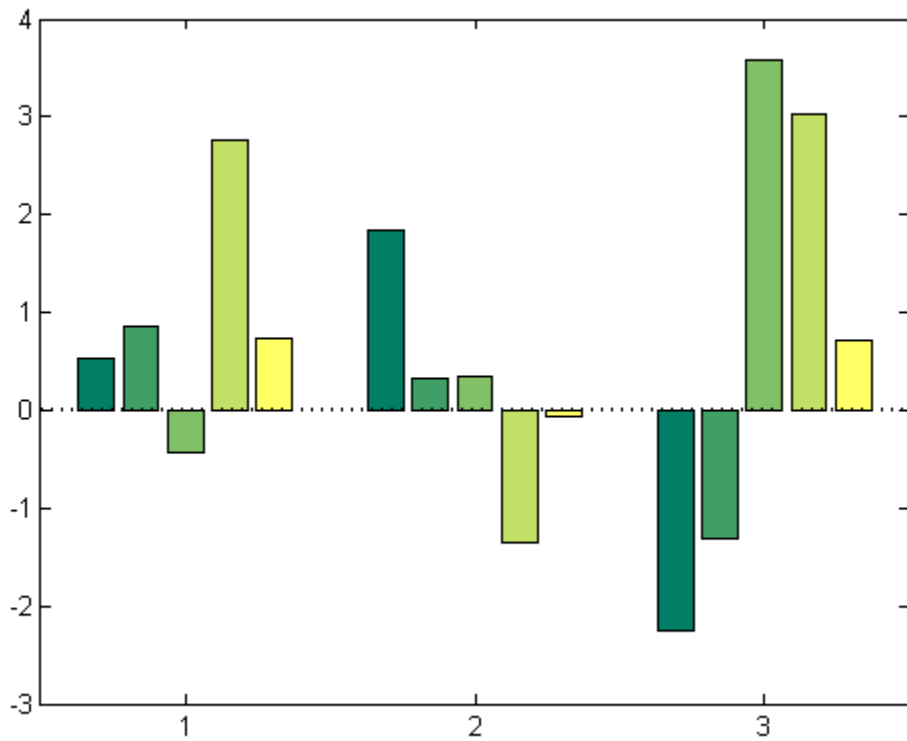


## Baseline Modifications

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',':')
```



# bar

---

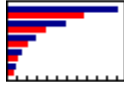
## See Also

[barh](#) | [bar3](#) | [bar3h](#) | [ColorSpec](#) | [stairs](#) | [hist](#) “Barseries Property Descriptions” on page 2-421

## Related Examples

- “Bar and Area Graphs”

**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

# barh

---

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

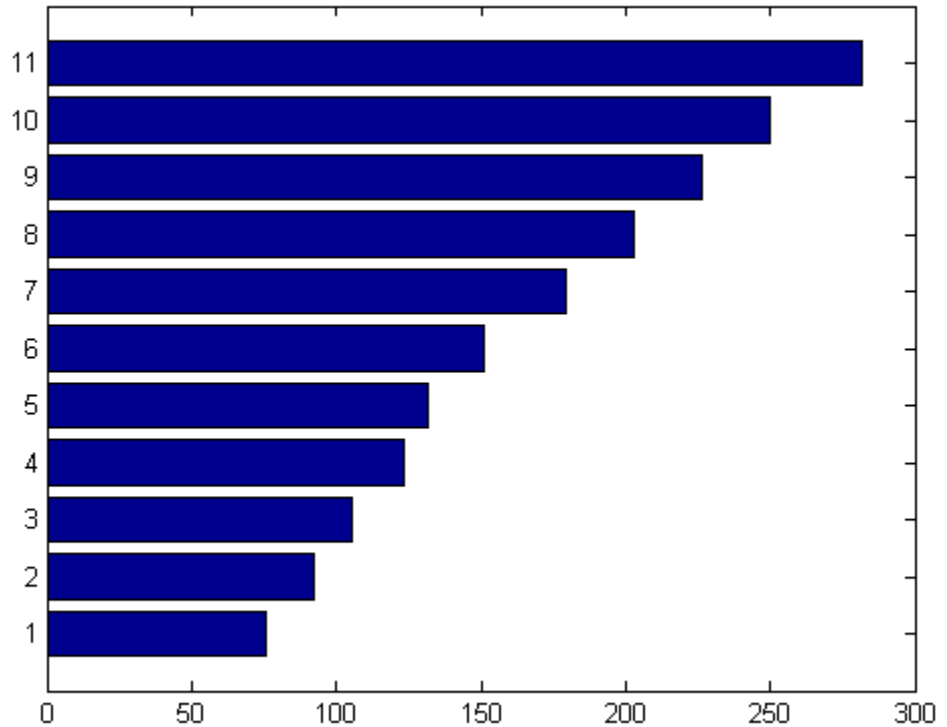
### Single Series of Data

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);
```

# barh

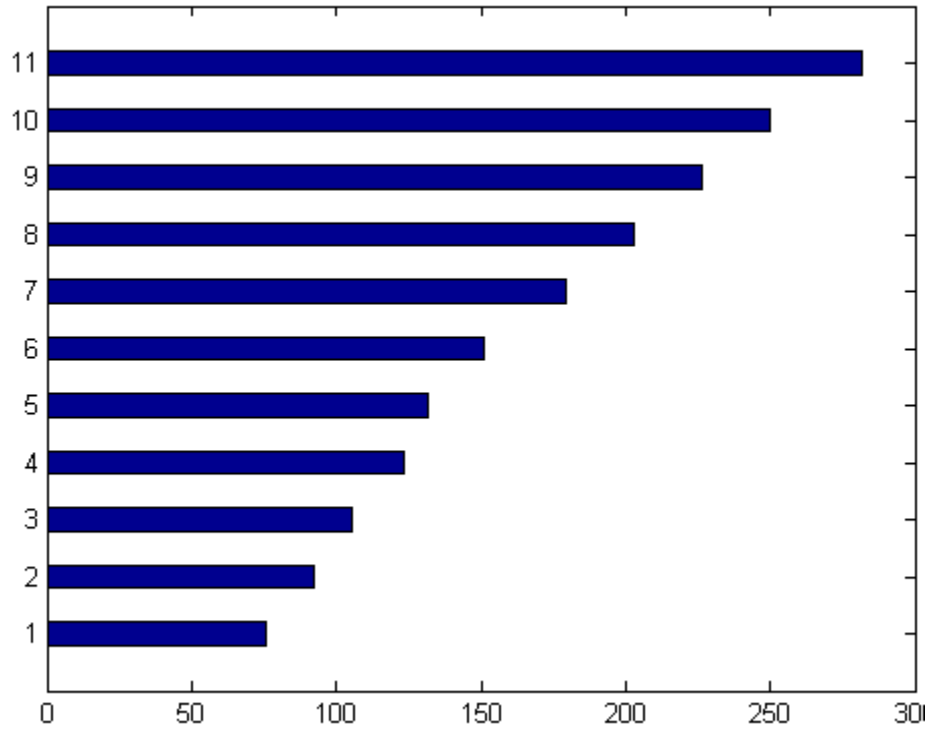
---



## 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);
```



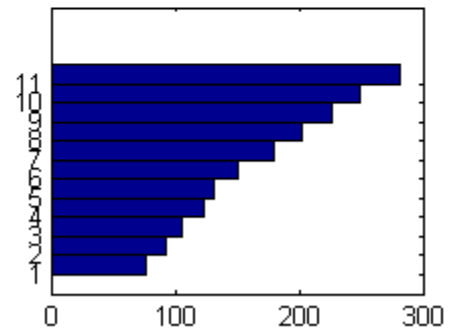
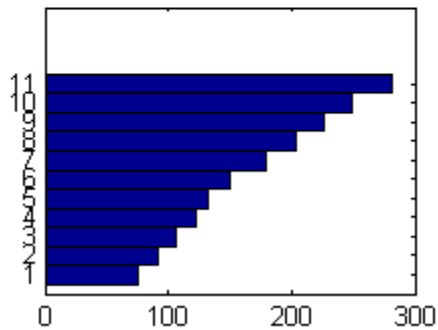
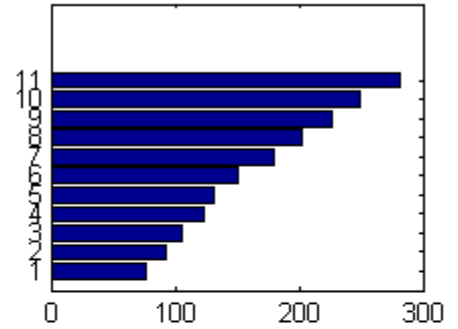
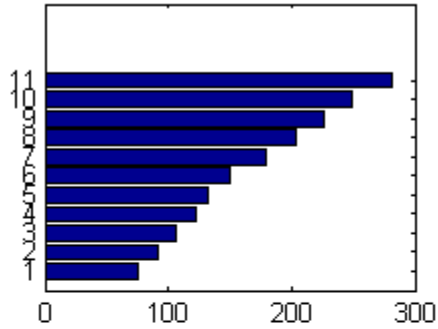
### Specifying style

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');
```

# barh

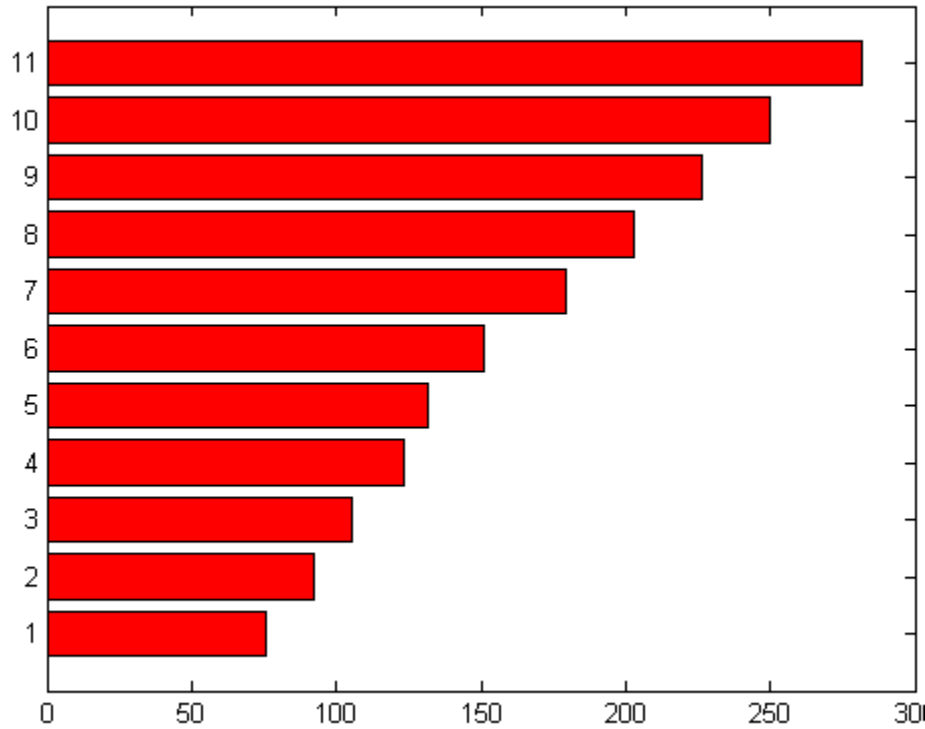
---



## Specifying color

This example shows how to specify color for the graph.

```
figure; barh(y, 'r');
```



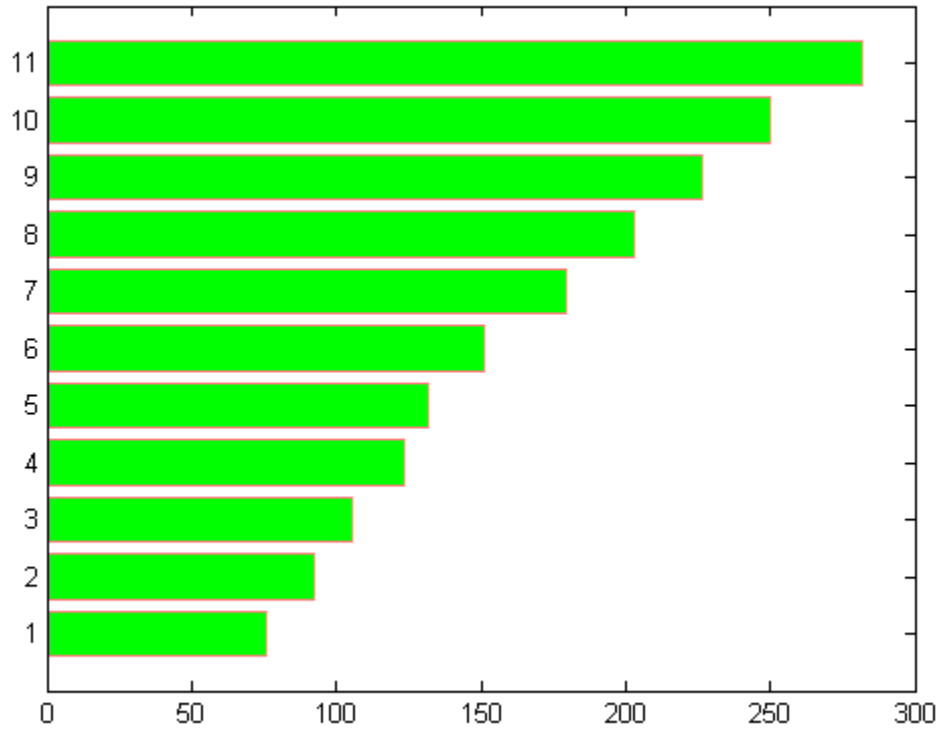
### Specifying Name-Value Pairs

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]);
```

# barh

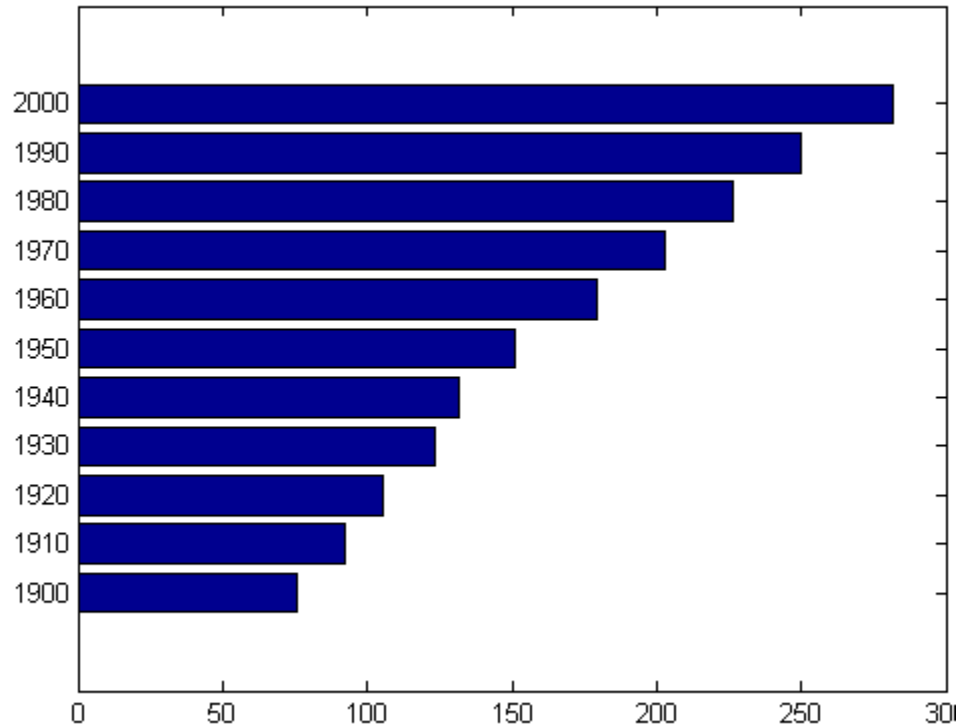
---



## 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);
```

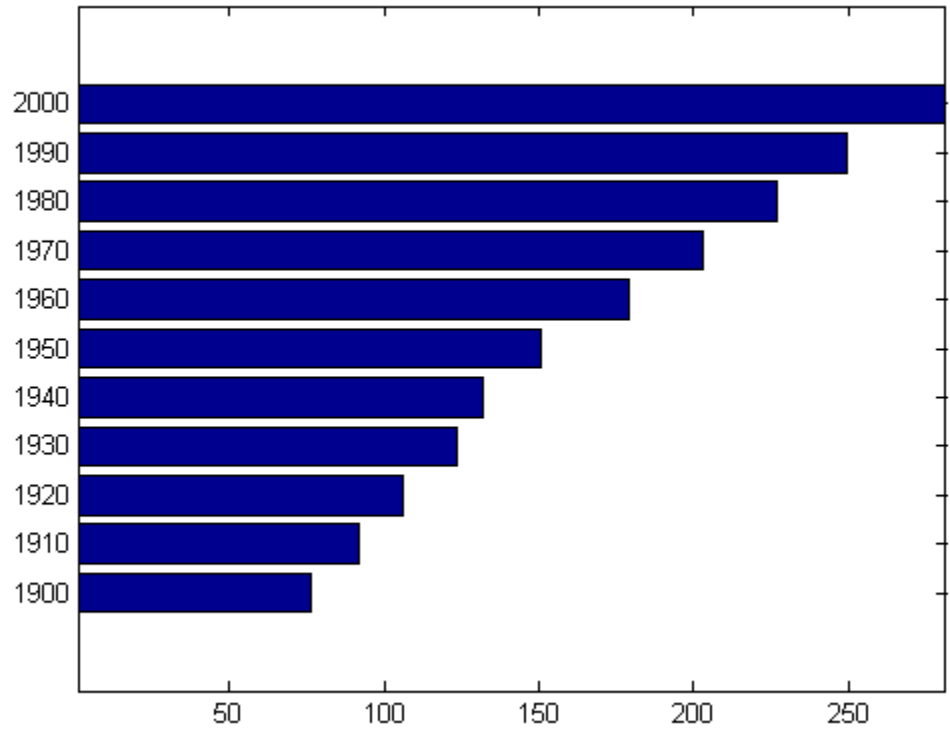


### Setting Axis Scale

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)]);
```

# barh

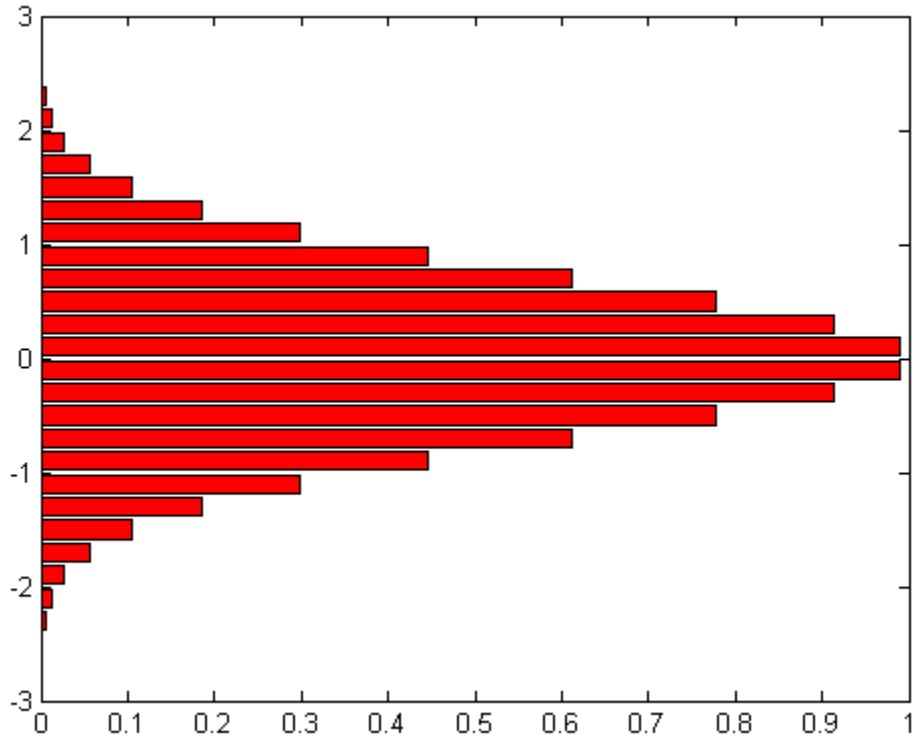


## Passing an Expression

You can also pass the expression with the `barh` function.

```
a = -2.9:0.2:2.9;  
barh(a,exp(-a.*a),'r')
```



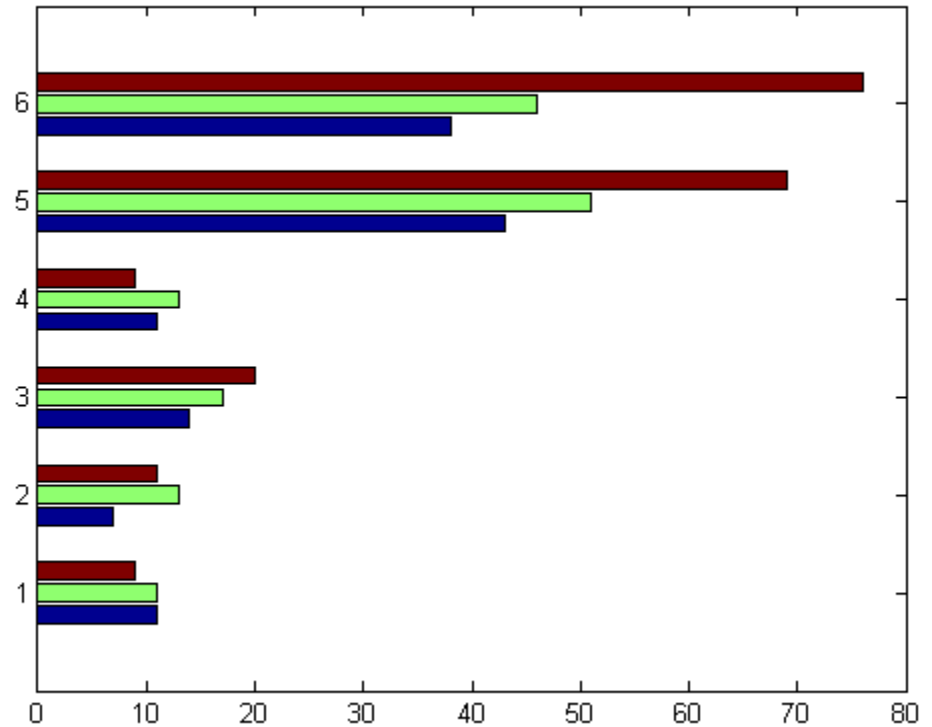


### Plotting Matrix Data

This example shows how to plot matrix data using `barh`.

```
load count.dat;  
yMat = count(1:6,:);  
figure; barh(yMat);
```

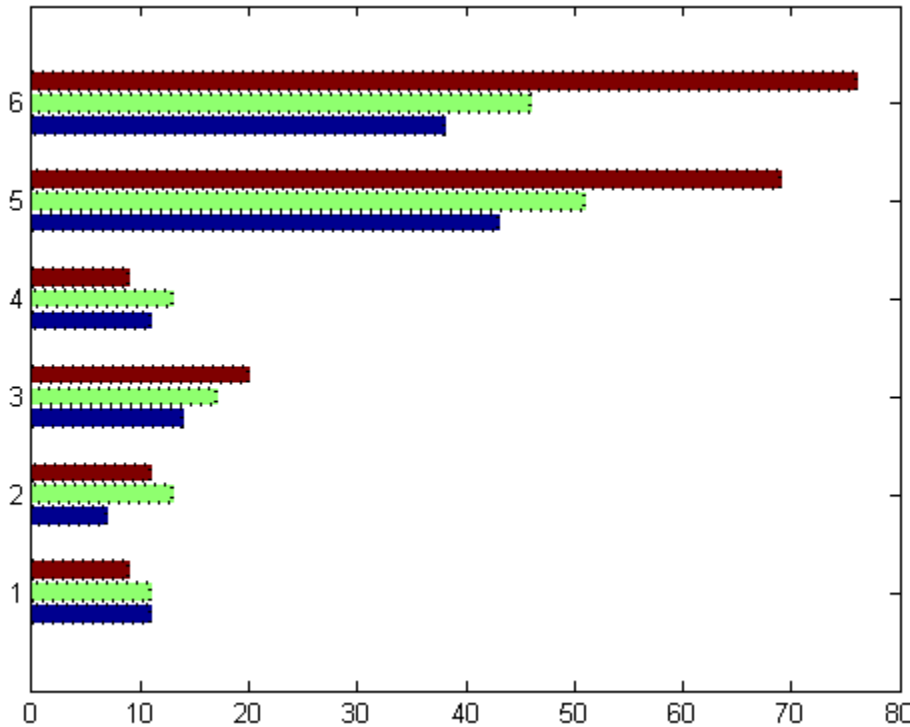
# barh



## 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',':');
```

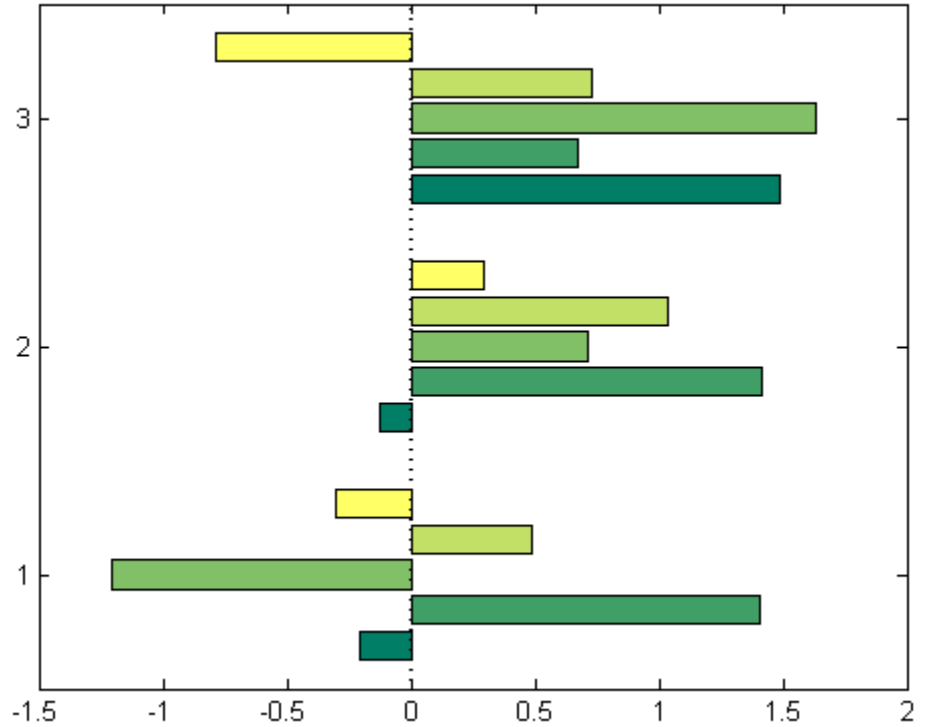


### Setting Properties with Multiobject Graphs

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
```

# barh



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

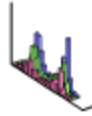
`bar` | `bar3` | `bar3h` | `ColorSpec` | `stairs` | `hist` | `Barseries`  
Properties

## How To

- “Bar and Area Graphs”

**Purpose**

Plot 3-D bar graph

**Syntax**

```
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineStyle)
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(...,LineStyle)` displays all bars using the color specified by `LineStyle`.

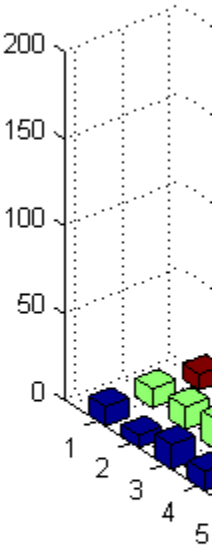
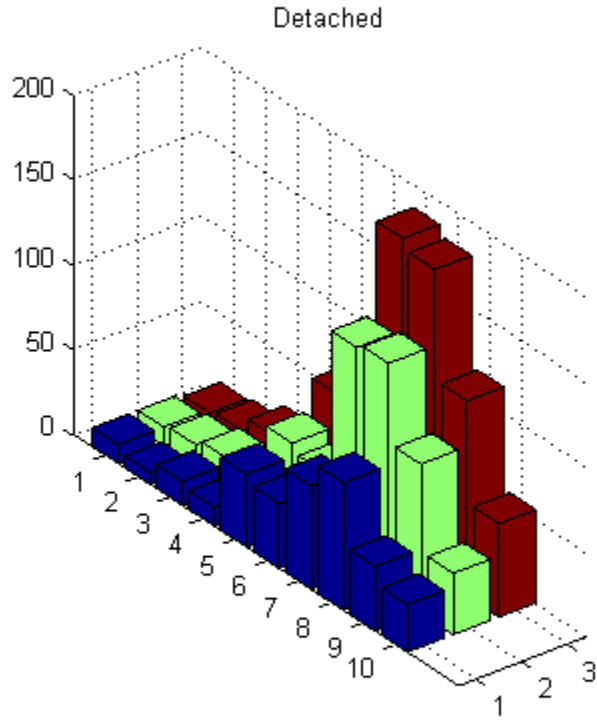
`bar3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`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');
```



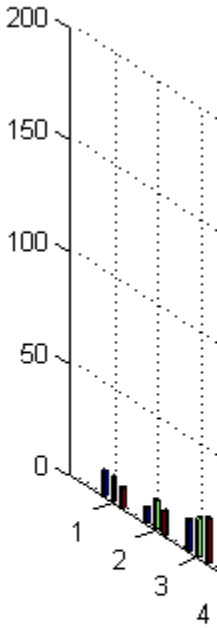
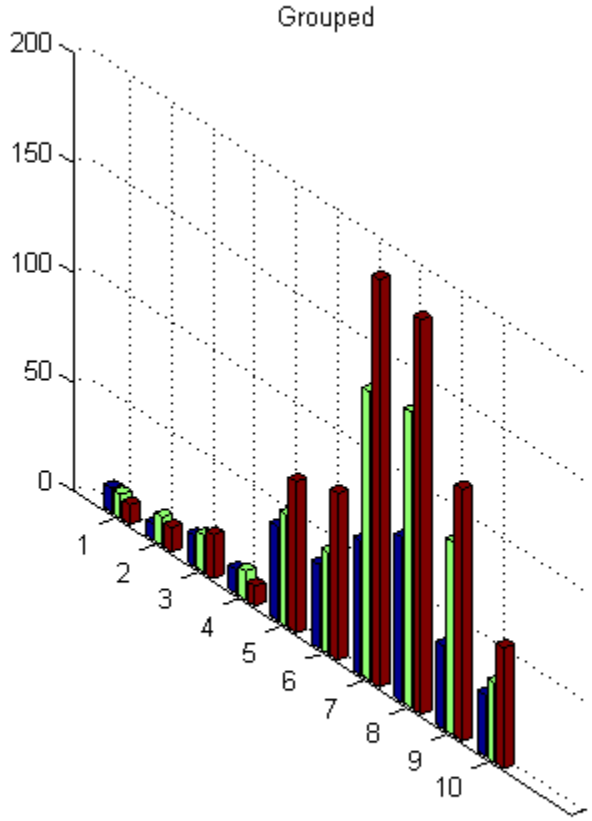
## bar3

---

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');
```



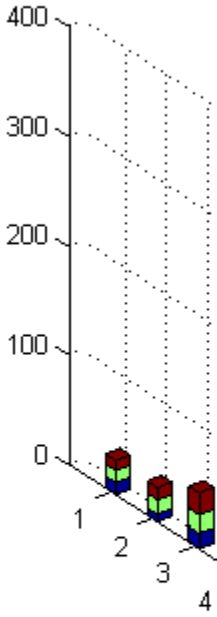
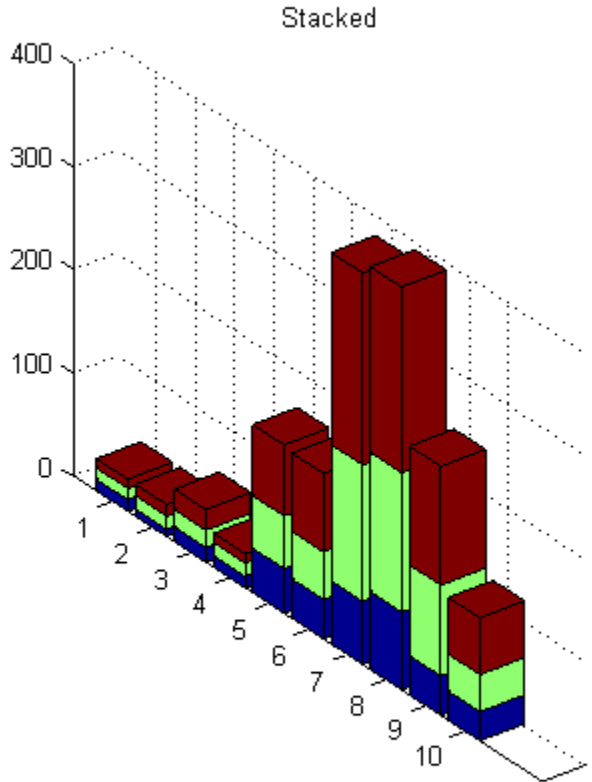


## bar3

---

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');
```



# bar3

---

## See Also

[bar](#) | [barh](#) | [bar3h](#) | [LineStyle](#)

## How To

- “Bar and Area Graphs”

**Purpose** Plot horizontal 3-D bar graph



**Syntax**

```
bar3h(Y)
bar3h(x,Y)
bar3h(...,width)
bar3h(...,'style')
bar3h(...,LineStyle)
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.

# bar3h

---

- '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(...,LineStyle)` displays all bars using the color specified by `LineStyle`.

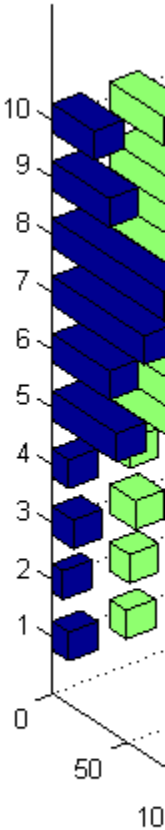
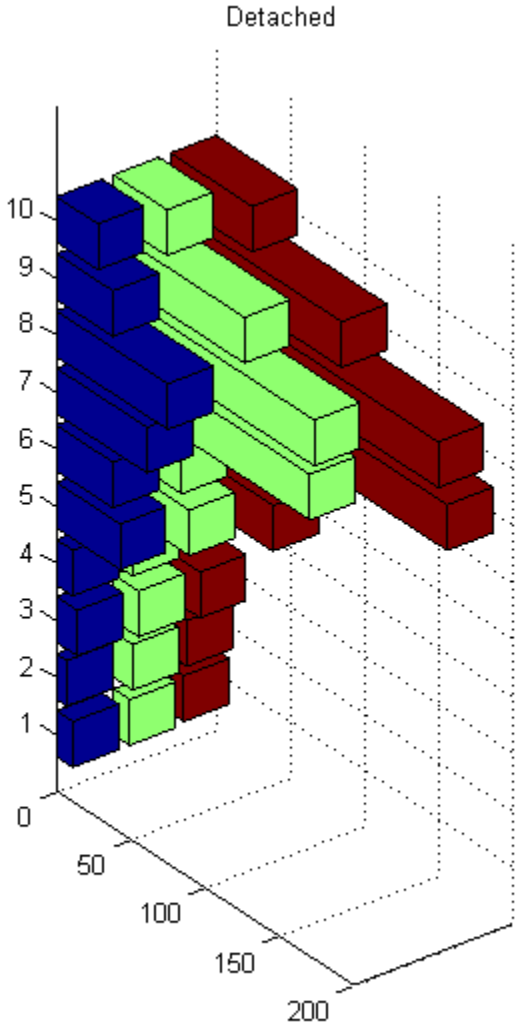
`bar3h(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`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');
```



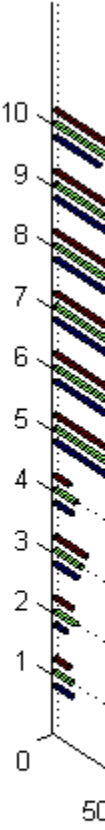
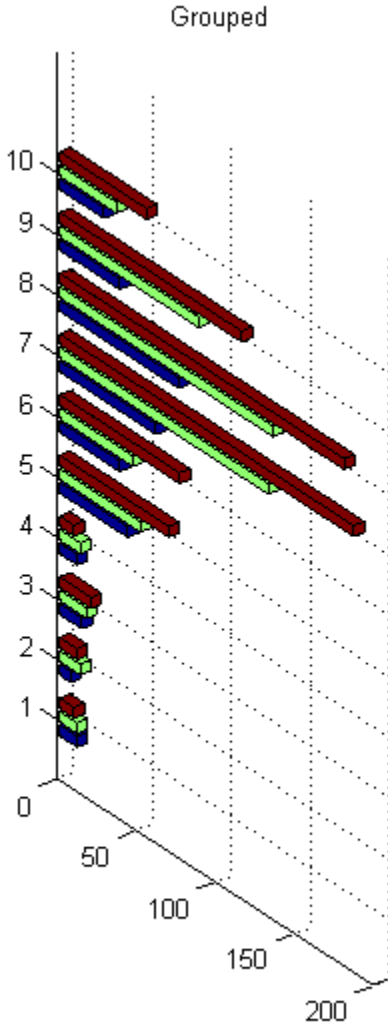
## bar3h

---

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');
```



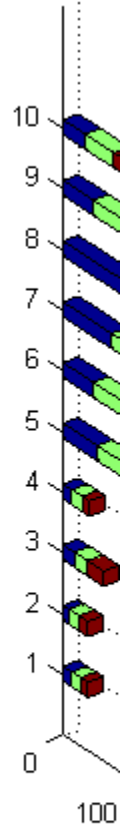
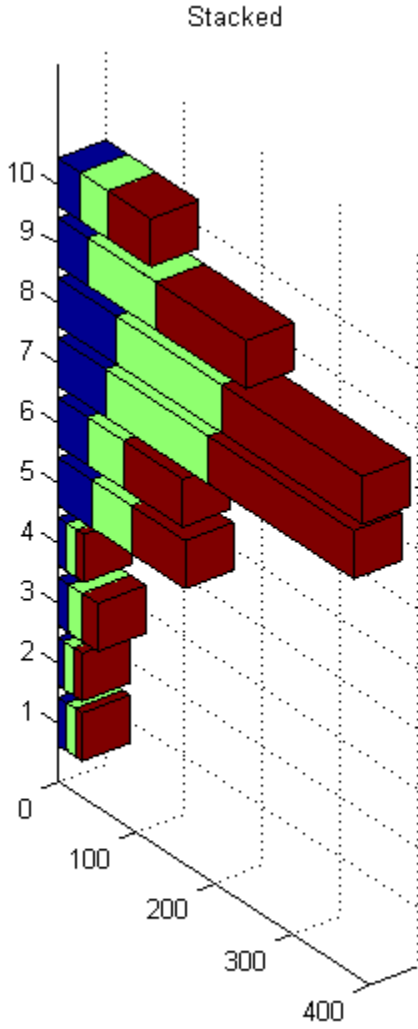


## bar3h

---

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');
```



# bar3h

---

## See Also

[bar](#) | [barh](#) | [bar3](#) | [LineStyle](#) | [patch](#)

## How To

- “Bar and Area Graphs”

## 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.

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
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

# Barseries Properties

---

## 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]

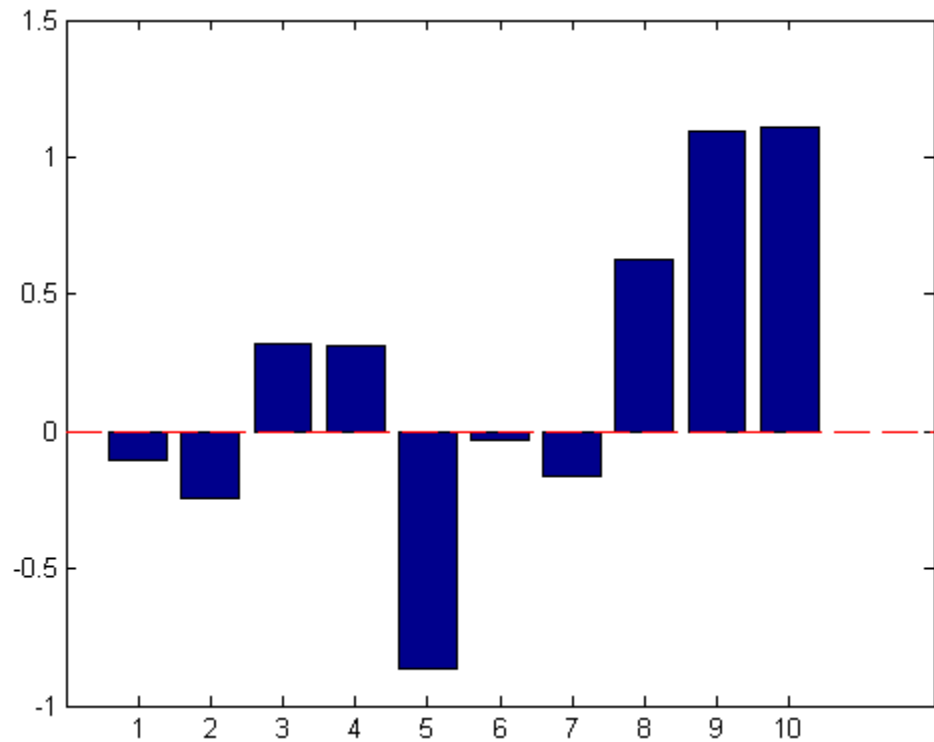
*Width of individual bars.* `BarWidth` specifies the relative bar width and controls the separation of bars within a group. The default is 0.8, so if you do not specify  $x$ , the bars within a group have a slight separation. If `BarWidth` is 1, the bars within a group touch one another.

`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

# Barseries Properties

---

*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-to-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:

# Barseries Properties

---

```
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.

## Barseries Properties

---

- 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`.

# Barseries Properties

---

- `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).

## Barseries Properties

---

- 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

# Barseries Properties

---

## 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, hgroup, or hgtransform

*Parent of object.* Handle of the object's parent. The parent is normally the axes, hgroup, 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 `'hgroup'`.

The following statement finds all the `hgroup` objects in the current axes.

```
t = findobj(gca, 'Type', 'hgroup');
```

UIContextMenu

handle of `uicontextmenu` object

## Barseries Properties

---

*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"

# Barseries Properties

---

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”

<b>Purpose</b>	Convert point coordinates from barycentric to Cartesian	
<b>Syntax</b>	<code>XC = baryToCart(TR, SI, B)</code>	
<b>Description</b>	<code>XC = baryToCart(TR, SI, B)</code> returns the Cartesian coordinates <code>XC</code> of each point in <code>B</code> that represents the barycentric coordinates with respect to its associated simplex <code>SI</code> .	
<b>Input Arguments</b>	<code>TR</code>	Triangulation representation.
	<code>SI</code>	Column vector of simplex indices that index into the triangulation matrix <code>TR.Triangulation</code>
	<code>B</code>	<code>B</code> is a matrix that represents the barycentric coordinates of the points to convert with respect to the simplices <code>SI</code> . <code>B</code> is of size <code>m-by-k</code> , where <code>m = length(SI)</code> , the number of points to convert, and <code>k</code> is the number of vertices per simplex.
<b>Output Arguments</b>	<code>XC</code>	Matrix of cartesian coordinates of the converted points. <code>XC</code> is of size <code>m-by-n</code> , where <code>n</code> is the dimension of the space where the triangulation resides. That is, the Cartesian coordinates of the point <code>B(j)</code> with respect to simplex <code>SI(j)</code> is <code>XC(j)</code> .
<b>Definitions</b>	A simplex is a triangle/tetrahedron or higher-dimensional equivalent.	
<b>Examples</b>	Compute the Delaunay triangulation of a set of points.	
	<pre>x = [0 4 8 12 0 4 8 12]'; y = [0 0 0 0 8 8 8 8]'; dt = DelaunayTri(x,y)</pre>	
	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;
```

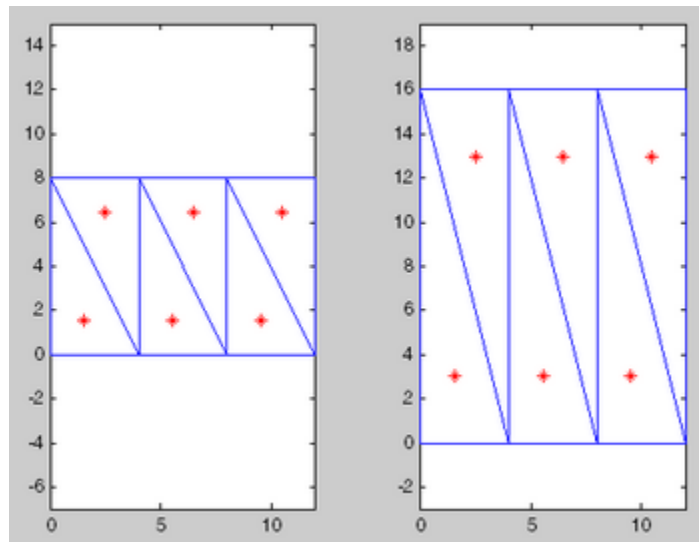
Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
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

[TriRep.cartToBary](#) | [DelaunayTri.pointLocation](#) | [DelaunayTri](#)

# base2dec

---

**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`

**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).

# BeginInvoke

---

<b>Purpose</b>	Initiate asynchronous .NET delegate call
<b>Syntax</b>	<code>result = BeginInvoke(arg1, ..., argN, callback, object)</code>
<b>Description</b>	<code>result = BeginInvoke(arg1, ..., argN, callback, object)</code> initiates asynchronous call to a .NET delegate. You must call <code>EndInvoke</code> to complete the asynchronous call.
<b>Input Arguments</b>	<p><b>arg1, ..., argN</b> Input arguments for delegate. The type and number of arguments must agree with the delegate signature.</p> <p><b>callback</b> .NET <code>System.AsyncCallback</code> delegate, or [ ] null value.</p> <p><b>object</b> User-defined object, or [ ] null value.</p>
<b>Output Arguments</b>	<p><b>result</b> .NET <code>System.IAsyncResult</code> object. Used to monitor the progress of the asynchronous call. Input argument to <code>EndInvoke</code>.</p>
<b>See Also</b>	<code>EndInvoke</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Calling .NET Methods Asynchronously”</li></ul>
<b>Related Links</b>	<ul style="list-style-type: none"><li>• MSDN® Calling Synchronous Methods Asynchronously</li></ul>

**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 comparison data for other computers is stored in the following text file. Updated versions of this file are available from MATLAB Central:

<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=1836&objectType=file#>

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`

**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

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2) y = 0,$$

where  $\nu$  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  $\nu$ .  $Y_\nu(z)$  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

# besselh

$$H_v^{(1)}(z) = J_v(z) + iY_v(z)$$

$$H_v^{(2)}(z) = J_v(z) - iY_v(z)$$

where  $J_v(z)$  is `besselj`, and  $Y_v(z)$  is `bessely`.

## Description

`H = besselh(nu,K,Z)` computes the Hankel function  $H_v^{(K)}(z)$  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  $H_v^{(K)}(z)$  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	<code>besselh</code> 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.

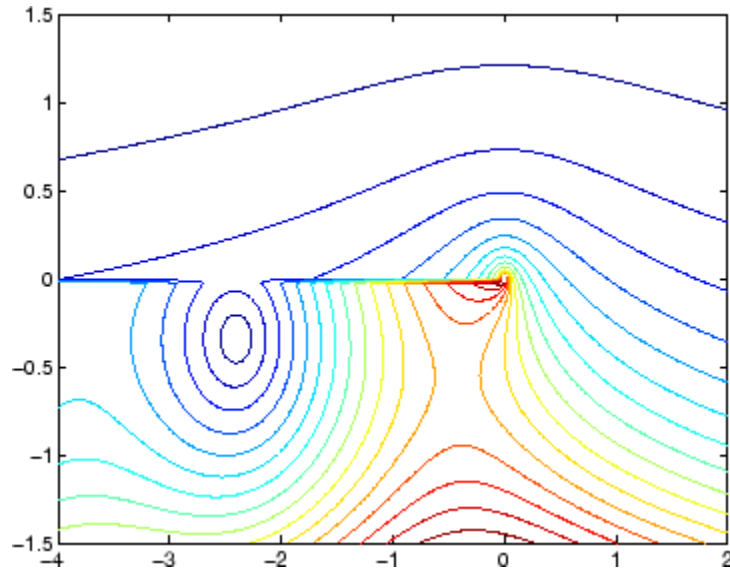
## Examples

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
```

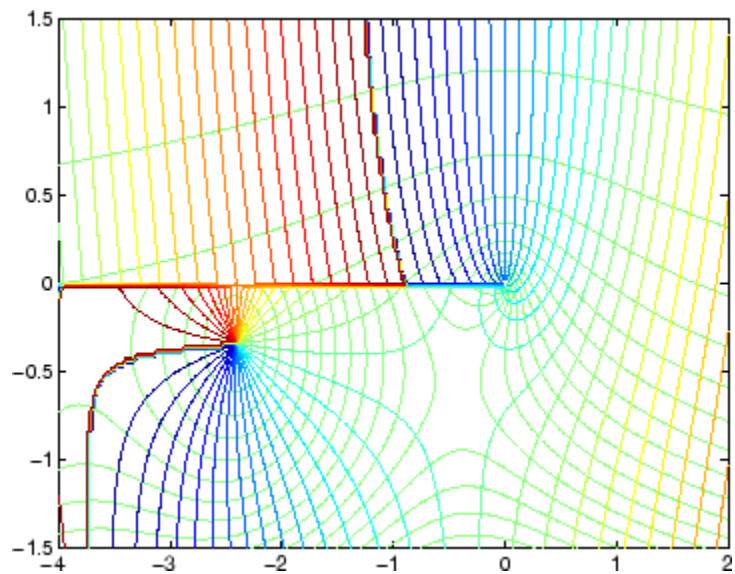


then adds the contour plot of the phase of the same function.

```
contour(X,Y,(180/pi)*angle(H),-180:10:180); hold off
```

# besselh

---



## 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

**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

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - (z^2 + \nu^2) y = 0,$$

where  $\nu$  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  $\nu$ .  $I_\nu(z)$  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`.

# besseli

## 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	<code>besseli</code> 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, <code>Z</code> or <code>nu</code> 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

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2)y = 0,$$

where  $\nu$  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  $\nu$ .  $J_\nu(z)$  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`.

<b>ierr</b>	<b>Description</b>
0	<code>besselj</code> 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, <code>Z</code> or <code>nu</code> 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

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - (z^2 + \nu^2)y = 0,$$

where  $\nu$  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  $\nu$ :

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k! \Gamma(\nu + k + 1)}$$

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`.

<b>ierr</b>	<b>Description</b>
0	<code>besselk</code> 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, <code>Z</code> or <code>nu</code> 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](#)

# 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

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2)y = 0,$$

where  $\nu$  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  $\nu$

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k! \Gamma(\nu + k + 1)},$$

and  $\Gamma(\alpha)$  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`.

<b>ierr</b>	<b>Description</b>
0	<code>bessely</code> 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, <code>Z</code> or <code>nu</code> 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)';

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 | bess elk

# beta

---

**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**

$\text{beta}(z,w) = \exp(\text{gammaIn}(z) + \text{gammaIn}(w) - \text{gammaIn}(z+w))$

**See Also**

[betainc](#) | [betaIn](#) | [gammaIn](#)

# betainc

---

**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:

$$1 - \text{betainc}(X,Z,W) = \text{betainc}(X,Z,W, \text{'upper'})$$

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 =  
1.000000000000000  
0.875000000000000  
0.687500000000000  
0.500000000000000  
0.343750000000000  
0.226562500000000  
0.144531250000000  
0.089843750000000  
0.054687500000000  
0.032714843750000  
0.019287109375000
```

**See Also**

[beta](#) | [betaIn](#)

# 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`

---

<b>Purpose</b>	Logarithm of beta function
<b>Syntax</b>	<code>L = betaIn(Z,W)</code>
<b>Description</b>	<p><code>L = betaIn(Z,W)</code> computes the natural logarithm of the beta function <code>log(beta(Z,W))</code>, for corresponding elements of arrays <code>Z</code> and <code>W</code>, without computing <code>beta(Z,W)</code>. Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.</p> <p><code>Z</code> and <code>W</code> must be real and nonnegative. They must be the same size, or either can be scalar.</p>
<b>Examples</b>	<pre>x = 510 betaIn(x,x)  ans =     -708.8616</pre> <p><code>-708.8616</code> is slightly less than <code>log(realmin)</code>. Computing <code>beta(x,x)</code> directly would underflow (or be denormal).</p>
<b>Algorithms</b>	<code>betaIn(z,w) = gammaIn(z)+gammaIn(w)-gammaIn(z+w)</code>
<b>See Also</b>	<code>beta</code>   <code>betainc</code>   <code>gammaIn</code>

## 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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $\text{inv}(M)*A*x = \text{inv}(M)*b$  for  $x$ . If  $M$  is `[]` then `bicg` applies



no preconditioner.  $M$  can be a function handle `mfun`, such that `mfun(x, 'notransp')` returns  $M \backslash x$  and `mfun(x, 'transp')` returns  $M' \backslash x$ .

`bicg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If  $x_0$  is `[]`, then `bicg` uses the default, an all-zero vector.

`[x,flag] = bicg(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	<code>bicg</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>bicg</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner $M$ was ill-conditioned.
3	<code>bicg</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>bicg</code> 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  $\|b - A \cdot x\| / \|b\|$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = bicg(A,b,...)` also returns the iteration number at which  $x$  was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = bicg(A,b,...)` also returns a vector of the residual norms at each iteration including  $\|b - A \cdot x_0\|$ .

## 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)           % y = A'*x
    if strcmp(transp_flag,'transp')
        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
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:

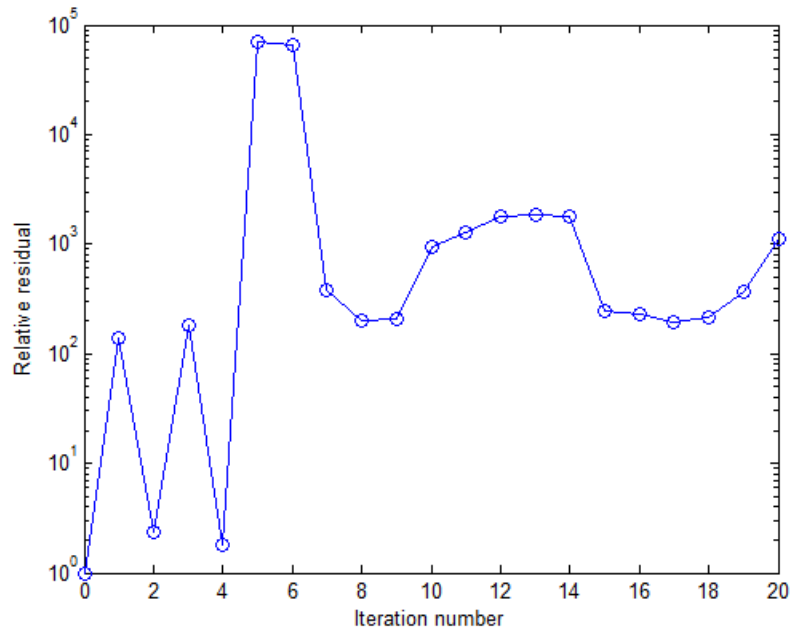
```
[x0,f10,rr0,it0,rv0] = bicg(A,b,tol,maxit);
```

`f10` 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:

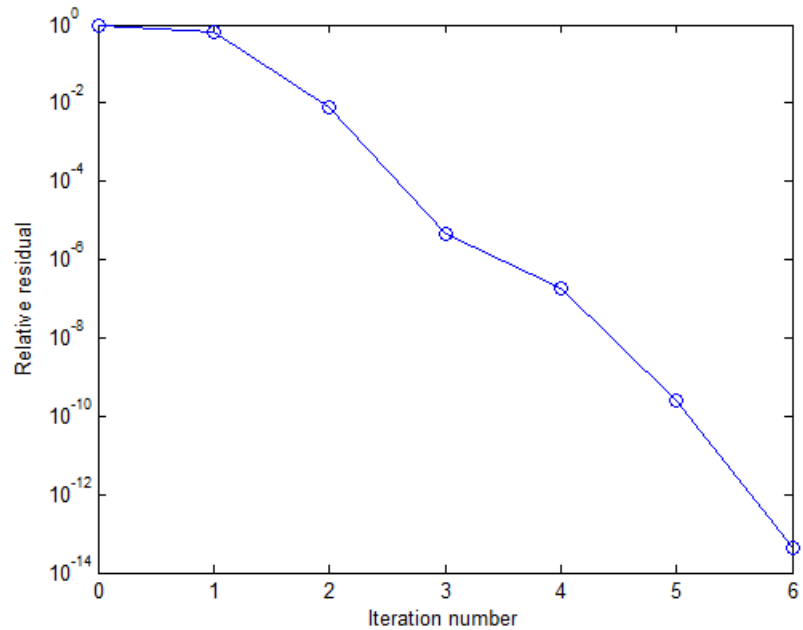
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));  
[x1,f11,rr1,it1,rv1] = bicg(A,b,tol,maxit,L,U);
```

f11 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`

# bicgstab

---

**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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $\text{inv}(M)*A*x = \text{inv}(M)*b$  for  $x$ . If  $M$  is `[]` then `bicgstab` applies no preconditioner.  $M$  can be a function handle `mfun`, such that `mfun(x)` returns  $M\backslash x$ .



`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	<code>bicgstab</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>bicgstab</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner <code>M</code> was ill-conditioned.
3	<code>bicgstab</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>bicgstab</code> 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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = bicgstab(A,b,...)` also returns the iteration number at which `x` was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ . `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  $\text{norm}(b-A*x_0)$ .

## 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:

# bicgstab

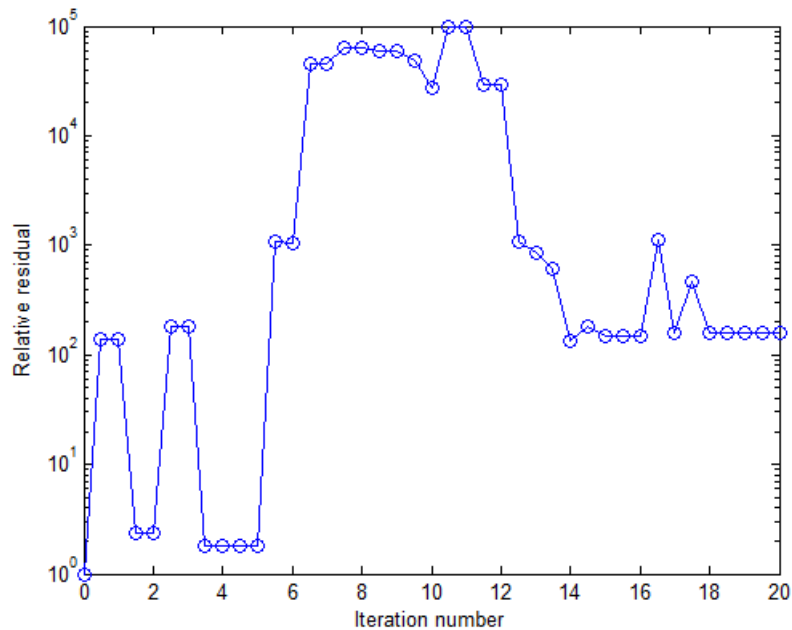
```
[x0,f10,rr0,it0,rv0] = bicgstab(A,b,tol,maxit);
```

f10 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 ( $x_0 = \text{zeros}(\text{size}(A,2),1)$ ) is the best solution and is returned as indicated by  $it_0 = 0$ . 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:

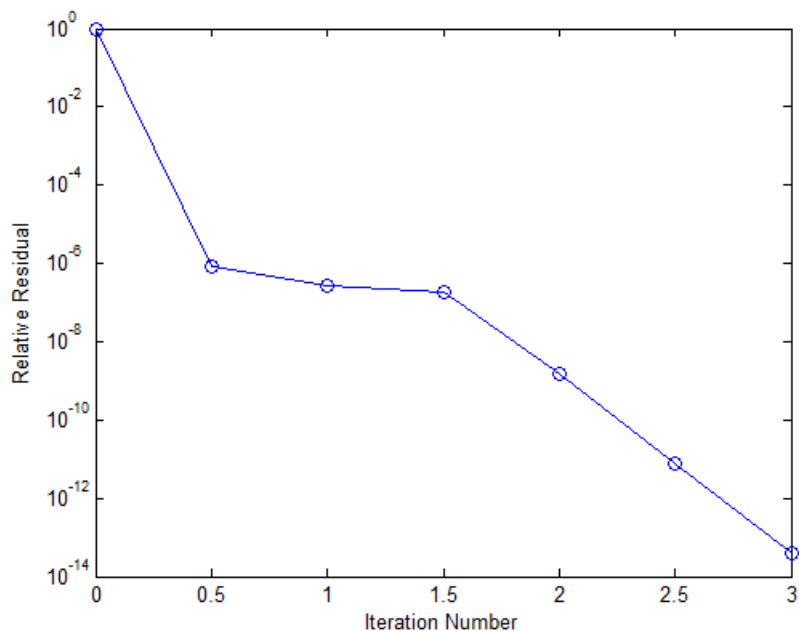
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,f11,rr1,it1,rv1] = bicgstab(A,b,tol,maxit,L,U);
```

`f11` 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

**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  $M\backslash x$ .

`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:

# bicgstabl

---

Flag	Convergence
0	bicgstabl converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	bicgstabl iterated <code>maxit</code> times but did not converge.
2	Preconditioner <code>M</code> 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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, `relres <= tol`.

`[x,flag,relres,iter] = bicgstabl(A,b,...)` also returns the iteration number at which `x` was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ . `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  $\text{norm}(b-A*x_0)$ .

## 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:

# bicgstabl

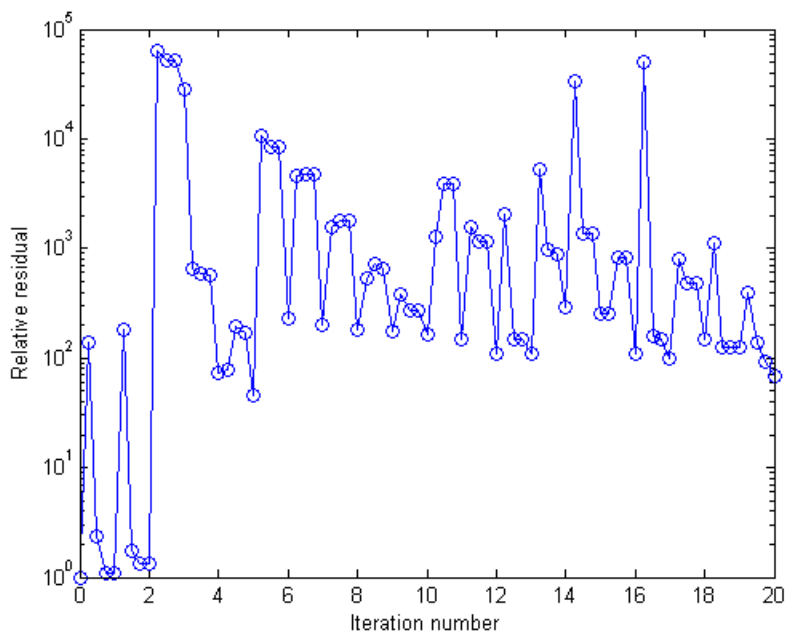
```
[x0,f10,rr0,it0,rv0] = bicgstabl(A,b,tol,maxit);
```

f10 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,f11,rr1,it1,rv1] = bicgstabl(A,b,tol,maxit,L,U);
```

`f11` 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');
```

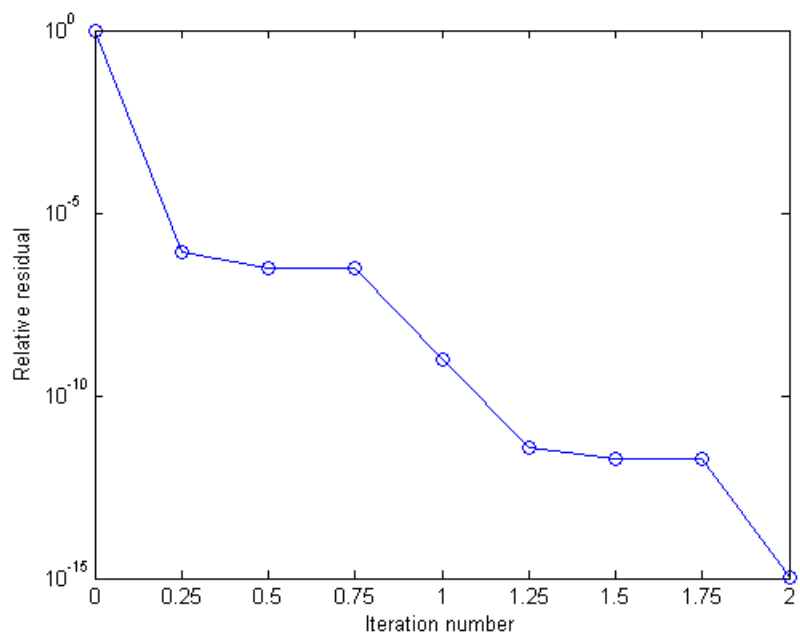
```
set(gca,'XTick',0:0.25:it1);
```

```
xlabel('Iteration number');
```

```
ylabel('Relative residual');
```

# bicgstabl

---



## See Also

[bicgstab](#) | [bicg](#) | [cgs](#) | [gmres](#) | [lsqr](#) | [luinc](#) | [minres](#) | [pcg](#) | [qmr](#) | [symmlq](#) | [function\\_handle](#) | [mldivide](#)

**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`

# FTP.binary

---

**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`

**Purpose**

Bitwise AND

**Syntax**

```
intout = bitand(integ1,integ2)
intout = bitand(integ1,integ2,assumedtype)

objout = bitand(netobj1,netobj2)
```

**Description**

`intout = bitand(integ1,integ2)` returns the bitwise 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 bitwise 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` - Bitwise AND result

signed integer array | unsigned integer array | double array

Bitwise 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` - Bitwise AND result

`.NET` enumeration object

Bitwise 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 bitwise inputs are 1.

### Negative Values

Explore how bitand handles negative values

MATLAB encodes signed integers using two's complement. Thus, the bitwise 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”

# bitcmp

---

<b>Purpose</b>	Bitwise complement
<b>Compatibility</b>	<code>bitcmp(A,N)</code> will not accept <code>N</code> in a future release. Use <code>bitcmp(A)</code> or <code>bitcmp(A,assumedtype)</code> instead.
<b>Syntax</b>	<pre>cmp = bitcmp(A) cmp = bitcmp(A,assumedtype)  cmp = bitcmp(A,N)</pre>
<b>Description</b>	<p><code>cmp = bitcmp(A)</code> returns the bitwise complement of <code>A</code>.</p> <p><code>cmp = bitcmp(A,assumedtype)</code> assumes that <code>A</code> is of <code>assumedtype</code>.</p> <p><code>cmp = bitcmp(A,N)</code> returns an <code>N</code>-bit complement of <code>A</code>. Elements of <code>A</code> cannot exceed <math>2^N-1</math>.</p>
<b>Input Arguments</b>	<p><b>A - Input value</b> signed integer array   unsigned integer array   double array</p> <p>Input value, specified as a signed integer array, unsigned integer array, or double array.</p> <ul style="list-style-type: none"><li>• If <code>A</code> is a double array, and <code>assumedtype</code> is not specified, then MATLAB treats <code>A</code> as an unsigned 64-bit integer.</li><li>• If <code>assumedtype</code> is specified, then all elements in <code>A</code> must have integer values within the range of <code>assumedtype</code>.</li></ul> <p><b>Data Types</b> double   int8   int16   int32   int64   uint8   uint16   uint32   uint64</p> <p><b>assumedtype - Assumed data type of A</b> 'uint64'   'uint32'   'uint16'   'uint8'   'int64'   'int32'   'int16'   'int8'</p> <p>Assumed data type of <code>A</code>, specified as 'uint64', 'uint32', 'uint16', 'uint8', 'int64', 'int32', 'int16', or 'int8'.</p>

- 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 - Bitwise complement**

signed integer array | unsigned integer array | double array

Bitwise 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 =
```

# bitcmp

---

```
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
```

<b>Purpose</b>	Get bit at specified position
<b>Syntax</b>	<code>b = bitget(A,bit)</code> <code>b = bitget(A,bit,assumedtype)</code>
<b>Description</b>	<p><code>b = bitget(A,bit)</code> returns the bit value at position <code>bit</code> in integer array <code>A</code>.</p> <p><code>b = bitget(A,bit,assumedtype)</code> assumes that <code>A</code> is of <code>assumedtype</code>.</p>
<b>Input Arguments</b>	<p><b>A - Input value</b> signed integer array   unsigned integer array   double array</p> <p>Input value, specified as a signed integer array, unsigned integer array, or double array.</p> <ul style="list-style-type: none"><li>• If <code>A</code> is a double array, and <code>assumedtype</code> is not specified, then MATLAB treats <code>A</code> as an unsigned 64-bit integer.</li><li>• If <code>assumedtype</code> is specified, then all elements in <code>A</code> must have integer values within the range of <code>assumedtype</code>.</li></ul> <p><b>Data Types</b> double   int8   int16   int32   int64   uint8   uint16   uint32   uint64</p> <p><b>bit - Bit position</b> integer   integer array</p> <p>Bit position, specified as an integer or integer array. <code>bit</code> must be between 1 (the least-significant bit) and the number of bits in the integer class of <code>A</code>.</p> <p><b>Data Types</b> double   int8   int16   int32   int64   uint8   uint16   uint32   uint64</p> <p><b>assumedtype - Assumed data type of A</b> 'uint64'   'uint32'   'uint16'   'uint8'   'int64'   'int32'   'int16'   'int8'</p>

# bitget

---

Assumed data type of, 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
```

# bitmax

---

**Purpose** Maximum double-precision floating-point integer

**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 =
    433fffffffffffffff
```

```
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  $1.111\dots111 \times 2^{52}$  with 52 trailing 1's, or  $2^{53} - 1$ .

**See Also**

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitshift` | `bitxor`

# bitnot

---

<b>Purpose</b>	Bit-wise NOT
<b>Syntax</b>	<code>C = bitnot(A)</code>
<b>Description</b>	<code>C = bitnot(A)</code> reverses all bits.
<b>Input Arguments</b>	<b>A</b> .NET enumeration object.
<b>Output Arguments</b>	<b>C</b> Result of operation, same type as input.
<b>See Also</b>	<code>bitand</code>   <code>bitor</code>   <code>bitxor</code>

**Purpose**

Bitwise OR

**Syntax**

```
intout = bitor(integ1,integ2)
intout = bitor(integ1,integ2,assumedtype)

objout = bitor(netobj1,netobj2)
```

**Description**

`intout = bitor(integ1,integ2)` returns the bitwise OR of `integ1` and `integ2`.

`intout = bitor(integ1,integ2,assumedtype)` assumes that `integ1` and `integ2` are of `assumedtype`.

`objout = bitor(netobj1,netobj2)` returns the bitwise 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` - Bitwise OR result

signed integer array | unsigned integer array | double array

Bitwise 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` - Bitwise OR result

`.NET` enumeration object

Bitwise 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 bitwise input is 1.

### Negative Values

Explore how bitor handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bitwise 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”

# bitset

---

## 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](#)

# bitshift

---

**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

# bitshift

---

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
```

# bitxor

---

## Purpose

Bitwise XOR

## Syntax

```
intout = bitxor(integ1,integ2)
intout = bitxor(integ1,integ2,assumedtype)

objout = bitxor(netobj1,netobj2)
```

## Description

`intout = bitxor(integ1,integ2)` returns the bitwise XOR of `integ1` and `integ2`.

`intout = bitxor(integ1,integ2,assumedtype)` assumes that `integ1` and `integ2` are of `assumedtype`.

`objout = bitxor(netobj1,netobj2)` returns the bitwise 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` - Bitwise XOR result

signed integer array | unsigned integer array | double array

Bitwise 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` - Bitwise XOR result

.NET enumeration object

Bitwise 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]);
```

# bitxor

---

```
B = uint8([0 0; 1 1]);  
TTable = bitxor(A, B)
```

```
TTable =
```

```
    0    1  
    1    0
```

bitxor returns 0 if both bitwise inputs are equal.

## Negative Values

Explore how bitxor handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bitwise 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”



**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`

# blkdiag

---

**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

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & \dots \end{bmatrix}$$

The input matrices do not have to be square, nor do they have to be of equal size.

**See Also** `diag` | `horzcat` | `vertcat`

**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`

# break

---

<b>Purpose</b>	Terminate execution of for or while loop
<b>Syntax</b>	break
<b>Description</b>	<p>break terminates the execution of a for or while loop. Statements in the loop that appear after the break statement are not executed.</p> <p>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.</p>
<b>Tips</b>	break is not defined outside a for or while loop. Use return in this context instead.
<b>Examples</b>	<p>The example below shows a while loop that reads the contents of the file <code>fft.m</code> 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 <code>fft</code> program.</p> <pre>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);</pre>
<b>See Also</b>	for   while   end   continue   return

**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  $0 < \text{beta} < 1$  and darker if  $-1 < \text{beta} < 0$ . `brighten(beta)`, followed by `brighten(-beta)`, where  $\text{beta} < 1$ , 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.

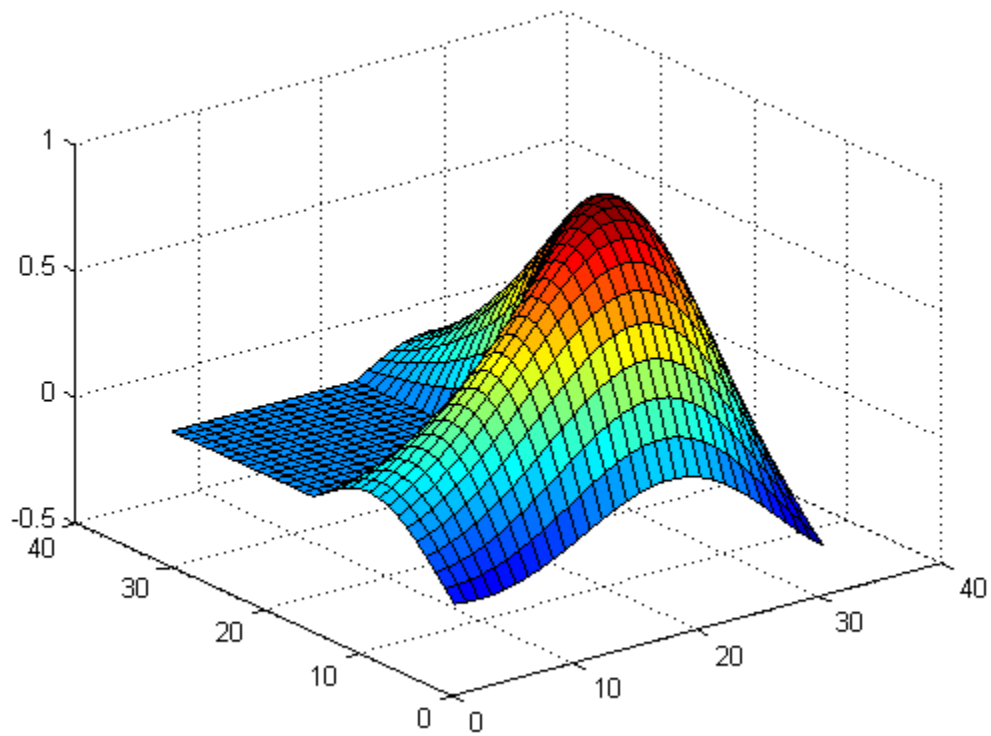
`newmap = brighten(cmap,beta)` returns a brighter or darker version of the colormap `cmap` without changing the display.

**Examples** Brighten the current colormap:

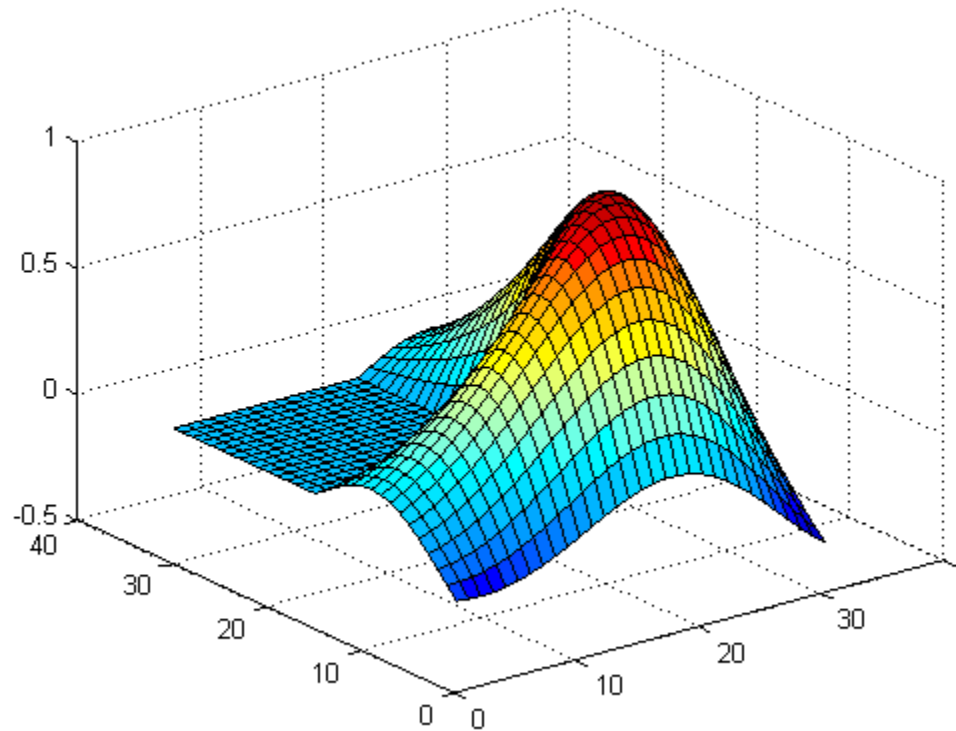
```
surf(membrane);
```

# brighten

---



```
beta = .5;  
brighten(beta);
```



## Algorithms

`brighten` raises values in the colormap to the power of gamma, where gamma is

$$\gamma = \begin{cases} 1 - \beta, & \beta > 0 \\ \frac{1}{1 + \beta}, & \beta \leq 0 \end{cases}$$

`brighten` has no effect on graphics objects defined with true color.

# brighten

---

## See Also

`colormap` | `rgbplot`

## How To

- “Altering Colormaps”



---

<b>Purpose</b>	Interactively mark, delete, modify, and save observations in graphs
<b>Syntax</b>	<pre>brush on brush off brush brush color brush(<i>figure_handle</i>,...) brushobj = brush(<i>figure_handle</i>)</pre>
<b>Description</b>	<p>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.</p> <p><code>brush on</code> turns on interactive data brushing mode.</p> <p><code>brush off</code> turns brushing mode off, leaving any brushed observations still highlighted.</p> <p><code>brush</code> by itself toggles the state of the data brushing tool.</p> <p><code>brush color</code> sets the current color used for brushing graphics to the specified <code>ColorSpec</code>. Changing brush color affects subsequent brushing, but does not change the color of observations already brushed or the brush tool's state.</p> <p><code>brush(<i>figure_handle</i>,...)</code> applies the function to the specified figure handle.</p> <p><code>brushobj = brush(<i>figure_handle</i>)</code> returns a <i>brush mode object</i> for that figure, useful for controlling and customizing the figure's brushing state. The following properties of such objects can be modified using <code>get</code> and <code>set</code>:</p>

Enable 'on'   {'off'}	Specifies whether this figure mode is currently enabled on the figure.
FigureHandle	The associated figure handle. This property supports <code>get</code> only.
Color	Specifies the color to be used for brushing.

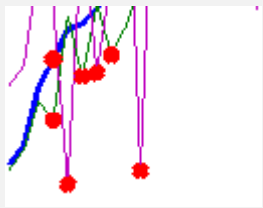
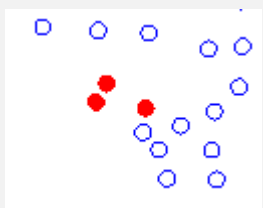
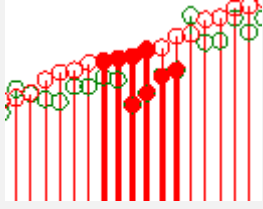
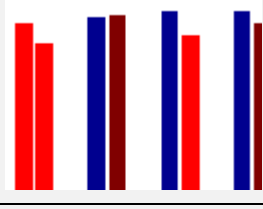
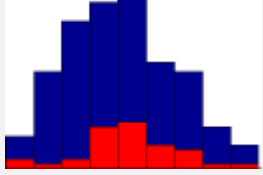
`brush` cannot return a brush mode object at the same time you are calling it to set a brushing option.

## Tips


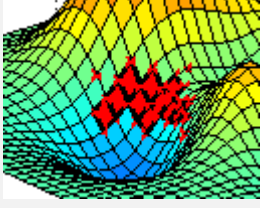
- “Types of Plots You Can Brush” on page 2-530
- “Plot Types You Cannot Brush” on page 2-532
- “Mode Exclusivity and Persistence” on page 2-533
- “How Data Linking Affects Data Brushing” on page 2-534
- “Mouse Gestures for Data Brushing” on page 2-535
- “Brush Mode Callbacks” on page 2-536

## 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	

# brush

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 plot 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 2-534 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`, `surf1`, 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 2-530 and “Plot Types You Cannot Brush” on page 2-532)
- 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 <b>Shift</b> 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 <b>Shift</b> or <b>Ctrl</b> 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)
```

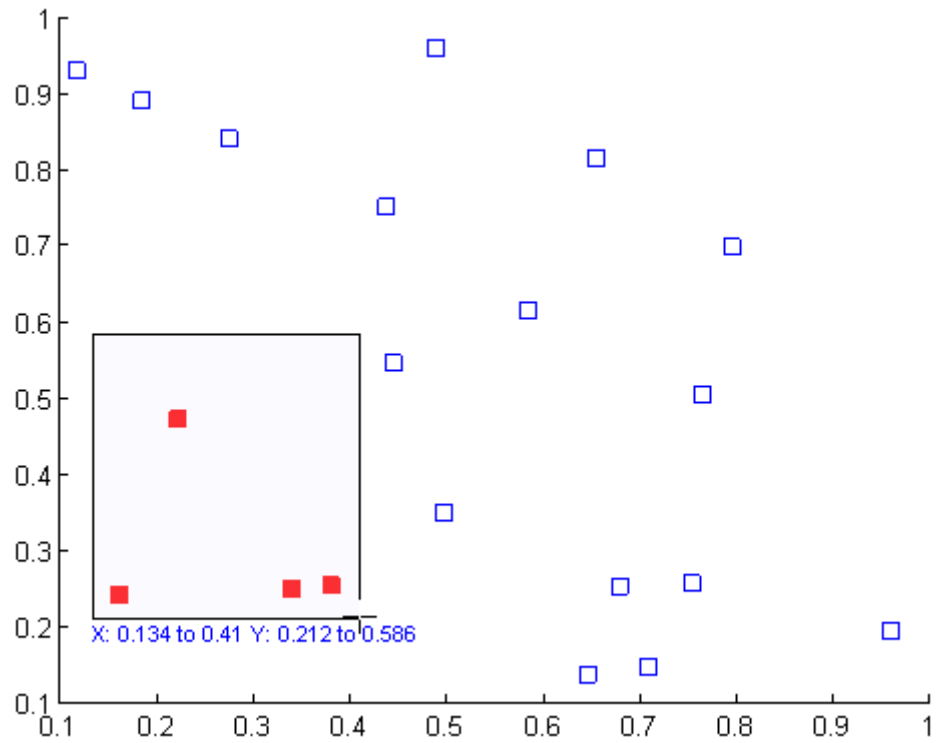
## Examples

### 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
```

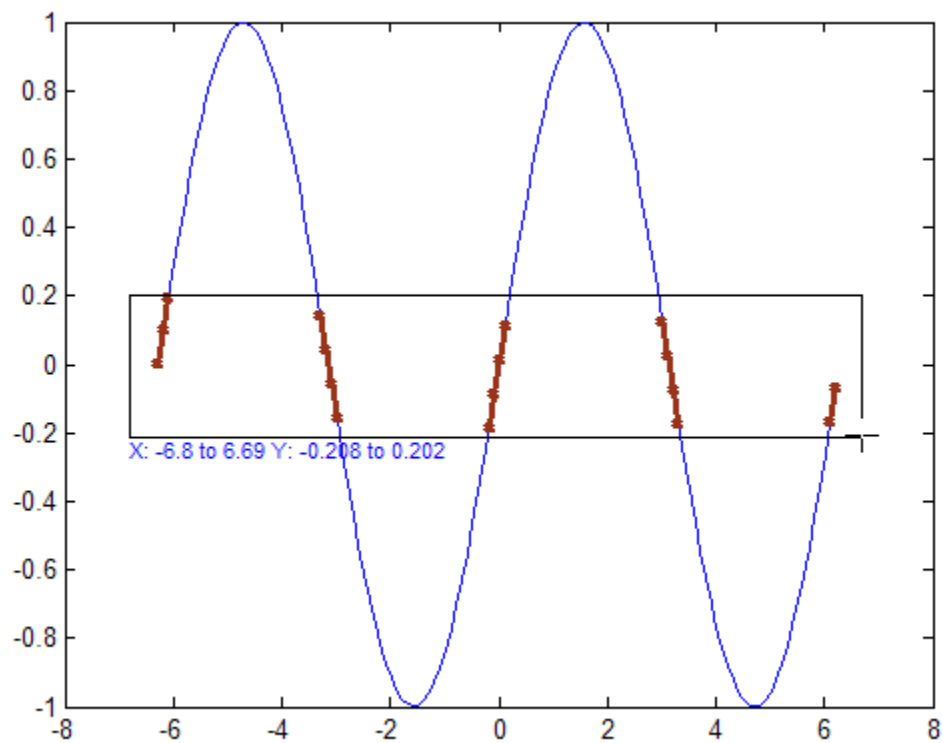
# brush



## 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”

# bsxfun

---

**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
```

# bsxfun

---

## See Also

[repmat](#) | [arrayfun](#)

<b>Purpose</b>	Build searchable documentation database
<b>Syntax</b>	<code>builddocsearchdb(folder)</code>
<b>Description</b>	<code>builddocsearchdb(folder)</code> builds a searchable database from HTML files in the specified folder. The <code>builddocsearchdb</code> function creates a subfolder named <code>helpsearch</code> 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.
<b>Input Arguments</b>	<p><b>folder</b> - Full path to folder with HTML files</p> <p>string</p> <p>Full path to a folder with HTML files, specified as a string.</p> <p>To include a particular HTML document in the search database, the <code>builddocsearchdb</code> function requires that:</p> <ul style="list-style-type: none"><li>• The document has a title.</li><li>• The content is different from the title.</li></ul> <p><b>Example:</b> <code>builddocsearchdb('c:\myfiles\html')</code></p>
<b>Examples</b>	<p><b>Search Custom Help Files</b></p> <p>Build a search database for custom help files.</p> <p>MATLAB includes a set of sample files to demonstrate how to create a custom toolbox and supporting documentation. This sample toolbox is called the <i>Upslope Area Toolbox</i>. The <code>upslope</code> folder includes a file named <code>info.xml</code>, which is required to display custom documentation, and a subfolder named <code>html</code>, which contains HTML documentation and supporting files.</p> <p>Copy the sample files to a temporary folder, and add the copied files to the path.</p> <pre>sample = fullfile(...</pre>

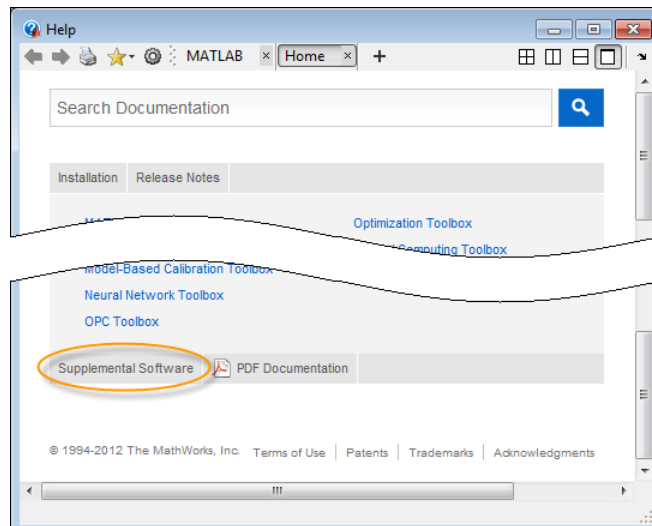
# builddocsearchdb

```
matlabroot,'help','techdoc','matlab_env','examples','upslope');  
tmp = tempname;  
mkdir(tmp);  
copyfile(sample,tmp);  
addpath(tmp);
```

Create a search database.

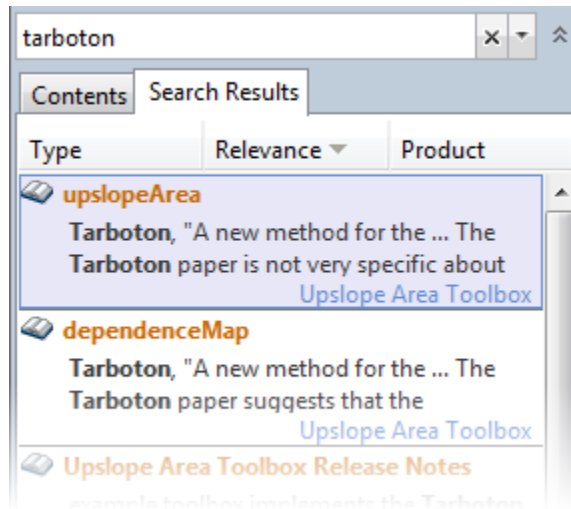
```
folder = fullfile(tmp,'html');  
builddocsearchdb(folder)
```

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”

# builtin

---

**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
end
```

Create an instance MyParticle.

```
p = MyParticle(1,2,4)
```

```
p =
```

```
MyParticle
```

```
Properties:
```

```
    velocity: [1x1 struct]
```

```
Methods
```

```
Velocity
```

```
x: 1
```

```
y: 2
```

```
z: 4
```

# builtin

---

## 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	<p>A function handle that evaluates the differential equations <math>f(x,y)</math>. It can have the form</p> <pre>dydx = odefun(x,y) dydx = odefun(x,y,parameters)</pre> <p>where <math>x</math> is a scalar corresponding to <math>x</math>, and <math>y</math> is a column vector corresponding to <math>y</math>. <code>parameters</code> is a vector of unknown parameters. The output <code>dydx</code> is a column vector.</p>
bcfun	<p>A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form <math>bc(y(a),y(b))</math>, <code>bcfun</code> can have the form</p> <pre>res = bcfun(ya,yb) res = bcfun(ya,yb,parameters)</pre> <p>where <code>ya</code> and <code>yb</code> are column vectors corresponding to <math>y(a)</math> and <math>y(b)</math>. <code>parameters</code> is a vector of unknown parameters. The output <code>res</code> is a column vector.</p> <p>See “Multipoint Boundary Value Problems” on page 2-552 for a description of <code>bcfun</code> for multipoint boundary value problems.</p>
solinit	<p>A structure containing the initial guess for a solution. You create <code>solinit</code> using the function <code>bvpinit</code>. <code>solinit</code> has the following fields.</p>

	x	Ordered nodes of the initial mesh. Boundary conditions are imposed at $a = \text{solinit.x}(1)$ and $b = \text{solinit.x}(\text{end})$ .
	y	Initial guess for the solution such that $\text{solinit.y}(:,i)$ is a guess for the solution at the node $\text{solinit.x}(i)$ .
	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 2-552. 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:

<code>sol.x</code>	Mesh selected by <code>bvp4c</code>
<code>sol.y</code>	Approximation to $y(x)$ at the mesh points of <code>sol.x</code>
<code>sol.yp</code>	Approximation to $y'(x)$ at the mesh points of <code>sol.x</code>
<code>sol.parameters</code>	Values returned by <code>bvp4c</code> for the unknown parameters, if any
<code>sol.solver</code>	' <code>bvp4c</code> '
<code>sol.stats</code>	Computational cost statistics (also displayed when the <code>stats</code> option is set with <code>bvpset</code> ).

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  $a = a_0 < a_1 < a_2 < \dots < a_n = b$  are boundary points in the interval  $[a,b]$ . The points  $a_1, a_2, \dots, a_{n-1}$  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

$$yp = \text{odefun}(x,y,k)$$

In the boundary conditions function

$$\text{bcfun}(y_{\text{left}}, y_{\text{right}})$$



`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 `bvp4c`, `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

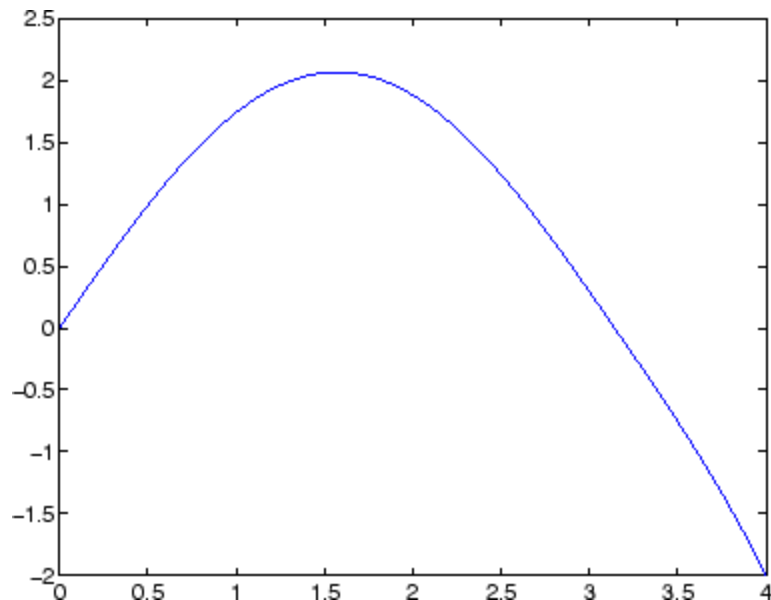
```
solinit = bvpinit(linspace(0,4,5),[1 0]);
```

Now solve the problem with

```
sol = bvp4c(@twoode,@twobc,solinit);
```

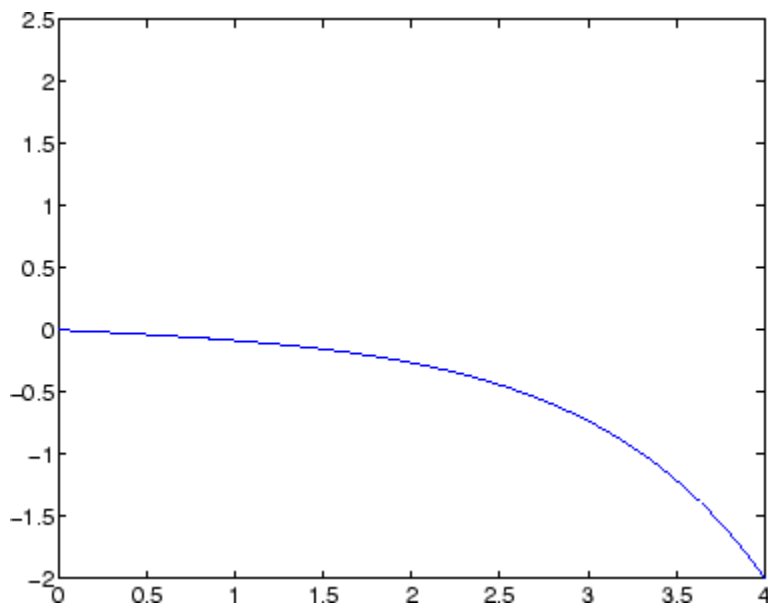
Evaluate the numerical solution at 100 equally spaced points and plot  $y(x)$  with

```
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  $\lambda$  of Mathieu's equation

$$y'' + (\lambda - 2q \cos 2x)y = 0.$$

Because the unknown parameter  $\lambda$  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)

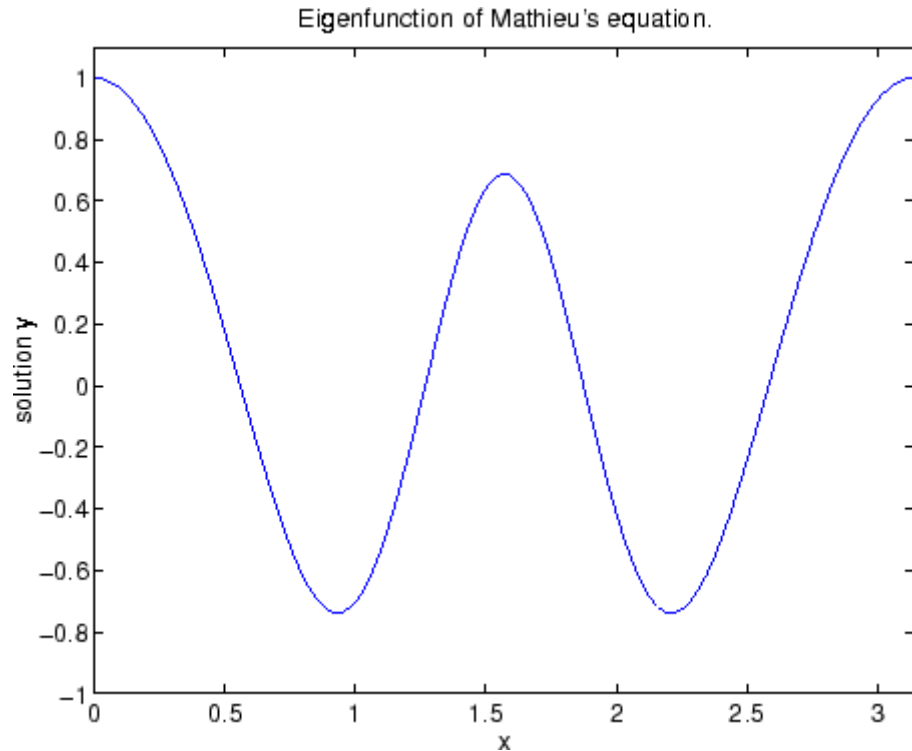
xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
% -----
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [ y(2)
        -(lambda - 2*q*cos(2*x))*y(1) ];
% -----
function res = mat4bc(ya,yb,lambda)
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  $\lambda$ .

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](http://www.mathworks.com/bvp_tutorial)

**See Also**

@ | bvp5c | bvpget | bvpinit | bvpset | bvpxtend | deval

# bvp5c

**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  $dydx = \text{odefun}(x,y)$ $dydx = \text{odefun}(x,y,\text{parameters})$  where $x$ is a scalar corresponding to $x$ , and $y$ is a column vector corresponding to $y$ . $\text{parameters}$ 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 $bc(y(a),y(b))$ , $\text{bcfun}$ can have the form  $\text{res} = \text{bcfun}(ya,yb)$ $\text{res} = \text{bcfun}(ya,yb,\text{parameters})$  where $ya$ and $yb$ are column vectors corresponding to $y(a)$ and $y(b)$ . $\text{parameters}$ is a vector of unknown parameters. The output $\text{res}$ is a column vector.	
solinit	A structure containing the initial guess for a solution. You create $\text{solinit}$ using the function $\text{bvpinit}$ . $\text{solinit}$ has the following fields.	
	$x$	Ordered nodes of the initial mesh. Boundary conditions are imposed at $a = \text{solinit}.x(1)$ and $b = \text{solinit}.x(\text{end})$ .



	<code>y</code>	Initial guess for the solution such that <code>solinit.y(:,i)</code> is a guess for the solution at the node <code>solinit.x(i)</code> .
	<code>parameters</code>	Optional. A vector that provides an initial guess for unknown parameters.
		The structure can have any name, but the fields must be named <code>x</code> , <code>y</code> , and <code>parameters</code> . You can form <code>solinit</code> with the helper function <code>bvpinit</code> . See <code>bvpinit</code> for details.
<code>options</code>		Optional integration argument. A structure you create using the <code>bvpset</code> function. See <code>bvpset</code> 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:

<code>sol.x</code>	Mesh selected by <code>bvp5c</code>
<code>sol.y</code>	Approximation to $y(x)$ at the mesh points of <code>sol.x</code>
<code>sol.parameters</code>	Values returned by <code>bvp5c</code> for the unknown parameters, if any
<code>sol.solver</code>	' <code>bvp5c</code> '
<code>sol.stats</code>	Computational cost statistics (also displayed when the <code>stats</code> option is set with <code>bvpset</code> ).

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  $a = a_0 < a_1 < a_2 < \dots < a_n = b$  are boundary points in the interval  $[a,b]$ . The points  $a_1, a_2, \dots, a_{n-1}$  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

$$yp = \text{odefun}(x,y,k)$$

In the boundary conditions function

$$\text{bcfun}(yleft,yright)$$

`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,

$$\text{yleft}(:, 1) = y(a)$$

and

$$\text{yright}(:, \text{end}) = y(b)$$

When you create an initial guess with

$$\text{solinit} = \text{bvpinit}(xinit,yinit),$$

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](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`

**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`

# bvpinit

---

**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=n linspace(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`.

<code>x</code>	Ordered nodes of the initial mesh.
<code>y</code>	Initial guess for the solution with <code>solinit.y(:,i)</code> a guess for the solution at the node <code>solinit.x(i)</code> .
<code>parameters</code>	Optional. A vector that provides an initial guess for unknown parameters.

# bvpinit

---

`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](#) | [bvpextend](#) | [deval](#)



**Purpose**

Create or alter options structure of boundary value problem

**Syntax**

```
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(olddopts,'name1',value1,...)
options = bvpset(olddopts,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(olddopts,'name1',value1,...)` alters an existing options structure `olddopts`. This overwrites any values in `olddopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = bvpset(olddopts,newopts)` combines an existing options structure `olddopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `olddopts`.

`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 2-570
- “Vectorization” on page 2-571
- “Analytical Partial Derivatives” on page 2-572
- “Singular BVPs” on page 2-573
- “Mesh Size Property” on page 2-574

- “Solution Statistic Property” on page 2-575

## 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.

$$\left| \left( res(i) / \max( abs(f(i)), AbsTol(i) / RelTol ) \right) \right| \leq RelTol$$

The following table describes the error tolerance properties.

### BVP Error Tolerance Properties

Property	Value	Description
<code>RelTol</code>	Positive scalar {1e-3}	<p>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 <math>f(x,y)</math>. The default, 1e-3, corresponds to 0.1% accuracy.</p> <p>The computed solution <math>S(x)</math> is the exact solution of <math>S'(x) = F(x, S(x)) + res(x)</math>. On each subinterval of the mesh, the residual <math>res(x)</math> satisfies</p> $\left  \left( res(i) / \max( abs(F(i)), AbsTol(i) / RelTol ) \right) \right  \leq RelTol$
<code>AbsTol</code>	Positive scalar or vector {1e-6}	<p>Absolute error tolerances that apply to the corresponding components of the residual vector. <code>AbsTol(i)</code> is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.</p>

## 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}	<p>Set on to inform <code>bvp4c</code> that you have coded the ODE function <code>F</code> so that <code>F([x1 x2 ...], [y1 y2 ...])</code> returns <code>[F(x1,y1) F(x2,y2) ...]</code>. 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.</p> <p>With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the <code>shockbvp</code> example shown previously, the <code>shockODE</code> function has been vectorized using colon notation into the subscripts and by using the array multiplication <code>(.*)</code> operator.</p> <pre>function dydx = shockODE(x,y,e)</pre>

## Vectorization Properties (Continued)

Property	Value	Description
		<pre> pix = pi*x; dydx = [ y(2,:)... -x/e.*y(2,)-pi^2*cos(pix)- pix/e.*sin(pix)]; </pre>

## 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 <code>@fjac</code> if <code>dfdy = fjac(x,y)</code> evaluates the Jacobian $\partial f/\partial y$ . If the problem involves unknown parameters $p$ , <code>[dfdy,dfdp] = fjac(x,y,p)</code> 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 <code>dfdy</code> or to a cell array <code>{dfdy,dfdp}</code> .
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 <code>@bcjac</code> if <code>[dbcodya,dbcodyb] = bcjac(ya,yb)</code> evaluates the partial derivatives $\partial bc/\partial ya$ , and $\partial bc/\partial yb$ . If the problem involves unknown parameters $p$ , <code>[dbcodya,dbcodyb,dbcdp] = bcjac(ya,yb,p)</code> must also return the partial derivative $\partial bc/\partial p$ . For problems with constant partial derivatives, set this property to a cell array <code>{dbcodya,dbcodyb}</code> or <code>{dbcodya,dbcodyb,dbcdp}</code> .

**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**

<b>Property</b>	<b>Value</b>	<b>Description</b>
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  $N_{Max}$ .

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: <ul style="list-style-type: none"> <li>• The number of points in the mesh</li> <li>• The maximum residual of the solution</li> <li>• The number of times it called the differential equation function odefun to evaluate <math>f(x,y)</math></li> </ul>

## BVP Solution Statistic Property (Continued)

Property	Value	Description
		<ul style="list-style-type: none"><li>The number of times it called the boundary condition function <code>bcfun</code> to evaluate <math>bc(y(a),y(b))</math></li></ul>

### 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`



**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`

# calendar

---

**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

---

<b>Purpose</b>	Call function in shared library
<b>Syntax</b>	<code>[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)</code>
<b>Description</b>	<code>[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)</code> calls function, <code>funcname</code> , in library, <code>libname</code> , passing input arguments, <code>arg1,...,argN</code> , and returns output values obtained from <code>funcname</code> in <code>x1,...,xN</code> .
<b>Input Arguments</b>	<p><b>libname - Name of shared library</b> string</p> <p>Name of shared library, specified as a string. If you call <code>loadlibrary</code> using the <code>alias</code> option, then you must use the alias name for the <code>libname</code> argument.</p> <p><b>funcname - Name of function in library</b> string</p> <p>Name of function in library, specified as a string.</p> <p><b>arg1,...,argN - Input arguments</b> any valid type</p> <p>Input arguments, 1 through N, required by <code>funcname</code> (if any), specified by any valid type.</p>
<b>Output Arguments</b>	<p><b>x1,...,xN - Output arguments</b> any valid type</p> <p>Output arguments, 1 through N, from <code>funcname</code> (if any), returned as any valid type.</p>
<b>Limitations</b>	<ul style="list-style-type: none"><li>• Use with libraries that are loaded using the <code>loadlibrary</code> function.</li></ul>
<b>Examples</b>	<p><b>Call <code>addStructByRef</code> Function</b></p> <p>Load the library.</p>

```
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”

# callSoapService

---

**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

[createClassFromWsd1](#) | [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”

# cameratoolbar

---

**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”

# camlight

---

**Purpose** Create or move light object in camera coordinates

**Syntax**

```
camlight('headlight')
camlight('right')
camlight('left')
camlight
camlight(az,e1)
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,e1)` creates a light at the specified azimuth (`az`) and elevation (`e1`) with respect to the camera position. The camera target is the center of rotation and `az` and `e1` 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”

# camlookat

---

**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”

# camorbit

---

## 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”

# campan

---

**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”

# campos

---

**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”

# camproj

---

**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.

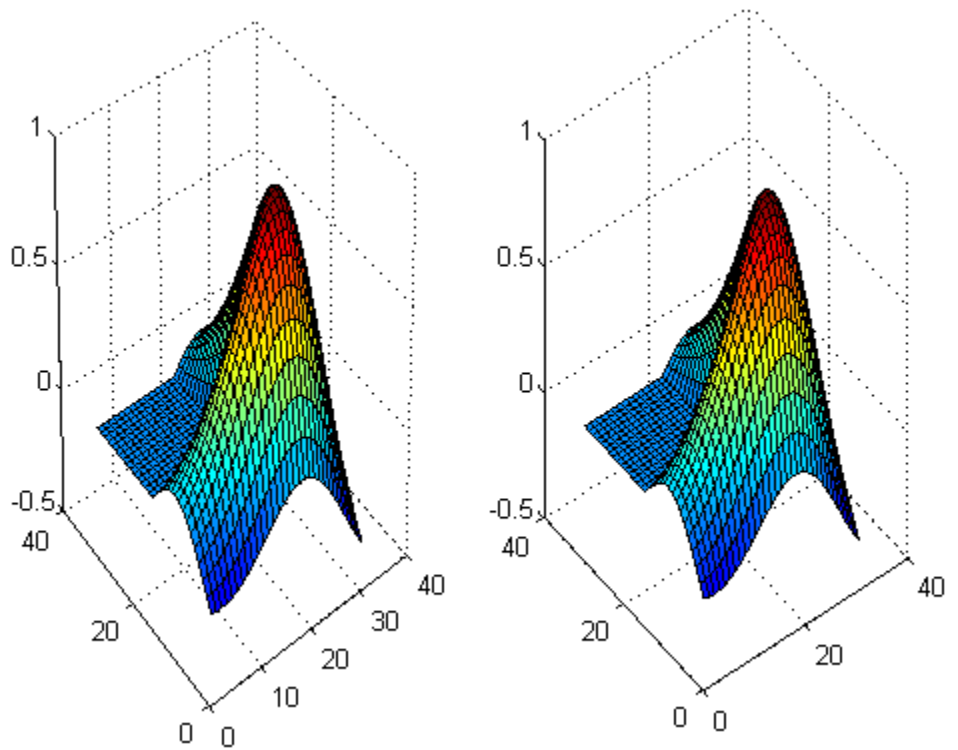
camproj('projection\_type') sets the projection type in the current axes to the specified value. Possible values for *projection\_type* are orthographic and perspective.

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](#) | [camtargt](#) | [camup](#) | [camva](#) | [Axes: CameraPosition](#) | [Axes: CameraTarget](#) | [Axes: CameraUpVector](#) | [Axes: CameraViewAngle](#) | [Axes: Projection](#)

## How To

- “Defining Scenes with Cameras”

# camroll

---

**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”



**Purpose** Set or query location of camera target

**Syntax**

```
camtarget
camtarget([camera_target])
camtarget('mode')
camtarget('auto')
camtarget('manual')
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
```

# camtarget

---

```
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”

**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`.

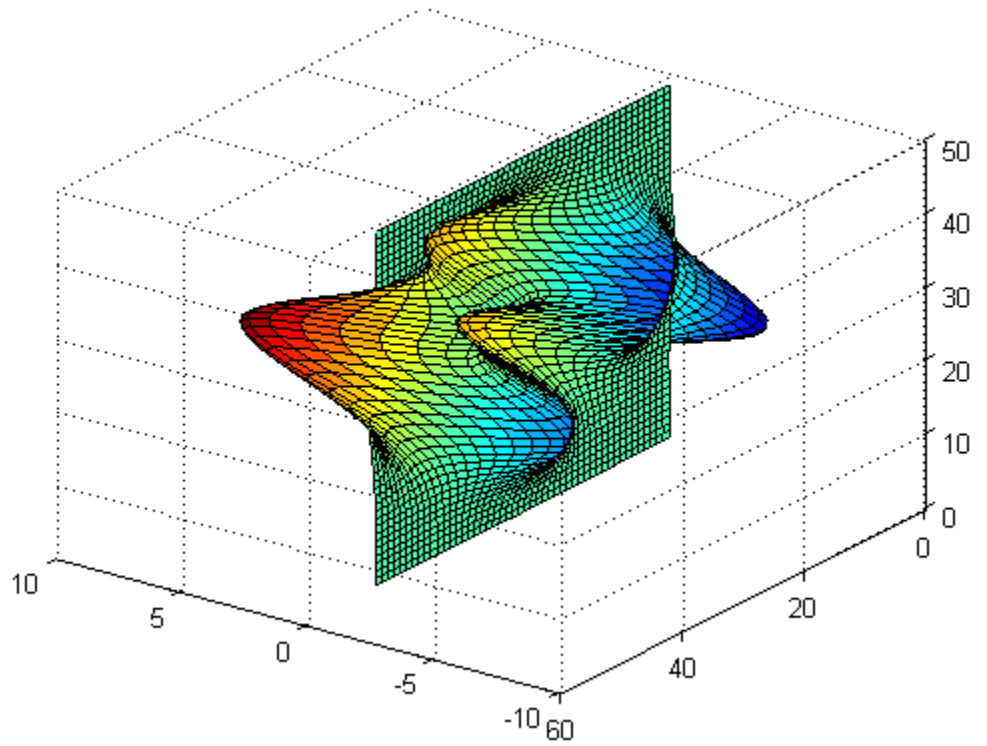
`camup('auto')` sets the camera up vector mode to `auto`. In `auto` mode,  $[0\ 1\ 0]$  is the up vector of for 2-D views. This means the  $y$ -axis points up. For 3-D views, the up vector is  $[0\ 0\ 1]$ , meaning the  $z$ -axis points up.

`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](#) | [camtargt](#) | [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”

<b>Purpose</b>	Zoom in and out on scene
<b>Syntax</b>	<code>camzoom(zoom_factor)</code> <code>camzoom(axes_handle,...)</code>
<b>Description</b>	<p><code>camzoom(zoom_factor)</code> zooms in or out on the scene depending on the value specified by <code>zoom_factor</code>. If <code>zoom_factor</code> is greater than 1, the scene appears larger; if <code>zoom_factor</code> is greater than zero and less than 1, the scene appears smaller.</p> <p><code>camzoom(axes_handle,...)</code> operates on the axes identified by the first argument, <code>axes_handle</code>. When you do not specify an axes handle, <code>camzoom</code> operates on the current axes.</p>
<b>Tips</b>	<p><code>camzoom</code> sets the axes <code>CameraViewAngle</code> property, which in turn causes the <code>CameraViewAngleMode</code> property to be set to <code>manual</code>. Note that setting the <code>CameraViewAngle</code> 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 <code>axes</code> function for more information on this behavior.</p>
<b>See Also</b>	<code>axes</code>   <code>camdolly</code>   <code>camorbit</code>   <code>campan</code>   <code>camroll</code>   <code>camva</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Defining Scenes with Cameras”</li></ul>

# TriRep.cartToBary

---

<b>Purpose</b>	Convert point coordinates from cartesian to barycentric						
<b>Syntax</b>	<code>B = cartToBary(TR, SI, XC)</code>						
<b>Description</b>	<code>B = cartToBary(TR, SI, XC)</code> returns the barycentric coordinates of each point in <code>XC</code> with respect to its associated simplex <code>SI</code> .						
<b>Input Arguments</b>	<table><tr><td><code>TR</code></td><td>Triangulation representation.</td></tr><tr><td><code>SI</code></td><td>Column vector of simplex indices that index into the triangulation matrix <code>TR.Triangulation</code>.</td></tr><tr><td><code>XC</code></td><td>Matrix that represents the Cartesian coordinates of the points to be converted. <code>XC</code> is of size <code>m-by-n</code>, where <code>m</code> is of <code>length(SI)</code>, the number of points to convert, and <code>n</code> is the dimension of the space where the triangulation resides.</td></tr></table>	<code>TR</code>	Triangulation representation.	<code>SI</code>	Column vector of simplex indices that index into the triangulation matrix <code>TR.Triangulation</code> .	<code>XC</code>	Matrix that represents the Cartesian coordinates of the points to be converted. <code>XC</code> is of size <code>m-by-n</code> , where <code>m</code> is of <code>length(SI)</code> , the number of points to convert, and <code>n</code> is the dimension of the space where the triangulation resides.
<code>TR</code>	Triangulation representation.						
<code>SI</code>	Column vector of simplex indices that index into the triangulation matrix <code>TR.Triangulation</code> .						
<code>XC</code>	Matrix that represents the Cartesian coordinates of the points to be converted. <code>XC</code> is of size <code>m-by-n</code> , where <code>m</code> is of <code>length(SI)</code> , the number of points to convert, and <code>n</code> is the dimension of the space where the triangulation resides.						
<b>Output Arguments</b>	<table><tr><td><code>B</code></td><td>Matrix of dimension <code>m-by-k</code> where <code>k</code> is the number of vertices per simplex.</td></tr></table>	<code>B</code>	Matrix of dimension <code>m-by-k</code> where <code>k</code> is the number of vertices per simplex.				
<code>B</code>	Matrix of dimension <code>m-by-k</code> where <code>k</code> is the number of vertices per simplex.						
<b>Definitions</b>	A simplex is a triangle/tetrahedron or higher dimensional equivalent.						
<b>Examples</b>	<p>Compute the Delaunay triangulation of a set of points.</p> <pre>x = [0 4 8 12 0 4 8 12]'; y = [0 0 0 0 8 8 8 8]'; dt = DelaunayTri(x,y)</pre> <p>Compute the barycentric coordinates of the incenters.</p> <pre>cc = incenters(dt); tri = dt(:,:);</pre> <p>Plot the original triangulation and reference points.</p>						



```
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.

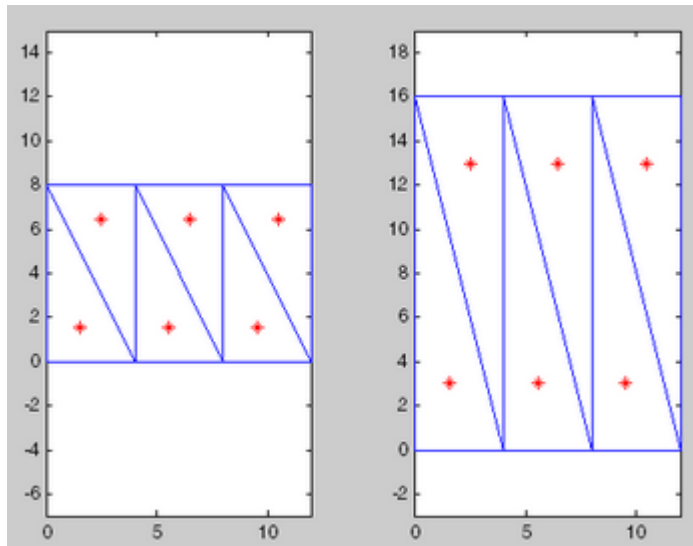
```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

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;
```

# TriRep.cartToBary

---



## See Also

[baryToCart](#) | [pointLocation](#)

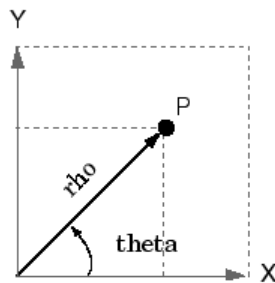
**Purpose** Transform Cartesian coordinates to polar or cylindrical

**Syntax** `[THETA,RHO,Z] = cart2pol(X,Y,Z)`  
`[THETA,RHO] = cart2pol(X,Y)`

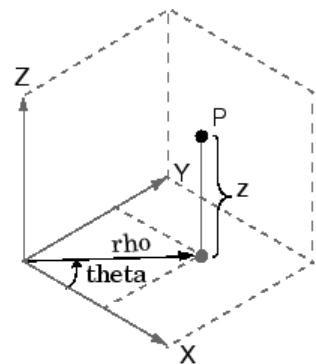
**Description** `[THETA,RHO,Z] = cart2pol(X,Y,Z)` transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays X, Y, and Z, into cylindrical coordinates. THETA is a counterclockwise angular displacement in radians from the positive x-axis, RHO is the distance from the 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**  
 $\text{theta} = \text{atan2}(y,x)$   
 $\text{rho} = \sqrt{x.^2 + y.^2}$



**Three-Dimensional Mapping**  
 $\text{theta} = \text{atan2}(y,x)$   
 $\text{rho} = \sqrt{x.^2 + y.^2}$   
 $Z = Z$

# cart2pol

---

## **See Also**

[cart2sph](#) | [pol2cart](#) | [sph2cart](#)

**Purpose** Transform Cartesian coordinates to spherical

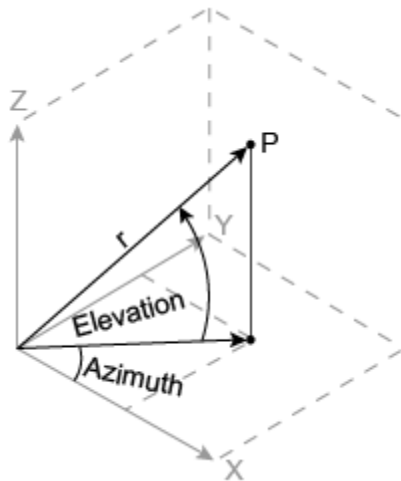
**Syntax** `[azimuth,elevation,r] = cart2sph(X,Y,Z)`

**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

## cart2sph

---

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

**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.

## case

---

### 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.');
```

### See Also

`switch` | `end` | `if` | `for` | `while`



---

<b>Purpose</b>	Cast variable to different data type
<b>Syntax</b>	$B = \text{cast}(A, \text{newclass})$
<b>Description</b>	$B = \text{cast}(A, \text{newclass})$ converts $A$ to class $\text{newclass}$ , where $\text{newclass}$ 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 $\text{newclass}$ .
<b>Examples</b>	<p>Convert an <code>int8</code> value to <code>uint8</code>:</p> <pre>a = int8(5); b = cast(a, 'uint8'); class(b)</pre> <p>MATLAB returns:</p> <pre>ans = uint8</pre>
<b>See Also</b>	<code>class</code>   <code>typecast</code>

# cat

---

**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)**

1	2
3	4

5	6
7	8

**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](#) | [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 occurred: 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
```

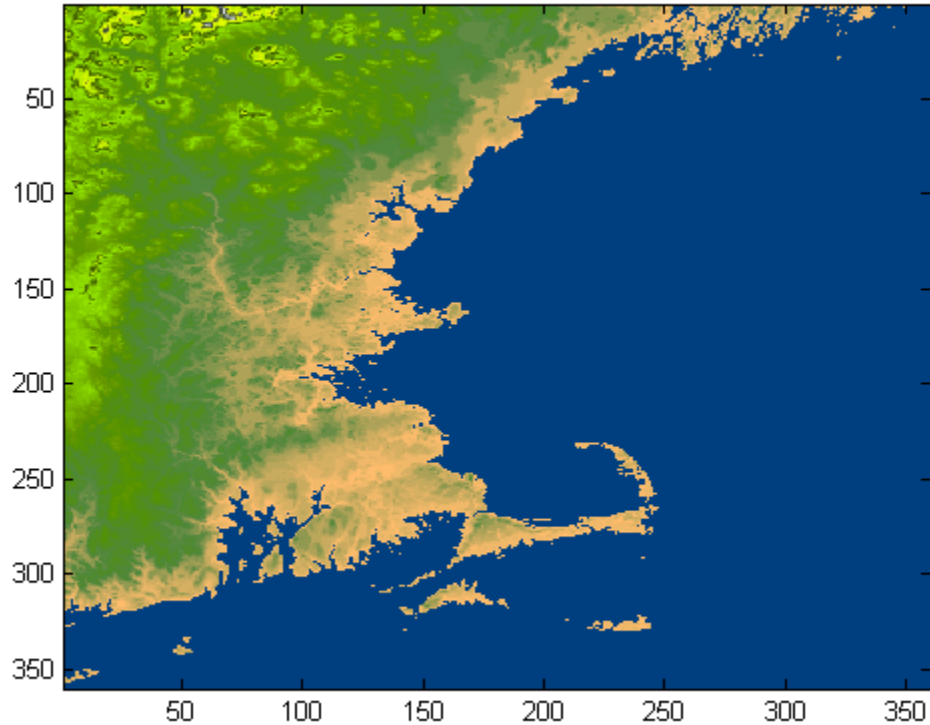
```
[-1 1]
```

## 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
```

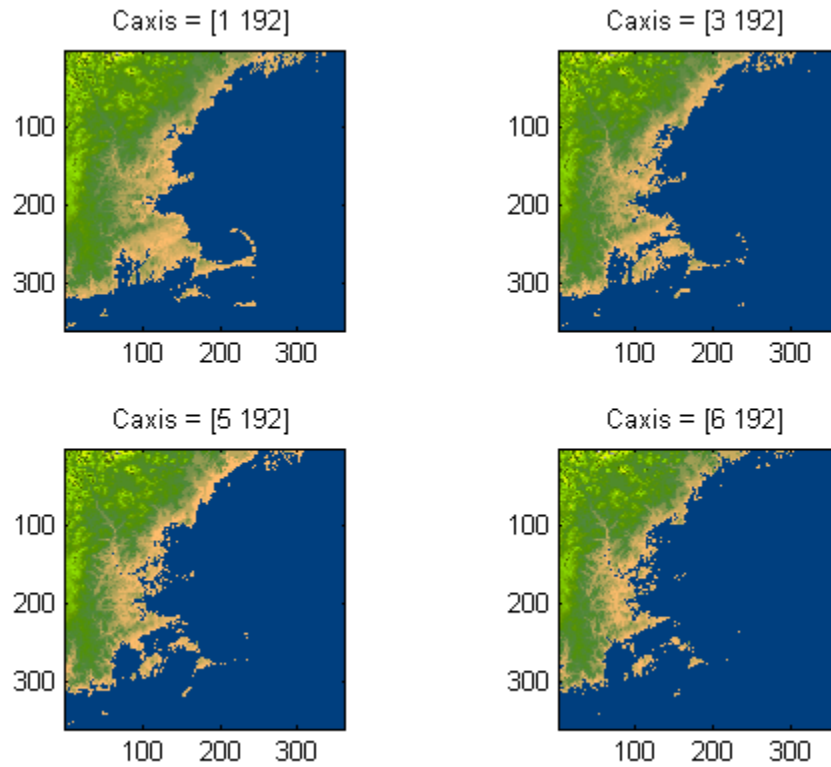




The blue color of the ocean is the first color in the colormap and is mapped to the lowest data value (1). You can effectively move sea level by changing the lower color limit value. For example, The following code compares four settings:

```
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 location  
% 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

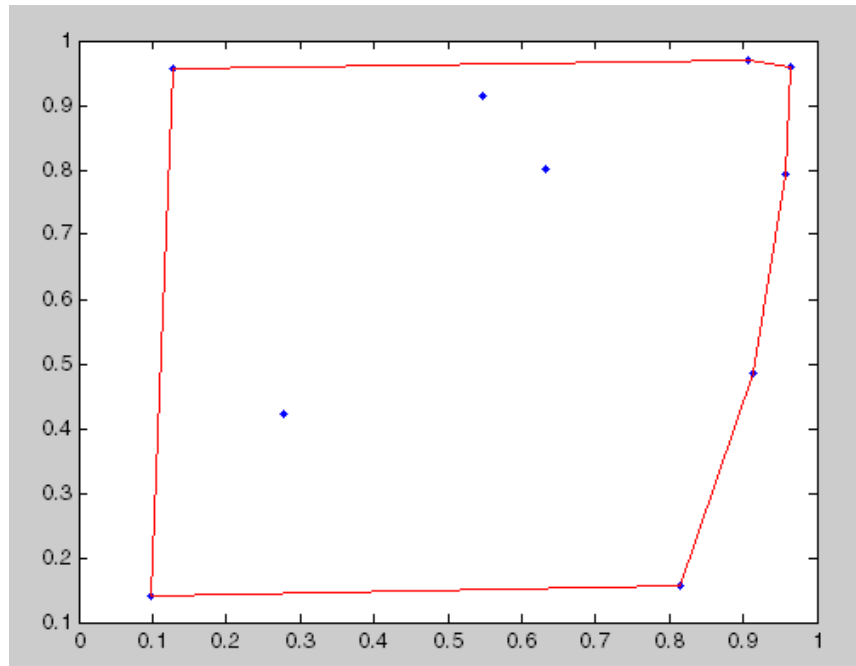
- “Path Names in MATLAB”
- “Making Files and Folders Accessible to MATLAB”

<b>Purpose</b>	Convex hull	
<b>Syntax</b>	<code>K = convexHull(DT)</code> <code>[K AV] = convexHull(DT)</code>	
<b>Description</b>	<code>K = convexHull(DT)</code> returns the indices into the array of points <code>DT.X</code> that correspond to the vertices of the convex hull. <code>[K AV] = convexHull(DT)</code> returns the convex hull and the area or volume bounded by the convex hull.	
<b>Input Arguments</b>	<code>DT</code>	Delaunay triangulation.
<b>Output Arguments</b>	<code>K</code>	If the points lie in 2-D space, <code>K</code> is a column vector of length <code>numf</code> . Otherwise <code>K</code> is a matrix of size <code>numf</code> -by- <code>ndim</code> , <code>numf</code> being the number of facets in the convex hull, and <code>ndim</code> the dimension of the space where the points reside.
	<code>AV</code>	The area or volume of the convex hull.
<b>Definitions</b>	The convex hull of a set of points <code>X</code> is the smallest convex region containing all of the points of <code>X</code> .	
<b>Examples</b>	<b>Example 1</b> Compute the convex hull of a set of random points located within a unit square in 2-D space. <pre>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;</pre>	

# DelaunayTri.convexHull

---

```
plot(dt.X(k,1),dt.X(k,2), 'r'); hold off;
```

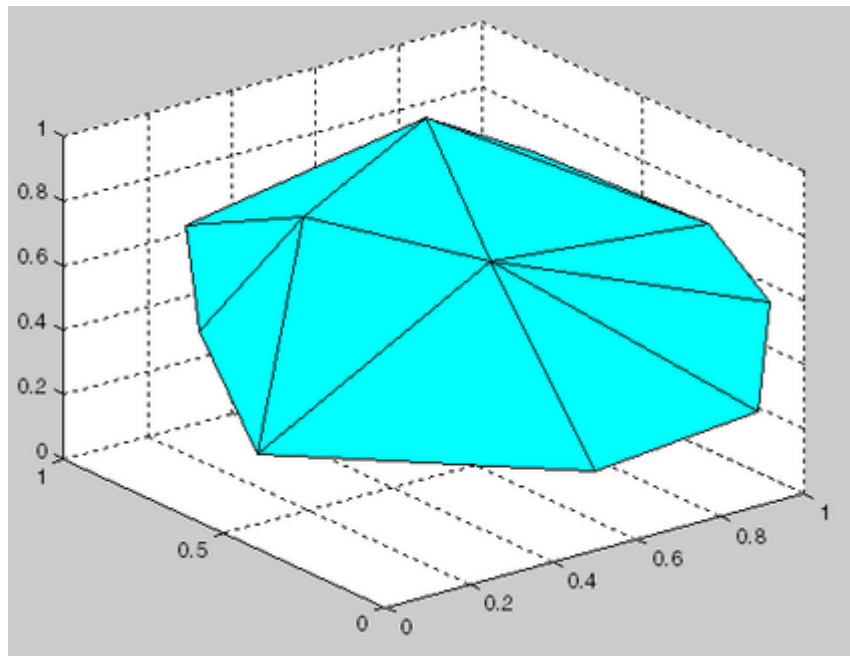


## Example 2

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**

`DelaunayTri.voronoiDiagram` | `TriRep` | `convhull` | `convhulln`

# FTP.cd

---

**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  $[V,D] = \text{eig}(X)$  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

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & -5 & 4 \end{bmatrix}$$

has a pair of complex eigenvalues.

$$[V,D] = \text{eig}(X)$$

$V =$

$$\begin{bmatrix} 1.0000 & -0.0191 - 0.4002i & -0.0191 + 0.4002i \\ 0 & 0 - 0.6479i & 0 + 0.6479i \\ 0 & 0.6479 & 0.6479 \end{bmatrix}$$

$D =$

$$\begin{bmatrix} 1.0000 & & \\ & 0 & \\ & & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4.0000 + 5.0000i & 0 \\ 0 & 0 & 4.0000 - 5.0000i \end{bmatrix}$$

Converting this to real block diagonal form produces

$$[V,D] = \text{cdf2rdf}(V,D)$$

V =

$$\begin{bmatrix} 1.0000 & -0.0191 & -0.4002 \\ 0 & 0 & -0.6479 \\ 0 & 0.6479 & 0 \end{bmatrix}$$

D =

$$\begin{bmatrix} 1.0000 & 0 & 0 \\ 0 & 4.0000 & 5.0000 \\ 0 & -5.0000 & 4.0000 \end{bmatrix}$$

## Algorithms

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

## See Also

[eig](#) | [rsf2csf](#)

---

<b>Purpose</b>	Convert MATLAB formatted dates to CDF formatted dates
<b>Syntax</b>	<code>E = cdfepoch(date)</code>
<b>Description</b>	<p><code>E = cdfepoch(date)</code> converts the date, specified by <code>date</code>, into a <code>cdfepoch</code> object. <code>date</code> must be a valid date string, returned by <code>datestr</code>, or a serial date number, returned by <code>datenum</code>. <code>date</code> can also be a <code>cdfepoch</code> object.</p> <p>When writing data to a CDF file using <code>cdfwrite</code>, use <code>cdfepoch</code> to convert MATLAB formatted dates to CDF formatted dates. The MATLAB <code>cdfepoch</code> object simulates the <code>CDFEPOCH</code> data type in CDF files.</p> <p>To convert a <code>cdfepoch</code> object into a MATLAB serial date number, use the <code>todatenum</code> function.</p>
<b>Definitions</b>	<p>The MATLAB serial date number calculates dates differently than CDF epochs.</p> <p>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.</p> <p>A CDF epoch is the number of milliseconds since 1-Jan-0000.</p>
<b>Examples</b>	<p>Convert the current time in serial date number format into a CDF epoch object.</p> <pre>% NOW function returns current time as serial date number dateobj = cdfepoch(now)  dateobj =      cdfepoch object:     11-Mar-2009 15:09:25</pre>

---

# cdfepoch

---

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

[cdfwrite](#) | [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

Field	Description	
Variables	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:	
	Column 1	Text string specifying name of variable
	Column 2	Double array specifying the dimensions of the variable, as returned by the <code>size</code> function
	Column 3	Double scalar specifying the number of records assigned for the variable
	Column 4	Text string specifying the data type of the variable, as stored in the CDF file
	Column 5	<p>Text string specifying the record and dimension variance settings for the variable. The single T or F to the left of the slash designates whether values vary by record. The zero or more T or F letters to the right of the slash designate whether values vary at each dimension. Here are some examples.</p> <p>T/ (scalar variable)  F/T (one-dimensional variable)</p> <p>T/TFF (three-dimensional variable)</p>



---

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'
```

# cdfinfo

---

```
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 several functions that provide high-level access to CDF files:

- `cdfinfo`
- `cdfread`
- `cdfwrite`

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

<code>cdflib</code>	Summary of Common Data Format (CDF) capabilities
<code>cdflib.getConstantNames</code>	Names of Common Data Format (CDF) library constants
<code>cdflib.getConstantValue</code>	Numeric value corresponding to Common Data Format (CDF) library constant
<code>cdflib.getFileBackward</code>	Return current backward compatibility mode setting
<code>cdflib.getLibraryCopyright</code>	Copyright notice of Common Data Format (CDF) library
<code>cdflib.getLibraryVersion</code>	Library version and release information
<code>cdflib.getValidate</code>	Library validation mode
<code>cdflib.setFileBackward</code>	Set backward compatibility mode
<code>cdflib.setValidate</code>	Specify library validation mode

## File Operations

<code>cdflib.close</code>	Close Common Data Format (CDF) file
<code>cdflib.create</code>	Create Common Data Format (CDF) file
<code>cdflib.delete</code>	Delete existing Common Data Format (CDF) file

---

<code>cdflib.getCacheSize</code>	Number of cache buffers used
<code>cdflib.getChecksum</code>	Checksum mode
<code>cdflib.getCompression</code>	Compression settings
<code>cdflib.getCompressionCacheSize</code>	Number of compression cache buffers
<code>cdflib.getCopyright</code>	Copyright notice in Common Data Format (CDF) file
<code>cdflib.getFormat</code>	Format of Common Data Format (CDF) file
<code>cdflib.getMajority</code>	Majority of variables
<code>cdflib.getName</code>	Name of Common Data Format (CDF) file
<code>cdflib.getReadOnlyMode</code>	Read-only mode
<code>cdflib.getStageCacheSize</code>	Number of cache buffers for staging
<code>cdflib.getVersion</code>	Common Data Format (CDF) library version and release information
<code>cdflib.inquire</code>	Basic characteristics of Common Data Format (CDF) file
<code>cdflib.open</code>	Open existing Common Data Format (CDF) file
<code>cdflib.setCacheSize</code>	Specify number of dotCDF cache buffers
<code>cdflib.setChecksum</code>	Specify checksum mode
<code>cdflib.setCompression</code>	Specify compression settings
<code>cdflib.setCompressionCacheSize</code>	Specify number of compression cache buffers

<code>cdflib.setFormat</code>	Specify format of Common Data Format (CDF) file
<code>cdflib.setMajority</code>	Specify majority of variables
<code>cdflib.setReadOnlyMode</code>	Specify read-only mode
<code>cdflib.setStageCacheSize</code>	Specify number of staging cache buffers for Common Data Format (CDF) file

## **Variables**

<code>cdflib.closeVar</code>	Close specified variable from multifile format Common Data Format (CDF) file
<code>cdflib.createVar</code>	Create new variable
<code>cdflib.deleteVar</code>	Delete variable
<code>cdflib.deleteVarRecords</code>	Delete range of records from variable
<code>cdflib.getVarAllocRecords</code>	Number of records allocated for variable
<code>cdflib.getVarBlockingFactor</code>	Blocking factor for variable
<code>cdflib.getVarCacheSize</code>	Number of multifile cache buffers
<code>cdflib.getVarCompression</code>	Information about compression used by variable
<code>cdflib.getVarData</code>	Single value from record in variable
<code>cdflib.getVarMaxAllocRecNum</code>	Maximum allocated record number for variable
<code>cdflib.getVarMaxWrittenRecNum</code>	Maximum written record number for variable
<code>cdflib.getVarName</code>	Variable name, given variable number

---

<code>cdflib.getVarNum</code>	Variable number, given variable name
<code>cdflib.getVarNumRecsWritten</code>	Number of records written to variable
<code>cdflib.getVarPadValue</code>	Pad value for variable
<code>cdflib.getVarRecordData</code>	Entire record for variable
<code>cdflib.getVarReservePercent</code>	Compression reserve percentage for variable
<code>cdflib.getVarsMaxWrittenRecNum</code>	Maximum written record number for CDF file
<code>cdflib.getVarSparseRecords</code>	Information about how variable handles sparse records
<code>cdflib.hyperGetVarData</code>	Read hyperslab of data from variable
<code>cdflib.hyperPutVarData</code>	Write hyperslab of data to variable
<code>cdflib.inquireVar</code>	Information about variable
<code>cdflib.putVarData</code>	Write single value to variable
<code>cdflib.putVarRecordData</code>	Write entire record to variable
<code>cdflib.renameVar</code>	Rename existing variable
<code>cdflib.setVarAllocBlockRecords</code>	Specify range of records to be allocated for variable
<code>cdflib.setVarBlockingFactor</code>	Specify blocking factor for variable
<code>cdflib.setVarCacheSize</code>	Specify number of multi-file cache buffers for variable
<code>cdflib.setVarCompression</code>	Specify compression settings used with variable

<code>cdflib.setVarInitialRecs</code>	Specify initial number of records written to variable
<code>cdflib.setVarPadValue</code>	Specify pad value used with variable
<code>cdflib.SetVarReservePercent</code>	Specify reserve percentage for variable
<code>cdflib.setVarsCacheSize</code>	Specify number of cache buffers used for all variables
<code>cdflib.setVarSparseRecords</code>	Specify how variable handles sparse records

## **Attributes**

<code>cdflib.createAttr</code>	Create attribute
<code>cdflib.deleteAttr</code>	Delete attribute
<code>cdflib.deleteAttrEntry</code>	Delete attribute entry
<code>cdflib.deleteAttrgEntry</code>	Delete entry in global attribute
<code>cdflib.getAttrEntry</code>	Value of entry in attribute with variable scope
<code>cdflib.getAttrgEntry</code>	Value of entry in global attribute
<code>cdflib.getAttrMaxEntry</code>	Number of last entry for variable attribute
<code>cdflib.getAttrMaxgEntry</code>	Number of last entry for global attribute
<code>cdflib.getAttrName</code>	Name of attribute, given attribute number
<code>cdflib.getAttrNum</code>	Attribute number, given attribute name
<code>cdflib.getAttrScope</code>	Scope of attribute
<code>cdflib.getNumAttrEntries</code>	Number of entries for attribute with variable scope



<code>cdflib.getNumAttrgEntries</code>	Number of entries for attribute with global scope
<code>cdflib.getNumAttributes</code>	Number of attributes with variable scope
<code>cdflib.getNumgAttributes</code>	Number of attributes with global scope
<code>cdflib.inquireAttr</code>	Information about attribute
<code>cdflib.inquireAttrEntry</code>	Information about entry in attribute with variable scope
<code>cdflib.inquireAttrgEntry</code>	Information about entry in attribute with global scope
<code>cdflib.putAttrEntry</code>	Write value to entry in attribute with variable scope
<code>cdflib.putAttrgEntry</code>	Write value to entry in attribute with global scope
<code>cdflib.renameAttr</code>	Rename existing attribute

**Utility Functions**

<code>cdflib.computeEpoch</code>	Convert time value to CDF_EPOCH value
<code>cdflib.computeEpoch16</code>	Convert time value to CDF_EPOCH16 value
<code>cdflib.epoch16Breakdown</code>	Convert CDF_EPOCH16 value to time value
<code>cdflib.epochBreakdown</code>	Convert CDF_EPOCH value into time value

**See Also**

`cdfread` | `cdfinfo` | `cdfwrite`

**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 these functions, 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”

<b>Purpose</b>	Close specified variable from multifile format Common Data Format (CDF) file
<b>Syntax</b>	<code>cdflib.closeVar(cdfId,varNum)</code>
<b>Description</b>	<p><code>cdflib.closeVar(cdfId,varNum)</code> closes a variable in a multifile format CDF.</p> <p><code>cdfId</code> identifies the CDF file and <code>varNum</code> is a numeric value that specifies the variable. Variable identifiers (variable numbers) are zero-based.</p> <p>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.</p>
<b>Examples</b>	<p>Create a multifile CDF, create a variable, and then close the variable. To run this example, you must be in a writable folder.</p> <pre>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</pre>
<b>References</b>	This function corresponds to the CDF library C API routine <code>CDFclosezVar</code> . To use this function, you must be familiar with the CDF

## cdflib.closeVar

---

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 these functions, 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.

# cdflib.computeEpoch

---

## 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];
```

# cdflib.computeEpoch16

---

```
epoch16 = cdflib.computeEpoch16(timeval);
```

## References

This function corresponds to the CDF library C API routine `computeEPOCH16`. To use these functions, 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”



<b>Purpose</b>	Create Common Data Format (CDF) file
<b>Syntax</b>	<code>cdfId = cdflib.create(filename)</code>
<b>Description</b>	<code>cdfId = cdflib.create(filename)</code> creates a new CDF file with the name specified by the text string <i>filename</i> . Returns the CDF file identifier <code>cdfId</code> .
<b>Examples</b>	<p>Create a CDF file. To run this example, you must have write permission in your current directory.</p> <pre>cdfId = cdflib.create('myfile.cdf');  % Clean up cdflib.delete(cdfId);  clear cdfId</pre>
<b>References</b>	This function corresponds to the CDF library C API routine <code>CDFcreateCDF</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the <a href="#">CDF Web site</a> . For copyright information, see the <code>cdfcopyright.txt</code> file.
<b>See Also</b>	<code>cdflib.open</code>   <code>cdflib.close</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>

# cdflib.createAttr

---

**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 these functions, 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.

<b>CDF Data Type</b>	<b>Description</b>
CDF_BYTE	1-byte, signed integer
CDF_CHAR	1 byte, signed character data type that maps to the MATLAB <code>char</code> class
CDF_INT1	1-byte, signed integer
CDF_UCHAR	1 byte, unsigned character data type that maps to the MATLAB <code>uint8</code> class
CDF_UINT1	1-byte, unsigned integer
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer

<b>CDF Data Type</b>	<b>Description</b>
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.

# cdflib.createVar

---

## 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 these functions, 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 these functions, 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

## cdflib.deleteAttr

---

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
```

# cdflib.deleteAttrEntry

---

```
    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 `CDFdeleteAttrEntry`. To use these functions, 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”

**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
```

# cdflib.deleteAttrgEntry

---

```
numgAttrs: 0

% Create an attribute with global scope in the file.
attrNum = cdfplib.createAttr(cdfid,'my_global_attr','global_scope');

% Write a value to an entry for the attribute
cdfplib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My global attr');

% Get the value of the global attribute entry
value = cdfplib.getAttrgEntry(cdfid,attrNum,0)

value =

My global attr

% Delete the entry
cdfplib.deleteAttrgEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdfplib.getAttrgEntry(cdfid,attrNum,0) % Should fail

% Clean up
cdfplib.delete(cdfid);

clear cdfid
```

## References

This function corresponds to the CDF library C API routine `CDFdeleteAttrgEntry`. To use these functions, 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

`cdfplib.deleteAttr` | `cdfplib.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'
```

# cdflib.deleteVar

---

```
        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 these functions, 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** 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);
```

# cdflib.deleteVarRecords

---

## References

This function corresponds to the CDF library C API routine `CDFdeletezVarRecords`. To use these functions, 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”

**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.

<b>timeVec Element</b>	<b>Description</b>	<b>Valid Values</b>
<code>timeVec(1,:)</code>	Year AD	e.g. 1994
<code>timeVec(2,:)</code>	Month	1 12
<code>timeVec(3,:)</code>	Day	1 31
<code>timeVec(4,:)</code>	Hour	0 23
<code>timeVec(5,:)</code>	Minute	0 59
<code>timeVec(6,:)</code>	Second	0 59
<code>timeVec(7,:)</code>	Millisecond	0 999
<code>timeVec(8,:)</code>	Microsecond	0 999
<code>timeVec(9,:)</code>	Nanosecond	0 999
<code>timeVec(10,:)</code>	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
```

# cdflib.epoch16Breakdown

---

12  
31  
23  
59  
59  
50  
100  
500  
999

## References

This function corresponds to the CDF library C API routine EPOCH16breakdown. To use these functions, 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”

**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
<code>timeVec(1,:)</code>	Year AD	e.g. 1994
<code>timeVec(2,:)</code>	Month	1 12
<code>timeVec(3,:)</code>	Day	1 31
<code>timeVec(4,:)</code>	Hour	0 23
<code>timeVec(5,:)</code>	Minute	0 59
<code>timeVec(6,:)</code>	Second	0 59
<code>timeVec(7,:)</code>	Millisecond	0 999

**Examples** Convert a CDF\_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
```

# cdflib.epochBreakdown

---

23  
59  
59  
0

## References

This function corresponds to the CDF library C API routine EPOCHbreakdown. To use these functions, 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”

<b>Purpose</b>	Value of entry in attribute with variable scope
<b>Syntax</b>	<code>value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)</code>
<b>Description</b>	<code>value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)</code> returns the value of an attribute entry in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>attrNum</b> Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.</p> <p><b>entryNum</b> Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.</p>
<b>Output Arguments</b>	<p><b>Value</b> Value of the entry.</p>
<b>Examples</b>	<p>Open the example CDF and get the value of an entry associated with an attribute with variable scope in the file.</p> <pre>cdfid = cdflib.open('example.cdf');  % The fourth attribute is of variable scope. attrscope = cdflib.getAttrScope(cdfid,3)  attrscope =  VARIABLE_SCOPE</pre>

# cdflib.getAttrEntry

---

```
% 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 these functions, 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”



<b>Purpose</b>	Value of entry in global attribute
<b>Syntax</b>	<code>value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)</code>
<b>Description</b>	<code>value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)</code> returns the value of a global attribute entry in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>attrNum</b> Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.</p> <p><b>entryNum</b> Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.</p>
<b>Output Arguments</b>	<p><b>Value</b> Value of the entry.</p>
<b>Examples</b>	<p>Open the example CDF, and then get the value of an entry associated with a global attribute in the file:</p> <pre>cdfid = cdflib.open('example.cdf');  % Any of the first three attributes have global scope. attrscope = cdflib.getAttrScope(cdfid,0)  attrscope =  GLOBAL_SCOPE</pre>

# cdflib.getAttrgEntry

---

```
% 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”

**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
```

# cdflib.getAttrMaxEntry

---

```
% 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 `CDFgetAttrMaxEntry`. To use these functions, 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”

<b>Purpose</b>	Number of last entry for global attribute
<b>Syntax</b>	<code>maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)</code>
<b>Description</b>	<code>maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)</code> returns the last entry number of a global attribute in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b><code>cdfId</code></b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b><code>attrNum</code></b> Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.</p>
<b>Output Arguments</b>	<p><b><code>maxEntry</code></b> Entry number of the last entry in the attribute. Entry numbers are zero-based.</p>
<b>Examples</b>	<p>Open the example CDF and get the number of the last entry associated with a global attribute in the file:</p> <pre>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)</pre>

# cdflib.getAttrMaxgEntry

---

```
entrynum =  
    4  
  
% Clean up  
cdflib.close(cdfid);  
  
clear cdfid
```

## References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxgEntry`. To use these functions, 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”

<b>Purpose</b>	Name of attribute, given attribute number
<b>Syntax</b>	<code>name = cdflib.getAttrName(cdfId,attrNum)</code>
<b>Description</b>	<code>name = cdflib.getAttrName(cdfId,attrNum)</code> returns the name of an attribute in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b><code>cdfId</code></b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b><code>attrNum</code></b> Numeric value that identifies the attribute. Attribute numbers are zero-based.</p>
<b>Output Arguments</b>	<p><b><code>name</code></b> Text string specifying the name of the attribute.</p>
<b>Examples</b>	<p>Open the example CDF and get name of an attribute.</p> <pre>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</pre>

# cdflib.getAttrName

---

## References

This function corresponds to the CDF library C API routine `CDFgetAttrName`. To use these functions, 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”



<b>Purpose</b>	Attribute number, given attribute name
<b>Syntax</b>	<code>attrNum = cdflib.getAttrNum(cdfId,attrName)</code>
<b>Description</b>	<code>attrNum = cdflib.getAttrNum(cdfId,attrName)</code> returns the number of an attribute in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b><code>cdfId</code></b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b><code>attrName</code></b> Text string specifying the name of an attribute.</p>
<b>Output Arguments</b>	<p><b><code>attrNum</code></b> Numeric value that identifies the attribute. Attribute numbers are zero-based.</p>
<b>Examples</b>	<p>Open the example CDF and get the attribute number associated with the <code>SampleAttribute</code> attribute.</p> <pre>cdfid = cdflib.open('example.cdf');  attrNum = cdflib.getAttrNum(cdfid,'SampleAttribute')  attrNum =      0  % Clean up cdflib.close(cdfid);  clear cdfid</pre>

# cdflib.getAttrNum

---

## References

This function corresponds to the CDF library C API routine `CDFgetAttrNum`. To use these functions, 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”

**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 =
```

# cdflib.getAttrScope

---

```
GLOBAL_SCOPE

% Clean up
cdflib.close(cdfid);

clear cdfid
```

## References

This function corresponds to the CDF library C API routine `CDFgetAttrScope`. To use these functions, 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”

<b>Purpose</b>	Number of cache buffers used
<b>Syntax</b>	<code>numBuffers = cdflib.getCacheSize(cdfId)</code>
<b>Description</b>	<code>numBuffers = cdflib.getCacheSize(cdfId)</code> returns the number of cache buffers used for the Common Data Format (CDF) file identified by <code>cdfId</code> . For a discussion of cache schemes, see the <i>CDF User's Guide</i> .
<b>Examples</b>	<p>Open the example CDF file and get the cache size:</p> <pre>cdfid = cdflib.open('example.cdf');  numBuf = cdflib.getCacheSize(cdfid)  numBuf =      300  % Clean up cdflib.close(cdfid) clear cdfid</pre>
<b>References</b>	This function corresponds to the CDF library C API routine <code>CDFgetCacheSize</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the <a href="#">CDF Web site</a> . For copyright information, see the <code>cdfcopyright.txt</code> file.
<b>See Also</b>	<code>cdflib.setCacheSize</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>

# cdflib.getChecksum

---

**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 these functions, 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”

# cdflib.getCompression

---

<b>Purpose</b>	Compression settings
<b>Syntax</b>	<code>[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)</code>
<b>Description</b>	<code>[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)</code> returns information about the compression settings of a Common Data Format (CDF) file.
<b>Input Arguments</b>	<b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code> .
<b>Output Arguments</b>	<b>ctype</b> 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 <code>cdflib.setCompression</code> .  <b>cparms</b> 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 <code>cdflib.setCompression</code> .  <b>cpercentage</b> The rate of compression, expressed as a percentage.
<b>Examples</b>	Open the example CDF file and check the compression settings in the file.  <code>cdfId = cdflib.open('example.cdf');</code>  <code>[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)</code>



```
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 these functions, 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”

# cdflib.getCompressionCacheSize

---

**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 these functions, 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”

- 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”

# cdflib.getConstantValue

---

**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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Copyright notice in Common Data Format (CDF) file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Syntax</b>      | <code>copyright = cdflib.getCopyright(cdfId)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Description</b> | <code>copyright = cdflib.getCopyright(cdfId)</code> returns the copyright notice in the CDF file identified by <code>cdfId</code> .                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Examples</b>    | <p>Create a CDF file, and then get the copyright notice in the file. To run this example, you must be in a writable folder.</p> <pre>cdfId = cdflib.create('your_file.cdf');<br/><br/>copyright = cdflib.getCopyright(cdfId)<br/><br/>copyright =<br/><br/>Common Data Format (CDF)<br/>(C) Copyright 1990-2009 NASA/GSFC<br/>Space Physics Data Facility<br/>NASA/Goddard Space Flight Center<br/>Greenbelt, Maryland 20771 USA<br/>(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)<br/><br/>% Clean up.<br/>cdflib.delete(cdfId)<br/>clear cdfId</pre> |
| <b>References</b>  | This function corresponds to the CDF library C API routine <code>CDFgetCopyright</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.                                                                                                                                                                                                                                                                  |
| <b>See Also</b>    | <code>cdflib.getLibraryCopyright</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>                                                                                                                                                                                                                                                                                                                                                                                           |

# 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
```

# cdflib.getFormat

---

## References

This function corresponds to the CDF library C API routine `CDFgetFormat`. To use these functions, 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”



|                    |                                                                                                                                                                                                                                                                                                                |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Copyright notice of Common Data Format (CDF) library                                                                                                                                                                                                                                                           |
| <b>Syntax</b>      | <pre>copyright = cdflib.getLibraryCopyright()</pre>                                                                                                                                                                                                                                                            |
| <b>Description</b> | <pre>copyright = cdflib.getLibraryCopyright()</pre> returns a text string containing the copyright notice of the CDF library.                                                                                                                                                                                  |
| <b>Examples</b>    | <pre>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)</pre> |
| <b>References</b>  | This function corresponds to the CDF library C API routine <code>CDFgetLibraryCopyright</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.                 |
| <b>See Also</b>    | <code>cdflib.getCopyright</code>                                                                                                                                                                                                                                                                               |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>                                                                                                                                                 |

# cdflib.getLibraryVersion

---

|                         |                                                                                                                                                                                                                                                                 |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Library version and release information                                                                                                                                                                                                                         |
| <b>Syntax</b>           | <code>[version,release,increment] = cdflib.getLibraryVersion()</code>                                                                                                                                                                                           |
| <b>Description</b>      | <code>[version,release,increment] = cdflib.getLibraryVersion()</code> returns information about the Common Data Format (CDF) library.                                                                                                                           |
| <b>Output Arguments</b> | <b>version</b><br>Numeric value indicating the version number of the CDF library.<br><b>release</b><br>Numeric value indicating the release number of the CDF library.<br><b>increment</b><br>Numeric value indicating the increment number of the CDF library. |
| <b>Examples</b>         | Get the version information of the CDF library:<br><pre>[version, release, increment] = cdflib.getLibraryVersion()<br/><br/>version =<br/>    3<br/><br/>release =<br/>    3<br/><br/>increment =<br/>    0</pre>                                               |
| <b>References</b>       | This function corresponds to the CDF library C API routine <code>CDFgetLibraryVersion</code> . To use these functions, 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 these functions, 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

% Determine the number of entries for the attribute
attrEntries = cdflib.getNumAttrEntries(cdfid,attrNum)

attrEntries =

    4

% Clean up
cdflib.close(cdfid);
```

# cdflib.getNumAttrEntries

---

## References

This function corresponds to the CDF library C API routine `CDFgetNumAttrzEntries`. To use these functions, 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”



|                    |                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Number of entries for attribute with global scope                                                                                                                                                                                                                                                                                                                                        |
| <b>Syntax</b>      | <code>nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)</code>                                                                                                                                                                                                                                                                                                                         |
| <b>Description</b> | <p><code>nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)</code> returns the number of entries written for the specified global attribute in the Common Data Format (CDF) file.</p> <p><code>cdfId</code> identifies the CDF file. <code>attrNum</code> is a numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.</p> |
| <b>Examples</b>    | <p>Open the example CDF and find out how many entries are associated with a global attribute in the file.</p> <pre>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</pre>                                                       |
| <b>References</b>  | This function corresponds to the CDF library C API routine <code>CDFgetNumAttrgEntries</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the <a href="#">CDF Web site</a> . For copyright information, see the <code>cdfcopyright.txt</code> file.                                                                           |
| <b>See Also</b>    | <code>cdflib.getNumAttrEntries</code>                                                                                                                                                                                                                                                                                                                                                    |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>“Importing Common Data File Format (CDF) Files”</li></ul>                                                                                                                                                                                                                                                                                          |

## **cdflib.getNumAttrgEntries**

---

- “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
numAttr = cdflib.getNumAttributes(cdfid)

numAttr =

     1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

**References** This function corresponds to the CDF library C API routine CDFgetNumvAttributes. To use these functions, 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”

# 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 these functions, 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 these functions, 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”

|                    |                                                                                                                                                                                                                                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Number of cache buffers for staging                                                                                                                                                                                                                                                                        |
| <b>Syntax</b>      | <code>numBuffers = cdflib.getStageCacheSize(cdfId)</code>                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p><code>numBuffers = cdflib.getStageCacheSize(cdfId)</code> 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 <i>CDF User's Guide</i>.</p> <p><code>cdfId</code> identifies the CDF file.</p> |
| <b>Examples</b>    | <p>Open the example CDF file and determine the number of cache buffers used for staging:</p> <pre>cdfId = cdflib.open('example.cdf');<br/><br/>numBuf = cdflib.getStageCacheSize(cdfId)<br/><br/>numBuf =<br/><br/>    125<br/><br/>% Clean up<br/>cdflib.close(cdfId)<br/>clear cdfId</pre>               |
| <b>References</b>  | <p>This function corresponds to the CDF library C API routine <code>CDFgetStageCacheSize</code>. To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.</p>         |
| <b>See Also</b>    | <code>cdflib.setStageCacheSize</code>                                                                                                                                                                                                                                                                      |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>                                                                                                                                             |

# cdflib.getValidate

---

**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 <code>cdflib.setValidate</code> . |
| '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 these functions, 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”



|                    |                                                                                                                                                                                                                                                                                                                                |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Number of records allocated for variable                                                                                                                                                                                                                                                                                       |
| <b>Syntax</b>      | <code>numrecs = cdflib.getVarAllocRecords(cdfId, varNum)</code>                                                                                                                                                                                                                                                                |
| <b>Description</b> | <p><code>numrecs = cdflib.getVarAllocRecords(cdfId, varNum)</code> returns the number of records allocated for a variable in a Common Data Format (CDF) file.</p> <p><code>cdfId</code> identifies the CDF file. <code>varNum</code> is a numeric value that identifies the variable. Variable numbers are zero-based.</p>     |
| <b>Examples</b>    | <p>Open example CDF and get the number of records allocated for a variable:</p> <pre>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</pre> |
| <b>References</b>  | <p>This function corresponds to the CDF library C API routine <code>CDFgetzVarAllocRecords</code>. To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.</p>                           |
| <b>See Also</b>    | <code>cdflib.setVarAllocBlockRecords</code>                                                                                                                                                                                                                                                                                    |

# **cdflib.getVarAllocRecords**

---

## **Tutorials**

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Blocking factor for variable                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>      | <code>blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)</code>                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p><code>blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)</code> returns the blocking factor for a variable in a Common Data Format (CDF) file. A variable's <i>blocking factor</i> specifies the minimum number of records the library allocates when you write to an unallocated record. <code>cdfId</code> identifies the CDF file. <code>varNum</code> is a numeric value that identifies the variable. Variable numbers are zero-based.</p> |
| <b>Examples</b>    | <p>Open the example CDF and determine the blocking factor of a variable.</p> <pre>cdfid = cdflib.open('example.cdf');<br/><br/>cdflib.getVarBlockingFactor(cdfid,0)<br/><br/>ans =<br/><br/>    0<br/><br/>% Clean up<br/>cdflib.close(cdfid)<br/>clear cdfid</pre>                                                                                                                                                                                        |
| <b>References</b>  | <p>This function corresponds to the CDF library C API routine <code>CDFgetzVarBlockingFactor</code>. To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.</p>                                                                                                                                                     |
| <b>See Also</b>    | <code>cdflib.setVarBlockingFactor</code>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>                                                                                                                                                                                                                                                                                             |

# cdflib.getVarCacheSize

---

**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 these functions, 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”

# cdflib.getVarCompression

---

**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-based)  
[ctype cparams 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 these functions, 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”

# cdflib.getVarData

---

|                         |                                                                                                                                                                                                 |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Single value from record in variable                                                                                                                                                            |
| <b>Syntax</b>           | <code>datum = cdflib.getVarData(cdfId,varNum,recNum,indices)</code>                                                                                                                             |
| <b>Description</b>      | <code>datum = cdflib.getVarData(cdfId,varNum,recNum,indices)</code> returns a single value from a variable in a Common Data Format (CDF) file.                                                  |
| <b>Input Arguments</b>  | <b><code>cdfId</code></b><br>Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code> .                                                           |
|                         | <b><code>varNum</code></b><br>Numeric value identifying the variable containing the datum. Variable numbers are zero-based.                                                                     |
|                         | <b><code>recNum</code></b><br>Numeric value identifying the location of the datum in the variable. In CDF terminology, this is called the <i>record number</i> . Record numbers are zero-based. |
|                         | <b><code>indices</code></b><br>Array of dimension indices within the record. Dimension indices are zero-based. If the variable has no dimensions, you can omit this parameter.                  |
| <b>Output Arguments</b> | <b><code>datum</code></b><br>Value of the specified record.                                                                                                                                     |
| <b>Examples</b>         | Open the example CDF file and retrieve data associated with a variable:<br><br><pre>cdfid = cdflib.open('example.cdf');<br/><br/>% Determine how many variables are in the file.</pre>          |



```
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.getVarData

---

```
cdflib.close(cdfid);  
clear cdfid
```

## References

This function corresponds to the CDF library C API routine `CDFgetzVarData`. To use these functions, 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”

|                    |                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Maximum allocated record number for variable                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <code>maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)</code>                                                                                                                                                                                                                                                                                                 |
| <b>Description</b> | <p><code>maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)</code> returns the record number of the maximum allocated record for a variable in a Common Data Format (CDF) file.</p> <p><code>cdfId</code> identifies the CDF file. <code>varNum</code> is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.</p> |
| <b>Examples</b>    | <p>Open example CDF and get the maximum allocated record number for a variable:</p> <pre>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</pre>                                                  |
| <b>References</b>  | <p>This function corresponds to the CDF library C API routine <code>CDFgetzVarMaxAllocRecNum</code>. To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.</p>                                                          |
| <b>See Also</b>    | <code>cdflib.getVarMaxWrittenRecNum</code>                                                                                                                                                                                                                                                                                                                      |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li></ul>                                                                                                                                                                                                                                                               |

## **cdflib.getVarMaxAllocRecNum**

---

- “Exporting to Common Data File Format (CDF) Files”

**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
```

# cdflib.getVarMaxWrittenRecNum

---

## References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxWrittenRecNum`. To use these functions, 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”

- 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 these functions, 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.getVarName

---

**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 these functions, 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”



<b>Purpose</b>	Variable number, given variable name
<b>Syntax</b>	<code>varNum = cdflib.getVarNum(cdfId, varname)</code>
<b>Description</b>	<code>varNum = cdflib.getVarNum(cdfId, varname)</code> returns the identifier (variable number) for a variable in a Common Data Format (CDF) file. <code>cdfId</code> identifies the CDF file. <code>varname</code> is a text string that identifies the variable. Variable names are case-sensitive.
<b>Examples</b>	<p>Open example CDF, and then get the number of a variable named Longitude:</p> <pre>cdfid = cdflib.open('example.cdf');  varNum = cdflib.getVarNum(cdfid, 'Longitude')  varNum =      1  % Clean up cdflib.close(cdfid);  clear cdfid</pre>
<b>References</b>	This function corresponds to the CDF library C API routine <code>CDFgetzVarNum</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the <a href="#">CDF Web site</a> . For copyright information, see the <code>cdfcopyright.txt</code> file.
<b>See Also</b>	<code>cdflib.getVarName</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>

# cdflib.getVarNumRecsWritten

---

**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 these functions, 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.getVarPadValue

---

**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 these functions, 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”

<b>Purpose</b>	Entire record for variable
<b>Syntax</b>	<code>data = cdflib.getVarRecordData(cdfId,varNum,recNum)</code>
<b>Description</b>	<code>data = cdflib.getVarRecordData(cdfId,varNum,recNum)</code> returns the data in a record associated with a variable in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>varNum</b> Numeric value that identifies the variable in the CDF file. Variable numbers are zero-based.</p> <p><b>recNum</b> Numeric value that identifies the record in the variable. Record numbers are zero-based.</p>
<b>Output Arguments</b>	<p><b>data</b> Data in the record.</p>
<b>Examples</b>	<p>Open the example CDF, and then get the data associated with a record in a variable:</p> <pre>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</pre>

# cdflib.getVarRecordData

---

```
% Clean up  
cdflib.close(cdfid)
```

```
clear cdfid
```

## References

This function corresponds to the CDF library C API routine `CDFgetzVarRecordData`. To use these functions, 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”

<b>Purpose</b>	Compression reserve percentage for variable
<b>Syntax</b>	<code>percent = cdflib.getVarReservePercent(cdfId,varNum)</code>
<b>Description</b>	<code>percent = cdflib.getVarReservePercent(cdfId,varNum)</code> returns the compression reserve percentage for a variable in a Common Data Format (CDF) file. This operation only applies to compressed variables. <code>cdfId</code> identifies the CDF file. <code>varNum</code> is a numeric value that identifies the variable. Variable numbers are zero-based.
<b>Definitions</b>	<b>reserve percentage</b> 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.
<b>Examples</b>	Open the example CDF file, get the number of a compressed variable, and then determine the reserve percent for the variable.  <pre>cdfid = cdflib.open('example.cdf'); varnum = cdflib.getVarNum(cdfid,'Temperature'); percent = cdflib.getVarReservePercent(cdfid,varnum); cdflib.close(cdfid);</pre>
<b>References</b>	This function corresponds to the CDF library C API routine <code>CDFgetzVarReservePercent</code> . To use these functions, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF Web site. For copyright information, see the <code>cdfcopyright.txt</code> file.

# cdflib.getVarReservePercent

---

## 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.

# cdflib.getVarSparseRecords

---

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 these functions, 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 =
```

# cdflib.getVersion

---

```
7
increment =
8
% Clean up
cdflib.close(cdfId)
clear cdfId
```

## References

This function corresponds to the CDF library C API routine `CDFgetVersion`. To use these functions, 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');
```

# cdflib.hyperGetVarData

---

```
% 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 these functions, 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”

<b>Purpose</b>	Write hyperslab of data to variable
<b>Syntax</b>	<code>cdfplib.hyperPutVarData(cdfId, varNum, recSpec, dimSpec, data)</code>
<b>Description</b>	<code>cdfplib.hyperPutVarData(cdfId, varNum, recSpec, dimSpec, data)</code> 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.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdfplib.create</code> or <code>cdfplib.open</code>.</p> <p><b>varNum</b> Specifies the variable containing the datum.</p> <p><b>recSpec</b> Three-element array described by <code>[RSTART RCOUNT RSTRIDE]</code>, where <code>RSTART</code>, <code>RCOUNT</code>, and <code>RSTRIDE</code> are scalar values giving the start, number of records, and sampling interval (or stride) between records. Record indices are zero-based.</p> <p><b>dimSpec</b> Three-element cell array described by <code>{DSTART DCOUNT DSTRIDE}</code>, where <code>DSTART</code>, <code>DCOUNT</code>, and <code>DSTRIDE</code> 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.</p> <p><b>data</b> Data to write to the variable.</p>
<b>Examples</b>	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 these functions, 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 =
```

# cdflib.inquire

---

```
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 these functions, 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
<code>name</code>	Attribute's name
<code>scope</code>	Either 'GLOBAL_SCOPE' or 'VARIABLE_SCOPE'
<code>maxgEntry</code>	The maximum entry number used for global attributes.
<code>maxEntry</code>	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');
```

# cdflib.inquireAttr

---

```
% 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”

<b>Purpose</b>	Information about entry in attribute with variable scope
<b>Syntax</b>	<pre>[datatype,numElements] = cdflib.inquireAttrEntry(cdfId, attrNum,entryNum)</pre>
<b>Description</b>	<pre>[datatype,numElements] = cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)</pre> returns the data type and the number of elements for an attribute entry in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>attrNum</b> Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have variable scope.</p> <p><b>entryNum</b> Numeric value identifying the entry in the attribute. Entry number are zero-based.</p>
<b>Output Arguments</b>	<p><b>datatype</b> Text string identifying a CDF data type. For a list of CDF data types, see <code>cdflib.putAttrEntry</code></p> <p><b>numElements</b> Numeric value indicating the number of elements in the entry.</p>
<b>Examples</b>	<p>Open example CDF, and then get information about entries associated with an attribute in the file:</p> <pre>cdfid = cdflib.open('example.cdf');</pre>

# cdflib.inquireAttrEntry

---

```
% 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 these functions, 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”

<b>Purpose</b>	Information about entry in attribute with global scope
<b>Syntax</b>	<pre>[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId, attrNum,entryNum)</pre>
<b>Description</b>	<pre>[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)</pre> returns the data type and the number of elements for a global attribute entry in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>attrNum</b> Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have global scope.</p> <p><b>entryNum</b> Numeric value identifying the entry in the attribute. Entry number are zero-based.</p>
<b>Output Arguments</b>	<p><b>datatype</b> Text string identifying a CDF data type. For a list of CDF data types, see <code>cdflib.putAttrgEntry</code></p> <p><b>numElements</b> Numeric value indicating the number of elements in the entry.</p>
<b>Examples</b>	<p>Open the example CDF, and then get information about entries associated with a global attribute in the file.</p> <pre>cdfid = cdflib.open('example.cdf');</pre>

# cdflib.inquireAttrEntry

---

```
% 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.inquireAttrEntry(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 `CDFinquireAttrEntry`. To use these functions, 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
<code>name</code>	Name of the variable
<code>datatype</code>	Data type
<code>numElements</code>	Number of elements of the datatype
<code>dims</code>	Sizes of the dimensions
<code>recVariance</code>	Record variance
<code>dimVariance</code>	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.

# cdflib.inquireVar

---

## 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 these functions, 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”

<b>Purpose</b>	Open existing Common Data Format (CDF) file
<b>Syntax</b>	<code>cdfId = cdflib.open(filename)</code>
<b>Description</b>	<p><code>cdfId = cdflib.open(filename)</code> opens an existing Common Data Format (CDF) file. <i>filename</i> is a text string that identifies the file.</p> <p>This function returns a CDF file identifier, <code>cdfId</code>.</p> <p>All CDF files opened this way have the <code>zMode</code> set to <code>zModeon2</code>. Refer to the <i>CDF User's Guide</i> for information about <code>zModes</code>.</p>
<b>Examples</b>	<p>Open the example CDF file:</p> <pre>cdfId = cdflib.open('example.cdf');  % Clean up cdflib.close(cdfId)  clear cdfId</pre>
<b>References</b>	<p>This function corresponds to the CDF library C API routine <code>CDFopenCDF</code>. To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the <a href="#">CDF Web site</a>. For copyright information, see the <code>cdfcopyright.txt</code> file.</p>
<b>See Also</b>	<code>cdflib.close</code>   <code>cdflib.create</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>• “Importing Common Data File Format (CDF) Files”</li><li>• “Exporting to Common Data File Format (CDF) Files”</li></ul>

# 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.

<b>CDF Data Type</b>	<b>MATLAB Equivalent</b>
CDF_BYTE	1-byte, signed integer
CDF_CHAR	1 byte, signed character data type that maps to the MATLAB <code>char</code> class
CDF_INT1	1-byte, signed integer.
CDF_UCHAR	1 byte, unsigned character data type that maps to the MATLAB <code>uint8</code> 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

```

# cdflib.putAttrEntry

---

```
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 Test');

% 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 `CDFputAttrEntry`. To use these functions, 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.putAttrEntry` |  
`cdflib.getAttrEntry` | `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.

<b>CDF Data Type</b>	<b>MATLAB Equivalent</b>
CDF_BYTE	1-byte, signed integer
CDF_CHAR	1 byte, signed character data type that maps to the MATLAB <code>char</code> class
CDF_INT1	1-byte, signed integer.
CDF_UCHAR	1 byte, unsigned character data type that maps to the MATLAB <code>uint8</code> class
CDF_UINT1	1-byte, unsigned integer

# cdflib.putAttrgEntry

---

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 these functions, 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 these functions, 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

---

<b>Purpose</b>	Write entire record to variable
<b>Syntax</b>	<code>cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)</code>
<b>Description</b>	<code>cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)</code> writes data to a record in a variable in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>varNum</b> Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.</p> <p><b>recNum</b> Numeric value identifying the location of the datum in the variable. Record numbers are zero-based.</p> <p><b>recordData</b> Data to be written to the variable.</p>

## 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 these functions, 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 these functions, 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 these functions, 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 these functions, 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 =
```

# cdflib.setCompression

---

0

```
% Clean up  
cdflib.delete(cdfId)  
clear cdfId
```

## References

This function corresponds to the CDF library C API routine `CDFsetCompression`. To use these functions, 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 these functions, 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
```

# cdflib.setFileBackward

---

## 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 =
```

# cdflib.setFormat

---

```
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)

```

# cdflib.setMajority

---

```
majority =  
  
COLUMN_MAJOR  
  
% Clean up  
cdflib.delete(cdfId)  
clear cdfId
```

## References

This function corresponds to the CDF library C API routine `CDFsetMajority`. To use these functions, 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
```

# cdflib.setReadOnlyMode

---

```
% Clean up
cdflib.close(cdfId)
clear cdfId
```

## References

This function corresponds to the CDF library C API routine `CDFsetReadOnlyMode`. To use these functions, 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)
```

# cdflib.setStageCacheSize

---

`clear cdfId`

## References

This function corresponds to the CDF library C API routine `CDFsetStageCacheSize`. To use these functions, 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 these functions, 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.

# cdflib.setValidate

---

## See Also

`cdflib.getValidate` | `cdflib.getConstantValue`

## Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

<b>Purpose</b>	Specify range of records to be allocated for variable
<b>Syntax</b>	<code>cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)</code>
<b>Description</b>	<code>cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)</code> specifies a range of records you want to allocate (but not write) for a variable in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b><code>cdfId</code></b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b><code>varNum</code></b> Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.</p> <p><b><code>firstRec</code></b> Numeric value identifying the record at which to start allocating. Record numbers are zero-based.</p> <p><b><code>lastRec</code></b> Numeric value identifying the record at which to stop allocating. Record numbers are zero-based.</p>
<b>Examples</b>	<p>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.</p> <pre>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.</pre>

# cdflib.setVarAllocBlockRecords

---

```
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”



<b>Purpose</b>	Specify blocking factor for variable
<b>Syntax</b>	<code>cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)</code>
<b>Description</b>	<code>cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)</code> specifies the blocking factor for a variable in a Common Data Format (CDF) file.
<b>Input Arguments</b>	<p><b>cdfId</b> Identifier of a CDF file, returned by a call to <code>cdflib.create</code> or <code>cdflib.open</code>.</p> <p><b>varNum</b> Numeric value identifying a variable in the file. Variable numbers are zero-based.</p> <p><b>blockingFactor</b> Numeric value that specifies the number of records to allocate when writing to an unallocated record.</p>
<b>Definitions</b>	<p><b>blocking factor</b></p> <p>A variable's <i>blocking factor</i> 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.</p>
<b>Examples</b>	<p>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.</p> <pre>cdfid = cdflib.create('your_file.cdf');  % Create a variable in the file. varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);</pre>

# cdflib.setVarBlockingFactor

---

```
% 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
```

# cdflib.setVarCacheSize

---

```
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 these functions, 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”

**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,

# cdflib.setVarCompression

---

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 these functions, 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”

# cdflib.setVarInitialRecs

---

**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 these functions, 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.setVarPadValue

---

**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 these functions, 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, expressed 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 these functions, 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”

# cdflib.setVarsCacheSize

---

**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 these functions, 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”

# cdflib.setVarSparseRecords

---

**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 these functions, 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”

# cdfread

---

**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. <code>cdfread</code> 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. <code>cdfread</code> 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, <code>cdfread</code> 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 <code>cdfread</code> automatically converts CDF epoch data types to MATLAB serial date numbers. If set to <code>false</code> (the default), <code>cdfread</code> wraps epoch values in MATLAB <code>cdfepoch</code> objects. Note: For better performance when reading large data sets, set this parameter to <code>true</code> .
'CombineRecords'	A Boolean value that determines how <code>cdfread</code> returns the CDF data sets read from the file. If set to <code>false</code> (the default), <code>cdfread</code> 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 <code>true</code> , <code>cdfread</code> combines

# cdfread

---

Parameter	Value
	<p>all records for a particular variable into one cell in the output cell array. In this cell, <code>cdfread</code> stores scalar data as a column array. <code>cdfread</code> extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, <code>cdfread</code> stores all the records in one cell as a 1000-by-20-by-30 array</p> <p>Note: If you use the 'Records' parameter to specify which records to read, you cannot use the 'CombineRecords' parameter.</p> <p>Note: When using the 'Variable' parameter to read one variable, if the 'CombineRecords' parameter is true, <code>cdfread</code> returns the data as an M-by-N numeric or character array; it does not put the data into a cell array.</p>

---

**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](#) | [cdfwrite](#)

## How To

- “Importing Common Data File Format (CDF) Files”

# cdfwrite

---

**Purpose** Write data to Common Data Format (CDF) file

**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

# cdfwrite

---

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](#)



**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`

# cell

---

## 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'
```

# cell

---

'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”

**Purpose** Convert cell array to numeric array

**Syntax** `A= cell2mat(C)`

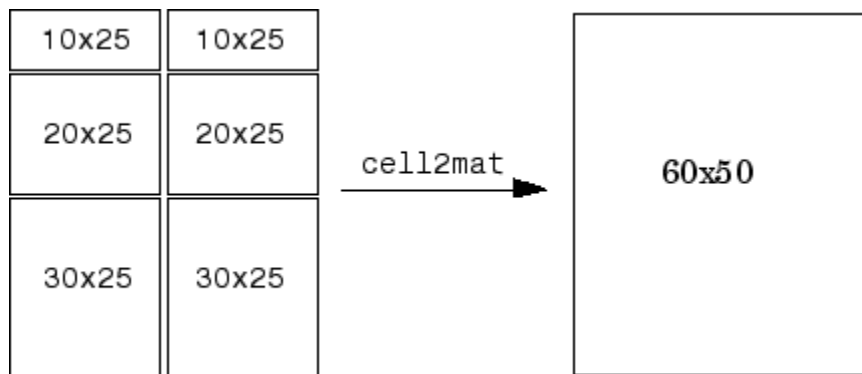
**Description** `A= cell2mat(C)` converts cell array `C` with contents of the same data type into a single array, `A`.

**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.



**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`.

# cell2mat

---

## 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](#)

**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

## cell2struct

---

	<b>5 Years</b>	<b>10 Years</b>	<b>15 Years</b>
<b>Quality</b>	Bates, Gray	Nash	Kay, Chase
<b>Documentation</b>	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:



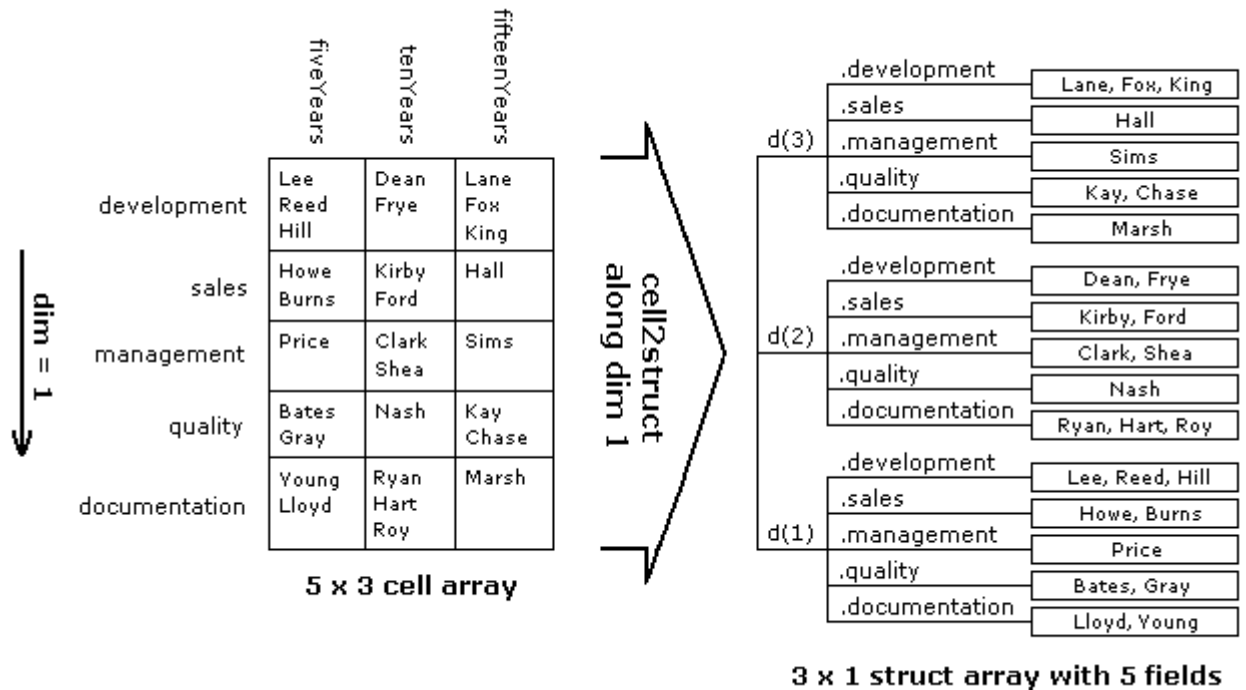
	fiveYears	tenYears	fifteenYears
development	Lee Reed Hill	Dean Frye	Lane Fox King
sales	Howe Burns	Kirby Ford	Hall
management	Price	Clark Shea	Sims
quality	Bates Gray	Nash	Kay Chase
documentation	Young Lloyd	Ryan Hart Roy	Marsh

### 5 × 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:

# cell2struct



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:

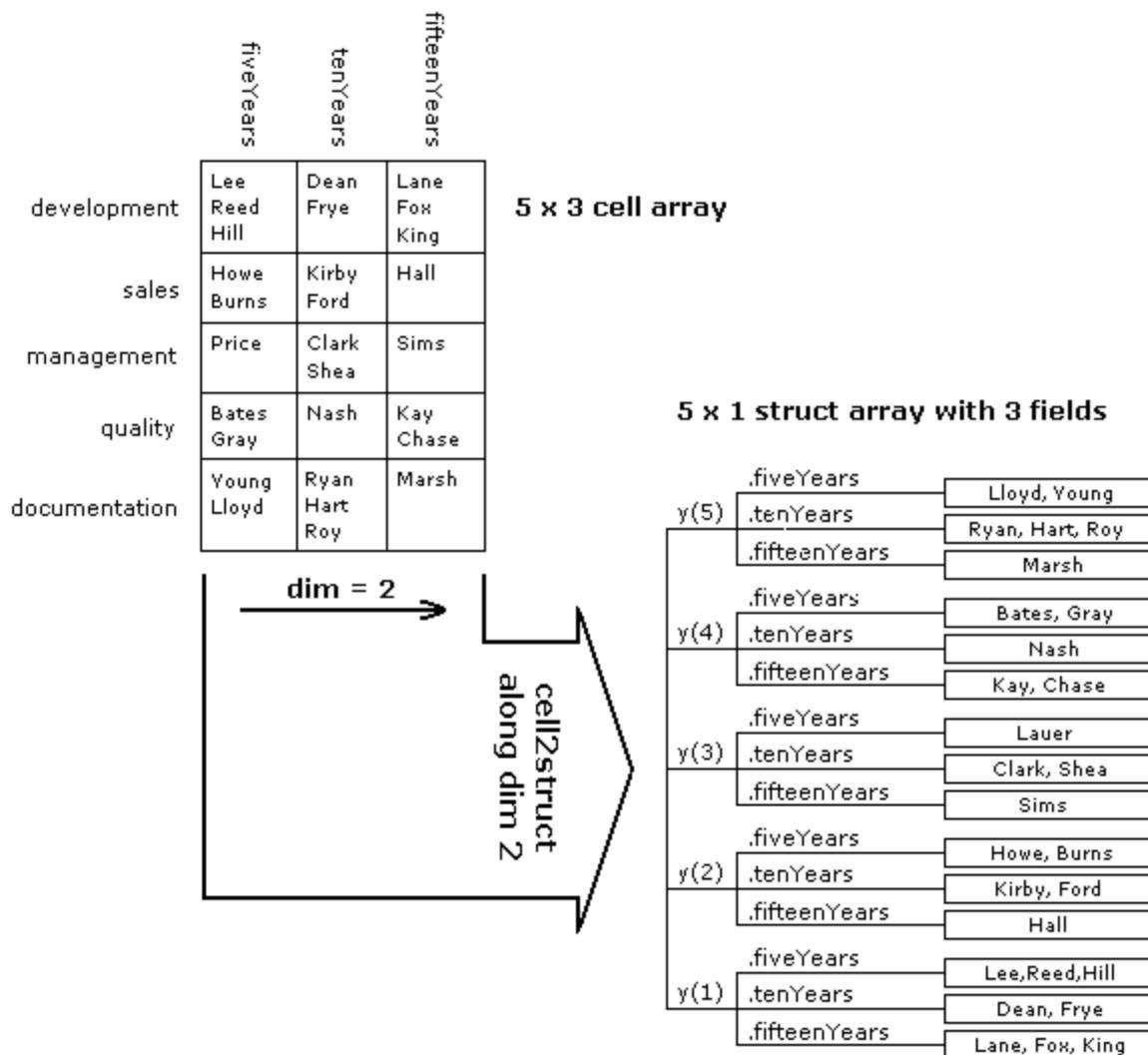
```
depts(1:2).development
ans =
    'Lee'    'Reed'    'Hill'
ans =
    'Dean'   'Frye'
```

---

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:

# cell2struct



- 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'
```

---

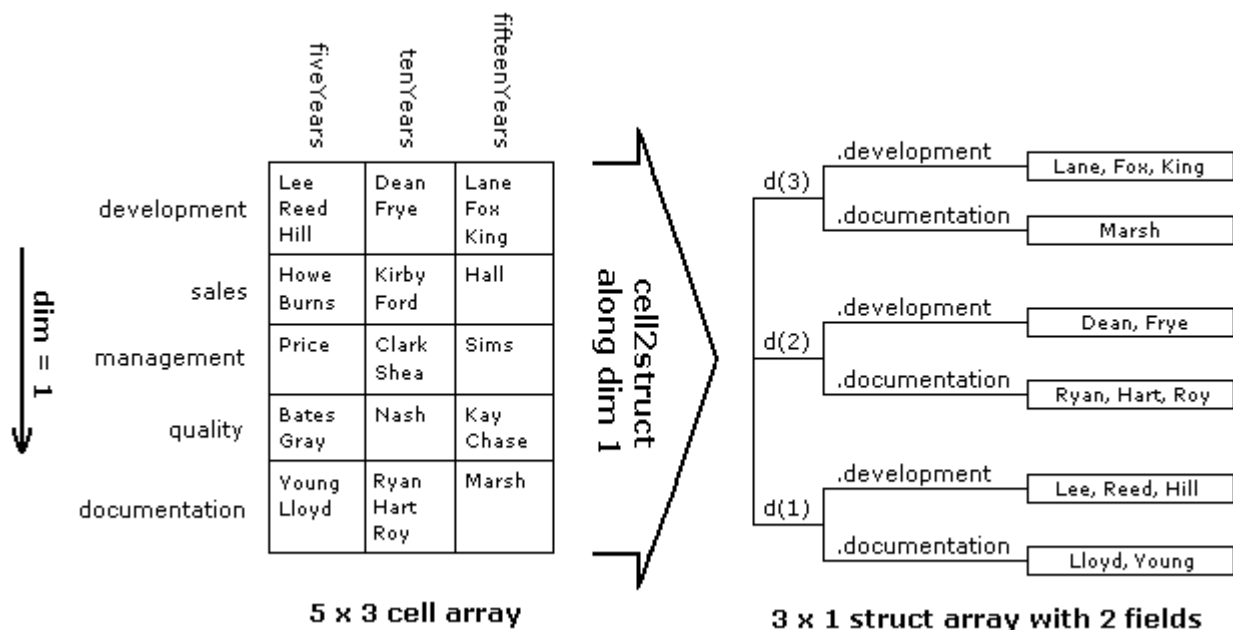
Convert only part of the cell array to a struct:

- 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
```

# cell2struct



**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, 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`.

## **UniformOutput**

Logical value, as follows:

- |                        |                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>true</code> (1)  | Indicates that for all inputs, each output from function <code>func</code> is a cell array or a scalar value that is always of the same type. The <code>cellfun</code> function combines the outputs in arrays <code>A1, ..., Am</code> , where <code>m</code> is the number of function outputs. Each output array is of the same type as the individual function outputs. |
| <code>false</code> (0) | Requests that the <code>cellfun</code> function combine the outputs into cell arrays <code>A1, ..., Am</code> . The outputs of function <code>func</code> 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:

<code>identifier</code>	Error identifier.
<code>message</code>	Error message text.
<code>index</code>	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  $C_1, \dots, C_n$ .

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”

# cellplot

---

**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`.

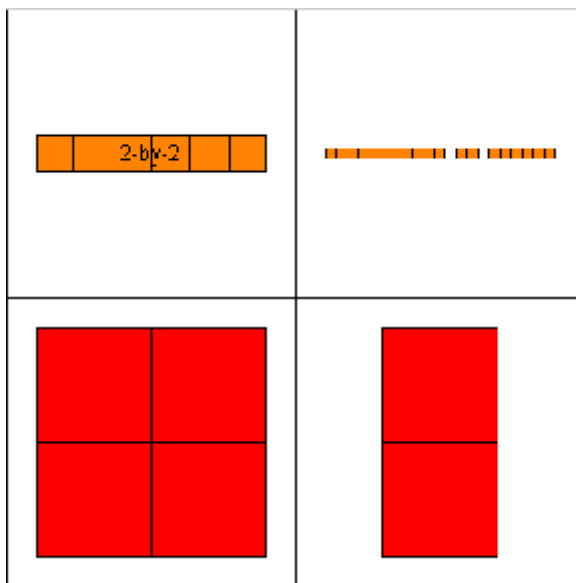
`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

**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”

# cellstr

---

**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](#)



**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  $\text{norm}(b-A*x) / \text{norm}(b)$  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  $\text{inv}(M)*A*x = \text{inv}(M)*b$  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	<code>cgs</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.
1	<code>cgs</code> iterated <code>maxit</code> times but did not converge.
2	Preconditioner <code>M</code> was ill-conditioned.
3	<code>cgs</code> stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during <code>cgs</code> 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 `mfun`.

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:

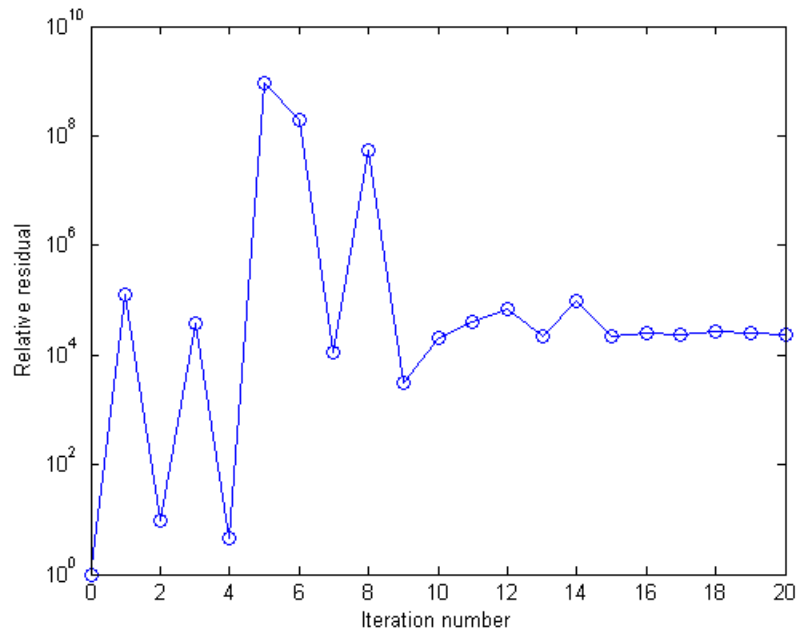
```
[x0,f10,rr0,it0,rv0] = cgs(A,b,tol,maxit);
```

**f10** 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 ( $x_0 = \text{zeros}(\text{size}(A,2),1)$ ) is the best solution and is returned as indicated by  $it_0 = 0$ . 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:

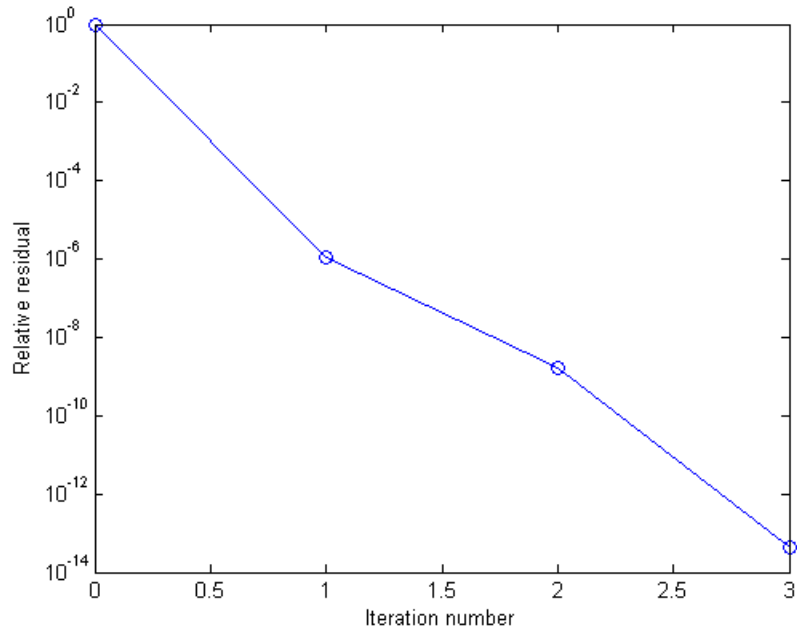
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));  
[x1,f11,rr1,it1,rv1] = cgs(A,b,tol,maxit,L,U);
```

f11 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

# char

---

**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”



<b>Purpose</b>	Check MATLAB code files for possible problems
<b>Alternatives</b>	For information on using the graphical user interface for checking code, see “Check Code for Errors and Warnings”.
<b>Syntax</b>	<pre>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')</pre>

**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.

---

# checkcode

---

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,filepath]=checkcode('filename')` additionally returns `filepath`, 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','-em1')` `'-em1'` 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 'nohandle' 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 STRCMP(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 'nohandle' 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 'nohandle' 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 'nohandle' 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 more in
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”.

**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 `filename2` 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.

# checkin

---

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)”



**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 `filename` 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 <code>undocheckout</code> followed by a <code>checkout</code> .
'force'	'off' (default)	Prevents you from checking out the file if you already have it checked out.

# checkout

---

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,p] = chol(A)
[L,p] = chol(A, 'lower')
[R,p,S] = chol(A)
[R,p,s] = chol(A, 'vector')
[L,p,s] = chol(A, 'lower', '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,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  $A$ .

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.

---

**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
```

# chol

---

```
-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.
```

# chol

---

## See Also

[cholinc](#) | [cholupdate](#)



**Purpose** Sparse incomplete Cholesky and Cholesky-Infinity factorizations

---

**Note** cholinc will be removed in a future version. Use ichol instead. For all syntaxes except cholinc(X, 'inf'), use ichol instead. Note that cholinc returns upper triangular factors, while ichol returns lower triangular factors by default.

---

**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  $\text{droptol} \cdot \text{norm}(X(:, j))$  (see `luinc`), the nonzero entries  $R(i, j)$  are bounded below by the local drop tolerance  $\text{droptol} \cdot \text{norm}(X(:, j)) / R(i, i)$ .

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)`.

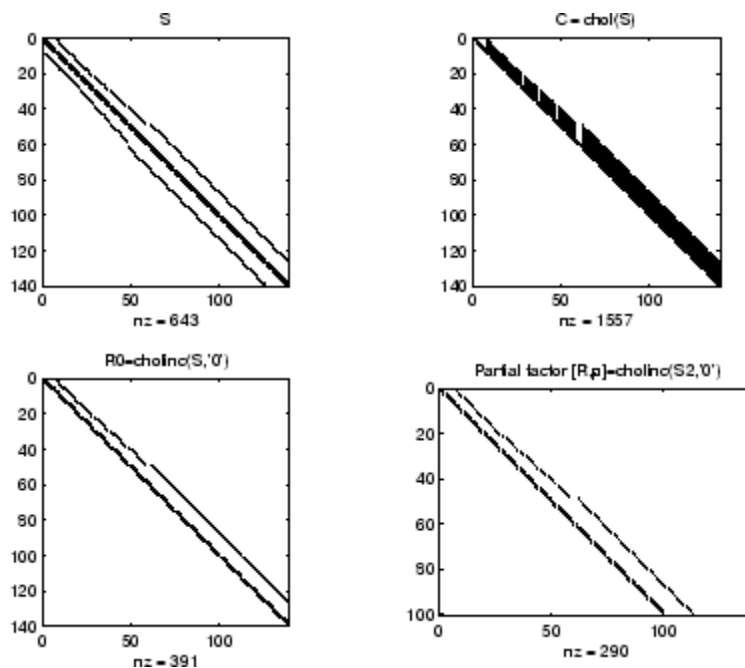
Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make `S` singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level 0.

```
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;
```

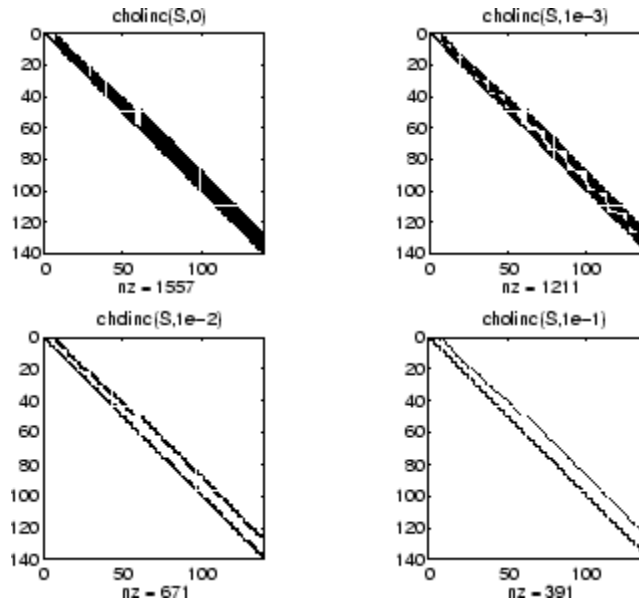
$D1$  has elements of the order of  $\text{eps}$ , showing that  $R0' * R0$  agrees with  $S$  over its sparsity pattern.  $D2$  has elements of the order of  $\text{eps}$  over its first 100 rows and first 100 columns,  $D2(1:100, :)$  and  $D2(:, 1:100)$ .



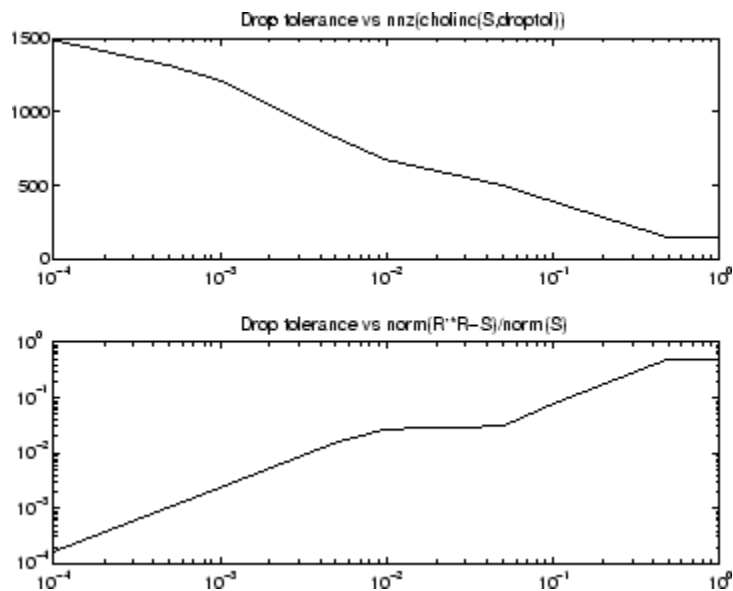
## Example 2

The first subplot below shows that  $\text{cholinc}(S, 0)$ , 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  $\text{norm}(R' * R - S, 1) / \text{norm}(S, 1)$  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.

# cholinc

---

`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`



**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  $R = \text{chol}(A)$  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 =
```

# cholupdate

---

```
      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  $\text{rank}(x*x')$  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  $R1 = \text{chol}(A + x*x')$ , 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

# circshift

---

**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`

<b>Purpose</b>	Circumcenters of specified simplices	
<b>Syntax</b>	<pre>CC = circumcenters(TR, SI) [CC RCC] = circumcenters(TR, SI)</pre>	
<b>Description</b>	<p>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.</p> <p>[CC RCC] = circumcenters(TR, SI) returns the circumcenters and the corresponding radii of the circumscribed circles or spheres.</p>	
<b>Input Arguments</b>	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.
<b>Output Arguments</b>	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.
<b>Definitions</b>	A simplex is a triangle/tetrahedron or higher-dimensional equivalent.	
<b>Examples</b>	<b>Example 1</b>	
	Load a 2-D triangulation.	

## TriRep.circumcenters

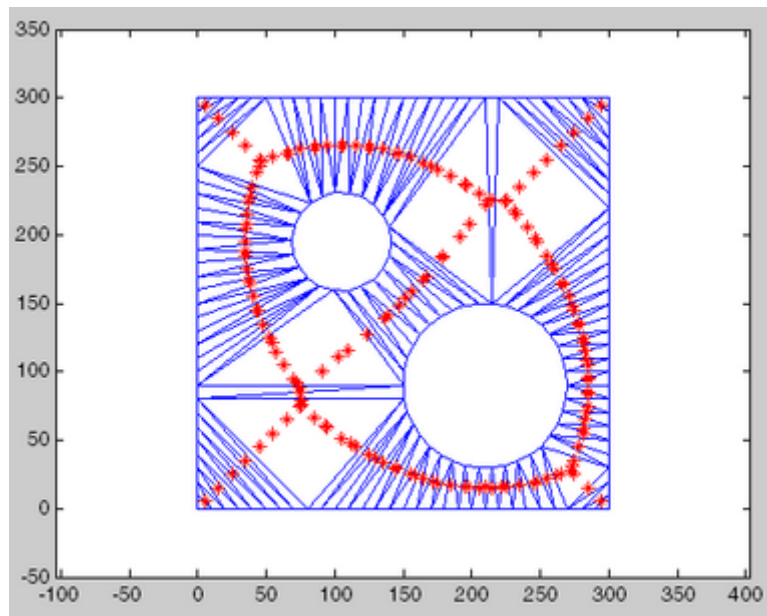
---

```
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

[incenters](#) | [DelaunayTri](#)

# cla

---

**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



**Purpose** Contour plot elevation labels

**Syntax**

```

clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')
clabel(C)
clabel(C,v)
clabel(C,'manual')
text_handles = clabel(...)
clabel(...,'PropertyName',propertyvalue,...)
clabel(...'LabelSpacing',points)

```

**Description**

The `clabel` function adds height labels to a 2-D contour plot.

`clabel(C,h)` rotates the labels and inserts them in the contour lines. The function inserts only those labels that fit within the contour, depending on the size of the contour.

`clabel(C,h,v)` creates labels only for those contour levels given in vector `v`, then rotates the labels and inserts them in the contour lines.

`clabel(C,h,'manual')` places contour labels at locations you select with a mouse. Press the left mouse button (the mouse button on a single-button mouse) or the space bar to label a contour at the closest location beneath the center of the cursor. 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.

`clabel(C)` adds labels to the current contour plot using the contour array `C` output from `contour`. The function labels all contours displayed and randomly selects label positions.

`clabel(C,v)` labels only those contour levels given in vector `v`.

`clabel(C,'manual')` places contour labels at locations you select with a mouse.

`text_handles = clabel(...)` returns the handles of text objects created by `clabel`. The `UserData` properties of the text objects contain the contour values displayed. If you call `clabel` without the `h` argument,

# clabel

---

`text_handles` also contains the handles of line objects used to create the '+' symbols.

`clabel(..., 'PropertyName', propertyvalue, ...)` enables you to specify text object property/value pairs for the label strings. (See Text Properties.)

`clabel(... 'LabelSpacing', points)` specifies the spacing between labels on the same contour line, in units of points (72 points equal one inch).

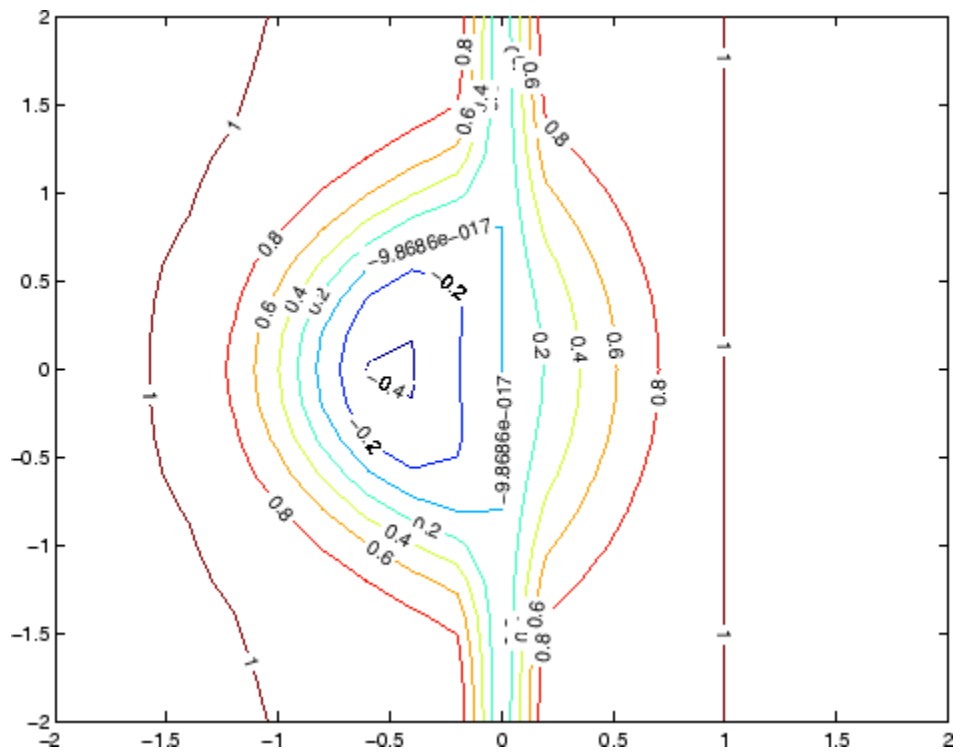
## Tips

When the syntax includes the argument `h`, this function rotates the labels and inserts them in the contour lines (see Examples). Otherwise, the labels are displayed upright and a '+' indicates which contour line the label is annotating.

## Examples

Generate, draw, and label a simple contour plot.

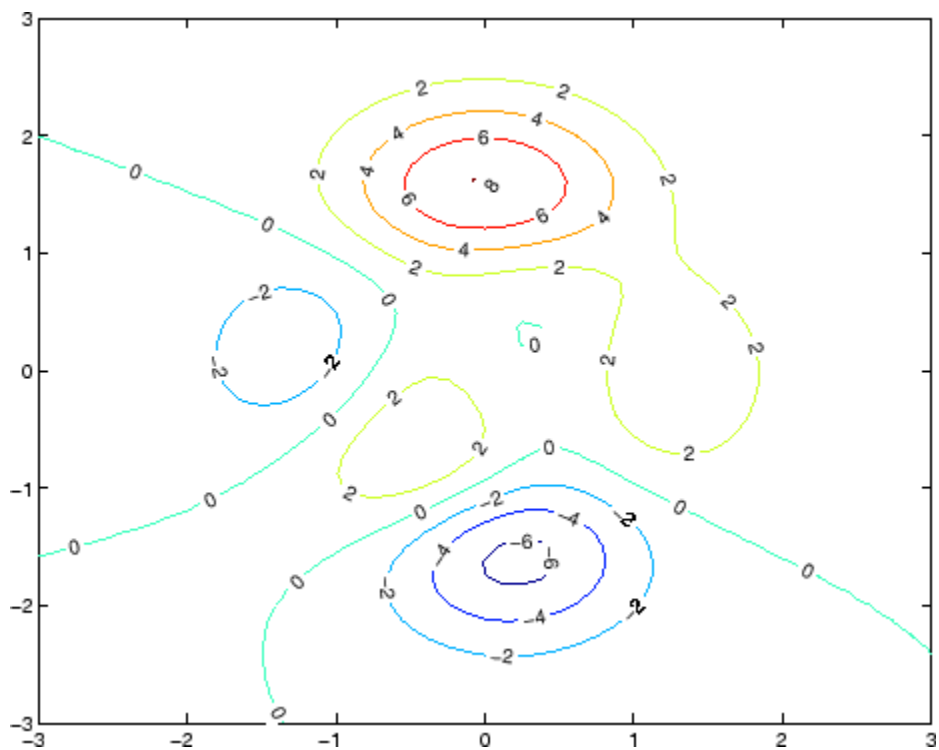
```
[x,y] = meshgrid(-2:.2:2);  
z = x.^exp(-x.^2-y.^2);  
[C,h] = contour(x,y,z);  
clabel(C,h);
```



Label a contour plot with label spacing set to 72 points (one inch).

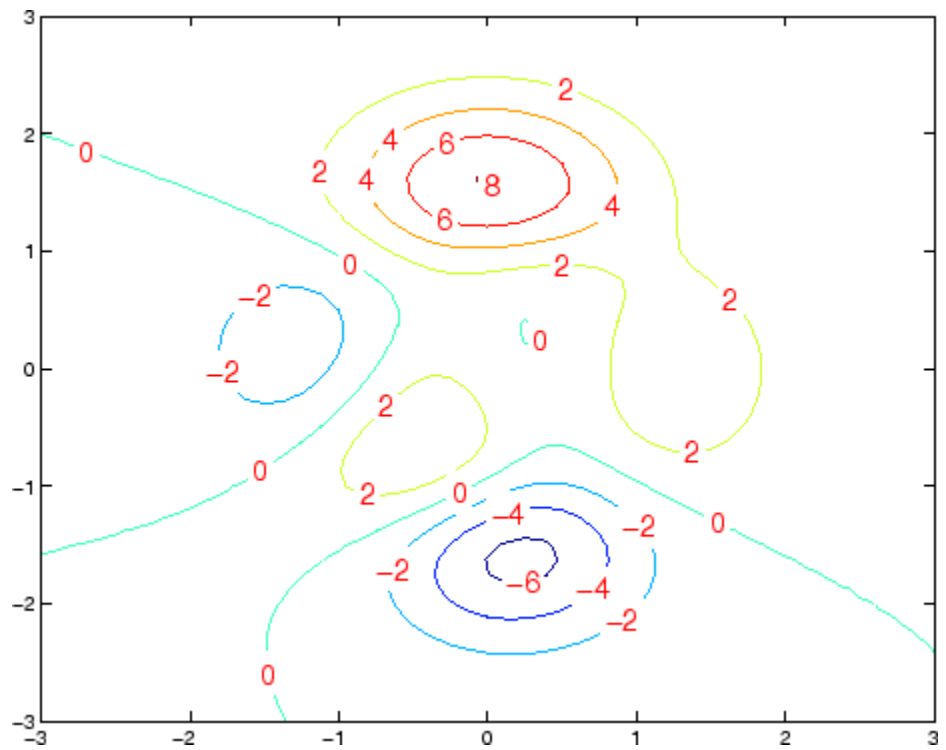
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```

# clabel



Label a contour plot with 15 point red text.

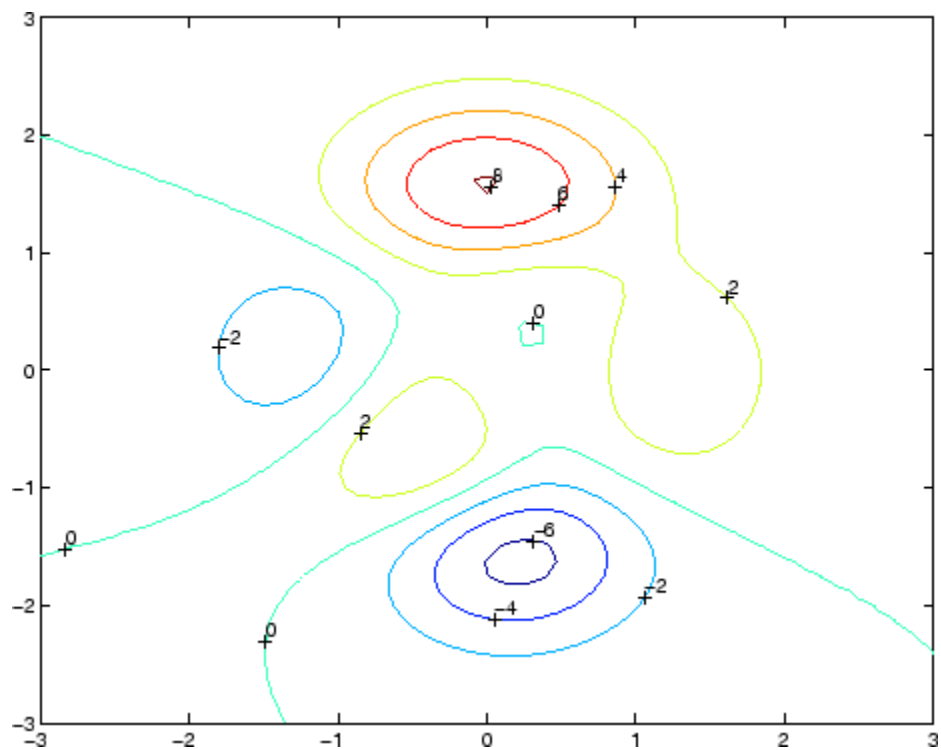
```
[x,y,z] = peaks;  
[C,h] = contour(x,y,z);  
clabel(C,h,'FontSize',15,'Color','r','Rotation',0)
```



Label a contour plot with upright text and '+' symbols indicating which contour line each label annotates.

```
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```

# clabel



## See Also

`contour` | `contourc` | `contourf`

## How To

- “Drawing Text in a Box”

**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.

# class

---

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”



**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, 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.

# classdef

---

`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`

<b>Purpose</b>	Remove items from workspace, freeing up system memory
<b>Syntax</b>	<pre>clear clear name1 ... nameN clear -regexp expr1 ... exprN clear ItemType</pre>
<b>Description</b>	<p><code>clear</code> removes all variables from the current workspace, releasing them from system memory. When called from within a function with persistent variables, <code>clear</code> reinitializes the persistent variables.</p> <p><code>clear name1 ... nameN</code> removes the variables, scripts, functions, or MEX-functions <code>name1 ... nameN</code> from memory. <code>clear</code> removes debugging breakpoints in MATLAB files.</p> <ul style="list-style-type: none"><li>• If function name is locked by <code>mlock</code>, then it remains in memory.</li><li>• If variable name is global, then <code>clear</code> removes it from the current workspace, but it remains in the global workspace.</li></ul> <p><code>clear -regexp expr1 ... exprN</code> clears all variables that match any of the regular expressions listed. This option only clears variables.</p> <p><code>clear ItemType</code> clears the types of items indicated by <code>ItemType</code>, such as <code>all</code>, <code>functions</code>, or <code>classes</code>.</p>
<b>Input Arguments</b>	<p><b>name1 ... nameN - Names of variables, scripts, functions, or MEX-functions to clear</b></p> <p><i>string</i></p> <p>Names of variables, scripts, functions, or MEX-functions to clear, specified as strings.</p> <p>Use a partial path to distinguish between different overloaded versions of a function. For example, <code>clear polynom/display</code> clears only the <code>display</code> method for <code>polynom</code> objects, leaving any other implementations in memory.</p>

# clear

---

## **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.

<b>ItemType</b>	<b>Items Cleared</b>
all	<p>Removes all variables from the base workspace, and all compiled scripts, functions, and MEX-functions from memory.</p> <p>If called from a function, <code>clear all</code> also clears the function workspace and the global workspace, leaving all workspaces empty.</p> <p><code>clear all</code> removes debugging breakpoints in MATLAB files and reinitializes persistent variables. When issued from the Command Window prompt, <code>clear all</code> also removes the Java packages import list.</p>
classes	<p>The same as <code>clear all</code>, but also clears MATLAB class definitions. <code>clear classes</code> 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).</p> <p>Call <code>clear classes</code> whenever you change a class definition, including when the number or names of properties, methods, or events or any of their attributes change.</p>

ItemType	Items Cleared
functions	Clears all the currently compiled functions, scripts, and MEX-functions from memory. If called from a function, <code>clear functions</code> 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, <code>clear global</code> also removes all global variables in the function workspace.  Use <code>clear global regexp expr1 ... exprN</code> 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 <code>clear all</code> , 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), <code>clear</code> issues a warning and does not remove the Java class definition. Issue a <code>clear java</code> command after modifying any files on the Java dynamic class path.
mex	The same as <code>clear functions</code> , but clears only MEX-functions, other than locked functions or functions that are currently in use. <code>clear mex</code> also clears breakpoints and persistent variables.
variables	Clears all variables from the workspace.

# clear

---

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”

# clearvars

---

**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?”

# clear (serial)

---

**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`

---

<b>Purpose</b>	Clear current figure window
<b>Syntax</b>	<pre>clf clf('reset') clf(fig) clf(fig, 'reset') figure_handle = clf(...)</pre>
<b>Description</b>	<p>clf deletes from the current figure all graphics objects whose handles are not hidden (i.e., their <code>HandleVisibility</code> property is set to on).</p> <p>clf('reset') deletes from the current figure all graphics objects regardless of the setting of their <code>HandleVisibility</code> property and resets all figure properties except <code>Position</code>, <code>Units</code>, <code>PaperPosition</code>, and <code>PaperUnits</code> to their default values.</p> <p>clf(fig) or clf(fig, 'reset') clears the single figure with handle fig.</p> <p>figure_handle = clf(...) returns the handle of the figure. This is useful when the figure <code>IntegerHandle</code> property is off because the noninteger handle becomes invalid when the reset option is used (i.e., <code>IntegerHandle</code> is reset to on, which is the default).</p>
<b>Tips</b>	The clf command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the <code>HandleVisibility</code> setting of callback. This means that when issued from within a callback routine, clf deletes only those objects whose <code>HandleVisibility</code> property is set to on.
<b>Alternatives</b>	Use <b>Clear Figure</b> from the figure window's <b>Edit</b> menu to clear the contents of a figure. You can also create a <i>desktop shortcut</i> to clear the current figure with one mouse click. See “Create Shortcuts to Rerun Commands”.
<b>See Also</b>	cla   clc   hold   reset

# clipboard

---

**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`

**Purpose** Current 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
```

**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`

# close

---

**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`

# Tiff.close

---

**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”

**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`

# FTP.close

---

**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`

<b>Purpose</b>	Close file after writing video data
<b>Syntax</b>	<code>close(writerObj)</code>
<b>Description</b>	<code>close(writerObj)</code> closes the file associated with object <code>writerObj</code> . The object remains in the workspace. If you call the <code>open</code> method after closing, <code>open</code> discards any existing contents of the file.
<b>Input Arguments</b>	<b>writerObj</b> VideoWriter object created by the <code>VideoWriter</code> function.
<b>Examples</b>	<b>AVI File from Animation</b> Write a sequence of frames to a compressed AVI file, <code>peaks.avi</code> .  Prepare the new file.  <pre>writerObj = VideoWriter('peaks.avi'); open(writerObj);</pre> Generate initial data and set axes and figure properties.  <pre>Z = peaks; surf(Z); axis tight set(gca, 'nextplot', 'replacechildren'); set(gcf, 'Renderer', 'zbuffer');</pre> Setting the <code>Renderer</code> property to <code>zbuffer</code> or <code>Painters</code> works around limitations of <code>getframe</code> with the OpenGL renderer on some Windows systems.  Create a set of frames and write each frame to the file.  <pre>for k = 1:20     surf(sin(2*pi*k/20)*Z,Z)     frame = getframe;     writeVideo(writerObj,frame);</pre>

# VideoWriter.close

---

```
end  
close(writerObj);
```

## See Also

[writeVideo](#) | [VideoWriter](#) | [open](#)

**Purpose** Default figure close request function

**Syntax** `closereq`

**Description** `closereq` deletes the current figure.

**See Also** | `CloseRequestFcn`

# cmopts

---

**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 <sup>®</sup> software from IBM <sup>®</sup> Rational <sup>®</sup>	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 <sup>®</sup> and ChangeMan <sup>®</sup> software	UNIX platforms
rcs	Revision Control System (RCS)	UNIX platforms
Any SCC-compliant source control system, for example, Microsoft Visual SourceSafe or	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”

# cmpermute

---

## 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`

---

<b>Purpose</b>	Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image
<b>Syntax</b>	<pre>[Y,newmap] = cmunique(X,map) [Y,newmap] = cmunique(RGB) [Y,newmap] = cmunique(I)</pre>
<b>Description</b>	<p>[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.</p> <p>[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.</p> <hr/> <p><b>Note</b> newmap might be very large, because the number of entries can be as many as the number of pixels in RGB.</p> <hr/> <p>[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.</p>
<b>Class Support</b>	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.
<b>Examples</b>	<ol style="list-style-type: none"><li>1 Use the magic function to create a sample 4-by-4 image that uses every value in the range between 1 and 16. <pre>X = magic(4)</pre></li></ol>

# cmunique

---

```
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](#)

# colamd

---

**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`.

<code>stats(1)</code>	Number of dense or empty rows ignored by <code>colamd</code>
<code>stats(2)</code>	Number of dense or empty columns ignored by <code>colamd</code>
<code>stats(3)</code>	Number of garbage collections performed on the internal data structure used by <code>colamd</code> (roughly of size <code>2.2*nnz(S) + 4*m + 7*n</code> integers)
<code>stats(4)</code>	0 if the matrix is valid, or 1 if invalid
<code>stats(5)</code>	Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists

<code>stats(6)</code>	Last seen duplicate or out-of-order row index in the column index given by <code>stats(5)</code> , or 0 if no such row index exists
<code>stats(7)</code>	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:

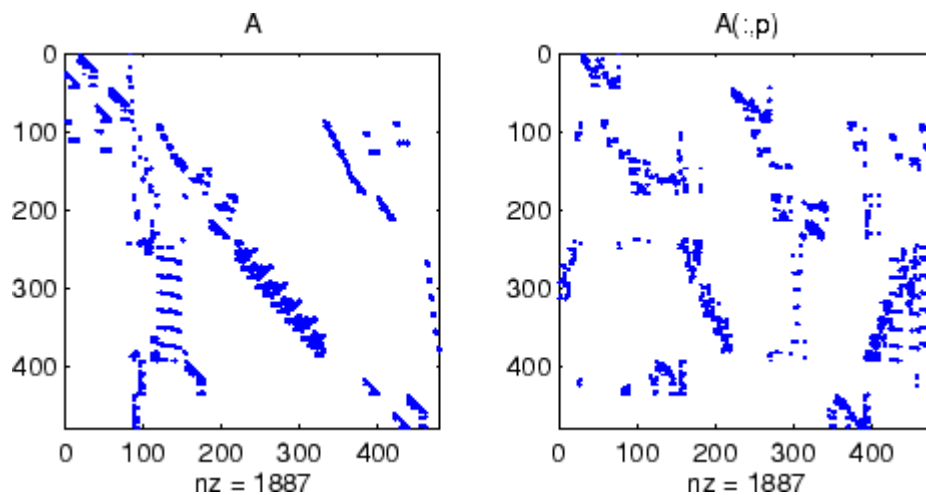
- If a row index appears two or more times in the same column, colamd ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, colamd sorts each column of its internal copy of the matrix `S` (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.

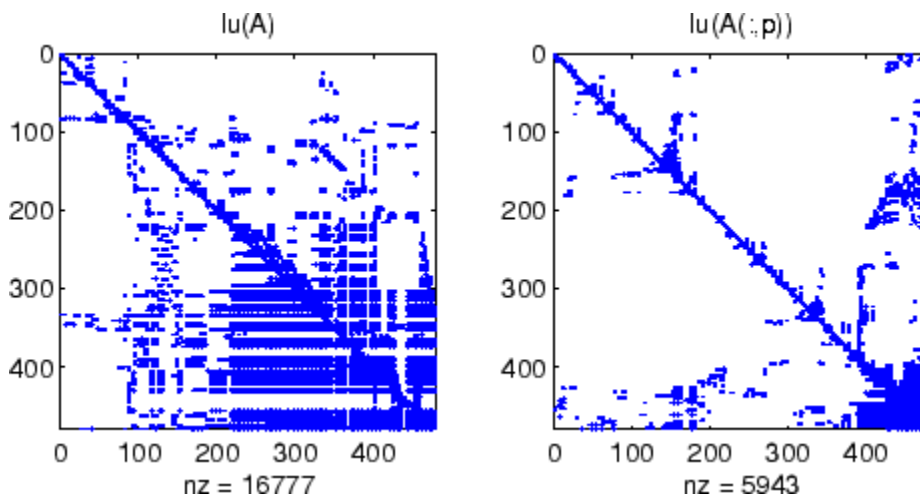
```
load west0479
A = west0479;
p = colamd(A);
subplot(1,2,1), spy(A,4), title('A')
subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
```



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

# colorbar

---

**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

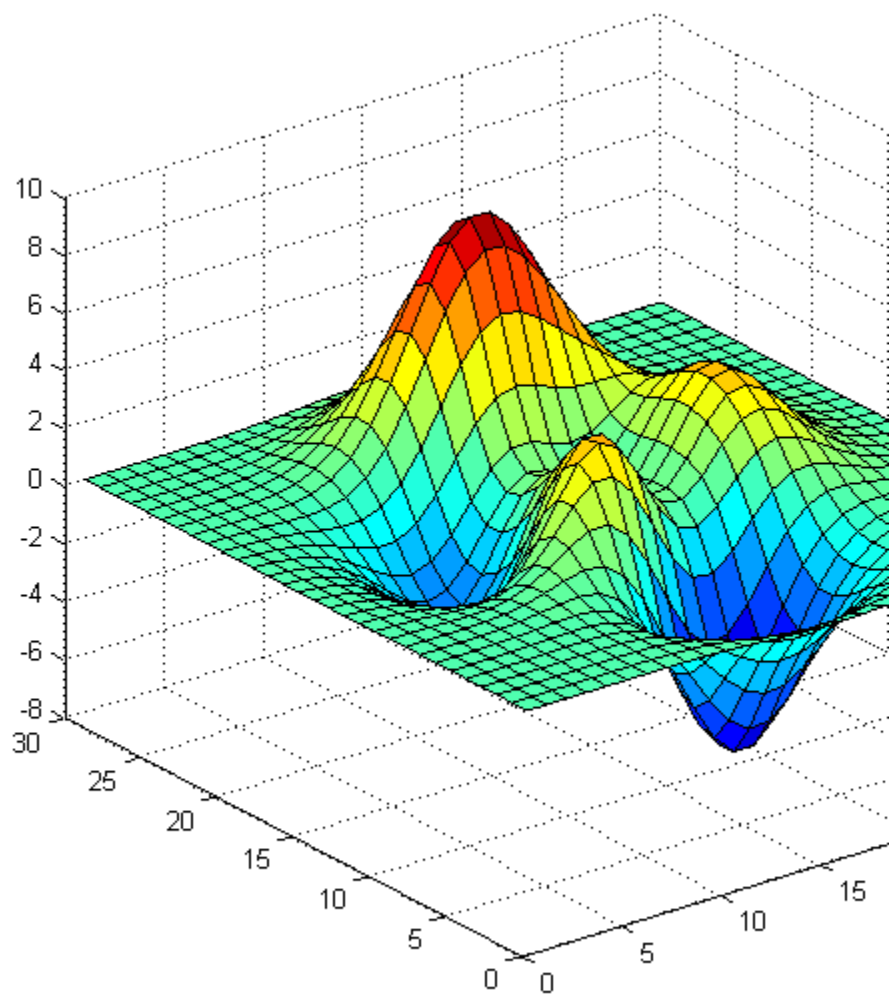
```
cbar_handle = findobj(figure_handle, 'tag', 'Colorbar')
```

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'})
```

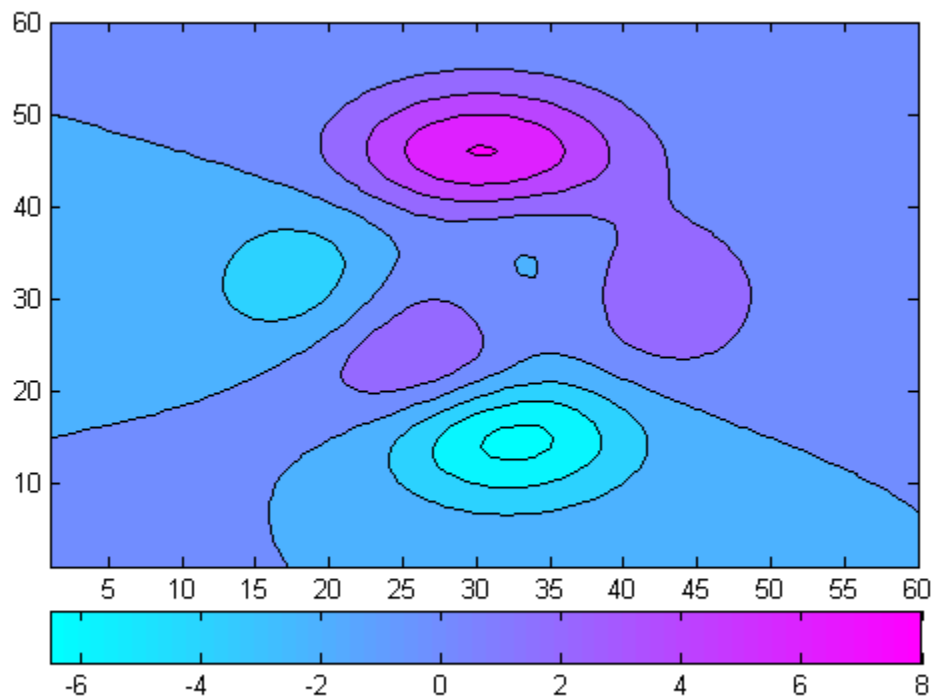


# colorbar


---

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`

# colordef

---

**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
```

# colormap

---

**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  $k$ th row of the colormap defines the  $k$ th color, where  $\text{map}(k, :) = [r(k) \ g(k) \ b(k)]$  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.

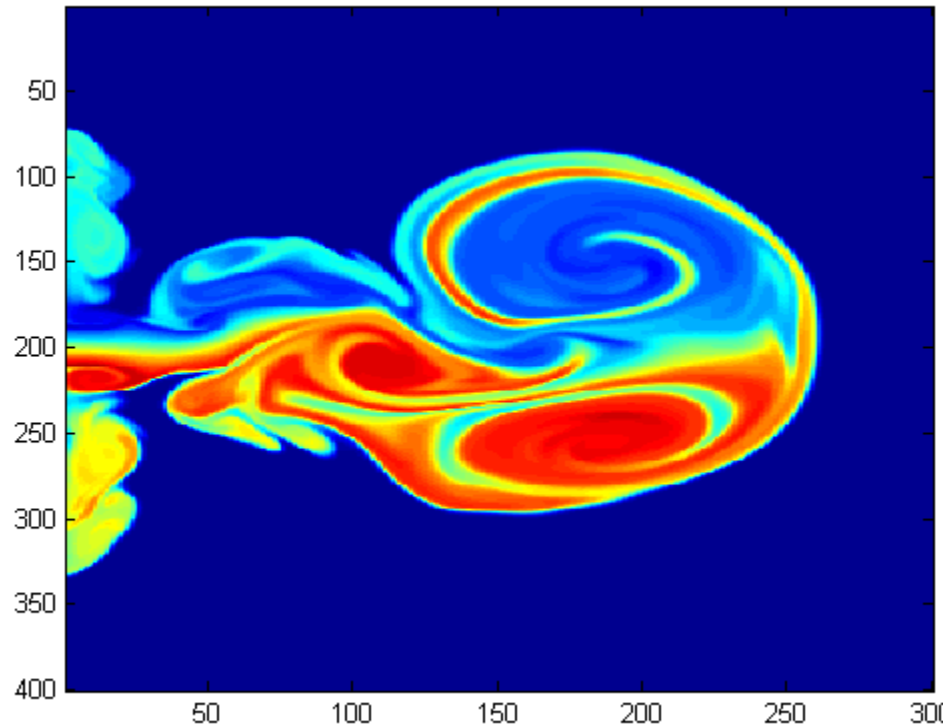
- `hot` varies smoothly from black through shades of red, orange, and yellow, to white.
- `hsv` varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for 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 2-940.
- `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)
```



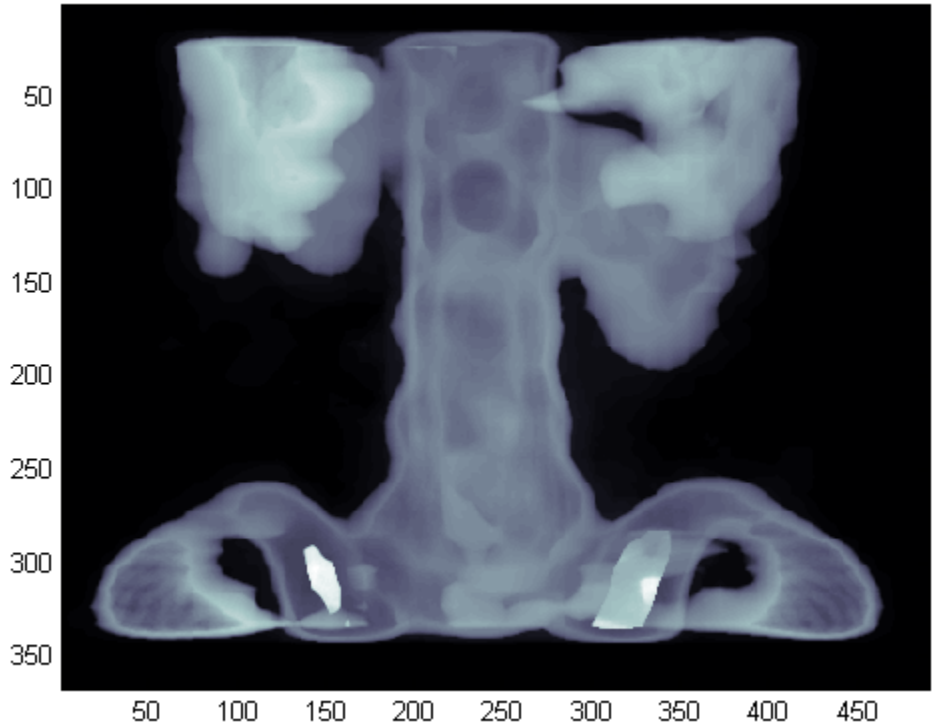
---

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
```

# colormap

---



## 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”

# colormapeditor

---

**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	<b>Tools &gt; Standard Colormaps</b>
Add a node	Click below the corresponding cell in the colormap strip.
Select a node	Left-click the node.

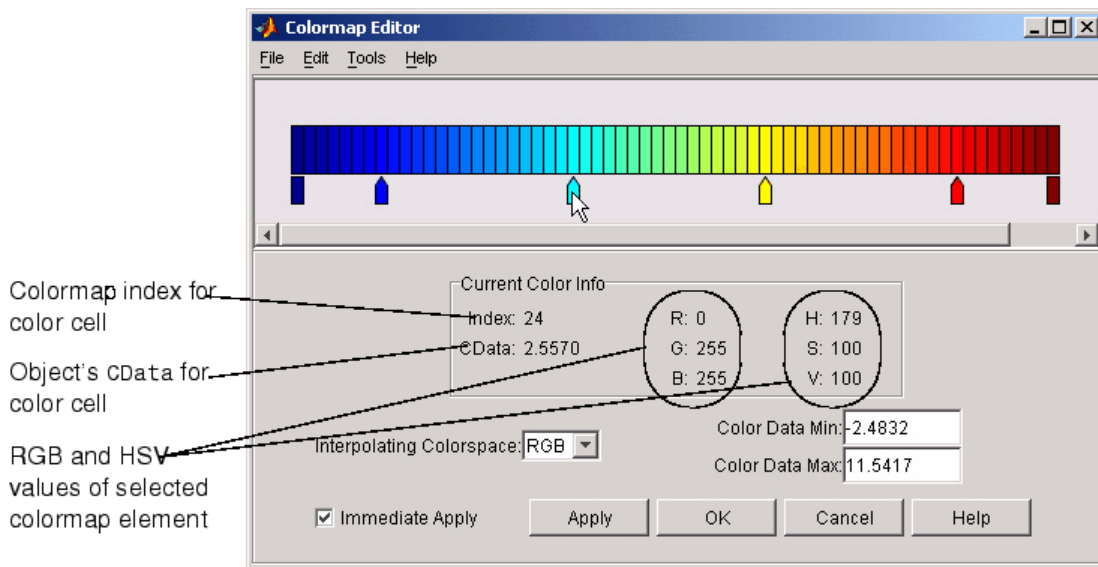


Operation	How to Perform
Select multiple nodes	Adjacent: left-click first node, <b>Shift+click</b> the last node. Nonadjacent: left-click first node, <b>Ctrl+click</b> subsequent nodes.
Move a node	Select and drag with the mouse or select and use the left and right arrow keys.
Move multiple nodes	Select 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 <b>Delete</b> key, or select <b>Delete</b> from the <b>Edit</b> menu, or type <b>Ctrl+x</b> .
Delete multiple nodes	Select the nodes and then press the <b>Delete</b> key, or select <b>Delete</b> from the <b>Edit</b> menu, or type <b>Ctrl+x</b> .
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

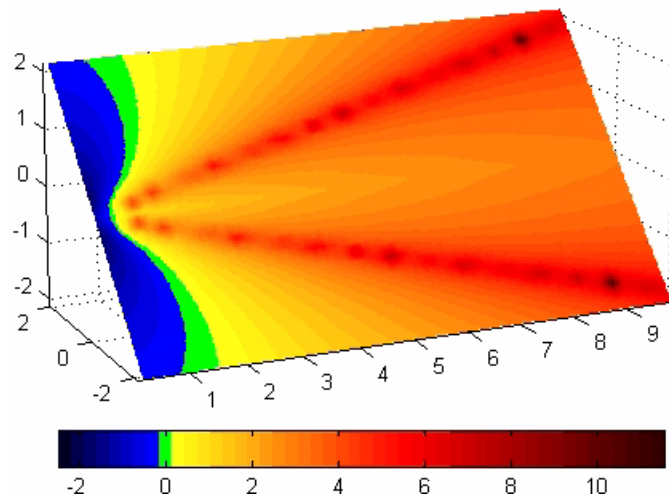
This example modifies a default MATLAB colormap so that ranges of data values are displayed in specific ranges of color. The graph is a slice plane illustrating a cross section of fluid flow through a jet nozzle. See the `slice` reference page for more information on this type of graph.

### 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.



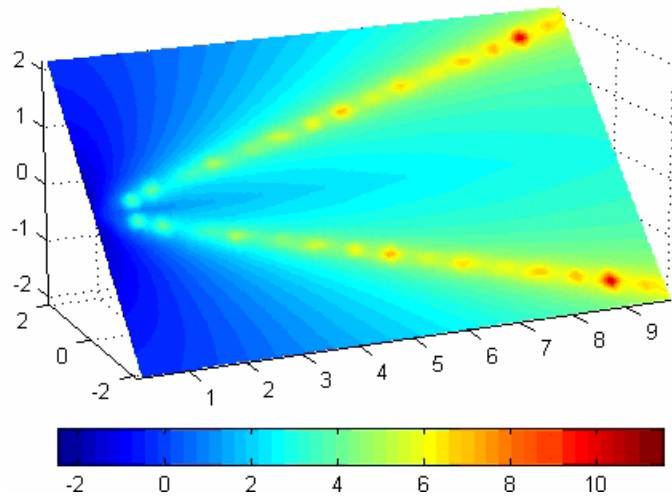
## Running the Example

---

**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).

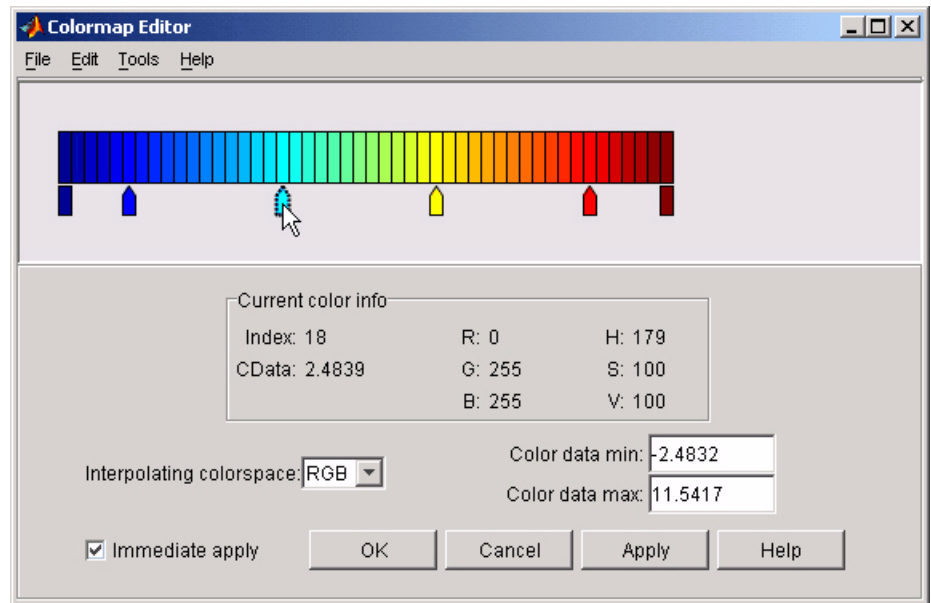
---

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).

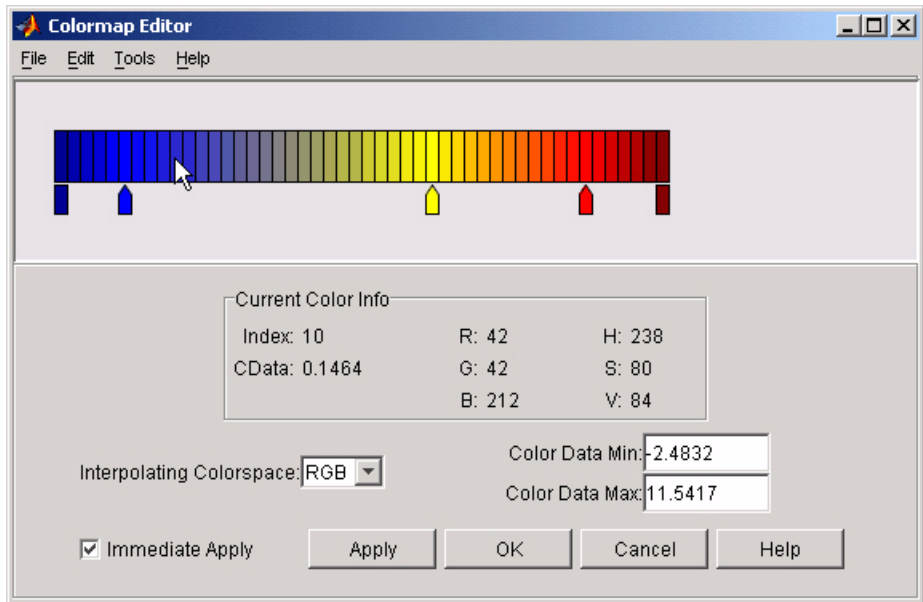


- 1 Start the colormap editor using the `colormapeditor` command. The color map editor displays the current figure's colormap, as shown in the following picture.

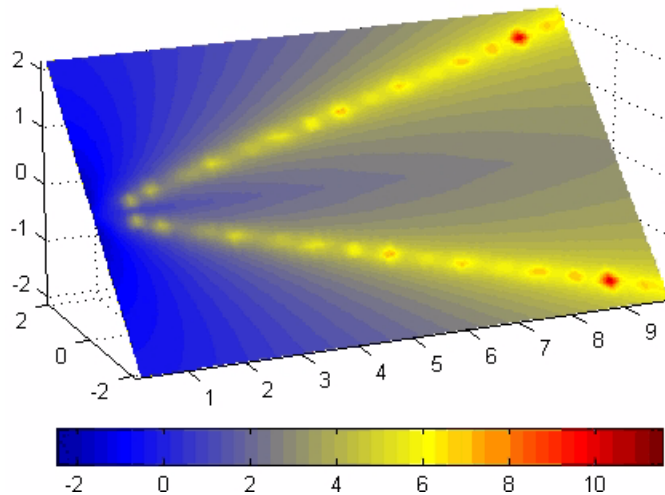
# colormapeditor



- 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.



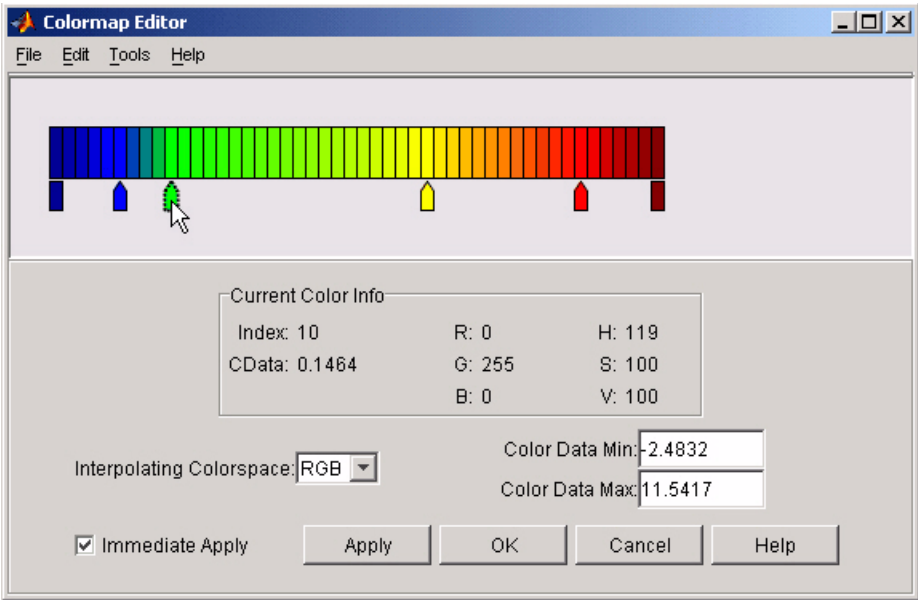
The **Immediate Apply** box is checked, so the graph displays the results of the changes made to the colormap.



- 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.

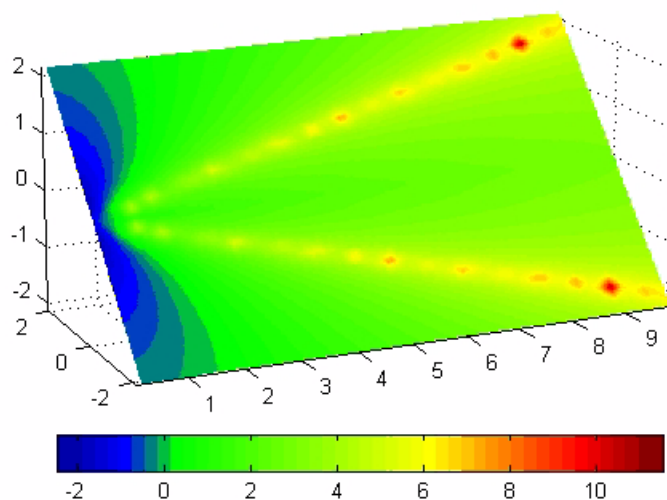




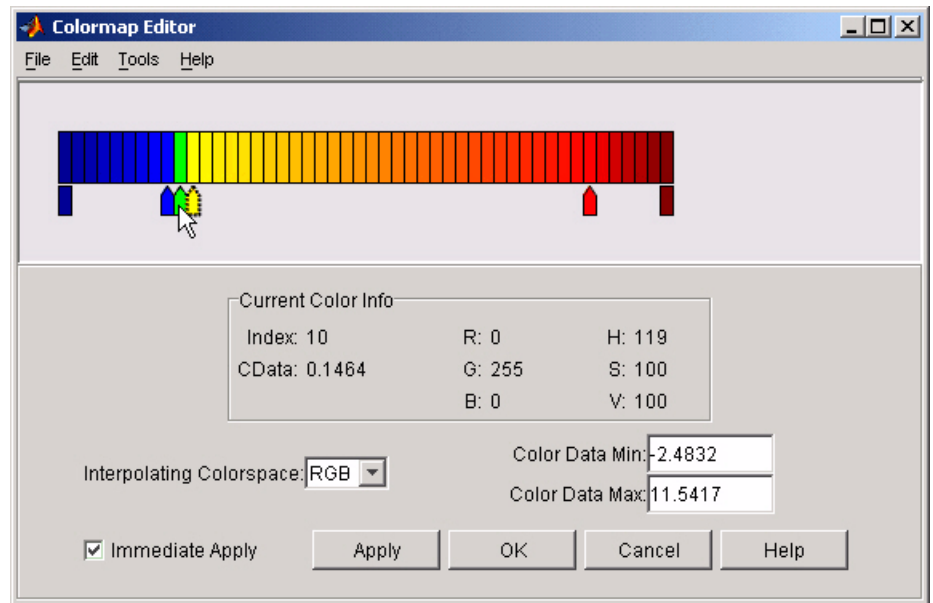
The graph continues to update to the modified colormap.

## colormapeditor

---



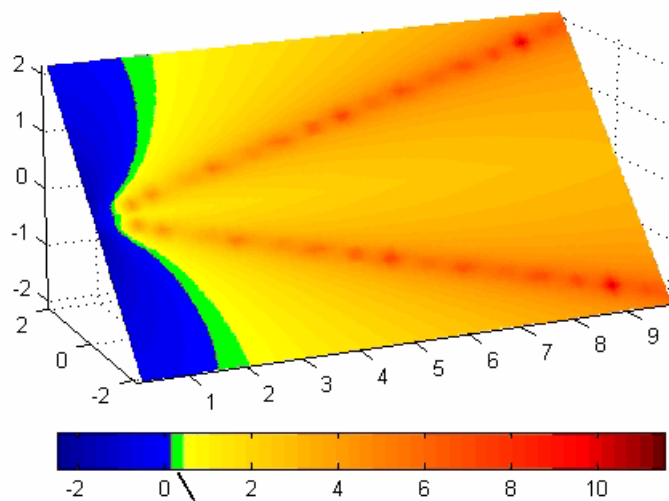
- 4** In the current state, the colormap colors are interpolated from the green node to the yellowish node about 20 cells away. We actually want only the single cell that is centered around zero to be colored green. To limit the color green to one cell, move the blue and yellow node pointers next to 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.

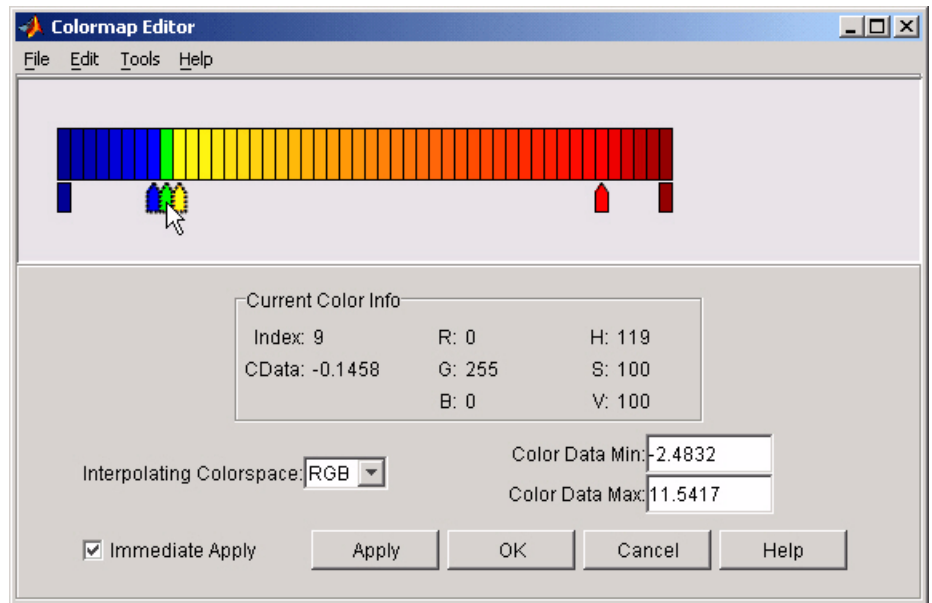
## colormapeditor

---



Note that green cell is not centered around zero.

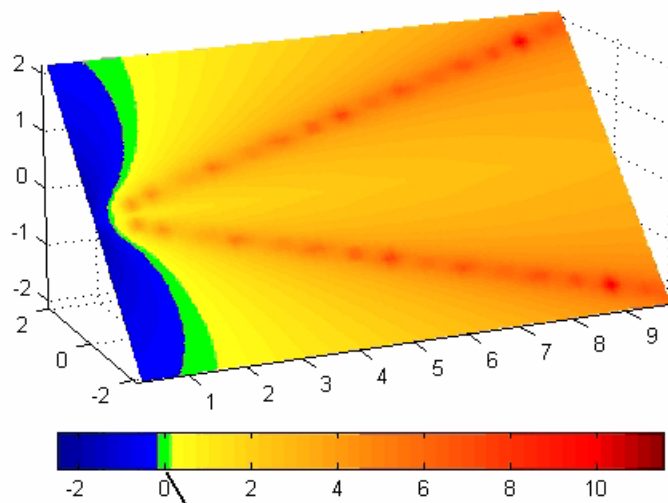
To recenter the green cell around zero, select the blue, green, and yellow node pointers (left-click blue, **Shift+click** yellow) and move them as a group using the left arrow key. Watch the colorbar in the figure window to see when the green color is centered around zero.



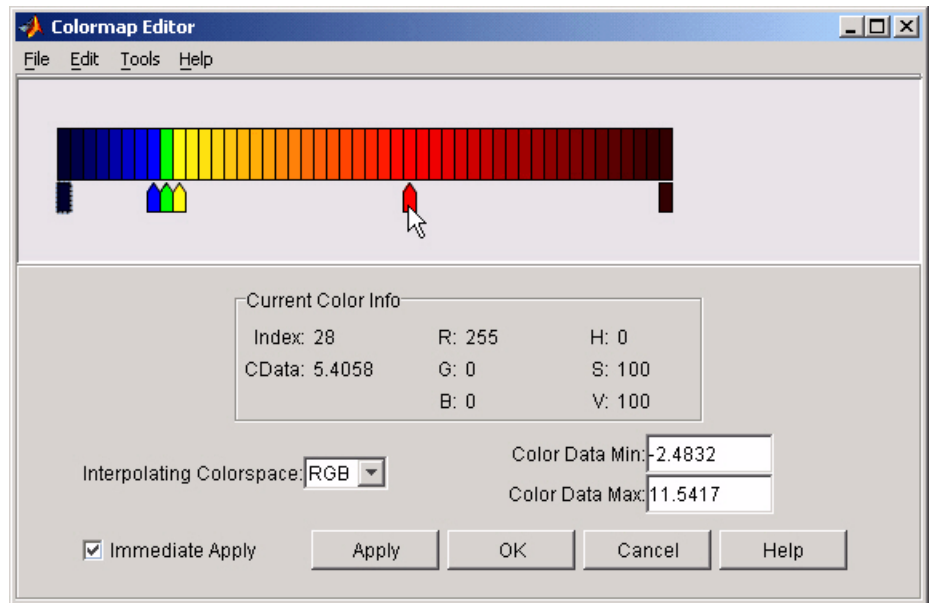
The slice plane now has the desired range of colors for negative, zero, and positive data.

# colormapeditor

---

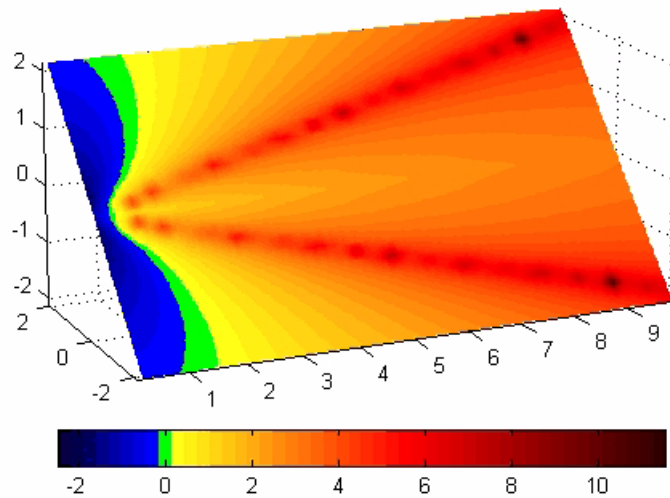


- 6 Increase the orange-red coloring in the slice by moving the red node pointer toward the yellow node.



- 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”

# ColorSpec (Color Specification)

---

**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](#)

# colperm

---

**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`

# Combine

---

**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](#)

**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.

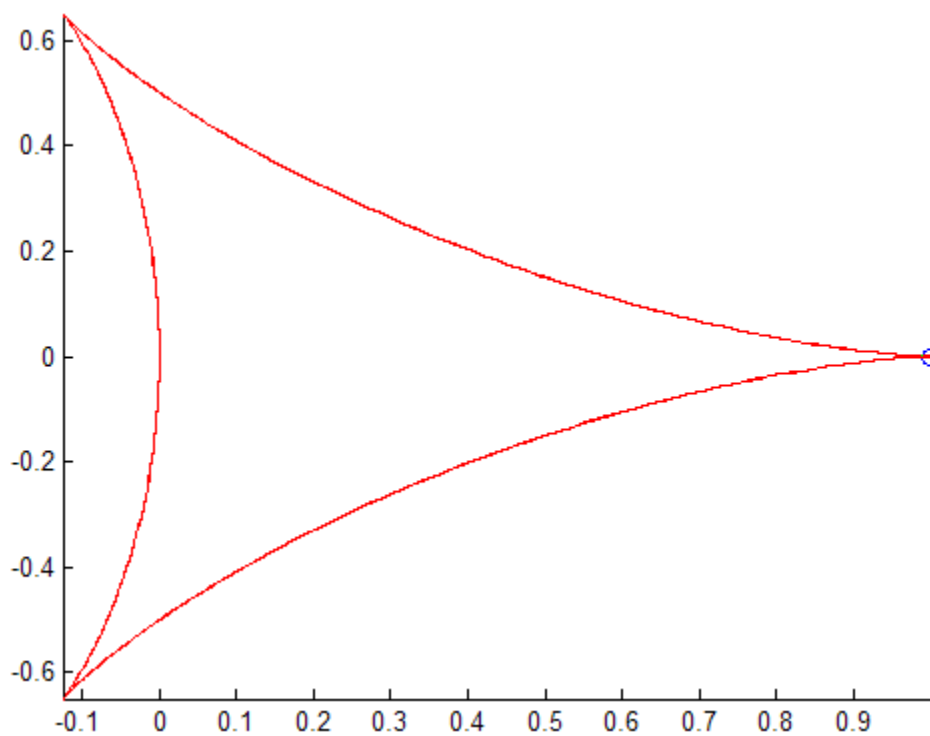
`comet(x,y)` displays a comet graph of vector `y` versus vector `x`.

`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)`.

`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”

<b>Purpose</b>	Open Command Window, or select it if already open
<b>Syntax</b>	<code>commandwindow</code>
<b>Description</b>	<code>commandwindow</code> opens the MATLAB Command Window when it is closed, and selects the Command Window when it is open.
<b>Tips</b>	<p>To determine the number of columns and rows that display in the Command Window, given its current size, use</p> <pre>get(0, 'CommandWindowSize')</pre> <p>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.</p>
<b>See Also</b>	<code>commandhistory</code>   <code>input</code>   <code>inputdlg</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Optimize Desktop Layout for Limited Screen Space”</li><li>• “Set Command Window Preferences”</li></ul>

# companion

---

**Purpose** Companion matrix

**Syntax** `A = companion(u)`

**Description** `A = companion(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 `companion(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 = companion(u)
A =
    0     7    -6
    1     0     0
    0     1     0
```

The eigenvalues are the polynomial roots:

```
eig(companion(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(...,LineStyle)
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(...,LineStyle)` draws a compass graph using the line type, marker symbol, and color specified by `LineStyle`.

`compass(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

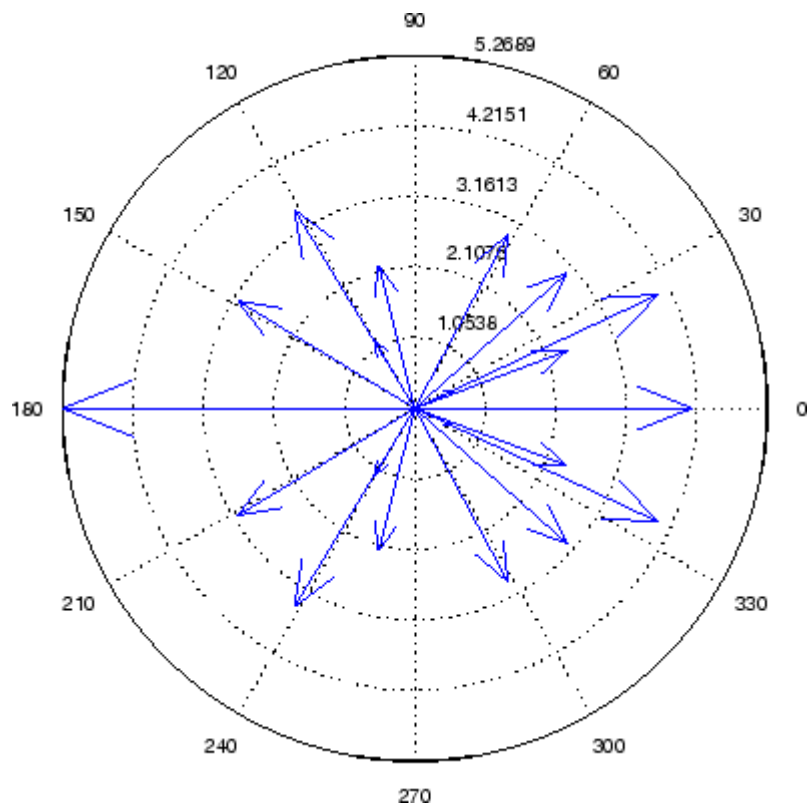
`h = compass(...)` returns handles to line objects.

**Examples** Draw a compass graph of the eigenvalues of a matrix.

```
Z = eig(randn(20,20));
compass(Z)
```

# compass

---



## See Also

`feather` | `polar` | `LineSpec` | `quiver` | `rose`

## How To

- “Compass Plots”

**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`

# complex

---

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`



- 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”

# Tiff.computeTile

---

**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”

# computer

**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	archstr	maxsize	endian	ispc	isunix	ismac	
Microsoft Windows	32-bit	PCWIN	win32	2 <sup>31</sup> - 1	L	1	0	0
	64-bit	PCWIN64	win64	2 <sup>48</sup> - 1	L	1	0	0
Linux	64-bit	GLNXA64	glnxa64	2 <sup>48</sup> - 1	L	0	1	0
Apple Macintosh	64-bit	MACI64	maci64	2 <sup>48</sup> - 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`

# cond

---

**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:

$\text{norm}(X,p) * \text{norm}(\text{inv}(X),p)$

If $p$ is...	Then <code>cond(X,p)</code> 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`

<b>Purpose</b>	Condition number with respect to eigenvalues
<b>Syntax</b>	<pre>c = condeig(A) [V,D,s] = condeig(A)</pre>
<b>Description</b>	<p><code>c = condeig(A)</code> returns a vector of condition numbers for the eigenvalues of <code>A</code>. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.</p> <p><code>[V,D,s] = condeig(A)</code> is equivalent to</p> <pre>[V,D] = eig(A); s = condeig(A);</pre> <p>Large condition numbers imply that <code>A</code> is near a matrix with multiple eigenvalues.</p>
<b>See Also</b>	<code>balance</code>   <code>cond</code>   <code>eig</code>

# condest

---

**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  $\text{norm}(A*v,1) = \text{norm}(A,1)*\text{norm}(v,1)/c$ .

---

**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  $\|A^{-1}\|_1$  without computing  $A^{-1}$ . This is posed as the convex but nondifferentiable optimization problem  $\max \|A^{-1}\mathbf{x}\|_1$  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)
```

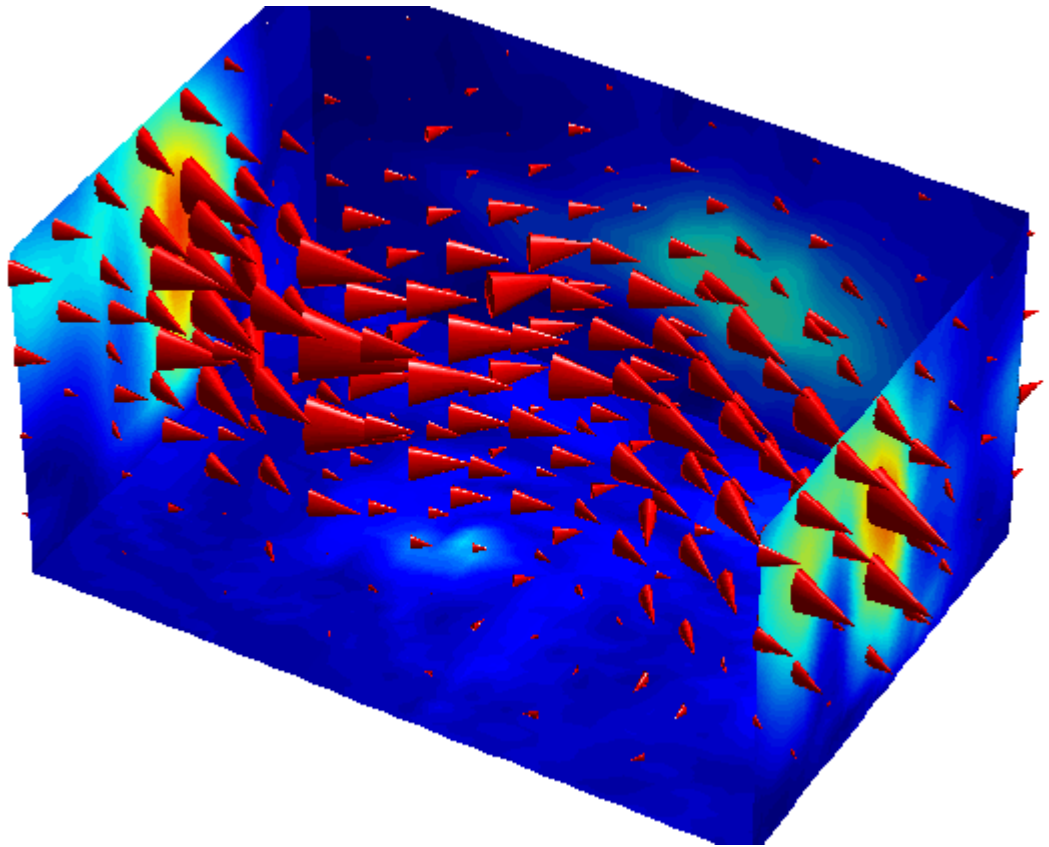
```
% The light source affects both the slice planes (surfaces) and the cone plots
% (patches). However, you can set the lighting characteristics of each independently:

% 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)
```

# coneplot

---



## See Also

[isosurface](#) | [patch](#) | [reducevolume](#) | [smooth3](#) | [streamline](#) | [stream2](#) | [stream3](#) | [subvolume](#)

## Tutorials

- “Overview of Volume Visualization”

**Purpose**           Complex conjugate

**Syntax**            $ZC = \text{conj}(Z)$

**Description**      $ZC = \text{conj}(Z)$  returns the complex conjugate of the elements of  $Z$ .

**Algorithms**       If  $Z$  is a complex array:

$$\text{conj}(Z) = \text{real}(Z) - i*\text{imag}(Z)$$

**See Also**         *i* | *j* | *imag* | *real*

# continue

---

**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`.

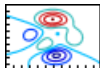
**Examples** Count the number of lines of code in the file `magic.m`, skipping all blank lines and comments:

```
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(...,LineStyle)
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  $[m,n] = \text{size}(Z)$ .

`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.

# contour

---

When  $X$  and  $Y$  are matrices, they must be the same size as  $Z$  and must be monotonically increasing.

`contour(...,LineStyle)` draws the contours using the line type and color specified by `LineStyle`. `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  $x$ ,  $y$  coordinates and contour levels for contour lines derived by the low-level `contourc` function, 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  $Y$ .

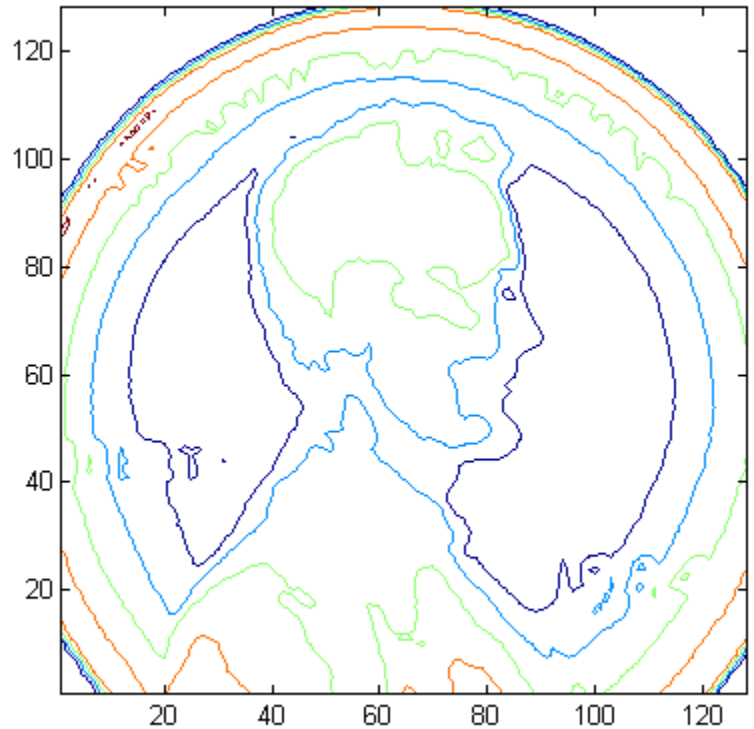
## Examples

### 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:



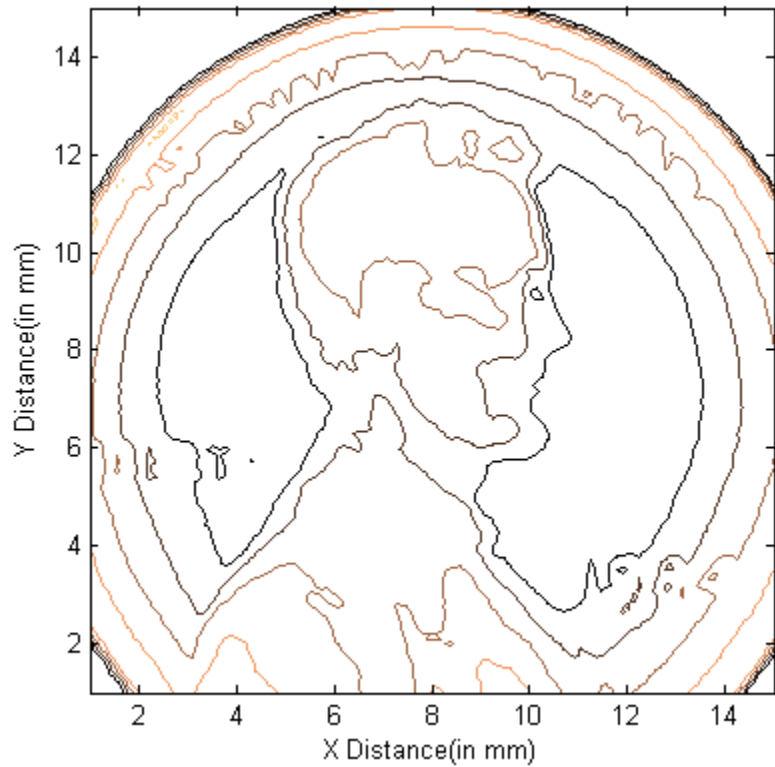
### Contour Graph of a Matrix

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

```
contour(x,y,flipud(P));  
axis square;  
colormap('Copper');  
xlabel('X Distance(in mm)');  
ylabel('Y Distance(in mm)');
```



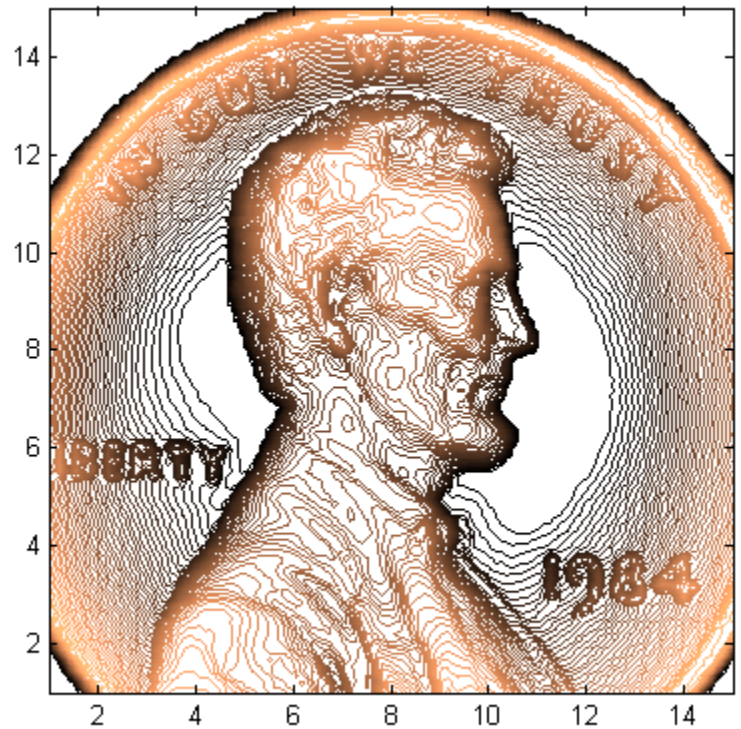
## Contour Graph with Specified Number of Contour Lines

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:

$$z = xe^{-(x_2 - y_2)}$$

## contour

---

over the range  $-2 \leq x \leq 2$ ,  $-2 \leq y \leq 3$ .

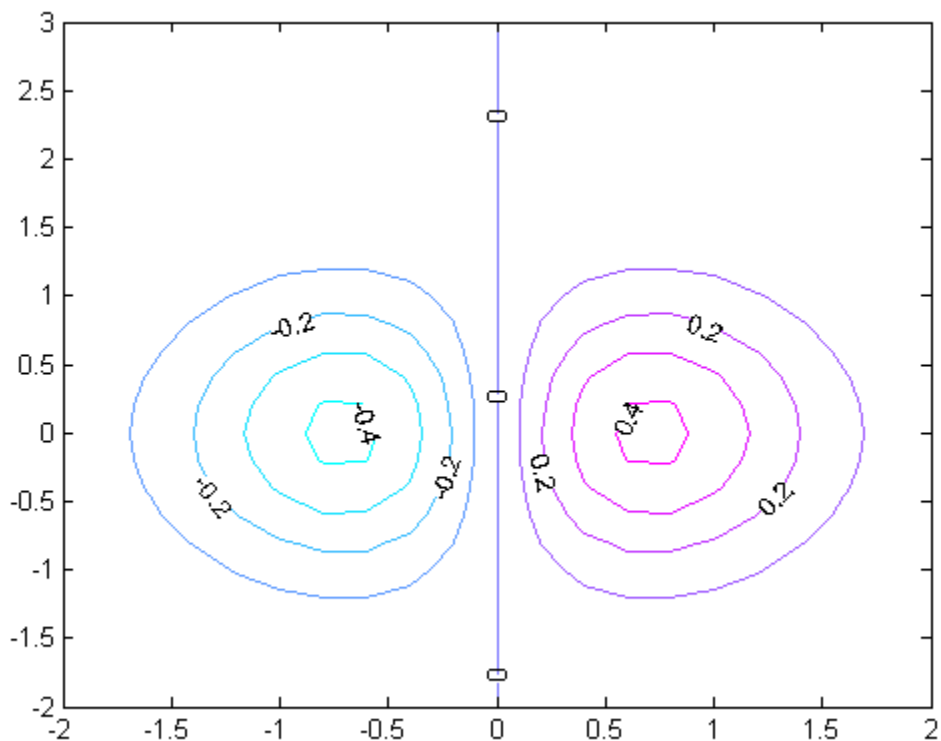
Evaluate the function to create matrix, Z. Use the `meshgrid` function to generate the values used to evaluate the function within the specified range:

```
[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
```

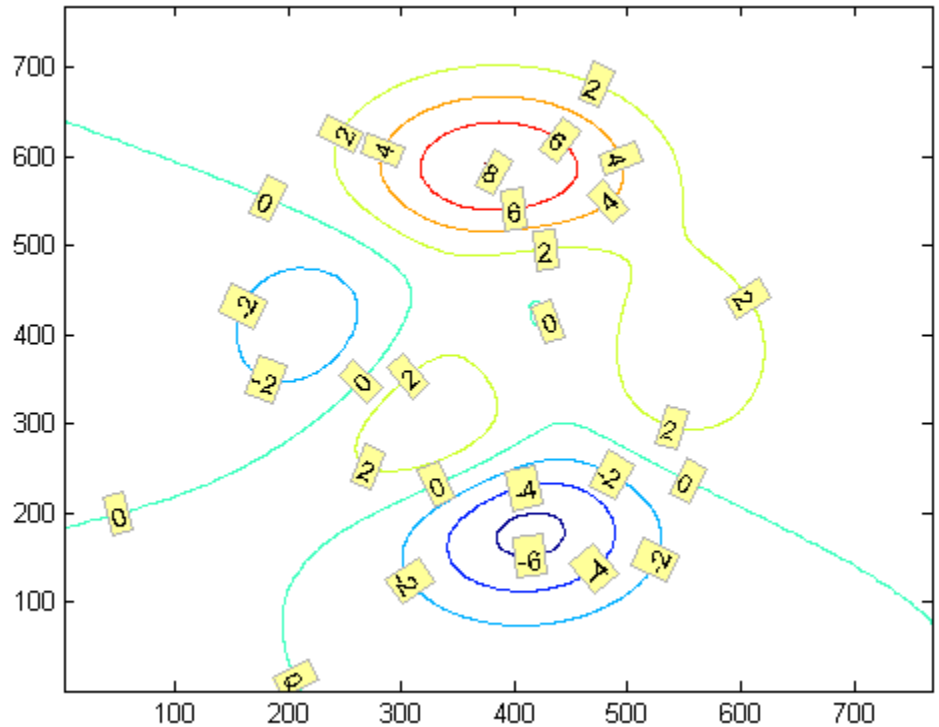


### Smoothing Contour Data

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])
```

# contour



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 NaNs 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

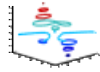
- “Contour Plots”

# contour3

---

## 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(...,LineStyle)
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(...,LineStyle)` draws the contour lines using the line type and color specified by `LineStyle`. `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  $x$ ,  $y$  coordinates and contour levels for contour lines derived by the low-level `contourc` function, 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 `LineStyle` 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 `LineStyle`, 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

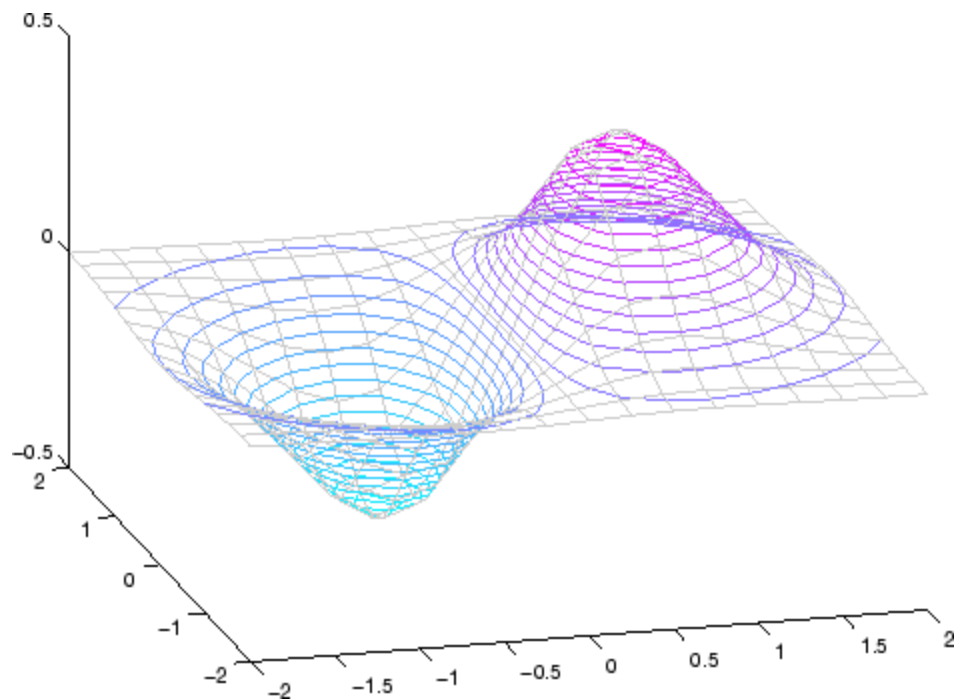
Plot the three-dimensional contour of a function and superimpose a surface plot to enhance visualization of the function.

```
[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
```

## contour3

---

```
view(-15,25)  
colormap cool
```



For more examples using `contour3`, see “Contour Plots”.

### See Also

`contour` | `contourc` | `contourf` | `meshc` | `meshgrid` | `surf` | `surf` | `contourgroup` properties

### How To

- “Contour Plots”

**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)
```

# contourc

---

```
dim1 ydata(1) ydata(2) ... ydata(dim1) dim2 ydata(1) ydata(2) ... ydat
```

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(...,LineStyle)
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

---

`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(...,LineStyle)` draws the contour lines using the line type and color specified by `LineStyle`. `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 `gckaxes_handle` instead of `gca`.

`[C,h] = contourf(...)` returns a contour matrix, `C`, that contains the  $x$ ,  $y$  coordinates and contour levels for contour lines derived by the low-level `contourc` function, 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`.

NaNs in the  $Z$ -data leave white holes with black borders in the contour plot.

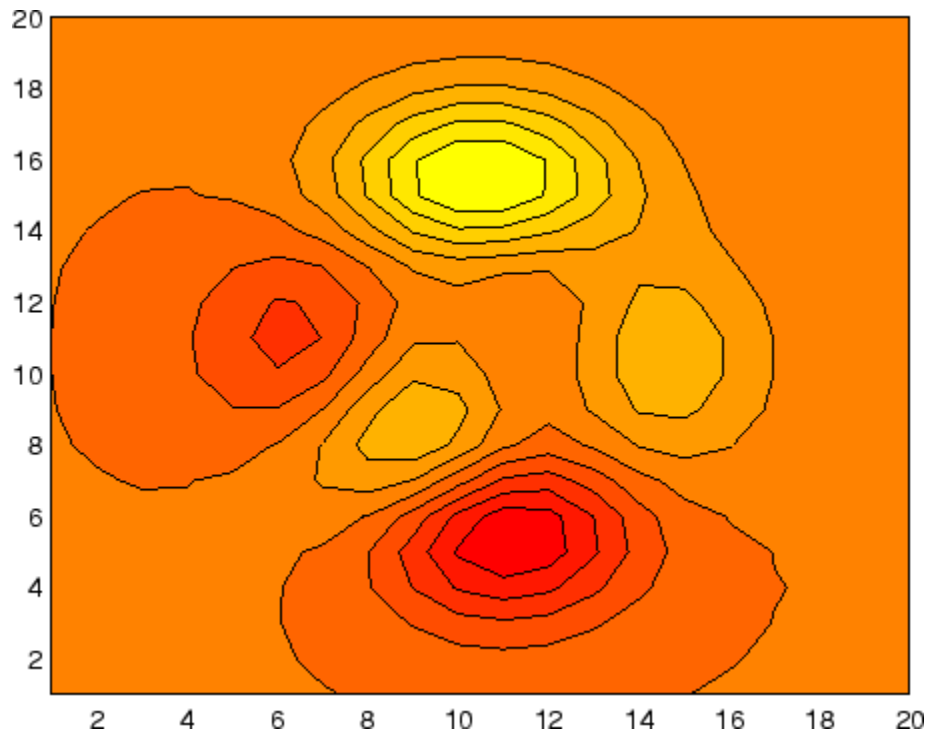
If  $X$  or  $Y$  is irregularly spaced, `contourf` calculates contours using a regularly spaced contour grid, and then transforms the data to  $X$  or  $Y$ .

## 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”

# Contourgroup Properties

---

**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.

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
<code>on</code>	Include the contourgroup object in a legend as one entry, but not its children objects
<code>off</code>	Do not include the contourgroup or its children in a legend (default)
<code>children</code>	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

## Contourgroup Properties

---

*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)...]
```

# Contourgroup Properties

---

That is:

$$C = [C(1) \ C(2) \ \dots \ C(I) \ \dots \ C(N)]$$

where N is the number of contour levels, and:

$$C(i) = [ \text{level}(i) \ x(1) \ x(2) \ \dots \ x(\text{numel}(i)); \\ \text{numel}(i) \ y(1) \ y(2) \ \dots \ y(\text{numel}(i)) ];$$

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.

# Contourgroup Properties

---

- 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.

# Contourgroup Properties

---

- `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}`

# Contourgroup Properties

---

## *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

# Contourgroup Properties

---

*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

# Contourgroup Properties

---

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 2-997.

`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 2-997 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

# Contourgroup Properties

---

`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 2-997 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

# Contourgroup Properties

---

*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

# Contourgroup Properties

---

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”

# contourslice

---

**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  $[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)$ , where  $[m,n,p] = \text{size}(v)$ .

`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.

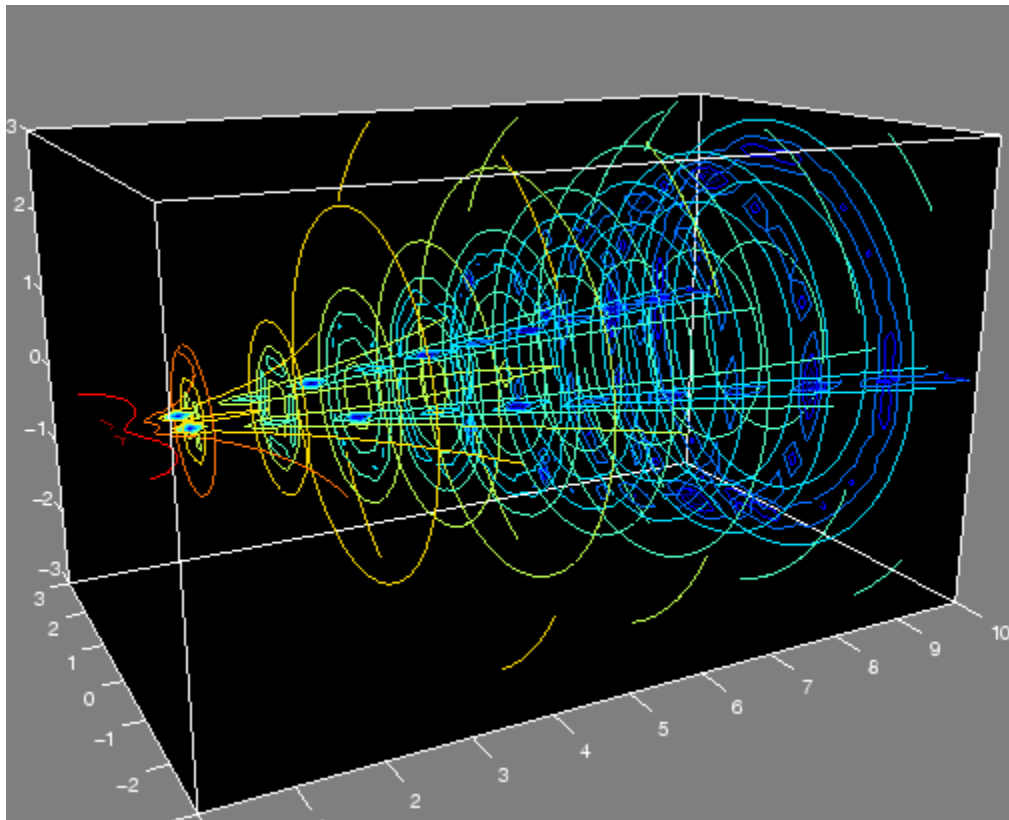
## Examples

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
```

# contourslice



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](#)

---

<b>Purpose</b>	Grayscale colormap for contrast enhancement
<b>Syntax</b>	<code>cmap = contrast(X)</code> <code>cmap = contrast(X,m)</code>
<b>Description</b>	<p>The <code>contrast</code> function enhances the contrast of an image. It creates a new gray colormap, <code>cmap</code>, that has an approximately equal intensity distribution. All three elements in each row are identical.</p> <p><code>cmap = contrast(X)</code> returns a gray colormap that is the same length as the current colormap. If there are NaN or Inf elements in <code>X</code> the length of the colormap increases.</p> <p><code>cmap = contrast(X,m)</code> returns an <code>m</code>-by-3 gray colormap.</p>
<b>Examples</b>	<p>Add contrast to the clown image defined by <code>X</code>.</p> <pre>load clown; cmap = contrast(X); image(X); colormap(cmap);</pre>
<b>See Also</b>	<code>brighten</code>   <code>colormap</code>   <code>image</code>

# conv

---

**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:

<code>full</code>	Returns the full convolution (default).
<code>same</code>	Returns the central part of the convolution of the same size as <code>u</code> .
<code>valid</code>	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, <code>length(w)</code> is <code>max(length(u) - max(0, length(v) - 1), 0)</code> .

**Definitions** Let  $m = \text{length}(u)$  and  $n = \text{length}(v)$ . 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`

# conv2

---

**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  $[ma,na] = \text{size}(A)$ ,  $[mb,nb] = \text{size}(B)$ , and  $[mc,nc] = \text{size}(C)$ , 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  $n1 = \text{length}(h1)$  and  $n2 = \text{length}(h2)$ , 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 <i>A</i> .
'valid'	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, $\text{size}(C) = \max([ma - \max(0, mb - 1), na - \max(0, nb - 1)], 0)$ .

---

**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
```

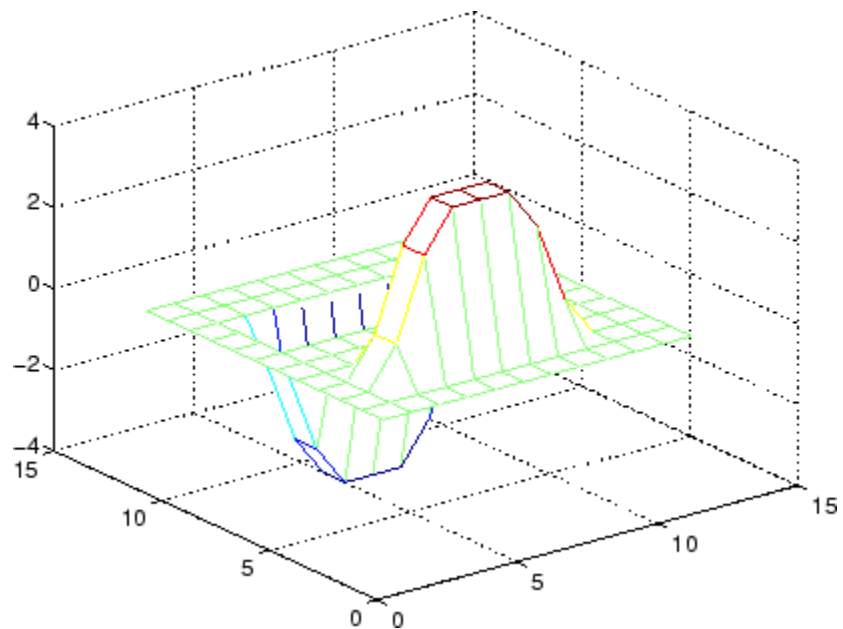
## Example 2

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.

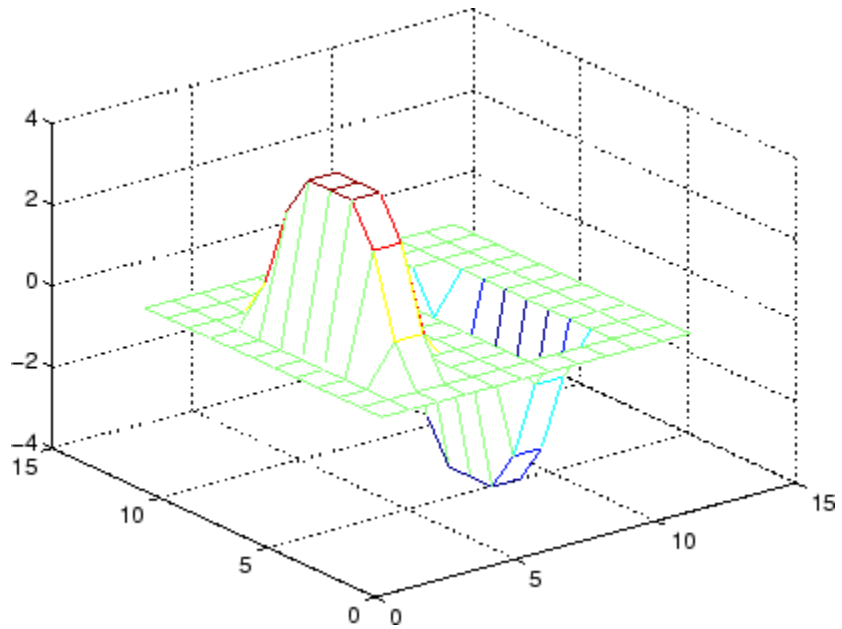
```
A = zeros(10);
A(3:7,3:7) = ones(5);
H = conv2(A,s);
mesh(H)
```



Transposing the filter  $s$  extracts the vertical edges of  $A$ .



```
V = conv2(A,s');  
figure, mesh(V)
```

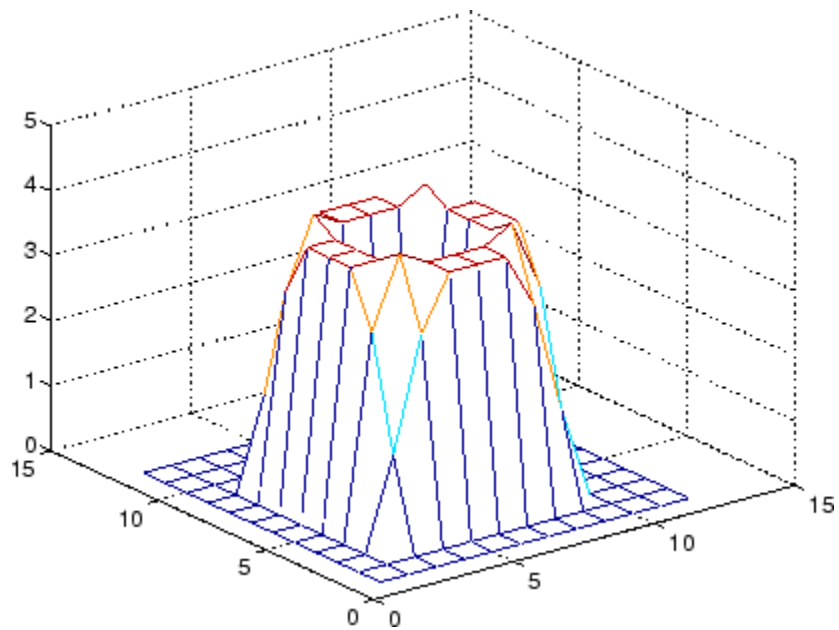


This figure combines both horizontal and vertical edges.

```
figure  
mesh(sqrt(H.^2 + V.^2))
```

# conv2

---



## See Also

[conv](#) | [convn](#) | [filter2](#) | [xcorr2](#)

**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,  $2 \leq \text{ndim} \leq 3$ . 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.

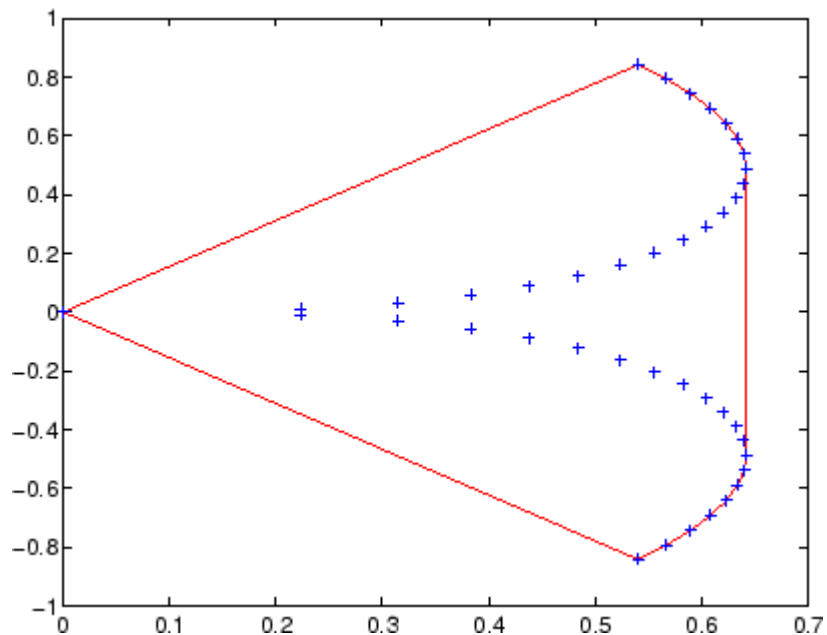
# convhull

`[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** [DelaunayTri/convexHull](#) | [DelaunayTri/voronoiDiagram](#) | [convhulln](#) | [delaunay](#) | [polyarea](#) | [voronoi](#)

<b>Purpose</b>	N-D convex hull
<b>Syntax</b>	<pre>K = convhulln(X) K = convhulln(X, options) [K, v] = convhulln(...)</pre>
<b>Description</b>	<p><code>K = convhulln(X)</code> returns the indices <code>K</code> of the points in <code>X</code> that comprise the facets of the convex hull of <code>X</code>. <code>X</code> is an <code>m</code>-by-<code>n</code> array representing <code>m</code> points in <code>N</code>-dimensional space. If the convex hull has <code>p</code> facets then <code>K</code> is <code>p</code>-by-<code>n</code>.</p> <p><code>convhulln</code> uses <code>Qhull</code>.</p> <p><code>K = convhulln(X, options)</code> specifies a cell array of strings <code>options</code> to be used as options in <code>Qhull</code>. The default options are:</p> <ul style="list-style-type: none"><li>• <code>{'Qt'}</code> for 2-, 3-, and 4-dimensional input</li><li>• <code>{'Qt', 'Qx'}</code> for 5-dimensional input and higher.</li></ul> <p>If <code>options</code> is <code>[]</code>, the default options are used. If <code>options</code> is <code>{''}</code>, no options are used, not even the default. For more information on <code>Qhull</code> and its options, see <a href="http://www.qhull.org/">http://www.qhull.org/</a>.</p> <p><code>[K, v] = convhulln(...)</code> also returns the volume <code>v</code> of the convex hull.</p>
<b>Visualization</b>	<p>Plotting the output of <code>convhulln</code> depends on the value of <code>n</code>:</p> <ul style="list-style-type: none"><li>• For <code>n = 2</code>, use <code>plot</code> as you would for <code>convhull</code>.</li><li>• For <code>n = 3</code>, you can use <code>trisurf</code> to plot the output. The calling sequence is</li></ul> <pre>K = convhulln(X); trisurf(K,X(:,1),X(:,2),X(:,3))</pre> <ul style="list-style-type: none"><li>• You cannot plot <code>convhulln</code> output for <code>n &gt; 3</code>.</li></ul>
<b>Examples</b>	The following example illustrates the options input for <code>convhulln</code> . The following commands

# convhulln

---

```
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

DelauayTri/convexHull | convhull | delaunayn | dsearchn | tsearchn | voronoin

---

<b>Purpose</b>	N-D convolution						
<b>Syntax</b>	<code>C = convn(A,B)</code> <code>C = convn(A,B, 'shape')</code>						
<b>Description</b>	<p><code>C = convn(A,B)</code> computes the N-dimensional convolution of the arrays A and B. The size of the result is <code>size(A)+size(B)-1</code>.</p> <p><code>C = convn(A,B, 'shape')</code> returns a subsection of the N-dimensional convolution, as specified by the <code>shape</code> parameter:</p> <table><tr><td><code>'full'</code></td><td>Returns the full N-dimensional convolution (default).</td></tr><tr><td><code>'same'</code></td><td>Returns the central part of the result that is the same size as A.</td></tr><tr><td><code>'valid'</code></td><td>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  <code>max(size(A)-size(B) + 1, 0)</code></td></tr></table>	<code>'full'</code>	Returns the full N-dimensional convolution (default).	<code>'same'</code>	Returns the central part of the result that is the same size as A.	<code>'valid'</code>	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  <code>max(size(A)-size(B) + 1, 0)</code>
<code>'full'</code>	Returns the full N-dimensional convolution (default).						
<code>'same'</code>	Returns the central part of the result that is the same size as A.						
<code>'valid'</code>	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  <code>max(size(A)-size(B) + 1, 0)</code>						
<b>See Also</b>	<code>conv</code>   <code>conv2</code>						

# matlab.mixin.Copyable

---

**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, DeepCp, ShallowCp)
            if nargin > 0
                obj.Prop1 = val1;
                obj.Prop2 = val2;
                obj.DeepObj = deepobj;
                obj.ShallowObj = shallowobj;
            end
        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
end
```

# matlab.mixin.Copyable

---

```
        methods
            function obj = DeepCp(val)
                ...
            end
        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 handle
```

```
    Properties:
```

```
        DpProp: 7
```

```
>> a.ShallowObj.ShProp
ans =
```

```
    ShallowCp handle
```

```
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 handle
```

```
Properties:  
  DpProp: 7
```

```
% The shallow copied object is still referencing the same data:
```

```
>> b.ShallowObj
```

```
ans =
```

```
ShallowCp handle
```

```
Properties:  
  ShProp: 5
```

# matlab.mixin.Copyable

---

## 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@matlab.mixin.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
        end
    end
end
```

```
        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.

# matlab.mixin.Copyable

---

---

**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 Programming Fundamentals](#) documentation.

## See Also

`handle`

## How To

- [Class Attributes](#)

**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`.

# matlab.mixin.Copyable.copy

---

## 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`



## 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”

# copyobj

---

**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`

# corrcoef

---

**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
```

# corrcoef

---

```
ans =  
    4     2  
    4     3  
    2     4  
    3     4
```

## See Also

[cov](#) | [mean](#) | [median](#) | [std](#) | [var](#) | [xcorr](#) | [xcov](#)



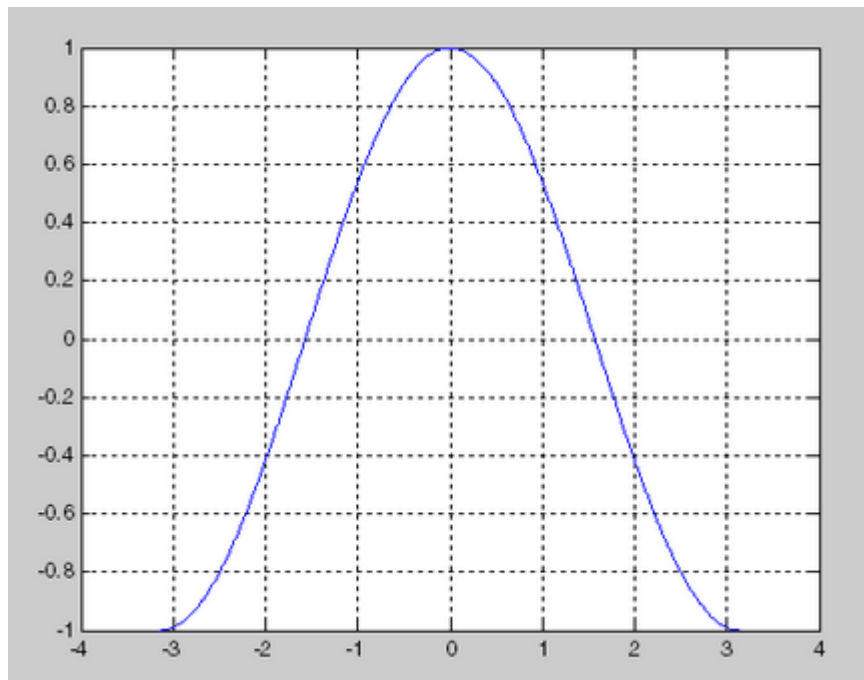
**Purpose** Cosine of argument in radians

**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  $-\pi \leq x \leq \pi$ :

```
x = -pi:0.01:pi;  
plot(x,cos(x)), grid on
```



**See Also** `cosd` | `acos` | `acosd` | `cosh`

# cosd

---

<b>Purpose</b>	Cosine of argument in degrees
<b>Syntax</b>	$Y = \text{cosd}(X)$
<b>Description</b>	$Y = \text{cosd}(X)$ returns the cosine for each element of $X$ , expressed in degrees.
<b>Input Arguments</b>	<p><b>X - Angle in degrees</b> scalar value   vector   matrix   N-D array</p> <p>Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>cosd</code> operation is element-wise when <math>X</math> is nonscalar.</p> <p><b>Data Types</b> single   double</p> <p><b>Complex Number Support:</b> Yes</p>
<b>Output Arguments</b>	<p><b>Y - Cosine of angle</b> scalar value   vector   matrix   N-D array</p> <p>Cosine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as <math>X</math>.</p>
<b>Examples</b>	<p><b>Cosine of 90 degrees compared to cosine of <math>\pi/2</math> radians</b></p> <pre>cosd(90)  ans =       0  cos(pi/2)  ans =   6.1232e-17</pre>

### 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](#)

# cosh

---

**Purpose** Hyperbolic cosine

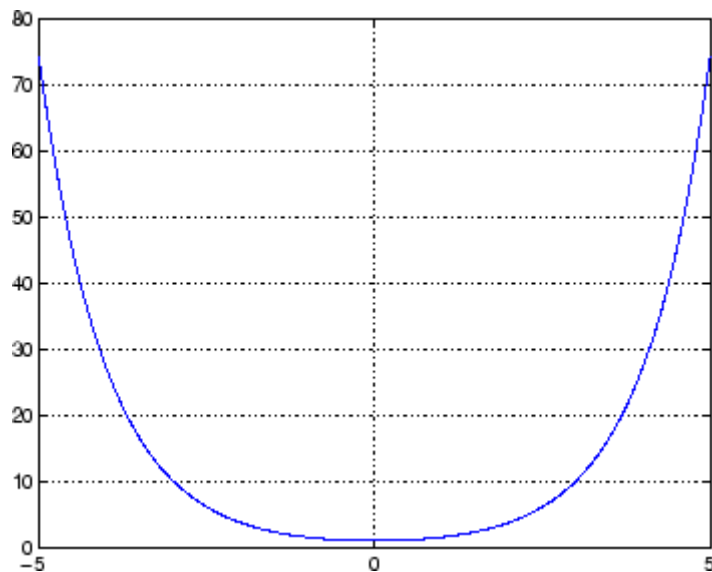
**Syntax**  $Y = \cosh(X)$

**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  $-5 \leq x \leq 5$ .

```
x = -5:0.01:5;  
plot(x,cosh(x)), grid on
```



**See Also** [acosh](#) | [cos](#) | [sinh](#) | [tanh](#)

**Purpose** Cotangent of argument in radians

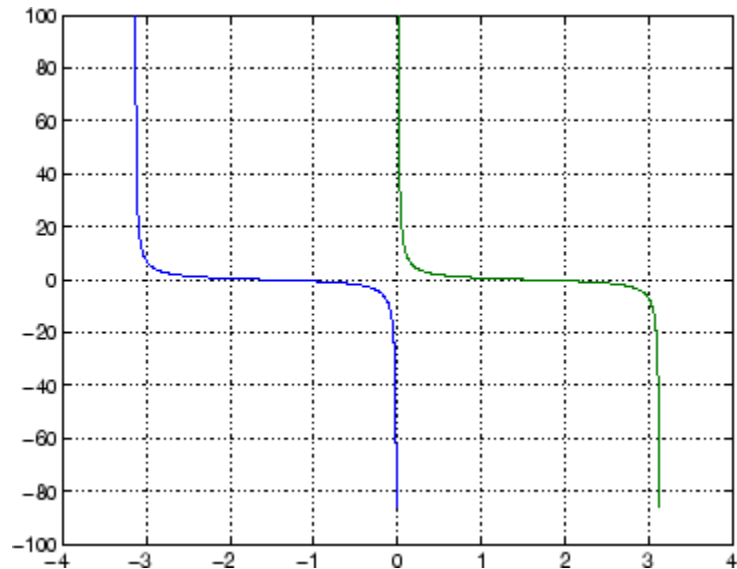
**Syntax**  $Y = \cot(X)$

**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](#)

# cotd

---

**Purpose** Cotangent of argument in degrees

**Syntax**  $Y = \text{cotd}(X)$

**Description**  $Y = \text{cotd}(X)$  is the cotangent of the elements of  $X$ . The result,  $Y$ , is 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  $90^\circ$  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  $90^\circ$ . 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  $90^\circ$ .

The behavior is similar for input angles near  $180^\circ$ .

```
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`

# coth

**Purpose** Hyperbolic cotangent

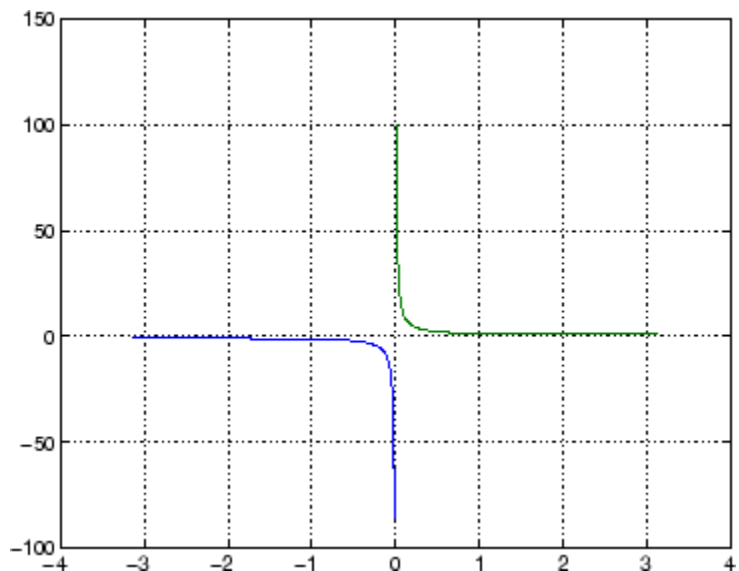
**Syntax**  $Y = \text{coth}(X)$

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

$Y = \text{coth}(X)$  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:

$$\text{cov}(x_1, x_2) = E[(x_1 - \mu_1)(x_2 - \mu_2)]$$

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  $100 \times \text{eps}$  relative to  $\text{abs}(A(i))$  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.
```

# cputime

---

**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`

**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 (`m1fg6331_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:

# RandStream.create

---

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 will be removed in a future release. 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](#)

# createClassFromWsd1

---

**Purpose** Create MATLAB class based on WSDL document

**Syntax** `createClassFromWsd1(source)`

**Description** `createClassFromWsd1(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. `createClassFromWsd1` 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:

```
createClassFromWsd1('path_to_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';
```



```
createClassFromWsd1(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”

# inputParser.createCopy

---

<b>Purpose</b>	Create copy of inputParser object
<b>Syntax</b>	<code>pNew = createCopy(p)</code>
<b>Description</b>	<code>pNew = createCopy(p)</code> creates a copy of inputParser object <code>p</code> .
<b>Tips</b>	<ul style="list-style-type: none"><li>Using <code>createCopy</code> is not the same as copying by assignment, such as <code>pNew = p</code>, which creates a new handle to the same object <code>p</code>.</li></ul>
<b>Input Arguments</b>	<b>p</b> Object of class <code>inputParser</code> .
<b>Output Arguments</b>	<b>pNew</b> <code>inputParser</code> object with the same properties as object <code>p</code> .

## 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](#) |

# createSoapMessage

---

**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
<code>namespace</code>	Location of the Web service in the form of a valid Uniform Resource Identifier (URI).
<code>method</code>	Name of the Web service operation you want to run.
<code>values</code>	Cell array of input you need to provide for the method.
<code>names</code>	Cell array of parameters for method.
<code>types</code>	Cell array defining the XML data types for <code>values</code> . Specifying <code>style</code> is optional; when you do not include the argument, MATLAB uses unspecified.
<code>style</code>	Style for structuring the SOAP message, either ' <b>document</b> ' or ' <b>rpc</b> '. Specifying <code>style</code> is optional; when you do not include the argument, MATLAB uses <b>rpc</b> . Use a style supported by the service you specified in <code>namespace</code> .

**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](#) | [createClassFromWsd1](#) | [parseSoapResponse](#) | [urlread](#) | [xmlread](#)

## How To

- “Access Web Services Using MATLAB SOAP Functions”

# **CROSS**

---

**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,  $C = A \times B$ . 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

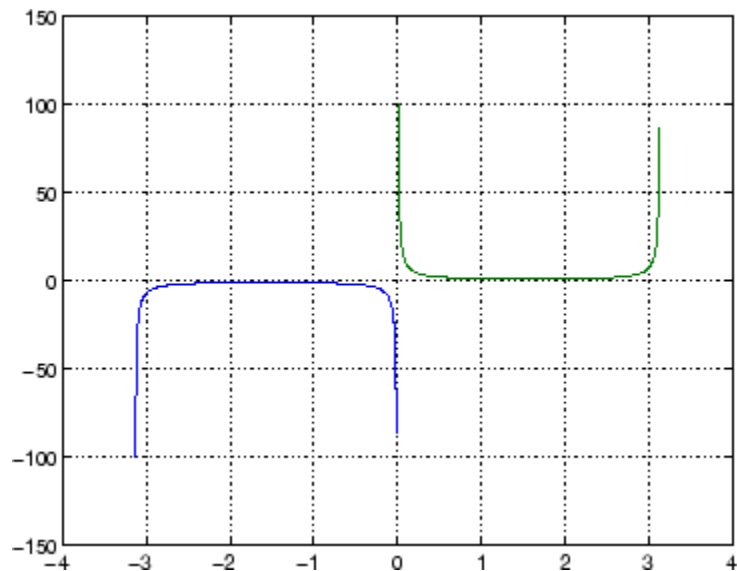
**Syntax**  $Y = \text{csc}(x)$

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

$Y = \text{csc}(x)$  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`

<b>Purpose</b>	Cosecant of argument in degrees
<b>Syntax</b>	$Y = \text{cscd}(X)$
<b>Description</b>	$Y = \text{cscd}(X)$ is the cosecant of the elements of $X$ . The result, $Y$ , is expressed in degrees.
<b>Input Arguments</b>	<b>X - Angle in degrees</b> 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 <code>cscd</code> operation is element-wise when $X$ is nonscalar.  <b>Data Types</b> single   double <b>Complex Number Support:</b> Yes
<b>Output Arguments</b>	<b>Y - Cosecant of angle</b> 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 $X$ .
<b>Examples</b>	<b>Cosecant of 180 degrees compared to cosecant of radians</b>  <code>cscd(180)</code> is infinite, whereas <code>csc(pi)</code> is large but finite.  <code>cscd(180)</code>  <code>ans =</code>  Inf  <code>csc(pi)</code>  <code>ans =</code>



---

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`

# csch

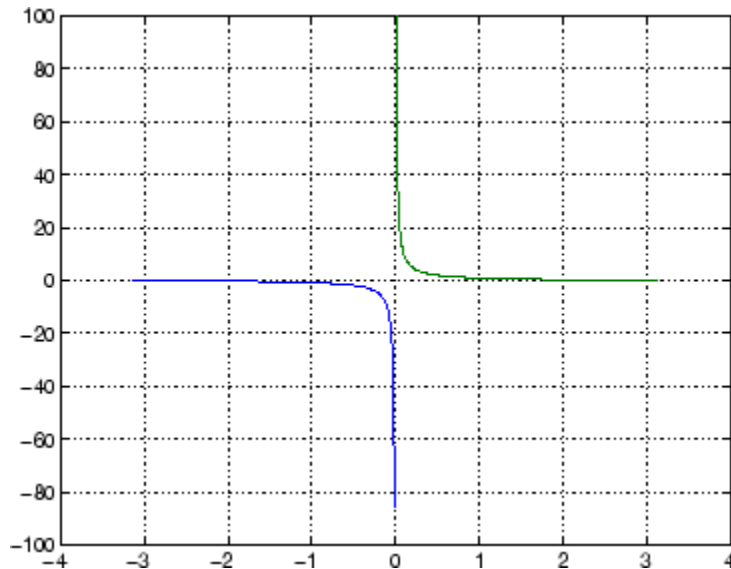
**Purpose** Hyperbolic cosecant

**Syntax**  $Y = \text{csch}(x)$

**Description** The `csch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.  $Y = \text{csch}(x)$  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](#)

**Purpose** Read comma-separated value file

**Syntax**

```
M = csvread(filename)
M = csvread(filename, row, col)
M = csvread(filename, row, col, range)
```

**Description**

`M = csvread(filename)` reads a comma-separated value formatted file, `filename`. The `filename` input is a string enclosed in single quotes. The result is returned in `M`. The file can only contain numeric values.

`M = csvread(filename, row, col)` reads data from the comma-separated value formatted 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, range)` reads only the range specified. Specify range using the notation `[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. You can also specify the range using spreadsheet notation, as in `range = 'A1..B7'`.

- Tips**
- `csvread` fills empty delimited fields with zero. Data files having lines that end with a nonspace delimiter, such as a semicolon, produce a result 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. Valid forms for a complex number are

Form	Example
<code>-&lt;real&gt;-&lt;imag&gt;i j</code>	<code>5.7-3.1i</code>
<code>-&lt;imag&gt;i j</code>	<code>-7j</code>

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

# csvread

---

## Examples

Given the file `csvlist.dat` that contains the comma-separated values

```
02, 04, 06, 08, 10, 12
03, 06, 09, 12, 15, 18
05, 10, 15, 20, 25, 30
07, 14, 21, 28, 35, 42
11, 22, 33, 44, 55, 66
```

To read the entire file, use

```
csvread('csvlist.dat')
```

ans =

```
2     4     6     8     10    12
3     6     9    12    15    18
5    10    15    20    25    30
7    14    21    28    35    42
11   22   33   44   55   66
```

To read the matrix starting with zero-based row 2, column 0, and assign it to the variable `m`,

```
m = csvread('csvlist.dat', 2, 0)
```

m =

```
5    10    15    20    25    30
7    14    21    28    35    42
11   22   33   44   55   66
```

To read the matrix bounded by zero-based (2,0) and (3,3) and assign it to `m`,

```
m = csvread('csvlist.dat', 2, 0, [2,0,3,3])
```

m =

```
5    10   15   20
7    14   21   28
```

## See Also

[csvwrite](#) | [dlmread](#) | [textscan](#) | [importdata](#) | [uiimport](#)

# ctranspose

---

**Purpose** Complex conjugate transpose

**Syntax**  $b = a'$   
 $b = \text{ctranspose}(a)$

**Description**  $b = a'$  computes the complex conjugate transpose of matrix  $a$  and returns the result in  $b$ .  
 $b = \text{ctranspose}(a)$  is called for the syntax  $a'$  (complex conjugate transpose) when  $a$  is an object.

**See Also** transpose

<b>Purpose</b>	Write comma-separated value file
<b>Syntax</b>	<pre>csvwrite(filename,M) csvwrite(filename,M,row,col)</pre>
<b>Description</b>	<p><code>csvwrite(filename,M)</code> writes matrix <code>M</code> into <code>filename</code> as comma-separated values. The <code>filename</code> input is a string enclosed in single quotes.</p> <p><code>csvwrite(filename,M,row,col)</code> writes matrix <code>M</code> into <code>filename</code> starting at the specified row and column offset. The row and column arguments are zero based, so that <code>row=0</code> and <code>C=0</code> specify the first value in the file.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• <code>csvwrite</code> terminates each line with a line feed character and no carriage return.</li><li>• <code>csvwrite</code> writes a maximum of five significant digits. If you need greater precision, use <code>dlmwrite</code> with a precision argument.</li><li>• <code>csvwrite</code> does not accept cell arrays for the input matrix <code>M</code>. To export a cell array that contains only numeric data, use <code>cell2mat</code> to convert the cell array to a numeric matrix before calling <code>csvwrite</code>. 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 <code>xlswrite</code>. 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.</li></ul>
<b>Examples</b>	<p>The following example creates a comma-separated value file from the matrix <code>m</code>.</p> <pre>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</pre>

## csvwrite

---

```
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 the cumulative product along different dimensions of an array.

If A is a vector, cumprod(A) returns a vector containing the cumulative product of the elements of A.

If A is a matrix, cumprod(A) returns a matrix the same size as A containing the cumulative products for each column of A.

If A is a multidimensional array, cumprod(A) works on the first nonsingleton dimension.

B = cumprod(A,dim) returns the cumulative product of the elements along the dimension of A specified by scalar dim. For example, cumprod(A,1) increments the column index, thus working along the columns of A. Thus, cumprod(A,1) and cumprod(A) will return the same thing. To increment the row index, use cumprod(A,2).

## Examples

```
cumprod(1:5)
ans =
     1     2     6    24   120
```

```
A = [1 2 3; 4 5 6];
```

```
cumprod(A,1)
ans =
     1     2     3
     4    10    18
```

```
cumprod(A,2)
ans =
     1     2     6
     4    20    120
```

# cumprod

---

## See Also

[cumsum](#) | [prod](#) | [sum](#)

**Purpose**

Cumulative sum

**Syntax**

```
B = cumsum(A)
B = cumsum(A,dim)
```

**Description**

`B = cumsum(A)` returns the cumulative sum along different dimensions of an array.

If `A` is a vector, `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.

If `A` is a matrix, `cumsum(A)` returns a matrix the same size as `A` containing the cumulative sums for each column of `A`.

If `A` is a multidimensional array, `cumsum(A)` works on the first nonsingleton dimension.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along the dimension of `A` specified by scalar `dim`. For example, `cumsum(A,1)` works along the first dimension (the columns); `cumsum(A,2)` works along the second dimension (the rows).

**Examples**

```
cumsum(1:5)
ans =
     1     3     6    10    15
```

```
A = [1 2 3; 4 5 6];
```

```
cumsum(A,1)
ans =
     1     2     3
     5     7     9
```

```
cumsum(A,2)
ans =
     1     3     6
     4     9    15
```

## **cumsum**

---

### **See Also**

`cumprod` | `prod` | `sum`

**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
```

# cumtrapz

---

```
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](#)

**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)`.

# curl

---

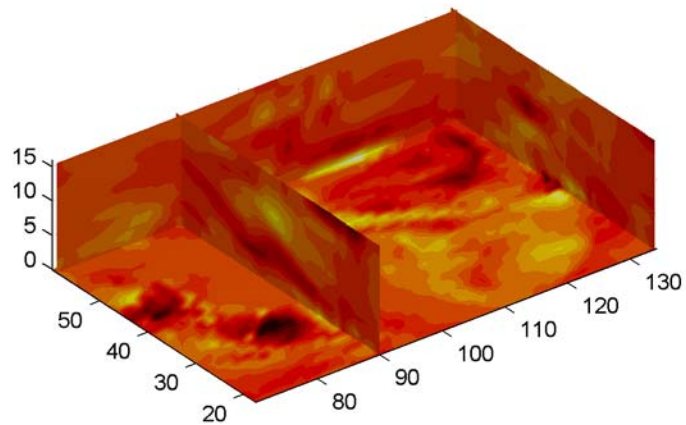
`[curlx,curly,curlz] = curl(...)`, `[curlx,curly] = curl(...)`  
returns only the curl.

`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.

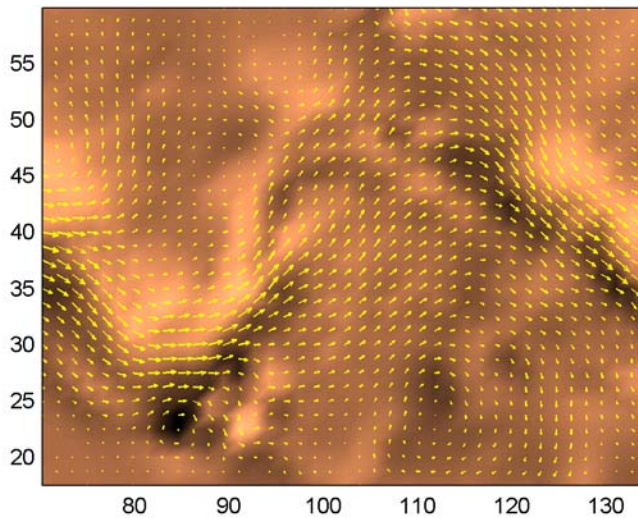
```
figure
load wind
cav = curl(x,y,z,u,v,w);
h = slice(x,y,z,cav,[90 134],59,0);
shading interp
daspect([1 1 1]);
axis tight
colormap hot(16)
camlight
set([h(1),h(2)], 'ambientstrength', .6)
```





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 `rsc.m` in `matlabroot\toolbox\matlab\verctrl`.

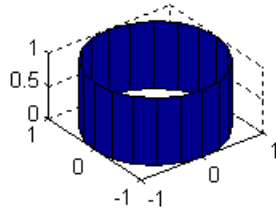
**See Also** `checkin` | `checkout` | `cmopts` | `undocheckout` | `verctrl`

# 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`.

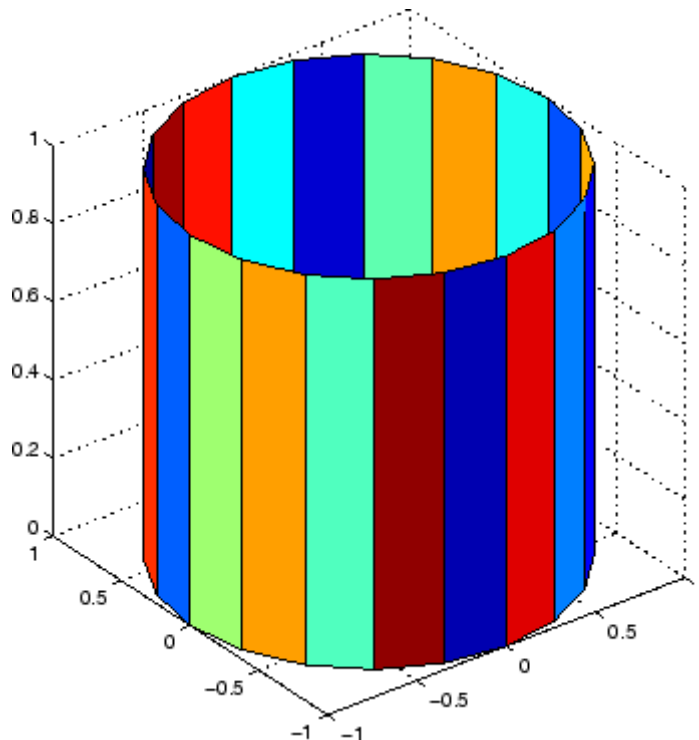
**Tips**

`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'))))
```



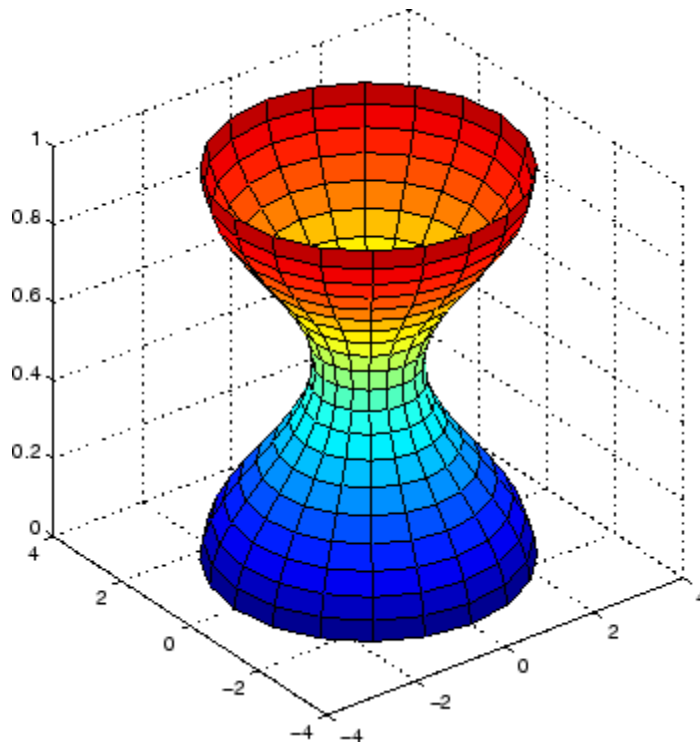
Generate a cylinder defined by the profile function  $2+\sin(t)$ .

```
t = 0:pi/10:2*pi;
```

# cylinder

---

```
[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 `theSamples`, `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`.

# daqread

---

`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 <code>doubles</code> (default) or <code>native</code> .
TimeFormat	Specify the time format as <code>vector</code> (default) or <code>matrix</code> .
OutputFormat	Specify the output format as <code>matrix</code> (the default) or <code>tscollection</code> . When you specify <code>tscollection</code> , <code>daqread</code> 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,

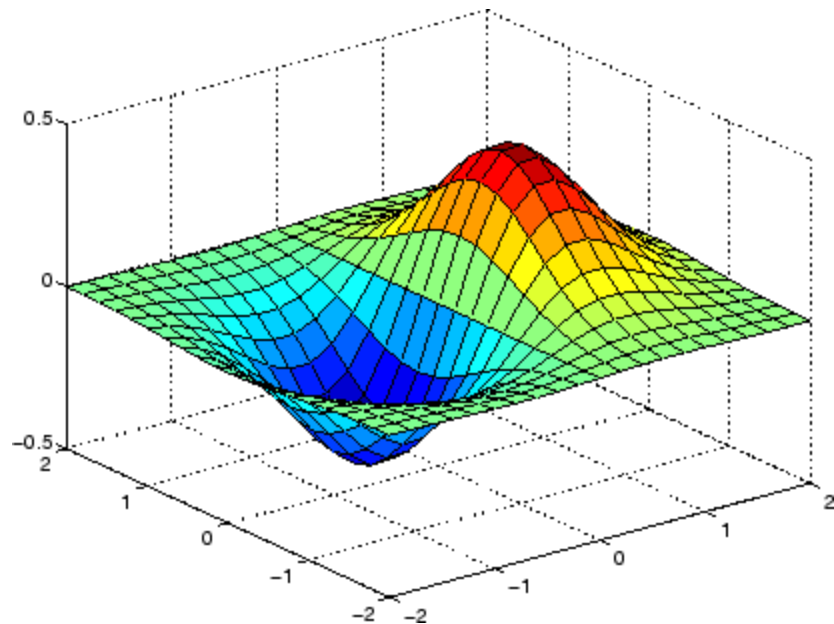
```
daspect(daspect)
```

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  $-2 \leq x \leq 2$ ,  $-2 \leq y \leq 2$ ,

```
[x,y] = meshgrid([-2:.2:2]);  
z = x.*exp(-x.^2 - y.^2);  
surf(x,y,z)
```



# daspect

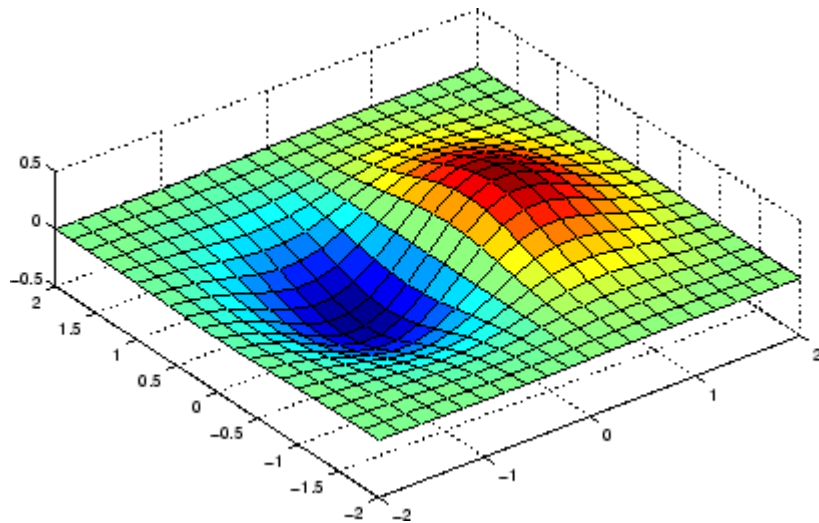
---

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 2-1121. 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 2-1125 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`, `KeyPressFcn`, 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.
-----------	----------------------------------------------------------------------------------------------------------------------

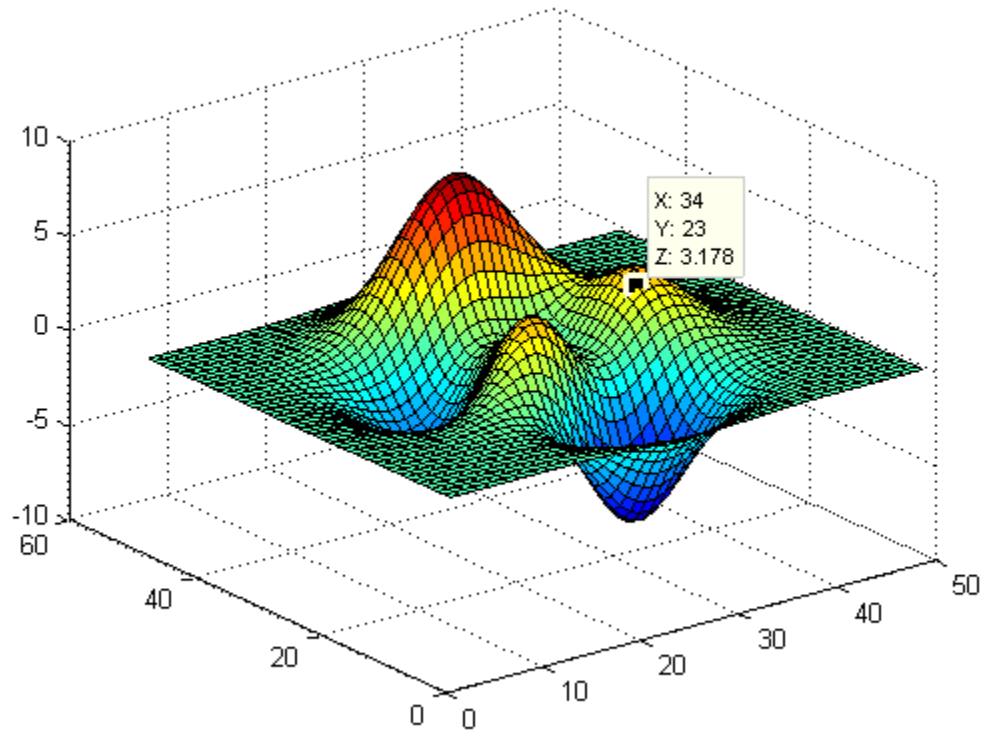
See “Output Arguments” on page 2-1121 for more details on data cursor mode objects.

## Examples

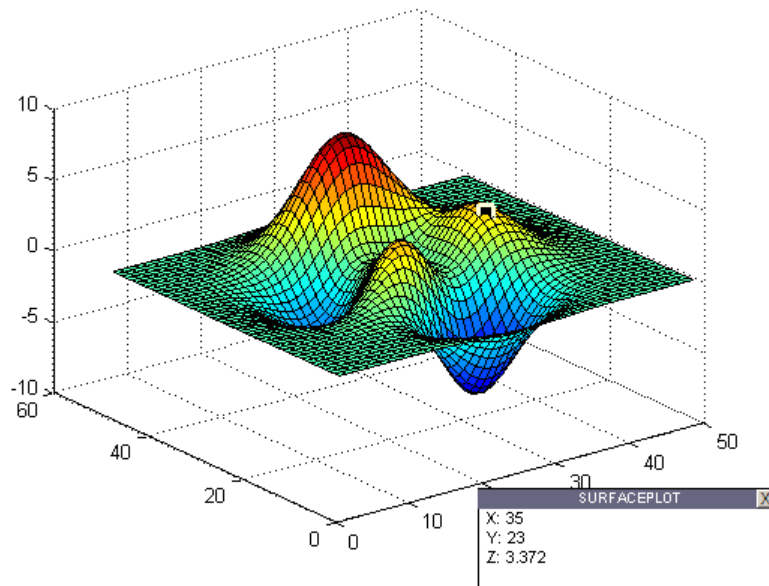
This example creates a plot and enables data cursor mode from the command line.

```
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))
```

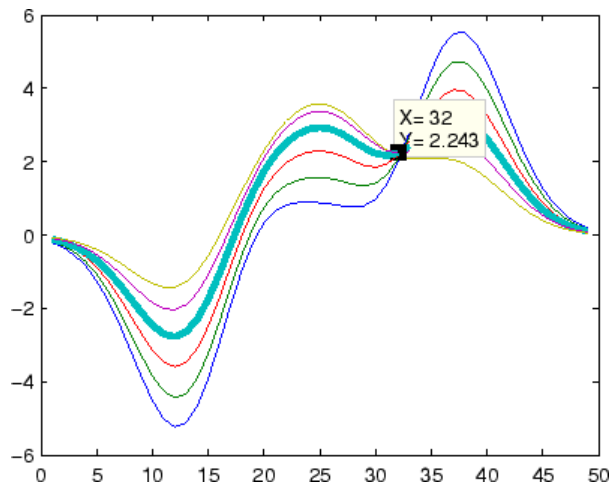
# datacursormode

---

```
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:

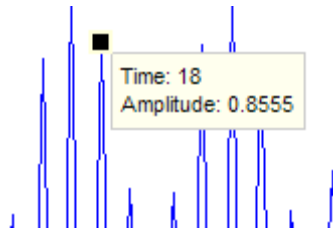
```
function txt = myupdatefcn(empty,event_obj)
% Customizes text of data tips

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` | `nargchk` | `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**

# datetime

---

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.

<b>Symbolic Identifier</b>	<b>Description</b>	<b>Example</b>
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**

# datenum

---

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.

<b>Symbolic Identifier</b>	<b>Description</b>	<b>Example</b>
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.

<b>Numeric Identifier</b>	<b>Date String Format</b>	<b>Example</b>
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	'mmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45



<b>Numeric Identifier</b>	<b>Date String Format</b>	<b>Example</b>
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	'yyyymmddTHMMSS' (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.

# datestr

---

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 2-1149.

---

## **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 | cell array

Date strings, returned as a column vector of length *m*, 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, 'mmm 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 = 'mmm-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(tickaxis,dateformnum)
datetick(...,'keeplimits')
datetick(...,'kepticks')
datetick(axes_handle,...)
```

**Description** `datetick(tickaxis)` labels the tick lines of an axis using dates, replacing the default numeric labels. `tickaxis` is the string 'x', 'y', or 'z'. The default is 'x'. `datetick` selects a label format based on the minimum and maximum limits of the specified axis. The axis data values should be generated by or be compatible with the output of the `datenum` function.

`datetick(tickaxis,dateformat)` formats the labels according to the string `dateformat`. A date format string can consist of the following elements (or combinations of them), identified by the format symbols in the left-hand column.

Date Format	Interpretation of Format Symbol
yyyy	Full year, e.g., 1990, 2001, or 2008
yy	Partial year, e.g. 90, 01, or 08
mmmm	Full name of the month, according to the calendar locale, e.g., "March" or "April" in the UK and USA English locales
mmm	First three letters of the month, according to the calendar locale, e.g., "Mar" or "Apr" in the UK and USA English
mm	Numeric month of year, padded with leading zeros, e.g., <code>../03/..</code> or <code>../12/..</code>



<b>Date Format</b>	<b>Interpretation of Format Symbol</b>
m	Capitalized first letter of the month, according to the calendar locale; for backwards compatibility, e.g., "D" for December
dddd	Full name of the weekday, according to the calendar locale, e.g., "Monday" or "Tuesday", for the UK and USA calendar locales
ddd	First three letters of the weekday, according to the calendar locale, e.g., "Mon" or "Tue", for the UK and USA calendar locales
dd	Numeric day of the month, padded with leading zeros, e.g., 05/./.. or 20/./..
d	Capitalized first letter of the weekday, e.g., "M" for Monday; for backwards compatibility
HH	Hour of the day, according to the time format. In case the time format AM   PM is set, HH does not pad with leading zeros. If AM   PM is not set, HH displays the hour of the day, padded with leading zeros; e.g., 10:20 PM, which is equivalent to 22:20; 9:00 AM, which is equivalent to 09:00.
MM	Minutes of the hour, padded with leading zeros, e.g., 10:05 or 10:05 AM
SS	Second of the minute, padded with leading zeros, e.g., 10:15:30, 10:05:30, 10:05:30 AM
FFF	Milliseconds field, padded with leading zeros, e.g., 10:15:30.015
PM	Setting the time format to morning or afternoon by appending AM or PM to the date string, as appropriate, without separating symbols

You can mix format symbols to create customized data symbols. For example:

# datetick

---

```
datetick('x','dd (ddd)')
```

generates ticks along the *x*-axis that display the day of the month followed by the three-letter abbreviation of the day of the week in parentheses, for example, 01 (Wed). To preface each date tick with an abbreviated month name, you could specify

```
datetick('x','mmm-dd (ddd)')
```

to yield ticks such as Apr-01 (Wed).

`datetick(tickaxis,dateformnum)` formats the labels according to the integer *dateformnum*, a date format index (see table). To produce correct results, the data for the specified axis must be serial date numbers (as produced by `datenum`).

<b>Date Format Number</b>	<b><i>dateformat</i> (string)</b>	<b>Example</b>
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2008 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2008
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	'mmyy'	Mar00

<b>Date Format Number</b>	<b><i>dateformat</i> (string)</b>	<b>Example</b>
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	'mmyyyy'	Mar2000
29	(ISO 8601) 'yyyy-mm-dd'	2000-03-01
30	(ISO 8601) 'yyyymmddTHMMSS'	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

`datetick(..., 'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits.

`datetick(..., 'keepticks')` changes the tick labels to date-based labels without changing their locations.

You can use both `keeplimits` and `keepticks` in the same call to `datetick`.

`datetick(axes_handle, ...)` uses the axes specified by the handle `ax` instead of the current axes.

`datetick` calls `datestr` to convert date numbers to date strings.

To change the tick spacing and locations, set the appropriate axes property (i.e., `XTick`, `YTick`, or `ZTick`) before calling `datetick`.

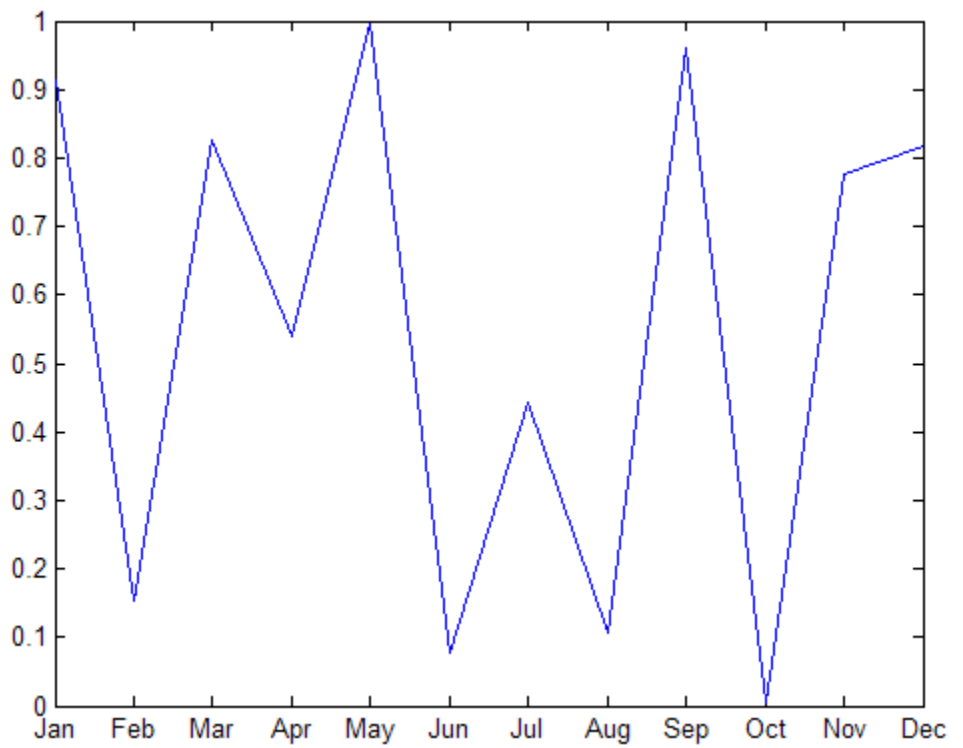
Calling `datetick` sets the `TickMode` of the specified axis to `'manual'`. This means that after zooming, panning or otherwise changing axis limits, you should call `datetick` again to update the ticks and labels.

## Examples

Add month labels to your plot:

```
% Select a starting date:
startDate = datenum('01-01-2009')
% Select an ending date:
endDate = datenum('12-31-2009')
% Create xdata to correspond to the number of
% months between the start and end dates:
xData = linspace(startDate, endDate, 12);
% For this example, plot random data:
plot(xData, rand(1, 12))
% Set the number of XTicks to the number of points
% in xData:
set(gca, 'XTick', xData)

% Convert the x tick labels to month names, keeping
% the total number of ticks by using the 'keepticks'
% option:
datetick('x', 'mmm', 'keepticks')
```



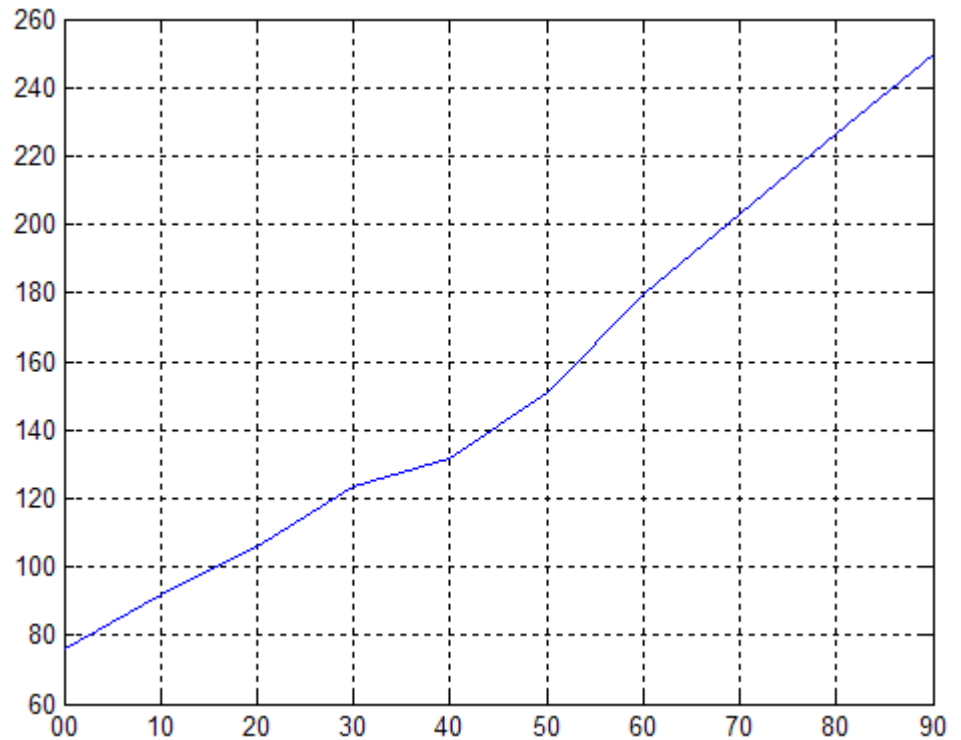
Graph population data for the 20th Century taken from the 1990 US census:

```
% 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 date numbers and plot
plot(datenum(t,1,1),p)
grid on
% Replace x-axis ticks with 2-digit years using date format 11
```

# datetick

---

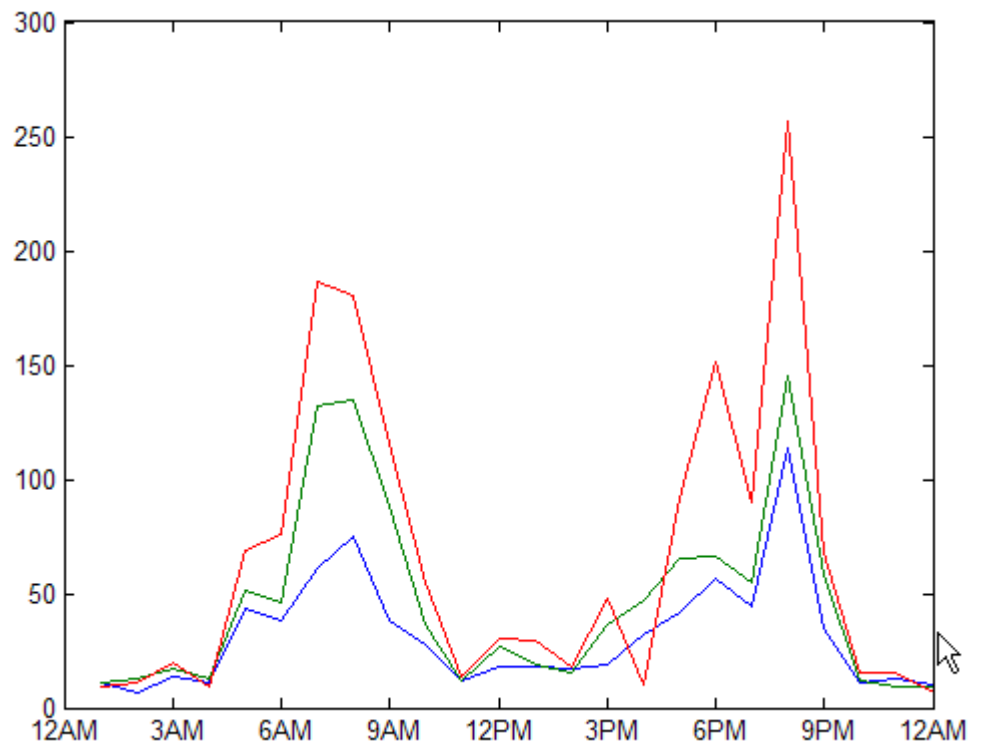
```
datetick('x',11)
```



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, here 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 datenums for the data (only hours change)  
xdate = datenum(year,month,day,hour,minutes,minutes);  
% Plot the traffic data against datenums  
plot(xdate,count)  
% Update the graph's x-axis with date ticks  
datetick('x','HHPM')
```



# datetick

---

Create multiple plots within a single figure and manipulate their *x*-labels using the `datetick` function:

```
% Select a starting date:
startDate = datenum('01-01-2009');
% Select an ending date:
endDate = datenum('12-31-2009');
% Create xdata to correspond to the number of
% months between the start and end dates:
xData = linspace(startDate,endDate,12);
% For this example, plot random data:
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)
% Convert the x tick labels to month names, keeping
% the total number of ticks by using the 'kepticks'
% option:
for i = 1:2
    % Handle input must be a scalar.
    datetick(s(i),'x','mmm','kepticks')
end
```

## 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.

<b>Symbolic Identifier</b>	<b>Description</b>	<b>Example</b>
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 =
```

# datevec

---

```
s =  
    0  
    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  $0 \leq \text{DateNumber} < 1$ , 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
<code>dbclear in file</code>	Removes all breakpoints in <code>file</code> . <code>file</code> must be the name of a MATLAB program file, and can include a MATLAB partial path.
<code>dbclear in file -completenames</code>	If the command includes the <code>-completenames</code> option, then <code>file</code> 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 <code>\\</code> or with a drive letter followed by a colon. On UNIX platforms, this is a file name that begins with <code>/</code> or <code>~</code> .) <code>file</code> can include a <code>&gt;</code> to specify the path to a particular local function or to a nested function within a code file.
<code>dbclear in file at lineno</code>	Removes the breakpoint set at line number <code>lineno</code> in <code>file</code> .
<code>dbclear in file at lineno@</code>	Removes the breakpoint set in the anonymous function at line number <code>lineno</code> in <code>file</code> .

# dbclear

Format	Action
<code>dbclear in file at lineno@n</code>	Removes the breakpoint set in the nth anonymous function at line number <code>lineno</code> in <code>file</code> .
<code>dbclear in file at locfun</code>	Removes all breakpoints in local function <code>locfun</code> in <code>file</code> .

`dbclear if error ...` formats are listed here:

Format	Action
<code>dbclear if error</code>	Removes the breakpoints set using the <code>dbstop if error</code> and <code>dbstop if error</code> identifier statements.
<code>dbclear if error identifier</code>	Removes the breakpoint set using <code>dbstop if error</code> identifier for the specified identifier. Running this produces an error if <code>dbstop if error</code> or <code>dbstop if error all</code> is set.
<code>dbclear if caught error</code>	Removes the breakpoints set using the <code>dbstop if caught error</code> and <code>dbstop if caught error</code> identifier statements.
<code>dbclear if caught error identifier</code>	Removes the breakpoints set using the <code>dbstop if caught error</code> identifier statement for the specified identifier. Running this produces an error if <code>dbstop if caught error</code> or <code>dbstop if caught error all</code> is set.

`dbclear if warning ...` formats are listed here:

<code>dbclear if warning</code>	Removes the breakpoints set using the <code>dbstop if warning</code> and <code>dbstop if warning</code> identifier statements.
<code>dbclear if warning identifier</code>	Removes the breakpoint set using <code>dbstop if warning</code> identifier for the specified identifier. Running this produces an error if <code>dbstop if warning</code> or <code>dbstop if warning all</code> 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”

<b>Purpose</b>	Reverse workspace shift performed by <code>dbup</code> , while in debug mode
<b>Syntax</b>	<code>dbdown</code>
<b>Description</b>	<p><code>dbdown</code> changes the current workspace context to the workspace of the called MATLAB code file when a breakpoint is encountered. You must have issued the <code>dbup</code> function at least once before you issue this function. <code>dbdown</code> is the opposite of <code>dbup</code>.</p> <p>Multiple <code>dbdown</code> 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.</p>
<b>See Also</b>	<code>dbcont</code>   <code>dbclear</code>   <code>dbquit</code>   <code>dbstack</code>   <code>dbstatus</code>   <code>dbstop</code>   <code>dbstep</code>   <code>dbtype</code>   <code>dbup</code>
<b>Related Examples</b>	<ul style="list-style-type: none"><li>• “Debugging Process and Features”</li></ul>

# dblquad

---

**Purpose** Numerically evaluate double integral over rectangle

**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  $\text{fun}(x,y)$  over the rectangle  $x_{\min} \leq x \leq x_{\max}$ ,  $y_{\min} \leq y \leq y_{\max}$ . 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 `@quad1` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quad1`.

**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  $\pi \leq x \leq 2\pi$ ,  $0 \leq y \leq \pi$ . 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` | `quad1` | `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 -Ddebugger`, 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
```

# 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:

<code>file</code>	The file in which the function appears. This field is the empty string if there is no file.
<code>name</code>	Function name within the file.
<code>line</code>	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.

<code>name</code>	Function name.
<code>file</code>	Full path for file containing breakpoints.
<code>line</code>	Vector of breakpoint line numbers.
<code>anonymous</code>	Vector of integers representing the anonymous functions in the <code>line</code> 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.
<code>expression</code>	Cell vector of breakpoint conditional expression strings corresponding to lines in the <code>line</code> 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** `dbclean` | `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 2-1192
- 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 2-1189. 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, and
"m_interpreter>clear" is not a MATLAB code file.
    Instead, the debugger will stop at the point right before
    "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.

# dbstop

---

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.

```
dbc clear 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

`dbc clear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbtype` | `dbup`

## Related Examples

- “Set Breakpoints”

---

<b>Purpose</b>	List text file with line numbers
<b>Syntax</b>	<code>dbtype filename</code> <code>dbtype filename start:end</code>
<b>Description</b>	<p>The <code>dbtype</code> command is used to list a text file with line numbers, which is helpful when setting breakpoints in a MATLAB code file with <code>dbstop</code>.</p> <p><code>dbtype filename</code> displays the contents of the specified text file, with the line number preceding each line. <code>filename</code> must be the full path name of a file, or a MATLAB relative partial path.</p> <p><code>dbtype filename start:end</code> displays the portion of the file specified by a range of line numbers from <code>start</code> to <code>end</code>.</p> <p>You cannot use <code>dbtype</code> for built-in functions.</p>
<b>Examples</b>	<p>To see only the input and output arguments for a function, that is, the first line of the file, use the syntax</p> <pre>dbtype filename 1</pre> <p>For example,</p> <pre>dbtype addpath 1</pre> <p>returns</p> <pre>1    function oldpath = addpath(varargin)</pre>
<b>See Also</b>	<code>dbclear</code>   <code>dbcont</code>   <code>dbdown</code>   <code>dbquit</code>   <code>dbstack</code>   <code>dbstatus</code>   <code>dbstep</code>   <code>dbstop</code>   <code>dbup</code>

# 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 you 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

$y'(t) = f(t, y(t), y(t - \tau_1), \dots, y(t - \tau_k))$ . 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  $y(t - \tau_j)$  for delay  $\tau_j = \text{lags}(j)$ . The output is a column vector

corresponding to  $f(t, y(t), y(t - \tau_1), \dots, y(t - \tau_k))$ .

lags

Vector of constant, positive delays  $\tau_1, \dots, \tau_k$ .

history

Specify history in one of three ways:

- A function of  $t$  such that  $y = \text{history}(t)$  returns the solution  $y(t)$  for  $t \leq t_0$  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

<code>tspan</code>	Interval of integration from <code>t0=tspan(1)</code> to <code>tf=tspan(end)</code> with <code>t0 &lt; tf</code> .
<code>options</code>	Optional integration argument. A structure you create using the <code>dde23set</code> function. See <code>dde23set</code> for details.

## Description

`sol = dde23(ddefun,lags,history,tspan)` integrates the system of DDEs

$$y'(t) = f(t, y(t), y(t - \tau_1), \dots, y(t - \tau_k))$$

on the interval  $[t_0, t_f]$ , where  $\tau_1, \dots, \tau_k$  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.

<code>sol.x</code>	Mesh selected by <code>dde23</code>
<code>sol.y</code>	Approximation to $y(x)$ at the mesh points in <code>sol.x</code> .
<code>sol.yp</code>	Approximation to $y'(x)$ at the mesh points in <code>sol.x</code>
<code>sol.solver</code>	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 `dde23set`. See `dde23set` 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  $t_0$  (the history) or in coefficients of the equations at known values of  $t$  after  $t_0$ .

Use the 'Events' option to specify a function that dde23 calls to find where functions  $g(t, y(t), y(t - \tau_1), \dots, y(t - \tau_k))$  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 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:

<code>sol.xe</code>	Row vector of locations of all events, i.e., times when an event function vanished
<code>sol.ye</code>	Matrix whose columns are the solution values corresponding to times in <code>sol.xe</code>
<code>sol.ie</code>	Vector containing indices that specify which event occurred at the corresponding time in <code>sol.xe</code>

## 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  $t \leq 0$ .

---

**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](http://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`

# ddensd

## 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

$$y'(t) = f(t, y(t), y(dy_1), \dots, y(dy_p), y'(dyp_1), \dots, y'(dyp_q)) \quad (2-1)$$

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.

<b>ddefun Argument</b>	<b>Description</b>
<code>t</code>	A scalar value representing the current value of time, $t$ .
<code>y</code>	A vector that represents $y(t)$ in Equation 2-1. The size of this vector is $n$ -by-1, where $n$ is the number of equations in the system you want to solve.

<b>ddefun Argument</b>	<b>Description</b>
ydel	A matrix whose columns, <code>ydel(:, i)</code> , 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 2-1.
ypdel	A matrix whose columns, <code>ypdel(:, j)</code> 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 2-1.
yp	The result returned by <code>ddefun</code> . It is an n-by-1 vector whose elements represent the right side of Equation 2-1.

### **dely - Solution delays**

function handle | vector

Solution delays, specified as a function handle, which returns  $dy_1, \dots, dy_p$  in Equation 2-1. 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.

<b>dely Argument</b>	<b>Description</b>
t	A scalar value representing the current value of time, $t$ .
y	A vector that represents $y(t)$ in Equation 2-1. 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 2-1. 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  $\text{dely}(i) = \tau_i$ . 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  $dy_{p_1}, \dots, dy_{p_q}$  in Equation 2-1. 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.

<b>delyp Argument</b>	<b>Description</b>
<b>t</b>	A scalar value representing the current value of time, $t$ .
<b>y</b>	A vector that represents $y(t)$ in Equation 2-1. The size of this vector is n-by-1, where n is the number of equations in the system you want to solve.
<b>dyp</b>	A vector returned by the <code>delyp</code> function whose values are the derivative delays, $dyp_j$ , in Equation 2-1. 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 <code>dyp</code> must be less than $t$ . There is one exception to this restriction: if you are solving an initial value DDE, the value of <code>dyp</code> can equal $t$ at $t = t_0$ . For more information, see “Initial Value Neutral Delay Differential Equations” on page 2-1207.

If you want specify constant derivative delays having the form  $dyp_j = t - \tau_j$ , then `delyp` must be a vector, where `delyp(j) =  $\tau_j$` . 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 2-1207 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  $t \leq t_0$ . 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, `yp0`, is a column vector whose elements are the initial derivatives,  $y'(t_0)$ . These vectors must be consistent, meaning that they satisfy Equation 2-1 at  $t_0$ . See “Initial Value Neutral Delay Differential Equations” on page 2-1207 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.

<code>sol.x</code>	Mesh selected by <code>ddensd</code> .
<code>sol.y</code>	An approximation to $y(t)$ at the mesh points.
<code>sol.yp</code>	An approximation to $y'(t)$ at the mesh points.
<code>sol.solver</code>	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  $y'(dyp_j) = y'(t_0)$ :

$$y'(t_0) = f(t_0, y(t_0), y(t_0), \dots, y(t_0), y'(t_0), \dots, y'(t_0)) \tag{2-2}$$

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 2-2 at  $t_0$ .

## Examples

### Neutral DDE with Two Delays

Solve the following neutral DDE, presented by Paul [1], for  $0 \leq t \leq \pi$ :

$$y'(t) = 1 + y(t) - 2y(t/2)^2 - y'(t - \pi)$$

with history:  $y(t) = \cos(t)$  for  $t \leq 0$ .

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);
```



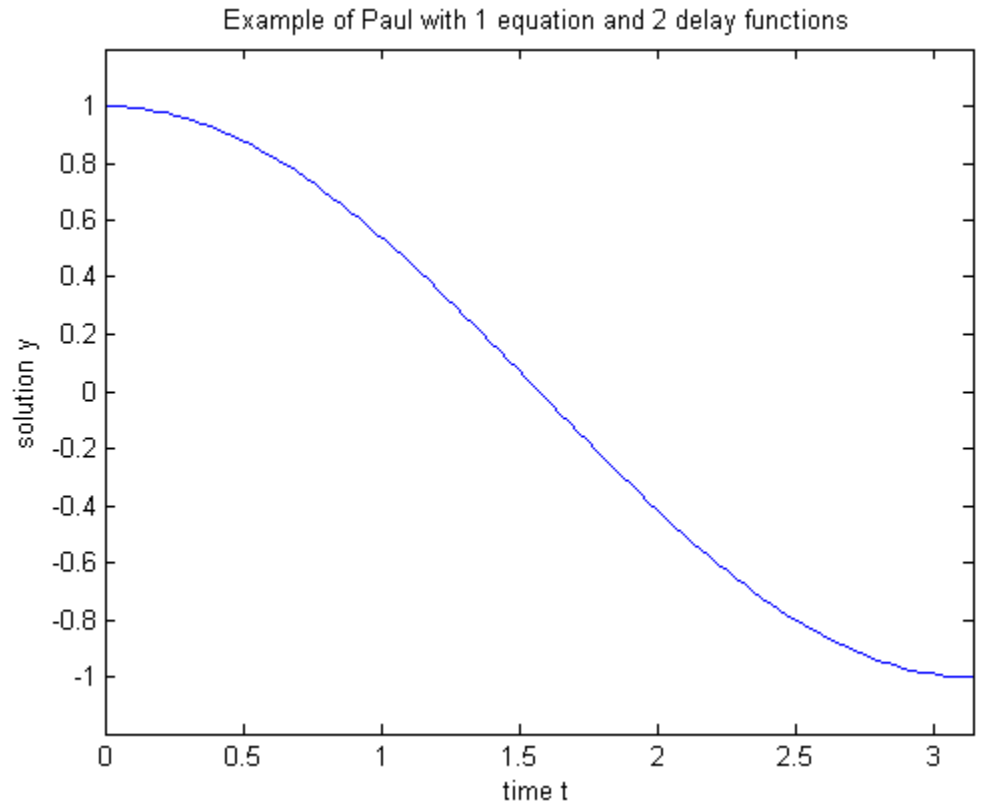
Evaluate the solution at 100 equally spaced points between 0 and  $\pi$ .  
Add this code to the main function.

```
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**

<code>ddefun</code>	Function handle that evaluates the right side of the differential equations $y'(t) = f(t,y(t),y(d(1)),\dots,y(d(k)))$ . The function must have the form  <code>dydt = ddefun(t,y,Z)</code>  where <code>t</code> corresponds to the current $t$ , <code>y</code> is a column vector that approximates $y(t)$ , and <code>Z(:,j)</code> approximates $y(d(j))$ for delay $d(j)$ given as component $j$ of <code>delays(t,y)</code> . The output is a column vector corresponding to $f(t,y(t),y(d(1)),\dots,y(d(k)))$ .
<code>delays</code>	Function handle that returns a column vector of delays $d(j)$ . The delays can depend on both $t$ and $y(t)$ . <code>ddesd</code> 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 <code>delays</code> to a constant vector <code>delays(j) = <math>\tau_j</math></code> . With delay functions of this form, <code>ddesd</code> is used exactly like <code>dde23</code> .

history	Specify history in one of three ways: <ul style="list-style-type: none"> <li>• A function of <math>t</math> such that <math>y = \text{history}(t)</math> returns the solution <math>y(t)</math> for <math>t \leq t_0</math> as a column vector</li> <li>• A constant column vector, if <math>y(t)</math> is constant</li> <li>• The solution <code>sol</code> from a previous integration, if this call continues that integration</li> </ul>
tspan	Interval of integration from <code>t0=tspan(1)</code> to <code>tf=tspan(end)</code> with <code>t0 &lt; tf</code> .
options	Optional integration argument. A structure you create using the <code>ddeset</code> function. See <code>ddeset</code> for details.

## Description

`sol = ddesd(ddefun,delays,history,tspan)` integrates the system of DDEs

$$y'(t) = f(t, y(t), y(d(1)), \dots, y(d(k)))$$

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.

<code>sol.x</code>	Mesh selected by <code>ddesd</code>
<code>sol.y</code>	Approximation to $y(x)$ at the mesh points in <code>sol.x</code> .
<code>sol.yp</code>	Approximation to $y'(x)$ at the mesh points in <code>sol.x</code>
<code>sol.solver</code>	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  $g(t,y(t),y(d(1)),\dots,y(d(k)))$  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:

<code>sol.xe</code>	Row vector of locations of all events, i.e., times when an event function vanished
<code>sol.ye</code>	Matrix whose columns are the solution values corresponding to times in <code>sol.xe</code>
<code>sol.ie</code>	Vector containing indices that specify which event occurred at the corresponding time in <code>sol.xe</code>

## 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`:

# ddesd

---

```
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(olddopts,'name1',value1,...)
options = ddeset(olddopts,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(olddopts,'name1',value1,...)` alters an existing options structure `olddopts`. This overwrites any values in `olddopts` that are specified using `name/value` pairs and returns the modified structure as the output argument.

`options = ddeset(olddopts,newopts)` combines an existing options structure `olddopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `olddopts`.

`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`.

$$|e(i)| * \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

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

Property	Value	Description
RelTol	Positive scalar {1e-3}	<p>A relative error tolerance that applies to all components of the solution vector <math>y</math>. 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 <math>AbsTol(i)</math>. The default, <math>1e-3</math>, corresponds to 0.1% accuracy.</p> <p>The estimated error in each integration step satisfies <math> e(i)  \leq \max(RelTol * abs(y(i)), AbsTol(i))</math>.</p>
AbsTol	Positive scalar or vector {1e-6}	<p>Absolute error tolerances that apply to the individual components of the solution vector. <math>AbsTol(i)</math> is a threshold below which the value of the <math>i</math>th 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 <math>y(i)</math> when it is small, you may have to specify <math>AbsTol(i)</math> small enough to get some correct digits in <math>y(i)</math> so that you can accurately compute more interesting components.</p> <p>If <math>AbsTol</math> is a vector, the length of <math>AbsTol</math> must be the same as the length of the solution vector <math>y</math>. If <math>AbsTol</math> is a scalar, the value applies to all components of <math>y</math>.</p>
NormControl	on   {off}	<p>Control error relative to norm of solution. Set this property on to request that the solvers control the error in each integration step with <math>norm(e) \leq \max(RelTol * norm(y), AbsTol)</math>. By default, the solvers use a more stringent component-wise error control.</p>

### 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}	<p>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,</p> <pre>options = ddeset('OutputFcn',... @myfun)</pre> <p>sets 'OutputFcn' to @myfun, a handle to the function myfun. See the <a href="#">function_handle</a> reference page for more information.</p> <p>The output function must be of the form</p> <pre>status = myfun(t,y,flag)</pre> <p>“Parameterizing Functions” explains how to provide additional parameters to myfun, if necessary.</p> <p>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:</p> <ul style="list-style-type: none"><li>• <code>init</code> — The solver calls <code>myfun(tspan,y0,'init')</code> before beginning the integration to allow the output function to initialize. <code>tspan</code> is the input argument to the solvers. <code>y0</code> is the initial value of the solution, either from <code>history(t0)</code> or specified in the <code>initialY</code> option.</li><li>• <code>{none}</code> — The solver calls <code>status = myfun(t,y)</code> after each integration step on which output is requested. <code>t</code> contains points where output was</li></ul>

**DDE Solver Output Properties (Continued)**

Property	Value	Description
		<p>generated during the step, and <i>y</i> is the numerical solution at the points in <i>t</i>. If <i>t</i> is a vector, the <i>ith</i> column of <i>y</i> corresponds to the <i>ith</i> element of <i>t</i>.</p> <p><i>myfun</i> must return a <b>status</b> output value of 0 or 1. If <code>literal &gt; status</code>, the solver halts integration. You can use this mechanism, for instance, to implement a <b>Stop</b> button.</p> <ul style="list-style-type: none"> <li>• <b>done</b> — The solver calls <code>myfun([], [], 'done')</code> when integration is complete to allow the output function to perform any cleanup chores.</li> </ul> <p>You can use these general purpose output functions or you can edit them to create your own. Type <code>help functionname</code> at the command line for more information.</p> <ul style="list-style-type: none"> <li>• <b>odeplot</b> – time series plotting (default when you call the solver with no output argument and you have not specified an output function)</li> <li>• <b>odephas2</b> – two-dimensional phase plane plotting</li> <li>• <b>odephas3</b> – three-dimensional phase plane plotting</li> <li>• <b>odeprint</b> – print solution as the solver computes it</li> </ul>
OutputSel	Vector of indices	<p>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 <code>odeplot</code> output function, but you want to plot only the first and third components of the solution, you can do this using</p> <pre>options = ddeaset... ('OutputFcn',@odeplot,...</pre>

## DDE Solver Output Properties (Continued)

Property	Value	Description
		<code>'OutputSel',[1 3]);</code>  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, <code>Stats</code> is <code>off</code> . If it is <code>on</code> , after solving the problem the solver displays: <ul style="list-style-type: none"><li>• The number of successful steps</li><li>• The number of failed attempts</li><li>• The number of times the DDE function was called</li></ul>

## 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. <code>InitialStep</code> sets an upper bound on the magnitude of the first step size the solver tries. If you do not set <code>InitialStep</code> , the solver bases the initial step size on the slope of the solution at the initial time <code>tspan(1)</code> . 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 <code>InitialStep</code> .
MaxStep	Positive scalar {0.1* abs(t0-tf)}	<p>Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set <code>MaxStep</code> 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 <i>not</i> reduce <code>MaxStep</code>:</p> <ul style="list-style-type: none"> <li>• When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance <code>RelTol</code>, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector <code>AbsTol</code>. (See “Error Control Properties” on page 2-1218 for a description of the error tolerance properties.)</li> <li>• 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 <code>RelTol</code> and <code>AbsTol</code>. Use <code>MaxStep</code> as a last resort.</li> </ul>

**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	<p>A function handle that includes one or more event functions. For <code>dde23</code> and <code>ddesd</code>, this function has the following syntax:</p> <pre>[value, isterminal, direction] = events(t, y, YDEL)</pre> <p>For <code>ddensd</code>, the syntax is:</p> <pre>[value, isterminal, direction] = events(t, y, YDEL, YPDEL)</pre> <p>The output arguments, <code>value</code>, <code>isterminal</code>, and <code>direction</code>, are vectors for which the <code>ith</code> element corresponds to the <code>ith</code> event function:</p> <ul style="list-style-type: none"><li>• <code>value(i)</code> is the value of the <code>ith</code> event function.</li><li>• <code>isterminal(i) = 1</code> if you want the integration to terminate at a zero of this event function, and <code>0</code> otherwise.</li><li>• <code>direction(i) = 0</code> if you want the solver to locate all zeros (the default), <code>+1</code> if only zeros where the event function is increasing, and <code>-1</code> if only zeros where the event function is decreasing.</li></ul> <p>If you specify an events function and events are detected, the solver returns three additional fields in the solution structure <code>sol</code>:</p> <ul style="list-style-type: none"><li>• <code>sol.xe</code> is a row vector of times at which events occur.</li><li>• <code>sol.ye</code> is a matrix whose columns are the solution values corresponding to times in <code>sol.xe</code>.</li><li>• <code>sol.ie</code> is a vector containing indices that specify which event occurred at the corresponding time in <code>sol.xe</code>.</li></ul>



### DDE Events Property (Continued)

String	Value	Description
		For examples that use an event function while solving ordinary differential equation problems, see “Event Location” ( <code>ballode</code> ) and “Advanced Event Location” ( <code>orbitode</code> ), 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
<code>Jumps</code>	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 <code>dde23</code> solver.
<code>InitialY</code>	Vector	Initial value of solution. By default the initial value of the solution is the value returned by <code>history</code> at the initial point. Supply a different initial value as the value of the <code>InitialY</code> 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);
```

# ddeset

---

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](#)

# dec2base

---

**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];
```

<code>A(:)</code>	<code>dec2hex(A)</code>
<code>ans =</code>	<code>ans =</code>
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](#)

# decic

---

## 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,resnorm] = 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  $f(t0, y0mod, yp0mod) = 0$ , 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,resnorm] = decic(odefun,t0,y0,fixed\_y0,yp0,fixed\_yp0...) returns the norm of odefun(t0,y0mod,yp0mod) as resnorm. 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**  $[q,r] = \text{deconv}(v,u)$

**Description**  $[q,r] = \text{deconv}(v,u)$  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  $v = \text{conv}(u,q)+r$ .

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**

$$L = \text{del2}(U)$$

$$L = \text{del2}(U, h)$$

$$L = \text{del2}(U, h_x, h_y)$$

$$L = \text{del2}(U, h_x, h_y, h_z, \dots)$$

**Definitions** If the matrix  $U$  is regarded as a function  $u(x,y)$  evaluated at the point on a square grid, then  $4*\text{del2}(U)$  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:

$$l_{ij} = \frac{1}{4} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - u_{i,j}$$

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables  $u(x,y,z,\dots)$ ,  $\text{del2}(U)$  is an approximation,

$$l = \frac{\nabla^2 u}{2N} = \frac{1}{2N} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \dots \right)$$

where  $N$  is the number of variables in  $u$ .

**Description**  $L = \text{del2}(U)$  where  $U$  is a rectangular array is a discrete approximation of

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)$$

# del2

---

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.

$L = \text{del2}(U)$  when  $U$  is an multidimensional array, returns an approximation of

$$\frac{\nabla^2 u}{2N}$$

where  $N$  is  $\text{ndims}(u)$ .

$L = \text{del2}(U, h)$  where  $H$  is a scalar uses  $H$  as the spacing between points in each direction ( $h=1$  by default).

$L = \text{del2}(U, hx, hy)$  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  $\text{size}(u, 2)$  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  $\text{size}(u, 1)$  and specifies the y-coordinates of the points.

$L = \text{del2}(U, hx, hy, hz, \dots)$  where  $U$  is multidimensional uses the spacing given by  $hx, hy, hz, \dots$

## 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** Delaunay triangulation in 2-D and 3-D

**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 Construct Delaunay triangulation

<b>Methods</b>	convexHull	Convex hull
	inOutStatus	Status of triangles in 2-D constrained Delaunay triangulation
	nearestNeighbor	Point closest to specified location
	pointLocation	Simplex containing specified location
	voronoiDiagram	Voronoi diagram

## Inherited methods

baryToCart	Convert point coordinates from barycentric to Cartesian
cartToBary	Convert point coordinates from cartesian to barycentric
circumcenters	Circumcenters of specified simplices
edgeAttachments	Simplices attached to specified edges
edges	Triangulation edges
faceNormals	Unit normals to specified triangles
featureEdges	Sharp edges of surface triangulation
freeBoundary	Facets referenced by only one simplex
incenters	Incenters of specified simplices
isEdge	Test if vertices are joined by edge
neighbors	Simplex neighbor information
size	Size of triangulation matrix
vertexAttachments	Return simplices attached to specified vertices

# DelaunayTri

---

## Properties

Constraints	<p>Constraints is a <code>numc-by-2</code> matrix that defines the constrained edge data in the triangulation, where <code>numc</code> is the number of constrained edges. Each constrained edge is defined in terms of its endpoint indices into <code>X</code>.</p> <p>The constraints can be specified when the triangulation is constructed or can be imposed afterwards by directly editing the constraints data.</p> <p>This feature is only supported for 2-D triangulations.</p>
<code>X</code>	<p>The dimension of <code>X</code> is <code>mpts-by-ndim</code>, where <code>mpts</code> is the number of points and <code>ndim</code> is the dimension of the space where the points reside. If column vectors of <code>x,y</code> or <code>x,y,z</code> coordinates are used to construct the triangulation, the data is consolidated into a single matrix <code>X</code>.</p>
Triangulation	<p>Triangulation is a matrix representing the set of simplices (triangles or tetrahedra etc.) that make up the triangulation. The matrix is of size <code>mtri-by-nv</code>, where <code>mtri</code> is the number of simplices and <code>nv</code> 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 <code>X</code>, where <code>X</code> is the array of point coordinates.</p>

**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**

TriScatteredInterp

# DelaunayTri

---

**Purpose** Construct Delaunay triangulation

**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.

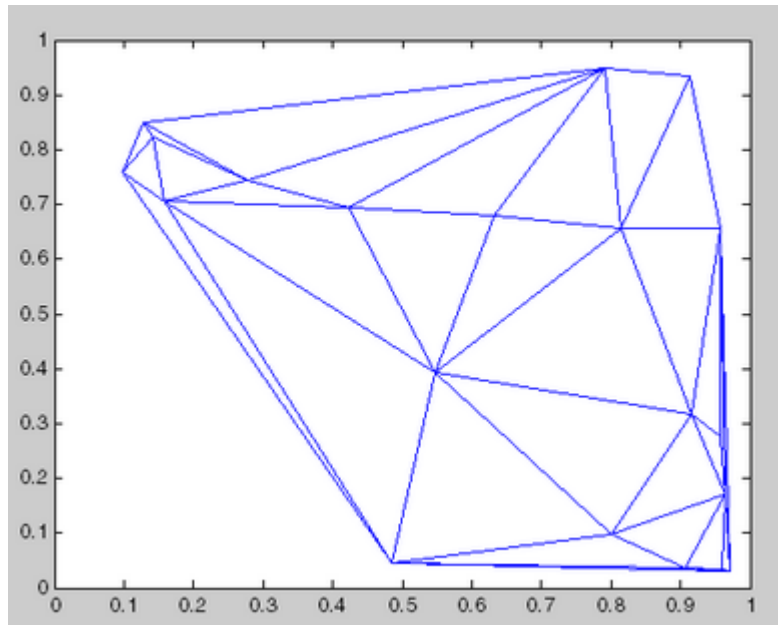
**Definitions**

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

[TriScatteredInterp](#) | [TriRep](#)

# delaunay

---

**Purpose** Delaunay triangulation

---

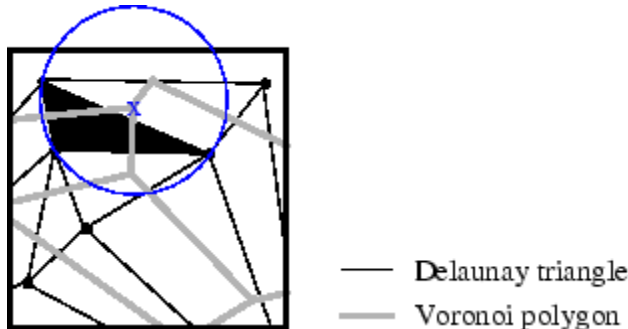
**Note** Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `delaunay`.

---

**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,  $2 \leq \text{ndim} \leq 3$ . 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 `DelaunayTri` instead.

## Visualization

Use one of these functions to plot the output of `delaunay`:

- |                      |                                                                                                                                                                                                                        |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>triplot</code> | Displays the triangles defined in the <code>m-by-3</code> matrix <code>TRI</code> .                                                                                                                                    |
| <code>trisurf</code> | Displays each triangle defined in the <code>m-by-3</code> matrix <code>TRI</code> 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)))
```

# delaunay

---

`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.

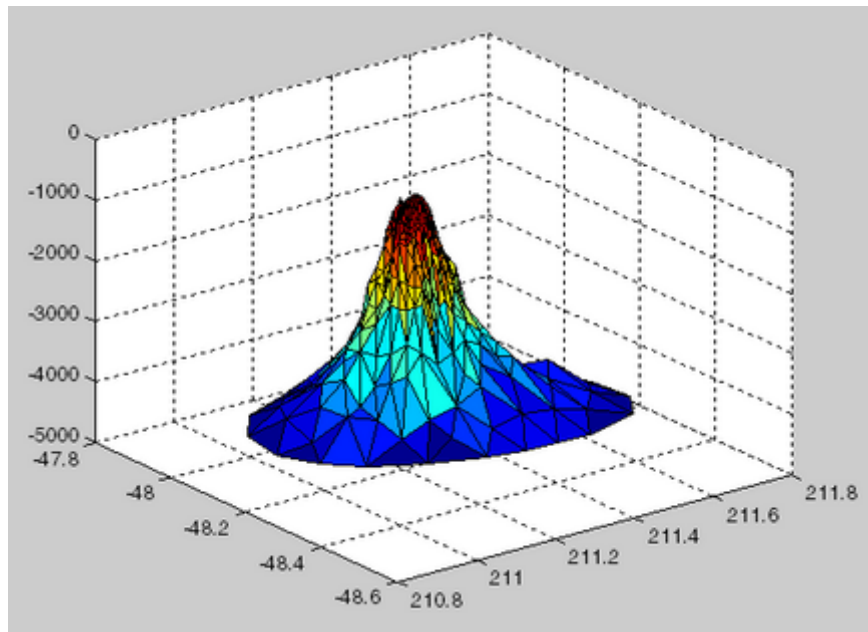
`tetramesh` Plots a triangulation composed of tetrahedra.

## Examples

### Example

Plot the Delaunay triangulation of a large dataset:

```
load seamount
tri = delaunay(x,y);
trisurf(tri,x,y,z);
```



## See Also

[DelaunayTri](#) | [TriScatteredInterp](#) | [plot](#) | [triplot](#) | [trimesh](#) | [trisurf](#)

# delaunay3

---

**Purpose** 3-D Delaunay triangulation

---

**Note** `delaunay3` has been removed. Use `delaunay` instead.

---

**Syntax** `T = delaunay3(x,y,z)`

**Description** `T = delaunay3(x,y,z)` returns an array `T`, each row of which contains the indices of the points in  $(x,y,z)$  that make up a tetrahedron in the triangulation of  $(x,y,z)$ . `T` is a `numtes`-by-4 array where `numtes` is the number of facets in the triangulation. `x`, `y`, and `z` are vectors of equal length. If the original data points are collinear or `x`, `y`, and `z` define an insufficient number of points, the triangles cannot be computed and `delaunay3` returns an empty matrix.

`T = delaunay3(x,y,z,options)` specifies a cell array of strings `options` that were previously used by `Qhull`. `Qhull`-specific options are no longer required and are currently ignored.

**Visualization** Use `tetramesh` to plot `delaunay3` output. `tetramesh` displays the tetrahedrons defined in `T` as mesh. `tetramesh` uses the default transparency parameter value `'FaceAlpha' = 0.9`.

## Examples **Example 1**

This example generates a 3-dimensional Delaunay triangulation, then uses `tetramesh` to plot the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
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.  
Tes = delaunay3(x,y,z)
```

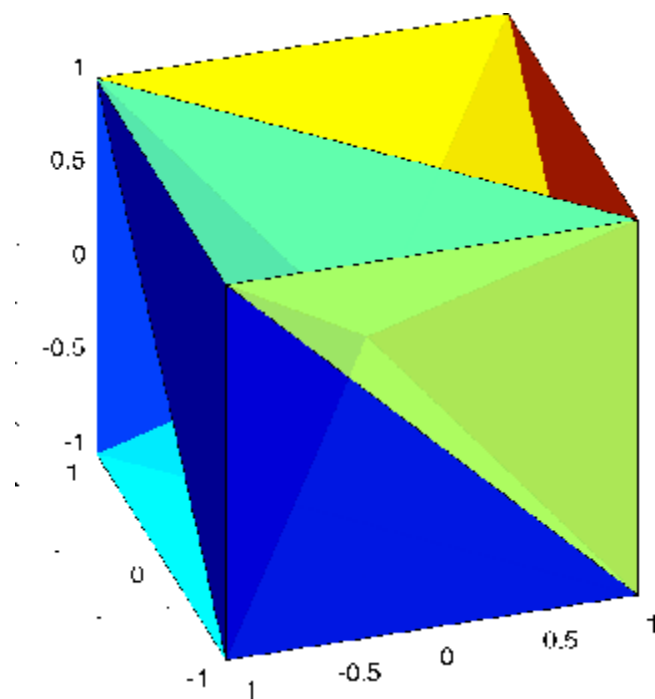
Tes =

9	1	5	6
3	9	1	5
2	9	1	6
2	3	9	4
2	3	9	1
7	9	5	6
7	3	9	5
8	7	9	6
8	2	9	6
8	2	9	4
8	3	9	4
8	7	3	9

```
X = [x(:) y(:) z(:)];  
tetramesh(Tes,X);camorbit(20,0)
```

# delaunay3

---



## See Also

[delaunay](#) | [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` as you would for `delaunay3`.

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
```

# delaunayn

---

```
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)
```

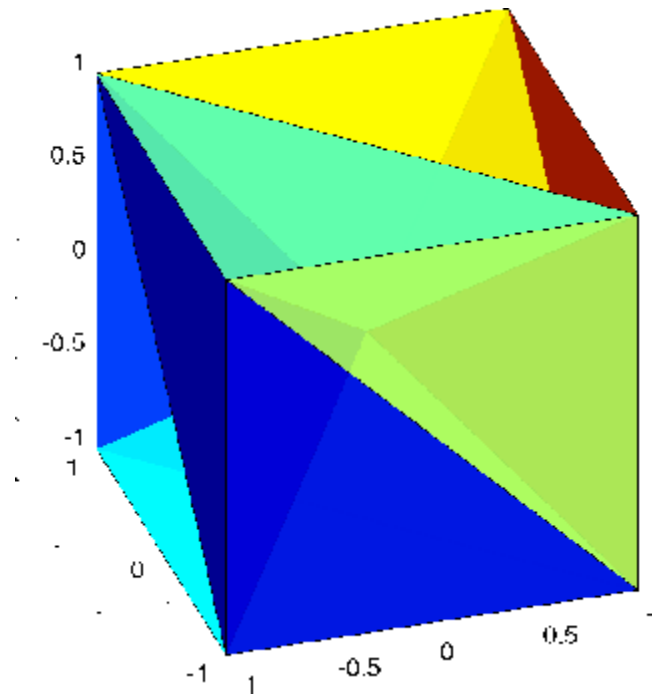
```
Tes =
```

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**

[DelaunayTri](#) | [convhulln](#) | [delaunay3](#) | [tetramesh](#) | [voronoin](#)

# delete

---

<b>Purpose</b>	Remove files or graphics objects
<b>Alternatives</b>	As an alternative to the <code>delete</code> function, use the Current Folder browser.
<b>Syntax</b>	<pre>delete('fileName1', 'filename2', ...) delete(h) delete(handle_array) delete fileName</pre>
<b>Description</b>	<p><code>delete('fileName1', 'filename2', ...)</code> deletes the files <code>fileName1</code>, <code>fileName2</code>, and so on, from the disk. <code>fileName</code> is a string and can be an absolute path or a path relative to the current folder. <code>fileName</code> also can include wildcards (*).</p> <p><code>delete(h)</code> deletes the graphics object with handle <code>h</code>. <code>h</code> 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.</p> <p><code>delete(handle_array)</code> is a method of the <code>handle</code> class. It removes from memory the handle objects referenced by <code>handle_array</code>.</p> <p>When deleted, any references to the objects in <code>handle_array</code> become invalid. To remove the handle variables, use the <code>clear</code> function.</p> <p><code>delete fileName</code> is the command syntax. Delete multiple files by appending filenames, separated by spaces. When filenames contain space characters, you must use the functional form.</p> <p>As <code>delete</code> 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 <code>delete</code>, use the <b>General</b> preference for <b>Deleting files</b>, or the <code>recycle</code> function.</p> <p>The <code>delete</code> function deletes files and graphics objects only. To delete folders, use <code>rmdir</code>.</p>

**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.

**Examples** Create a Microsoft Calendar application. Then create a `TitleFont` interface and use it to change the appearance of the font of the calendar's title:

```
f = figure('position',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);
TFont = cal.TitleFont
```

MATLAB software displays information similar to:

```
TFont =
  Interface.Microsoft_Forms_2.0_Object_Library.Font
```

Make the following changes and observe the results:

```
TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the `TitleFont` interface:

```
TFont.release;
```

Now create a `GridFont` interface and use it to modify the size of the calendar's date numerals:

```
GFont = cal.GridFont
```

MATLAB displays:

```
GFont =  
    Interface.Microsoft_Forms_2.0_Object_Library.Font
```

Make the following changes and observe the results:

```
GFont.Size = 16;
```

When you're done, delete the `cal` object and the figure window. Deleting the `cal` object also releases all interfaces to the object (for example, `GFont`):

```
cal.delete;  
delete(f);  
clear f;
```

Note that, although the object and interfaces themselves have been destroyed, the variables assigned to them still reside in the MATLAB workspace until you remove them with `clear`:

```
whos
```

MATLAB displays (in part):

Name	Size	Bytes	Class
GFont	1x1	0	handle
TFont	1x1	0	handle
cal	1x1	0	handle

## See Also

[release](#) | [save \(COM\)](#) | [load \(COM\)](#) | [actxcontrol](#) | [actxserver](#)

# FTP.delete

---

**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`

**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`

# delete (serial)

---

**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`



**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`

# deleteproperty

---

**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](#)

# delevent

---

**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 `n`th `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`

**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`

# demo

---

**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`

# demdir

---

**Purpose** List dependent folders for function or P-file

**Syntax**

```
list = demdir('file_name')
[list, prob_files, prob_sym,
  prob_strings] = demdir('file_name')
[...] = demdir('file_name1', 'file_name2',...)
```

**Description** The demdir 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 = demdir('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] = demdir('file_name')` creates three additional cell arrays containing information about any problems with the demdir search. `prob_files` contains filenames that demdir was unable to parse. `prob_sym` contains symbols that demdir was unable to find. `prob_strings` contains callback strings that demdir was unable to parse.

`[...] = demdir('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 = demdir('mesh')
```

**See Also** depfun



**Purpose**

List dependencies of function or P-file

**Syntax**

```
list = depfun('fun')
[list, builtins, classes] = depfun('fun')
[list, builtins, classes, prob_files, prob_sym, eval_strings,
    called_from, java_classes] = depfun('fun')
[...] = depfun('fun1', 'fun2',...)
[...] = depfun({'fun1', 'fun2', ...})
[...] = depfun('fig_file')
[...] = depfun(..., options)
```

**Description**

The `depfun` function lists the paths of all files a specified function or P-file needs to operate.

---

**Note** It cannot be guaranteed that `depfun` will find every dependent file. Some dependent files can be hidden in callbacks, or can be constructed dynamically for evaluation, for example. Also note that the list of functions returned by `depfun` often includes extra files that would never be called if the specified function were actually evaluated.

---

`list = depfun('fun')` creates a cell array of strings containing the paths of all the files that function `fun` uses. This includes the second-level files that are called directly by `fun`, and the third-level files that are called by the second-level files, and so on.

Function `fun` must be on the MATLAB path, as determined by the `which` function. If the MATLAB path contains any relative folders, then files in those folders will also have a relative path.

---

**Note** If MATLAB returns a parse error for any of the input functions, or if the `prob_files` output below is nonempty, then the rest of the output of `depfun` might be incomplete. You should correct the problematic files and invoke `depfun` again.

---

`[list, builtins, classes] = depfun('fun')` creates three cell arrays containing information about dependent functions. `list` contains the paths of all the files that function `fun` and its subordinates use. `builtins` contains the built-in functions that `fun` and its subordinates use. `classes` contains the MATLAB classes that `fun` and its subordinates use.

`[list, builtins, classes, prob_files, prob_sym, eval_strings, called_from, java_classes] = depfun('fun')` creates additional cell arrays or structure arrays containing information about any problems with the `depfun` search and about where the functions in `list` are invoked. The additional outputs are

- `prob_files` — Indicates which files `depfun` was unable to parse, find, or access. Parsing problems can arise from MATLAB syntax errors. `prob_files` is a structure array having these fields:
  - `name` (path to the file)
  - `listindex` (index of the file in `list`)
  - `errmsg` (problems encountered)
- `unused` — This is a placeholder for an output argument that is not fully implemented at this time. MATLAB returns an empty structure array for this output.
- `called_from` — Cell array of the same length as `list` that indicates which functions call other functions. This cell array is arranged so that the following statement returns all functions in function `fun` that invoke the function `list{i}`:

```
list(called_from{i})
```

- `java_classes` — Cell array of Java class names used by `fun` and its subordinate functions.

`[...] = depfun('fun1', 'fun2', ...)` performs the same operation for multiple functions. The dependent functions of all files are listed together in the output arrays.

[...] = depfun({'fun1', 'fun2', ...}) performs the same operation, but on a cell array of functions. The dependent functions of all files are listed together in the output array.

[...] = depfun('fig\_file') looks for dependent functions among the callback strings of the GUI elements that are defined in the figure file named fig\_file.

[...] = depfun(..., options) modifies the depfun operation according to the options specified (see table below).

Option	Description
'-all'	Computes all possible left-side arguments and displays the results in the report(s). Only the specified arguments are returned.
'-calltree'	Returns a call list in place of a called_from list. This is derived from the called_from list as an extra step.
'-print', 'file'	Prints a full report to file.
'-expand'	Includes both indices and full paths in the call or called_from list. Requires the '-print', 'file' option. Will not appear in Command Window.
'-quiet'	Displays only error and warning messages, and not a summary report.
'-toponly'	Examines <i>only</i> the files listed explicitly as input arguments. It does not examine the files on which they depend.
'-verbose'	Outputs additional internal messages.

## Examples

```
list = depfun('mesh'); % Files mesh.m depends on
list = depfun('mesh','-toponly') % Files mesh.m depends on
directly
[list,builtins,classes] = depfun('gca');
```

# depfun

---

## **See Also**

depdir

<b>Purpose</b>	Matrix determinant
<b>Syntax</b>	<code>d = det(X)</code>
<b>Description</b>	<code>d = det(X)</code> returns the determinant of the square matrix <code>X</code> .
<b>Tips</b>	Testing singularity using <code>abs(det(X)) &lt;= tolerance</code> is not recommended as it is difficult to choose the correct tolerance. The function <code>cond(X)</code> can check for singular and nearly singular matrices.
<b>Algorithms</b>	<p>The determinant is computed from the triangular factors obtained by Gaussian elimination</p> <pre>[L,U] = lu(A) s = det(L)           % This is always +1 or -1 det(A) = s*prod(diag(U))</pre>
<b>Examples</b>	<p>The statement <code>A = [1 2 3; 4 5 6; 7 8 9]</code> produces</p> <pre>A =      1     2     3      4     5     6      7     8     9</pre> <p>This happens to be a singular matrix, so <code>det(A)</code> produces a very small number. Changing <code>A(3,3)</code> with <code>A(3,3) = 0</code> turns <code>A</code> into a nonsingular matrix. Now <code>d = det(A)</code> produces <code>d = 27</code>.</p>
<b>See Also</b>	<code>cond</code>   <code>condest</code>   <code>inv</code>   <code>lu</code>   <code>rref</code>   <code>mldivide</code>   Arithmetic Operators <code>\</code> , <code>/</code>

# detrend

## Purpose

Remove linear trends

## Syntax

```
y = detrend(x)
y = detrend(x, 'constant')
y = detrend(x, 'linear', bp)
```

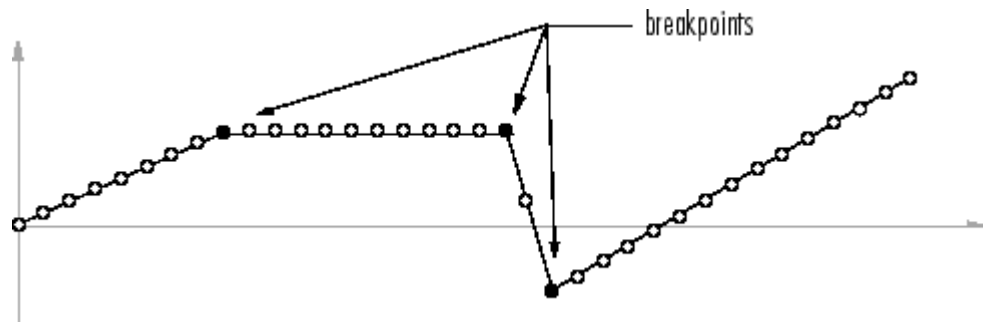
## Description

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`.

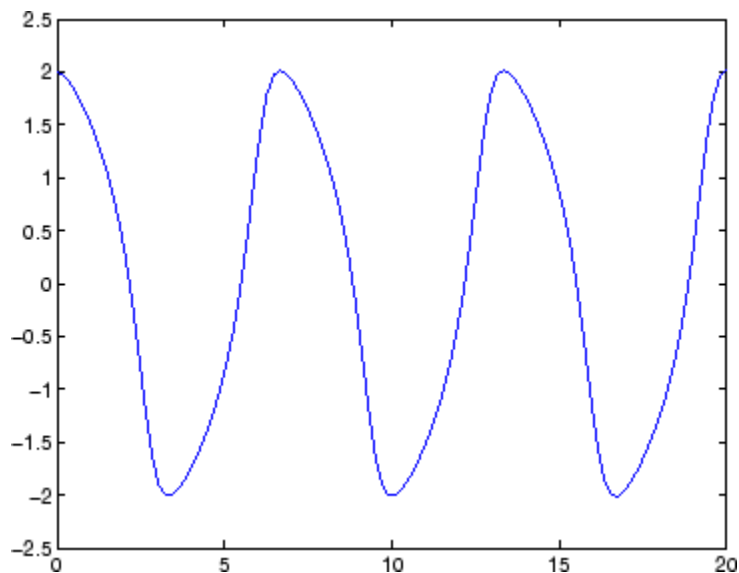
---



## 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

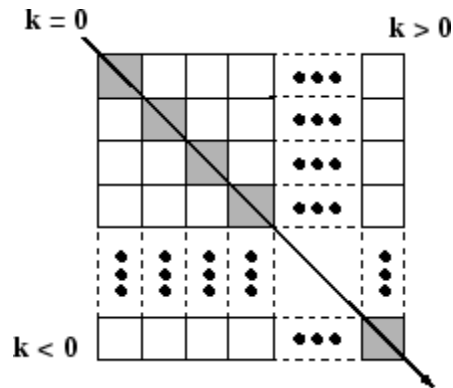
[ode45](#) | [ode23](#) | [ode113](#) | [ode15s](#) | [ode23s](#) | [ode23t](#) | [ode23tb](#) | [ode15i](#) | [dde23](#) | [ddesd](#) | [ddensd](#) | [bvp4c](#) | [bvp5c](#)

# diag

**Purpose** Diagonal matrices and diagonals of matrix

**Syntax**  
 $X = \text{diag}(v, k)$   
 $X = \text{diag}(v)$   
 $v = \text{diag}(X, k)$   
 $v = \text{diag}(X)$

**Description**  $X = \text{diag}(v, k)$  when  $v$  is a vector of  $n$  components, returns a square matrix  $X$  of order  $n + \text{abs}(k)$ , 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.



$X = \text{diag}(v)$  puts  $v$  on the main diagonal, same as above with  $k = 0$ .

$v = \text{diag}(X, k)$  for matrix  $X$ , returns a column vector  $v$  formed from the elements of the  $k$ th diagonal of  $X$ .

$v = \text{diag}(X)$  returns the main diagonal of  $X$ , same as above with  $k = 0$ .

**Tips**  $\text{diag}(\text{diag}(X))$  is a diagonal matrix.

$\text{sum}(\text{diag}(X))$  is the trace of  $X$ .

$\text{diag}([])$  generates an empty matrix,  $([])$ .

$\text{diag}(m\text{-by-}1, k)$  generates a matrix of size  $m + \text{abs}(k)$ -by- $m + \text{abs}(k)$ .

`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(gcf)), close(gcf), 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(gcf))`, `close(gcf)`, `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 2-1286 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”

# diff

---

**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 `n`th difference. Thus, `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,n,dim)` is the `n`th 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](#) | [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:  $R = \cos(\text{PSI})$  where `PSI` is the angle between the surface normal and light source.

**See Also** `specular` | `surfnorm` | `surf1`

**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: []
```

# dir

---

```
isdir: 0
datenum: []
```

The most common occurrence is on UNIX<sup>1</sup> 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`

**Purpose** Display text or array

**Syntax** `disp(X)`

**Description** `disp(X)` displays an array, without printing the array name. If `X` contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading "X=", which is not always desirable.

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 “Example 3 — Display multiple items on the same line” on page 2-1299.

Note that `disp` does not display empty arrays.

## Examples

### Example 1 — Display a matrix with column labels

This example uses `disp` to display a matrix with column labels:

```
disp('    Corn    Oats    Hay')
x = gallery('uniformdata',[5 3],0);
disp(x)
```

This results in

	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

### Example 2 — Display a hyperlink in the Command Window

You also can use the `disp` command to display a hyperlink in the Command Window. Include the full hypertext string on a single line as input to `disp`:

```
disp('<a href = "http://www.mathworks.com">MathWorks Web Site</a>')
```

which generates this hyperlink in the Command Window:

```
MathWorks Web Site
```

Click the link to display the MathWorks home page in a MATLAB Web browser.

### Example 3 – Display multiple items on the same line

Use any of the following techniques to display multiple items on the same line:

- Concatenate all substrings together using the `[]` operator. Convert any numeric values to characters using the `num2str` function:

```
name = 'Alice'; age = 12;  
str = [name, ' will be ', num2str(age), ' this year.']
```

```
Alice will be 12 this year.
```

- Use `sprintf` to create the string, and `disp` to display it. Terminate the `sprintf` command with a semicolon. This keeps "str = " from being displayed:

```
name = 'Alice'; age = 12;  
str = sprintf('%s will be %d this year.', name, age);  
disp(str);
```

```
Alice will be 12 this year.
```

- Use `fprintf` to create and display the string. Unlike the `sprintf` function, `fprintf` does not display the "str = " text. However, you do 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);
```

# disp

---

```
Alice will be 12 this year.
```

## **See Also**

```
format | int2str | num2str | rats | sprintf | fprintf | colon (:)
```

**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: []
```

## disp (timer)

---

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**

- “Overload `disp` Or `disp` and `display`”
- “Overloading Functions for Your Class”
- “Relationship Between `disp` and `display`”

# dither

---

<b>Purpose</b>	Convert image, increasing apparent color resolution by dithering
<b>Syntax</b>	<pre>X = dither(RGB, map) X = dither(RGB, map, Qm, Qe) BW = dither(I)</pre>
<b>Description</b>	<p><code>X = dither( RGB, map )</code> creates an indexed image approximation of the RGB image in the array <code>RGB</code> by dithering the colors in the colormap <code>map</code>. The colormap cannot have more than 65,536 colors.</p> <p><code>X = dither( RGB, map, Qm, Qe )</code> creates an indexed image from <code>RGB</code>, where <code>Qm</code> specifies the number of quantization bits to use along each color axis for the inverse color map, and <code>Qe</code> specifies the number of quantization bits to use for the color space error calculations. If <code>Qe &lt; Qm</code>, dithering cannot be performed, and an undithered indexed image is returned in <code>X</code>. If you omit these parameters, <code>dither</code> uses the default values <code>Qm = 5</code>, <code>Qe = 8</code>.</p> <p><code>BW = dither( I )</code> converts the grayscale image in the matrix <code>I</code> to the binary (black and white) image <code>BW</code> by dithering.</p>
<b>Class Support</b>	<p><code>RGB</code> can be <code>uint8</code>, <code>uint16</code>, <code>single</code>, or <code>double</code>. <code>I</code> can be <code>uint8</code>, <code>uint16</code>, <code>int16</code>, <code>single</code>, or <code>double</code>. All other input arguments must be <code>double</code>. <code>BW</code> is <code>logical</code>. <code>X</code> is <code>uint8</code>, if it is an indexed image with 256 or fewer colors; otherwise, it is <code>uint16</code>.</p>
<b>Algorithms</b>	<code>dither</code> increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.
<b>Examples</b>	<p>Read and display the cameraman image. Display the image as a gray scale image using <code>imagesc</code>.</p> <pre>I = imread('cameraman.tif'); figure; imagesc(I); colormap(gray);</pre>

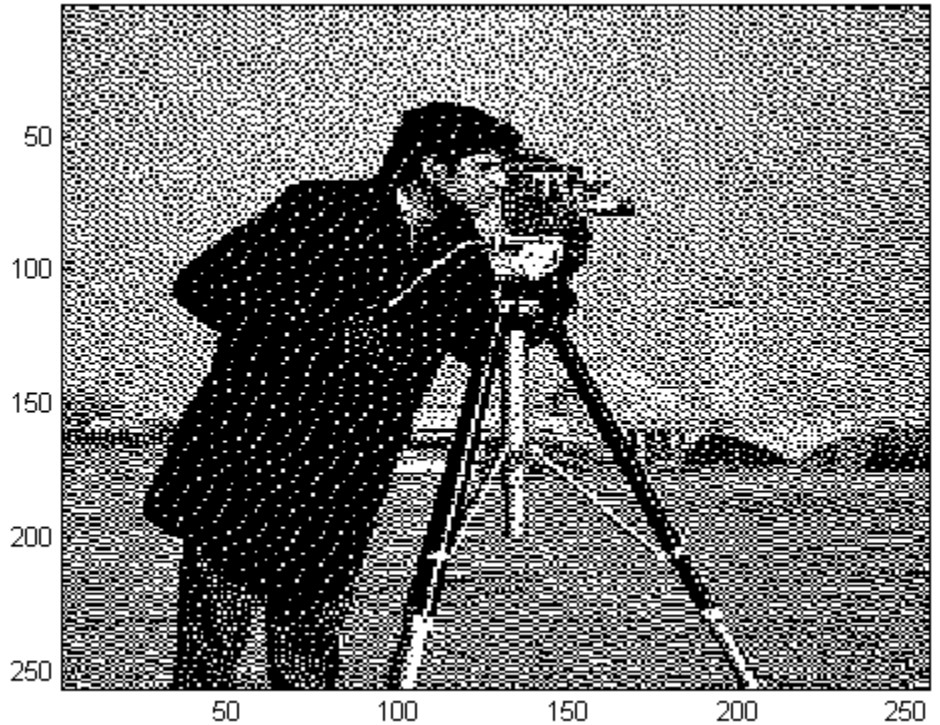


Apply dithering to get an indexed image as,

```
BW = dither(I);  
figure; imagesc(BW);  
colormap(gray);
```

# dither

---



## 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

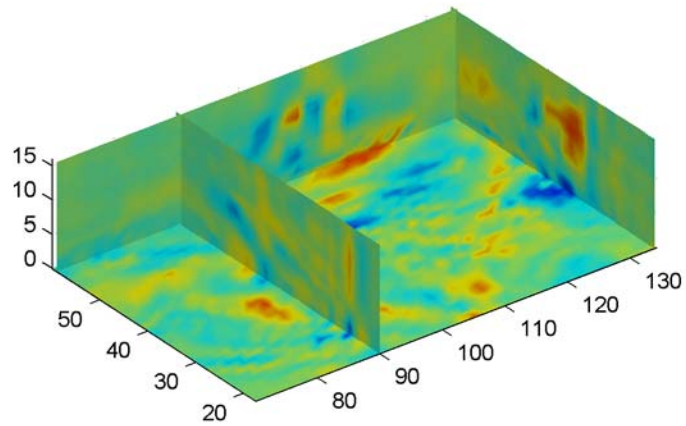
This example displays the divergence of vector volume data as slice planes, using color to indicate divergence.

```
load wind
div = divergence(x,y,z,u,v,w);
```

# divergence

---

```
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)
```

# dlmread

---

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

```

# dlmread

---

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).

# dlmwrite

Attribute	Value
<b>delimiter</b>	Delimiter string to be used in separating matrix elements
<b>newline</b>	Character(s) to use in terminating each line (see table below)
<b>roffset</b>	Offset, in rows, from the top of the destination file to where matrix data is to be written. Offset is zero based.
<b>coloffset</b>	Offset, in columns, from the left side of the destination file to where matrix data is to be written. Offset is zero based.
<b>precision</b>	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**  $p = \text{dmperm}(A)$   
 $[p,q,r,s,cc,rr] = \text{dmperm}(A)$

**Description**  $p = \text{dmperm}(A)$  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  $\text{sprank}(A) = \text{sum}(p > 0)$ .

$[p,q,r,s,cc,rr] = \text{dmperm}(A)$  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  $\text{sprank}(A) = \text{rr}(4) - 1$ , which is an upper bound on the numerical rank of A.  $\text{sprank}(A) = \text{rank}(\text{full}(\text{sprand}(A)))$  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  $b = \text{length}(r) - 1$ . 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

---

<b>Purpose</b>	Reference page in Help browser
<b>Syntax</b>	<code>doc</code> <code>doc name</code>
<b>Description</b>	<p><code>doc</code> opens the Help browser. If the Help browser is already open, but not visible, then <code>doc</code> brings it to the foreground and opens a new tab.</p> <p><code>doc name</code> displays documentation for the functionality specified by <code>name</code>, such as a function, class, block, or product.</p> <ul style="list-style-type: none"><li>• If there is a MathWorks reference page or product page corresponding to <code>name</code>, then <code>doc</code> displays the page in the Help browser. The <code>doc</code> command does not display third-party or custom HTML documentation.</li><li>• If there is no reference or product page corresponding to <code>name</code>, then <code>doc</code> searches for help text in a file named <code>name.m</code>. When help text is available, <code>doc</code> displays it in a Web browser.</li><li>• If there is no reference or product page and no help text associated with <code>name</code>, then <code>doc</code> searches the documentation for <code>name</code> and displays the search results in the Help browser.</li></ul> <p>When the browser is already open, the page displays in the most recently used tab.</p>
<b>Input Arguments</b>	<p><b>name - Name of function, class, block, product folder, or other functionality</b></p> <p>string</p> <p>Name of a function, class, block, product folder, or other functionality, specified as a string. Alternatively, an operator symbol.</p> <p>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:</p>

```
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 Toolbox™ 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”

<b>Purpose</b>	Help browser search
<b>Syntax</b>	<code>docsearch</code> <code>docsearch expression</code>
<b>Description</b>	<p><code>docsearch</code> opens the Help browser and displays the documentation home page. If the Help browser is already open, but not visible, then <code>docsearch</code> brings it to the foreground.</p> <p><code>docsearch expression</code> searches MathWorks documentation for pages with words that match the specified expression. The <code>docsearch</code> command does not search third-party or custom documentation.</p>
<b>Input Arguments</b>	<p><b>expression - Expression that defines search terms</b> <i>string</i></p> <p>Expression that defines search terms, specified as a string. Expressions can include:</p> <ul style="list-style-type: none"><li>• Quotation marks to specify exact phrases, such as "plot tools".</li><li>• Boolean operator keywords in uppercase (listed here in order of precedence): NOT, OR, AND.</li><li>• 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 <i>not</i> wildcard characters.</li></ul>
<b>Examples</b>	<p><b>Single Words</b></p> <p>Find all documentation pages that contain the word <i>plot</i>.</p> <pre>docsearch plot</pre> <p><b>Multiple Words</b></p> <p>Find documentation pages with the words <i>plot</i> and <i>tools</i>.</p> <pre>docsearch plot tools</pre>

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 result

**Syntax**

```
dos command

status = dos(command)

status = dos(command &)

[status,result] = dos(command)
[status,result] = dos(command, '-echo')
```

**Description** `dos` command executes the specified DOS command for Windows platforms.

`status = dos(command)` executes the specified DOS command and *waits* for the command to finish execution before returning the completion status to the `status` variable.

`status = dos(command &)` executes the specified DOS command in the *background* and immediately returns status to the `status` variable. This syntax is useful for Windows console programs, such as `notepad` and `telnet`, that require interactive user command input while they run, and that do not run correctly in the MATLAB command window.

`[status,result] = dos(command)` executes the specified DOS command and waits for the command to finish execution before returning the command result to the `result` variable. This syntax is most useful for DOS console commands that do not require user input, such as `dir`.

`[status,result] = dos(command, '-echo')` in addition to returning the output to the `result` variable, 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.

## Output Arguments

### **status - Command status**

0 | 1

Command status, returned as a logical value, where 0 indicates success and 1 indicates failure.

- When you use syntax with the ampersand character (&), then `status` is the command launch status.
- When you use syntax without the ampersand (&) character, then `status` is the command completion status.

### **Data Types**

logical

### **result - Result of running command**

string

Result of running 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

### View DOS Command Results in MATLAB Command Window

In the Command Window, list the files in your current folder using the DOS `dir` command.

```
dos dir;
```

### Save DOS Command Completion Status

Issue a DOS command to create a new folder named `mynew` and save completion status to a variable.

```
[status] = dos('mkdir mynew');  
status
```

```
status =
```

```
0
```

The status of zero indicates that `mynew` folder was created successfully.

### Open and Run a Windows UI Command

Open Notepad and immediately return status to MATLAB by appending an ampersand (&) to the `notepad` command.

```
[status] = dos('notepad &');  
status
```

```
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 `results` contains the command output.

```
[status, results] = dos('dir');
status

status =

    0

results

results =

    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 Results

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 Results in MATLAB Command Window and Save Output**

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
```

# dos

---

```
Compare more files (Y/N) ? N  
N  
>>
```

## 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`

# double

---

**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.



---

<b>Purpose</b>	Drag rectangles with mouse
<b>Syntax</b>	<code>[finalrect] = dragrect(initialrect)</code> <code>[finalrect] = dragrect(initialrect, stepsize)</code>
<b>Description</b>	<p><code>[finalrect] = dragrect(initialrect)</code> tracks one or more rectangles anywhere on the screen. The n-by-4 matrix <code>initialrect</code> defines the rectangles. Each row of <code>initialrect</code> must contain the initial rectangle position as <code>[left bottom width height]</code> values. <code>dragrect</code> returns the final position of the rectangles in <code>finalrect</code>.</p> <p><code>[finalrect] = dragrect(initialrect, stepsize)</code> moves the rectangles in increments of <code>stepsize</code>. The lower left corner of the first rectangle is constrained to a grid of size equal to <code>stepsize</code> starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.</p> <p><code>[finalrect] = dragrect(...)</code> returns the final positions of the rectangles when the mouse button is released. The default step size is 1.</p>
<b>Tips</b>	<p><code>dragrect</code> returns immediately if a mouse button is not currently pressed. Use <code>dragrect</code> in a <code>ButtonDownFcn</code>, or from the command line in conjunction with <code>waitforbuttonpress</code>, to ensure that the mouse button is down when <code>dragrect</code> is called. <code>dragrect</code> returns when you release the mouse button.</p> <p>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.</p>

---

**Note** You cannot use normalized figure units with `dragrect`.

---

# 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](#)

**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)

# drawnow

---

- 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`

**Purpose** Search Delaunay triangulation for nearest point

---

**Note** `dsearch` has been removed. Use `DelaunayTri/nearestNeighbor` instead.

---

**Syntax**

```
K = dsearch(x,y,TRI,xi,yi)
K = dsearch(x,y,TRI,xi,yi,S)
```

**Description** `K = dsearch(x,y,TRI,xi,yi)` returns the index into `x` and `y` of the nearest point to the point `(xi,yi)`. `dsearch` requires a triangulation `TRI` of the points `x,y` obtained using `delaunay`. If `xi` and `yi` are vectors, `K` is a vector of the same size.

`K = dsearch(x,y,TRI,xi,yi,S)` uses the sparse matrix `S` instead of computing it each time:

```
S = sparse(TRI(:,[1 1 2 2 3 3]),TRI(:,[2 3 1 3 1 2]),1,nxy,nxy)
```

where `nxy = prod(size(x))`.

**See Also** `DelaunayTri` | `delaunay` | `voronoi`

# 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** `DelaunayTri` | `dsearch`

**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 <i>fcnname</i> on	Turns on echoing of the named function file
echo <i>fcnname</i> off	Turns off echoing of the named function file



<code>echo <i>fcnname</i></code>	Toggles the echo state of the named function file
<code>echo on all</code>	Sets echoing on for all function files
<code>echo off all</code>	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`

# echodemo

---

**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

---

<b>Purpose</b>	Simplices attached to specified edges	
<b>Syntax</b>	SI = edgeAttachments(TR, V1, V2) SI = edgeAttachments(TR, EDGE)	
<b>Description</b>	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.	
<b>Input Arguments</b>	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.
<b>Output Arguments</b>	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.
<b>Definitions</b>	A simplex is a triangle/tetrahedron or higher dimensional equivalent.	
<b>Examples</b>	<b>Example 1</b>	
	Load a 3-D triangulation to compute the tetrahedra attached to an edge.	
	<pre>load tetmesh trep = TriRep(tet, X); v1 = [15 21]'; v2 = [936 716]'; t1 = edgeAttachments(trep, v1, v2);</pre>	
	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

`TriRep.edges`

# TriRep.edges

---

**Purpose**                    Triangulation edges

**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**                    TriRep.edgeAttachments

**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`.

If `W` is a matrix such that  $W'*A = D*W'$ , the columns of `W` are the *left eigenvectors* of `A`. Use `[W,D] = eig(A, '');`; `W = conj(W)` to compute the left eigenvectors.

`[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 <code>A</code> and <code>B</code> using the Cholesky factorization of <code>B</code> . This is the default for symmetric (Hermitian) <code>A</code> and symmetric (Hermitian) positive definite <code>B</code> .
'qz'	Ignores the symmetry, if any, and uses the QZ algorithm as it would for nonsymmetric (non-Hermitian) <code>A</code> and <code>B</code> .

---

**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  $\lambda$  is a scalar. The  $n$  values of  $\lambda$  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  $\lambda$ , 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  $\lambda$  is a scalar. The values of  $\lambda$  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`

# eigs

---

**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 <i>sigma</i> . If A is a function, Afun must return $Y = (A - \textit{sigma} * B) \backslash x$ (i.e., $Y = A \backslash x$ when <i>sigma</i> = 0).
'lm'	Largest magnitude (default).
'sm'	Smallest magnitude. Same as <i>sigma</i> = 0. If A is a function, Afun must return $Y = A \backslash x$ .

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 $\leq$ tol*norm(A).	[scalar   {eps}]
opts.maxit	Maximum number of iterations.	[integer   {300}]
opts.p	Number of Lanczos basis vectors. p $\geq$ 2k (p $\geq$ 2k+1 real nonsymmetric) advised. p must satisfy k < p $\leq$ n for real symmetric, k+1 < p $\leq$ 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:

<code>A*x</code>	if <i>sigma</i> is not specified, or is a string other than 'sm'
<code>A\x</code>	if <i>sigma</i> is 0 or 'sm'
<code>(A-sigma*I)\x</code>	if <i>sigma</i> is a nonzero scalar (standard eigenvalue problem). <code>I</code> is an identity matrix of the same size as <code>A</code> .
<code>(A-sigma*B)\x</code>	if <i>sigma</i> 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
```

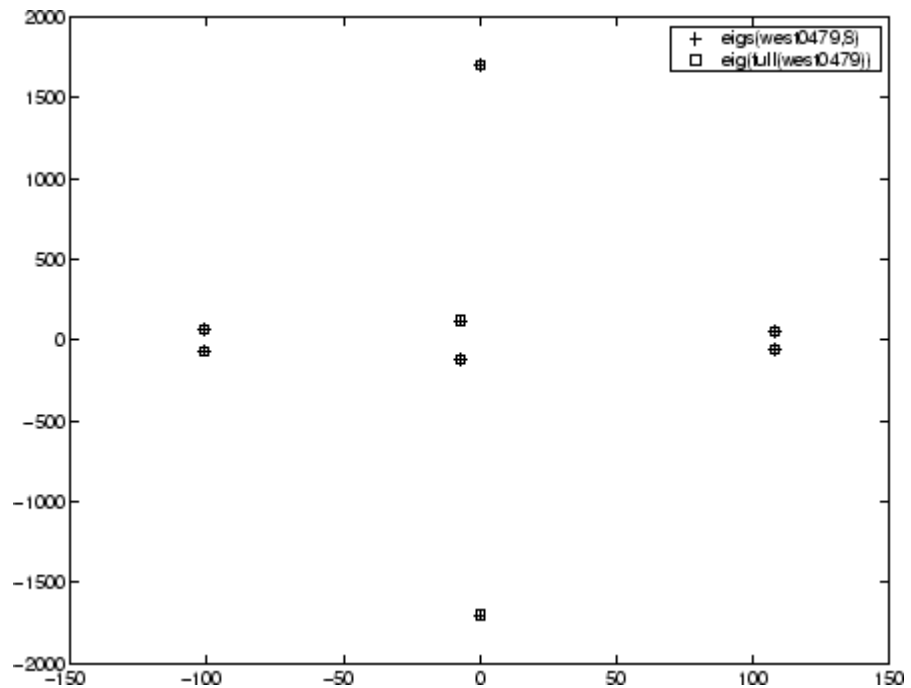


### Example 3

`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))
d1m = eigs(west0479,8)
[dum,ind] = sort(abs(d));
plot(d1m,'k+')
hold on
plot(d(ind(end-7:end)),'ks')
hold off
legend('eigs(west0479,8)', 'eig(full(west0479))')
```



## Example 4

$A = \text{delsq}(\text{numgrid}('C', 30))$  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));  
d1m = eigs(A);  
dsm = eigs(A,6,'sm');
```

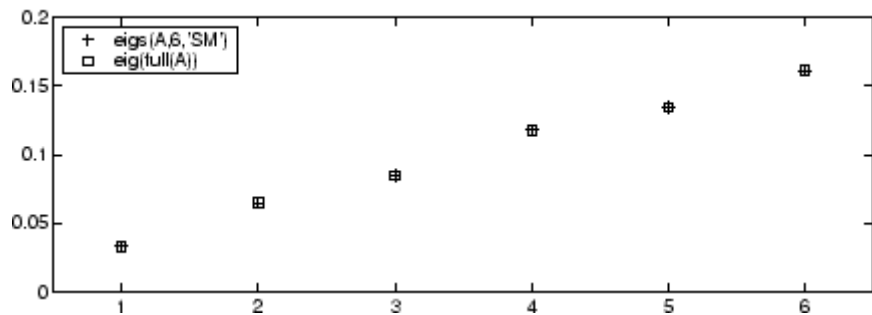
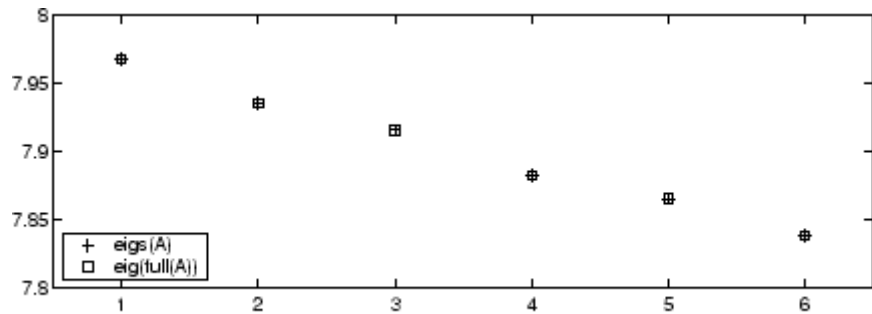
```
subplot(2,1,1)  
plot(d1m,'k+')
```

```

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])

```

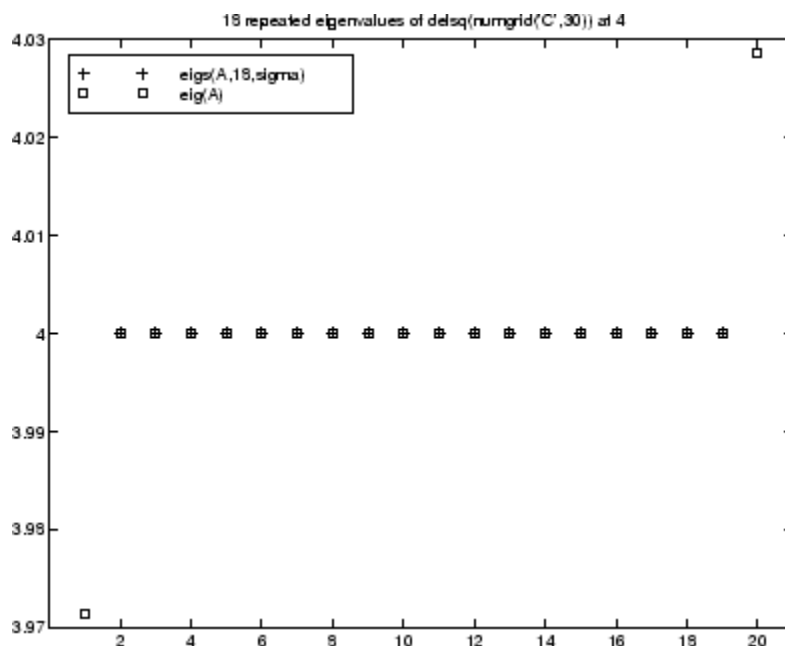


# eigs

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  $A - 4.0 \cdot I$ . 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

# ellipj

---

**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

$$\operatorname{sn}(u) = \sin \phi, \operatorname{cn}(u) = \cos \phi, \operatorname{dn}(u) = \sqrt{1 - m \sin^2 \phi}.$$

Some definitions of the elliptic functions use the modulus  $k$  instead of the parameter  $m$ . They are related by

$$k^2 = m = \sin^2 \alpha,$$

where  $\alpha$  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  $0 \leq m \leq 1$ . 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

# 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  $0 \leq m \leq 1$ .

**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.



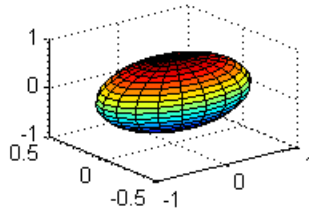
**See Also**      ellipj

# ellipsoid

---

## 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:

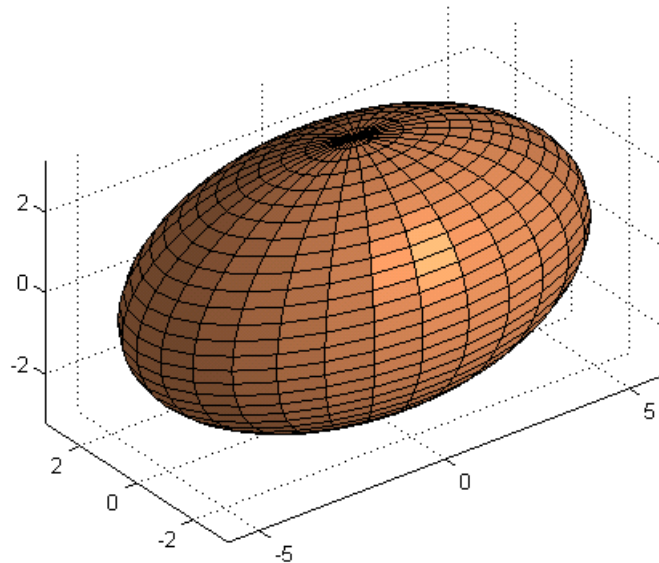
$$\frac{(x-xc)^2}{xr^2} + \frac{(y-yc)^2}{yr^2} + \frac{(z-zc)^2}{zr^2}$$

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);  
surf1(x, y, z)  
colormap copper  
axis equal
```



## See Also

[cylinder](#) | [sphere](#) | [surf](#)

# else

---

**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

**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
```

# elseif

---

```
        myData(r,c) = -1;
    else
        myData(r,c) = 0;
    end

    end
end
```

## See Also

if | end | for | while | switch

<b>Purpose</b>	Create empty array
<b>Syntax</b>	<pre>A = <i>ClassName</i>.empty A = <i>ClassName</i>.empty(n,m,p,...) A = <i>ClassName</i>.empty([n,m,p,...])</pre>
<b>Description</b>	<p>Use <code>empty</code> to create empty arrays of the specified class, <i>ClassName</i>. Specify at least one dimension of the array as 0. MATLAB treats negative values as 0.</p> <p><code>A = <i>ClassName</i>.empty</code> returns an empty 0-by-0 array of the class of <i>ClassName</i>.</p> <p><code>A = <i>ClassName</i>.empty(n,m,p,...)</code> returns an empty rectangular array with the specified dimensions. At least one of the dimensions must be 0.</p> <p><code>A = <i>ClassName</i>.empty([n,m,p,...])</code> 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 <code>size</code> function to define an empty array that is the same size as an existing empty array:</p> <pre>A = <i>ClassName</i>.empty(size(otherEmptyArray));</pre>
<b>Input Arguments</b>	<p><b>n,m,p,...</b></p> <p>Dimensions of the empty array. At least one of the specified dimensions must be 0.</p>
<b>Output Arguments</b>	<p><b>A</b></p> <p>An empty array of the specified dimensions and of the class used in the method invocation.</p>
<b>Attributes</b>	<p><code>empty</code> is a hidden, public, static method of all nonabstract MATLAB classes.</p>

# empty

---

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
```



```

        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 empty ExEmpty

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)

```

# empty

---

```
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:

```
A = 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”

# enableNETfromNetworkDrive

---

<b>Purpose</b>	Enable access to .NET commands from network drive
<b>Syntax</b>	<code>enableNETfromNetworkDrive</code>
<b>Description</b>	<code>enableNETfromNetworkDrive</code> adds an entry for the MATLAB interface to .NET module to the security policy on your machine.
<b>How To</b>	<ul style="list-style-type: none"><li>• “Troubleshooting Security Policy Settings From Network Drives”</li></ul>

**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`

# enableservice

---

## 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 kth 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
```

# 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`



<b>Purpose</b>	Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method
<b>Syntax</b>	<pre>result = EndInvoke(asyncResult) [res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)</pre>
<b>Description</b>	<p>result = EndInvoke(asyncResult) retrieves result of asynchronous call initiated by BeginInvoke method.</p> <p>[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult) for methods with out and/or ref parameters.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• 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”.</li></ul>
<b>Input Arguments</b>	<p><b>asyncResult</b> .NET System.IAsyncResult object returned by BeginInvoke.</p> <p><b>res0,...,resN</b> For methods with out and/or ref parameters, results of the asynchronous call. The number of arguments is the sum of:</p> <ul style="list-style-type: none"><li>• Number of return values (0 or 1).</li><li>• Number of out and ref arguments.</li></ul>
<b>Output Arguments</b>	<p><b>result</b> Results of the asynchronous call.</p> <p><b>res0,...,resN</b> For methods with out and/or ref parameters, results of the asynchronous call,</p>

## 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.IAsyncResultRetVal  
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
```

# EndInvoke

---

**3** MATLAB creates the following signatures. For `BeginInvoke`:

```
System.IAsyncResultRetVal
BeginInvoke (
    SignatureExamples.del2int this,
    int32 scalar arg1,
    int32 scalar arg2,
    System.AsyncCallback callback,
    System.Object object)
```

The `EndInvoke` signature:

```
int32 scalarRetVal
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);
```

# EndInvoke

---

```
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,  
    single scalar argIn,  
    System.AsyncCallback callback,  
    System.Object object)
```

```
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](#)

# eomday

---

**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`

# enumeration

---

**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 =
```

# enumeration

---

```
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”

**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  $\text{eps} = 2^{-52}$ .

$d = \text{eps}(X)$  is the positive distance from  $\text{abs}(X)$  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  $X$ ,

$$\text{eps}(X) = \text{eps}(-X) = \text{eps}(\text{abs}(X))$$

$\text{eps}('double')$  is the same as  $\text{eps}$  or  $\text{eps}(1.0)$ .

$\text{eps}('single')$  is the same as  $\text{eps}(\text{single}(1.0))$  or  $\text{single}(2^{-23})$ .

Except for numbers whose absolute value is smaller than  $\text{realmin}$ , if  $2^E \leq \text{abs}(X) < 2^{(E+1)}$ , then

$$\begin{aligned} \text{eps}(X) &= 2^{(E-23)} \text{ if } \text{isa}(X, 'single') \\ \text{eps}(X) &= 2^{(E-52)} \text{ if } \text{isa}(X, 'double') \end{aligned}$$

For all  $X$  of class double such that  $\text{abs}(X) \leq \text{realmin}$ ,  $\text{eps}(X) = 2^{-1074}$ . Similarly, for all  $X$  of class single such that  $\text{abs}(X) \leq \text{realmin}('single')$ ,  $\text{eps}(X) = 2^{-149}$ .

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 =
```

# eq

---

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



<b>Purpose</b>	Compare scalar MException objects for equality
<b>Syntax</b>	<code>eObj1 == eObj2</code>
<b>Description</b>	<code>eObj1 == eObj2</code> tests scalar MException objects <code>eObj1</code> and <code>eObj2</code> for equality, returning logical 1 (true) if the two objects are identical, otherwise returning logical 0 (false).
<b>See Also</b>	<code>try</code>   <code>catch</code>   <code>error</code>   <code>assert</code>   <code>MException</code>   <code>isequal(MException)</code>   <code>ne(MException)</code>   <code>getReport(MException)</code>   <code>disp(MException)</code>   <code>throw(MException)</code>   <code>rethrow(MException)</code>   <code>throwAsCaller(MException)</code>   <code>addCause(MException)</code>   <code>last(MException)</code>

# erf

---

**Purpose** Error function

**Syntax**  $Y = \text{erf}(X)$

**Definitions** The error function  $\text{erf}(X)$  is twice the integral of the Gaussian distribution with 0 mean and variance of 1/2.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

**Description**  $Y = \text{erf}(X)$  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**  $Y = \text{erfc}(X)$

**Definitions** The complementary error function  $\text{erfc}(X)$  is defined as

$$\begin{aligned}\text{erfc}(x) &= \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt \\ &= 1 - \text{erf}(x)\end{aligned}$$

**Description**  $Y = \text{erfc}(X)$  computes the value of the complementary error function.

**Tips** The relationship between the complementary error function  $\text{erfc}$  and the standard normal probability distribution returned by the Statistics Toolbox function  $\text{normcdf}$  is

$$\text{normcdf}(x) = \left(\frac{1}{2}\right) \times \text{erfc}\left(\frac{-x}{\sqrt{2}}\right)$$

**See Also** [erf](#) | [erfcinv](#) | [erfcx](#) | [erfinv](#)

# erfcinv

---

**Purpose** Inverse complementary error function

**Syntax**  $X = \text{erfcinv}(Y)$

**Description**  $X = \text{erfcinv}(Y)$  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  $\text{erfcinv}$  satisfies  $y = \text{erfc}(x)$  for  $2 \geq y \geq 0$  and  $-\infty \geq x \geq \infty$ .

**Tips** The relationship between the inverse complementary error function  $\text{erfcinv}$  and the inverse standard normal probability distribution returned by the Statistics Toolbox function  $\text{norminv}$  is:

$$\text{norminv}(p) = (-\sqrt{2}) \times \text{erfcinv}(2p).$$

**See Also** `erf` | `erfc` | `erfcx` | `erfinv`

**Purpose** Scaled complementary error function

**Syntax**  $Y = \text{erfcx}(X)$

**Definitions** The scaled complementary error function  $\text{erfcx}(X)$  is defined as

$$\text{erfcx}(x) = e^{x^2} \text{erfc}(x)$$

For large  $X$ ,  $\text{erfcx}(X)$  is approximately  $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$

**Description**  $Y = \text{erfcx}(X)$  computes the value of the scaled complementary error function.

**See Also** `erf` | `erfcinv` | `erfc` | `erfinv`

# 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  $y = \text{erf}(x)$  for  $-1 \leq y \leq 1$  and  $-\infty \leq x \leq \infty$ .

**Examples** `erfinv(1)` is Inf  
`erfinv(-1)` is -Inf.

**See Also** `erf` | `erfcinv` | `erfcx` | `erfc`

**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\errrtest1.m'
    name: 'errrtest1'
    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**

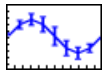
`assert` | `try` | `dbstop` | `errordlg` | `warning` | `warndlg` |  
`MException` | `throw(MException)` | `rethrow(MException)`  
| `throwAsCaller(MException)` | `addCause(MException)` |  
`getReport(MException)` | `last(MException)`

# errorbar

---

## Purpose

Plot error bars along curve



## Syntax

```
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineStyle)
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  $2 \cdot E(i)$  long.

`errorbar(X,Y,E)` plots  $Y$  versus  $X$  with symmetric error bars  $2 \cdot E(i)$  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(...,LineStyle)` uses the color and line style specified by the string 'LineStyle'. 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

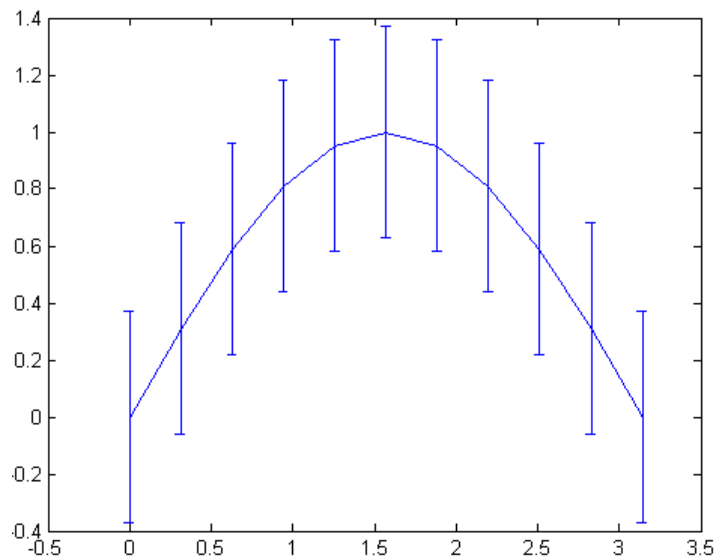
one object per column for matrix input arguments. See `errorbarseries` properties for more information.

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)
```

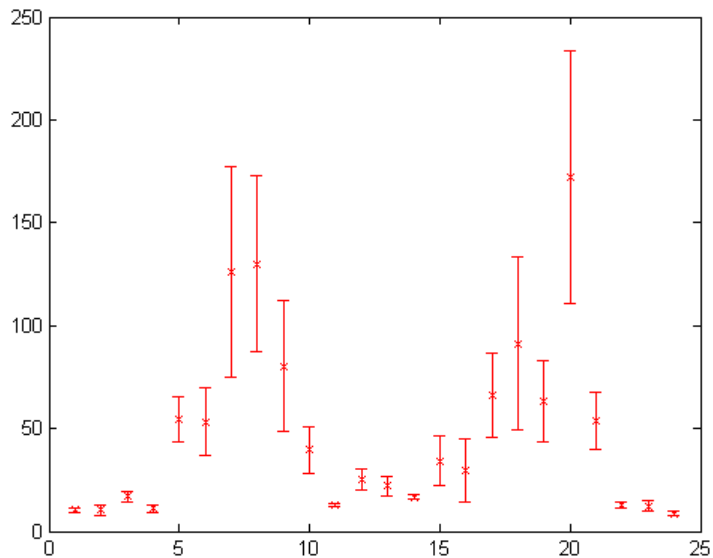


Plot the computed average traffic volume and computed standard deviations for three street locations over the course of a day using red 'x' markers:

```
load count.dat;  
y = mean(count,2);
```

# errorbar

```
e = std(count,1,2);  
figure  
errorbar(y,e,'xr')
```



## See Also

[corrcoef](#) | [linespec](#) | [plot](#) | [std](#) | [Errorbarseries](#) Properties

## How To

- [ConfidenceBounds](#)

## 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.

# Errorbarseries Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
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:

# Errorbarseries Properties

---

- 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 `gcb0`.

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.

## Errorbarseries Properties

---

- 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.

# Errorbarseries Properties

---

- **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}

# Errorbarseries Properties

---

## *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

# Errorbarseries Properties

---

*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

## Errorbarseries Properties

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

# Errorbarseries Properties

---

*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` (the factory default for axes objects), 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, `hgroup`, or `hgtransform`

*Parent of object.* Handle of the object's parent. The parent is normally the axes, `hgroup`, 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)

# Errorbarseries Properties

---

*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`.

# Errorbarseries Properties

---

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.

---

# errordlg

---

**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 titled `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.

<b>createmode Value</b>	<b>Description</b>
modal	Replaces the error dialog box having the specified <code>Title</code> , 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 `WindowState` in the `Figure Properties`.

---

If `CreateMode` is a structure, it can have fields `WindowState` and `Interpreter`. `WindowState` 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.

# errordlg

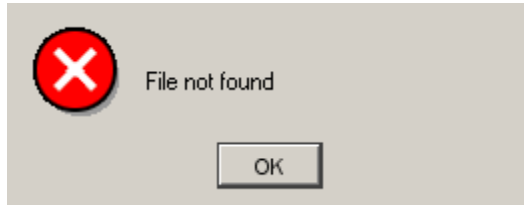
---

## 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](#)

<b>Purpose</b>	Time elapsed between date vectors
<b>Syntax</b>	<code>e = etime(t2,t1)</code>
<b>Description</b>	<code>e = etime(t2,t1)</code> returns the number of seconds between two date vectors or matrices of date vectors, <code>t1</code> and <code>t2</code> .
<b>Input Arguments</b>	<p><b>t2,t1 - Date vectors</b> 1-by-6 vector   m-by-6 matrix</p> <p>Date vectors, specified as 1-by-6 vectors or m-by-6 matrices containing m full date vectors in the format:</p> <p>[Year Month Day Hour Minute Second]</p> <p><b>Example:</b> [2012 03 27 11 50 01]</p> <p><b>Data Types</b> double</p>
<b>Examples</b>	<p><b>Compute Elapsed Time</b></p> <p>Compute the time elapsed between a specific time and the current time, to 0.01-second accuracy.</p> <p>Define the initial date and time and convert to date vector form.</p> <pre>format shortg str = 'March 28, 2012 11:51:00'; t1 = datevec(str,'mmm dd, yyyy HH:MM:SS')</pre> <pre>t1 =       2012          3          28          11          51</pre> <p>Determine the current date and time.</p> <pre>t2 = clock</pre> <pre>t2 =</pre>

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  $A^T * A$ .

`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`

# etreeplot

---

<b>Purpose</b>	Plot elimination tree
<b>Syntax</b>	<code>etreeplot(A)</code> <code>etreeplot(A,nodeSpec,edgeSpec)</code>
<b>Description</b>	<code>etreeplot(A)</code> plots the elimination tree of $A$ (or $A+A'$ , if non-symmetric). <code>etreeplot(A,nodeSpec,edgeSpec)</code> allows optional parameters <code>nodeSpec</code> and <code>edgeSpec</code> to set the node or edge color, marker, and linestyle. Use <code>' '</code> to omit one or both.
<b>See Also</b>	<code>etree</code>   <code>treepplot</code>   <code>treelayout</code>



---

<b>Purpose</b>	Execute MATLAB expression in text string
<b>Syntax</b>	<code>eval(expression)</code> <code>[output1,...,outputN] = eval(expression)</code>
<b>Description</b>	<p><code>eval(expression)</code> evaluates the MATLAB code in the string <code>expression</code>. If you use <code>eval</code> within an anonymous function, nested function, or function that contains a nested function, the evaluated expression cannot create a variable.</p> <p><code>[output1,...,outputN] = eval(expression)</code> stores output from <code>expression</code> in the specified variables.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Many common uses of the <code>eval</code> 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 <code>eval</code> Function”.</li><li>• Whenever possible, do not include output arguments within the input to the <code>eval</code> function, such as <code>eval(['output = ',expression])</code>. The preferred syntax,  <code>output = eval(expression)</code>  allows the MATLAB parser to perform stricter checks on your code, preventing untrapped errors and other unexpected behavior.</li></ul>
<b>Input Arguments</b>	<p><b>expression</b></p> <p>String that contains a valid MATLAB expression.</p> <p>To include a numeric value in the expression, convert it to a string with <code>int2str</code>, <code>num2str</code>, or <code>sprintf</code>.</p>
<b>Output Arguments</b>	<p><b>output1,...,outputN</b></p> <p>Outputs from the evaluated expression.</p>

## 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”

---

<b>Purpose</b>	Evaluate MATLAB expression with capture
<b>Syntax</b>	<code>T = evalc(S)</code> <code>[T, X, Y, Z, ...] = evalc(S)</code>
<b>Description</b>	<p><code>T = evalc(S)</code> is the same as <code>eval(S)</code> except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array <code>T</code> (lines in <code>T</code> are separated by <code>\n</code> characters).</p> <p><code>[T, X, Y, Z, ...] = evalc(S)</code> is the same as <code>[X, Y, Z, ...] = eval(S)</code> except that any output is captured into <code>T</code>.</p>
<b>Tips</b>	When you are using <code>evalc</code> , <code>diary</code> , <code>more</code> , and <code>input</code> are disabled.
<b>See Also</b>	<code>eval</code>   <code>evalin</code>   <code>assignin</code>   <code>feval</code>   <code>diary</code>   <code>input</code>   <code>more</code>

# evalin

---

**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`

# event.EventData

---

## **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”

## 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.

# event.listener

---

## 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	<p>When false (the default), this listener does not execute recursively. Therefore, if the callback triggers its own event, the listener does not execute again.</p> <p>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.</p>

## See Also

`addlistener` | `delete` | `event.proplistener`



<b>Purpose</b>	Listener for property events
<b>Description</b>	The <code>event.PropertyEvent</code> class defines the data objects passed to listeners of the <code>meta.property</code> events <code>PreGet</code> , <code>PostGet</code> , <code>PreSet</code> , and <code>PostSet</code> . <code>event.PropertyEvent</code> is a sealed subclass of <code>event.EventData</code> (i.e., you cannot subclass <code>event.PropertyEvent</code> ).
<b>Properties</b>	<code>event.PropertyEvent</code> inherits the <code>EventName</code> and <code>Source</code> properties from <code>event.EventData</code> and defines one new property: <ul style="list-style-type: none"><li>• <code>AffectedObject</code> — The instance of the class to which this event refers.</li></ul>
<b>See Also</b>	<code>event.EventData</code>   <code>meta.property</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Listen for Changes to Property Values”</li></ul>

# event.proplistener

---

**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`

**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, `Db1Click` and `MouseDown`:

```
h.registerevent({'Db1Click', 'my2click'; 'MouseDown' 'mymoused'});
h.eventlisteners
```

# 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
```

---

<b>Purpose</b>	Event names
<b>Syntax</b>	<pre>events('classname') events(obj) e = events(...)</pre>
<b>Description</b>	<p><code>events('classname')</code> displays the names of the public events for the MATLAB class <code>classname</code>, including events inherited from superclasses.</p> <p><code>events(obj)</code> <code>obj</code> is a scalar or array of objects of a MATLAB class.</p> <p><code>e = events(...)</code> returns the event names in a cell array of strings.</p> <p>An event is public when the value of its <code>ListenAccess</code> attribute is <code>public</code> and its <code>Hidden</code> attribute value is <code>false</code> (default values for both attributes). See “Event Attributes” for a complete list of attributes.</p> <p><code>events</code> is also a MATLAB class-definition keyword. See <code>classdef</code> for more information on class definition keywords.</p>
<b>Examples</b>	<p>Get the names of the public events for the <code>handle</code> class:</p> <pre>events('handle')</pre> <p>Events for class <code>handle</code>:</p> <pre>    ObjectBeingDestroyed</pre>
<b>See Also</b>	<code>properties</code>   <code>methods</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>“Events”</li></ul>

# events (COM)

---

**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](#)  
| [unregisterallevnts](#)

# Execute

---

**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`

**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: <ul style="list-style-type: none"> <li>• name exists on your MATLAB search path as a file with extension <code>.m</code>.</li> <li>• name is the name of an ordinary file on your MATLAB search path.</li> <li>• name is the full pathname to any file.</li> </ul>
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. ( <code>exist</code> returns 0 for Java classes if you start MATLAB with the <code>-nojvm</code> 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:

<code>builtin</code>	Checks only for built-in functions.
<code>class</code>	Checks only for classes.
<code>dir</code>	Checks only for folders.
<code>file</code>	Checks only for files or folders.
<code>var</code>	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.

<b>Order of Evaluation</b>	<b>Return Value</b>	<b>Type of Entity</b>
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`

<b>Purpose</b>	Terminate MATLAB program (same as quit)
<b>Alternatives</b>	As an alternative to the <code>exit</code> function, click the Close box in the MATLAB desktop.
<b>Syntax</b>	<code>exit</code>
<b>Description</b>	<code>exit</code> terminates the current session of MATLAB after running <code>finish.m</code> , if the file <code>finish.m</code> exists. It performs the same as <code>quit</code> and takes the same termination options, such as <b>force</b> . For more information, see <code>quit</code> .
<b>See Also</b>	<code>quit</code>   <code>finish</code>

# exp

---

**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`



**Purpose** Exponential integral

**Syntax**  $Y = \text{expint}(X)$

**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

$$\text{Ei}(x) = \int_{-\infty}^x e^t / t dt$$

which, for real positive  $x$ , is related to `expint` as

$$E_1(-x) = -\text{Ei}(x) - i\pi$$

**Description**  $Y = \text{expint}(X)$  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.

# expm

---

**Purpose** Matrix exponential

**Syntax**  $Y = \text{expm}(X)$

**Description**  $Y = \text{expm}(X)$  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

$$[V,D] = \text{EIG}(X) \text{ and } \text{EXPM}(X) = V \cdot \text{diag}(\exp(\text{diag}(D))) / V$$

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, `expdemo1.m`, `expdemo2.m`, and `expdemo3.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](#)

# expm1

---

**Purpose** Compute  $\exp(x)-1$  accurately for small values of  $x$

**Syntax**  $y = \text{expm1}(x)$

**Description**  $y = \text{expm1}(x)$  computes  $\exp(x) - 1$ , compensating for the roundoff in  $\exp(x)$ .

For small  $x$ ,  $\text{expm1}(x)$  is approximately  $x$ , whereas  $\exp(x) - 1$  can be zero.

**See Also** [exp](#) | [expm](#) | [log1p](#)

## 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

---

`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');
```

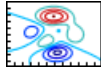
---

<b>Purpose</b>	Identity matrix
<b>Syntax</b>	<pre>Y = eye(n) Y = eye(m,n) Y = eye([m n]) Y = eye(size(A)) Y = eye Y = eye(m, n, classname)</pre>
<b>Description</b>	<p><code>Y = eye(n)</code> returns the <math>n</math>-by-<math>n</math> identity matrix.</p> <p><code>Y = eye(m,n)</code> or <code>Y = eye([m n])</code> returns an <math>m</math>-by-<math>n</math> matrix with 1's on the diagonal and 0's elsewhere. The size inputs <math>m</math> and <math>n</math> should be nonnegative integers. Negative integers are treated as 0.</p> <p><code>Y = eye(size(A))</code> returns an identity matrix the same size as <code>A</code>.</p> <p><code>Y = eye</code> with no arguments is the scalar 1.</p> <p><code>Y = eye(m, n, classname)</code> is an <math>m</math>-by-<math>n</math> matrix with 1's of class <code>classname</code> on the diagonal and zeros of class <code>classname</code> elsewhere. <code>classname</code> is a string specifying the data type of the output. <code>classname</code> can take the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.</p> <p>The identity matrix is not defined for higher-dimensional arrays. The assignment <code>y = eye([2,3,4])</code> results in an error.</p>
<b>Examples</b>	<p>Return a 2-by-3 matrix of class <code>int8</code>:</p> <pre>x = eye(2,3,'int8');</pre>
<b>See Also</b>	<code>speye</code>   <code>ones</code>   <code>zeros</code>   <code>magic</code>

# ezcontour

---

**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

---

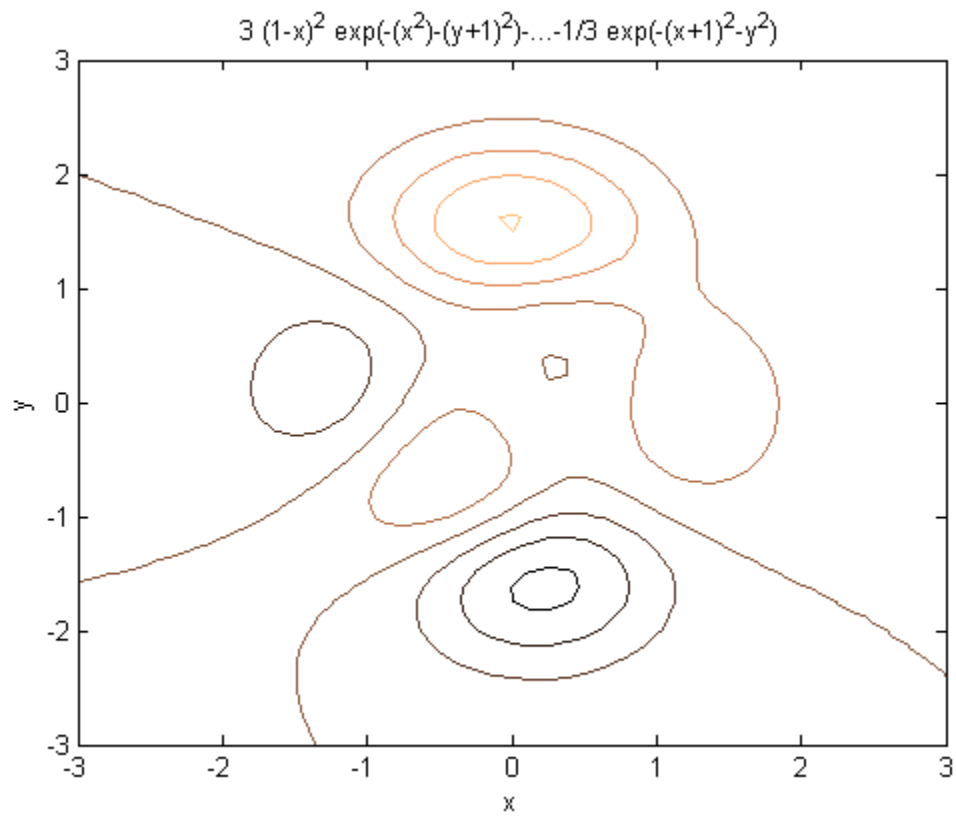
`ezcontour` requires a function handle argument that expresses this function using MATLAB syntax. This example uses an anonymous function, which you can define in the command window without creating a separate file.

```
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.

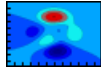
### See Also

[contour](#) | [ezcontourf](#) | [ezmesh](#) | [ezmeshc](#) | [ezplot](#) | [ezplot3](#) | [ezpolar](#) | [ezsurf](#) | [ezsurfz](#) | [function\\_handle](#)

# ezcontourf

---

**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:  $-2\pi < x < 2\pi$ ,  $-2\pi < y < 2\pi$ .

`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

---

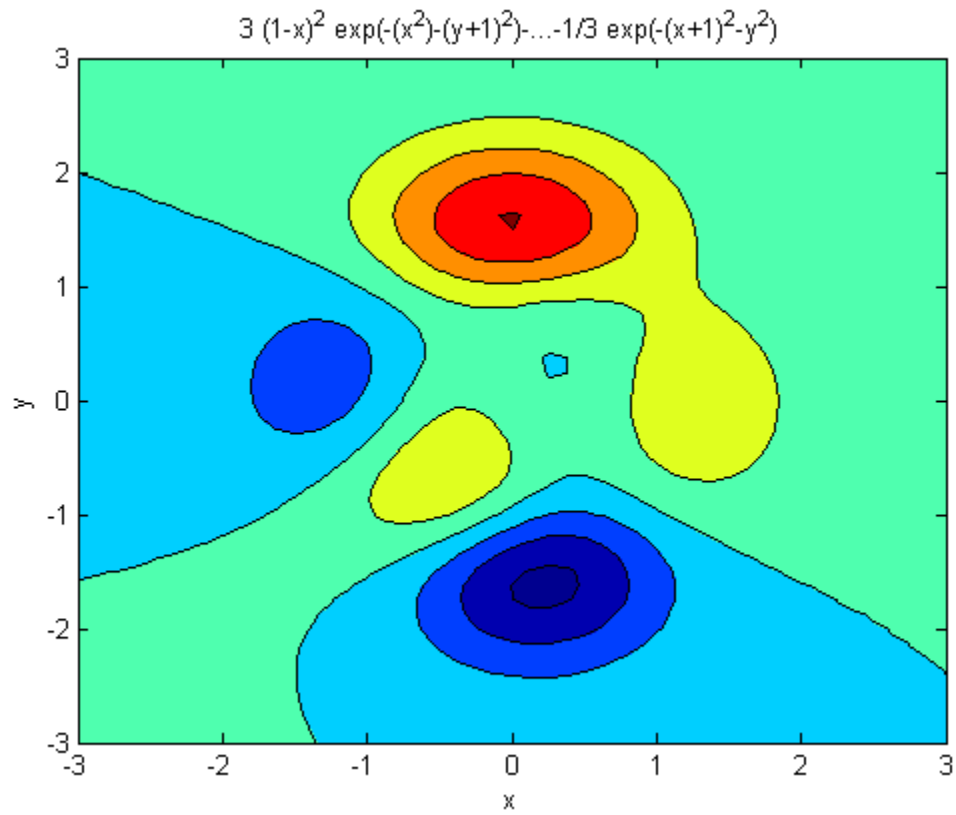
`ezcontourf` requires a string argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the string

```
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](#) | [ezsurfz](#) | [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  $\text{fun}(x,y)$  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  $\text{min} < x < \text{max}$ ,  $\text{min} < y < \text{max}$ ).

`ezmesh(funx,funy,funz)` plots the parametric surface  $\text{funx}(s,t)$ ,  $\text{funy}(s,t)$ , and  $\text{funz}(s,t)$  over the square:  $-2\pi < s < 2\pi$ ,  $-2\pi < t < 2\pi$ .

`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))
```

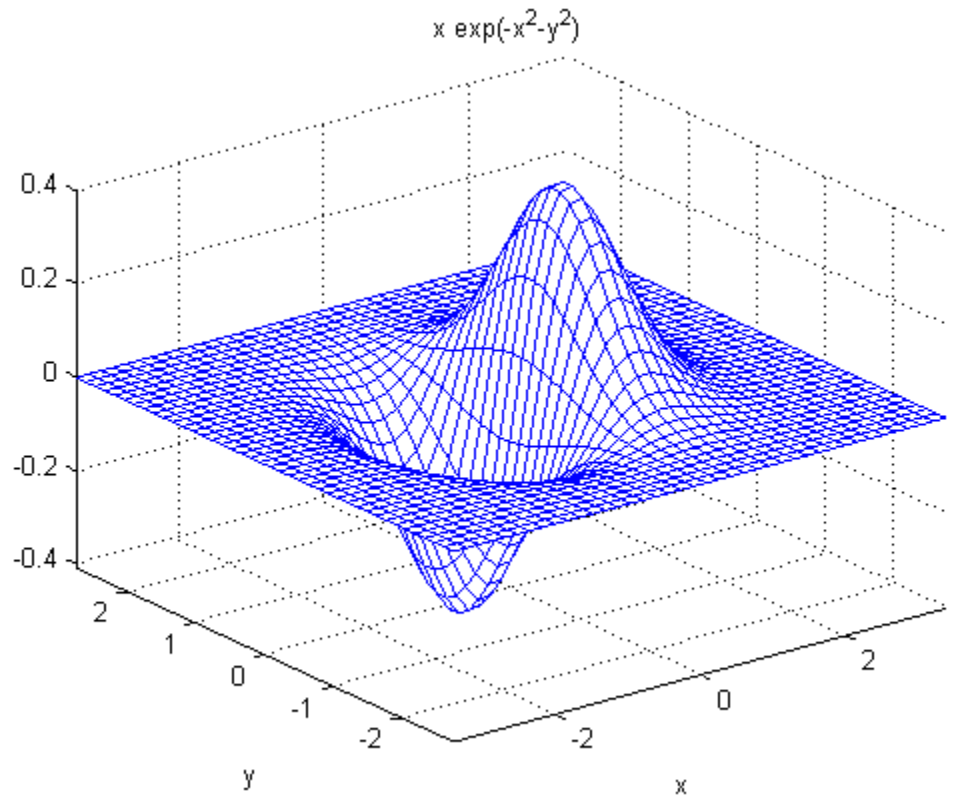
## Examples

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

# ezmeshc

---

**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  $\text{fun}(x, y)$  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  $\text{min} < x < \text{max}$ ,  $\text{min} < y < \text{max}$ ).

`ezmeshc(funx, funy, funz)` plots the parametric surface  $\text{funx}(s, t)$ ,  $\text{funy}(s, t)$ , and  $\text{funz}(s, t)$  over the square:  $-2\pi < s < 2\pi$ ,  $-2\pi < t < 2\pi$ .

`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

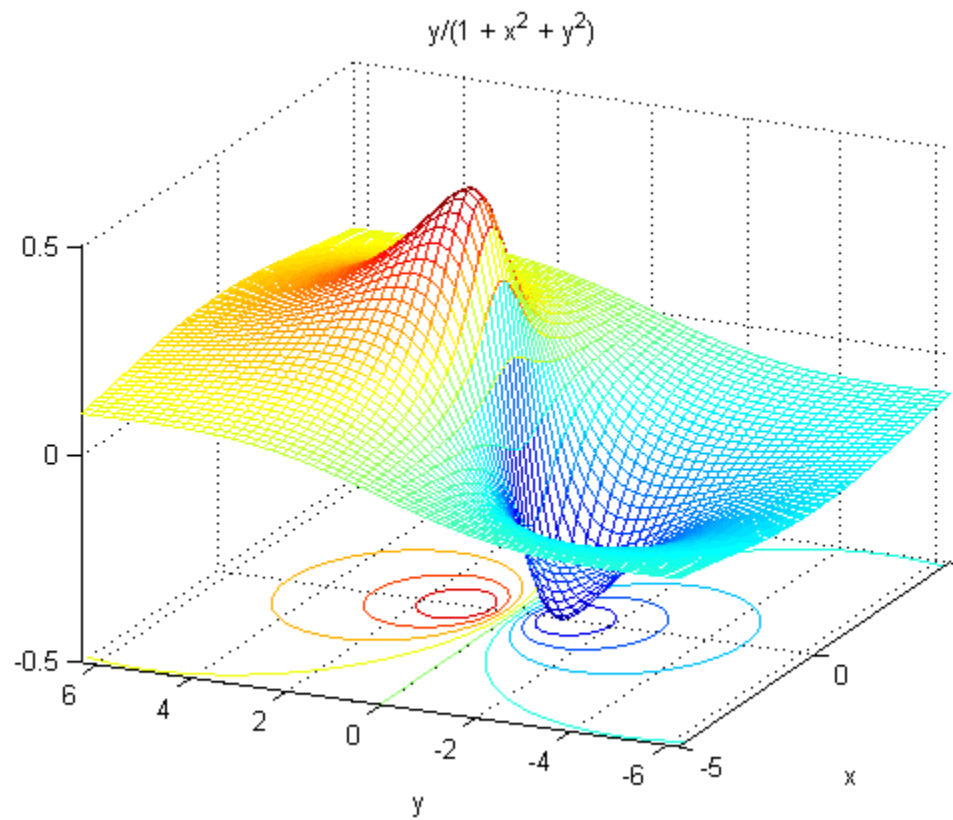
Create a mesh/contour graph of the expression

$$f(x,y) = \frac{y}{1+x^2+y^2}$$

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](#) | [ezsurf](#) | [function\\_handle](#) | [meshc](#)

**How To**

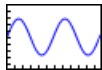
- [Anonymous Functions](#)

# ezplot

---

## 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  $\text{fun}(x)$  over the default domain  $-2\pi < x < 2\pi$ , where  $\text{fun}(x)$  is an explicit function of only  $x$ .

`fun` can be a function handle or a string.

`ezplot(fun,[xmin,xmax])` plots  $\text{fun}(x)$  over the domain:  $x_{\min} < x < x_{\max}$ .

For an implicit function, `fun2(x,y)`:

`ezplot(fun2)` plots  $\text{fun2}(x,y) = 0$  over the default domain  $-2\pi < x < 2\pi$ ,  $-2\pi < y < 2\pi$ .

`ezplot(fun2,[xymin,xymax])` plots  $\text{fun2}(x,y) = 0$  over  $x_{\min} < x < x_{\max}$  and  $y_{\min} < y < y_{\max}$ .

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots  $\text{fun2}(x,y) = 0$  over  $x_{\min} < x < x_{\max}$  and  $y_{\min} < y < y_{\max}$ .

`ezplot(funx,funy)` plots the parametrically defined planar curve  $\text{funx}(t)$  and  $\text{funy}(t)$  over the default domain  $0 < t < 2\pi$ .

`ezplot(funx,funy,[tmin,tmax])` plots  $\text{funx}(t)$  and  $\text{funy}(t)$  over  $t_{\min} < t < t_{\max}$ .



`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

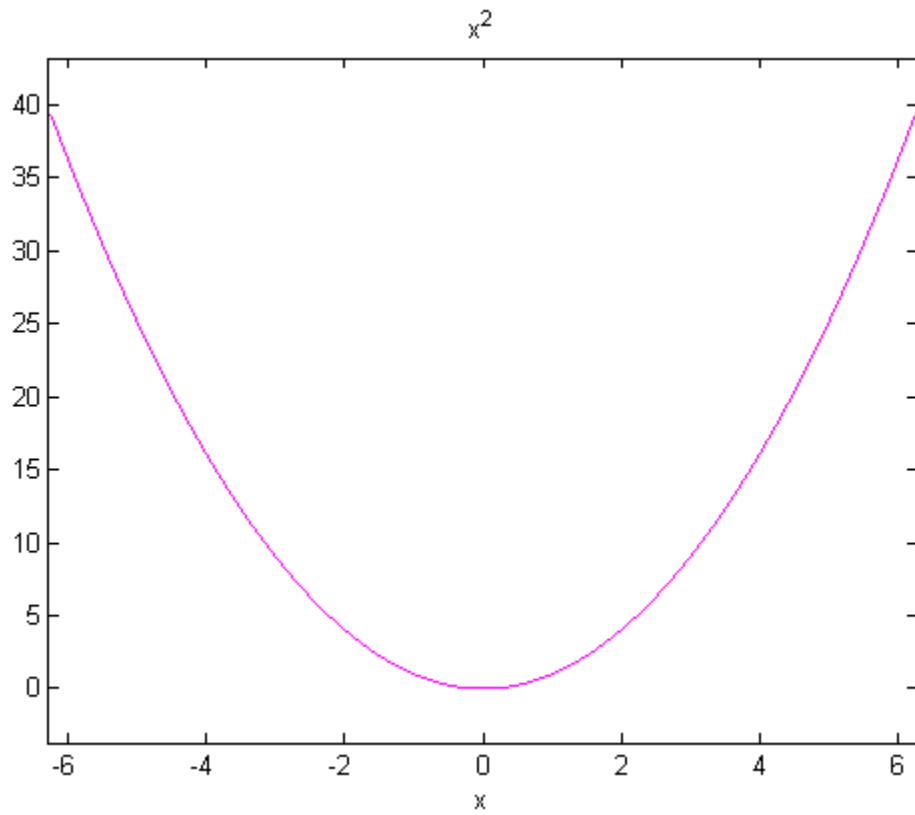
### Plotting an Explicit Function

This example plots an explicit function:

$$x^2$$

over the domain  $[-2\pi, 2\pi]$ .

```
h = ezplot('x^2')  
set(h, 'Color', 'm'); % Make the line magenta
```



### Plotting an Implicit Function

This example plots the implicitly defined function

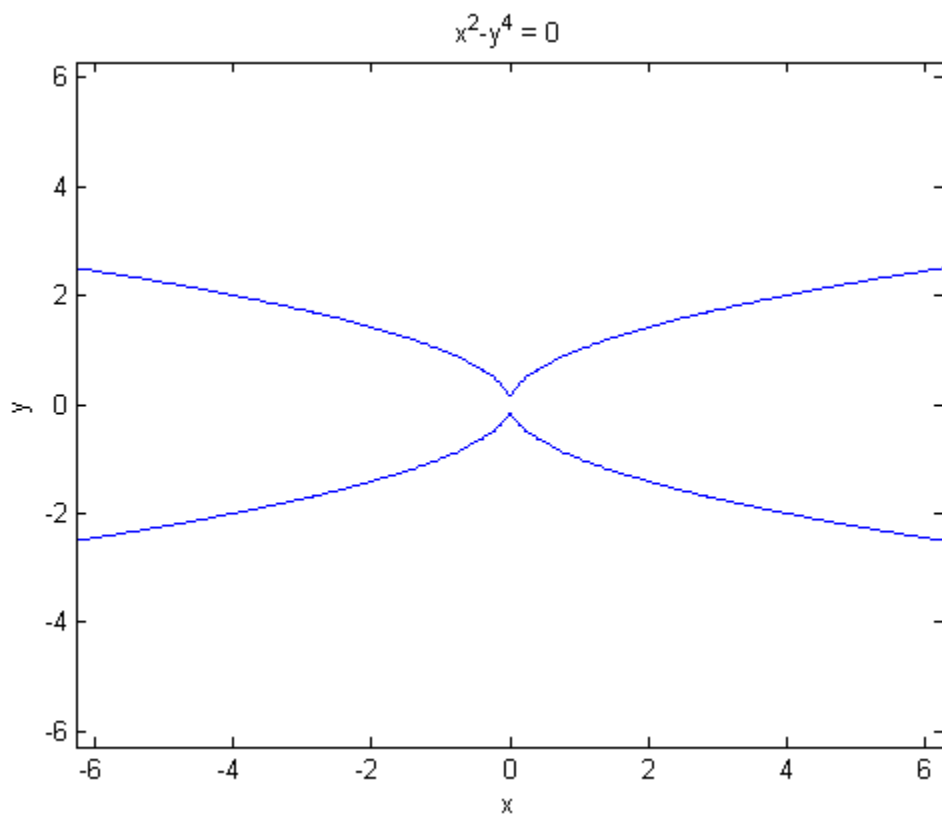
$$x^2 - y^4 = 0$$

over the domain  $[-2\pi, 2\pi]$ .

```
ezplot('x^2-y^4')  
colormap([0 0 1]) % Make the line blue
```

# ezplot

---



## See Also

[ezplot3](#) | [ezpolar](#) | [function\\_handle](#) | [plot](#)

## How To

- [Anonymous Functions](#)

**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  $\text{funx}(t)$ ,  $\text{funy}(t)$ , and  $\text{funz}(t)$  over the default domain  $0 < t < 2\pi$ .

`funx`, `funy`, and `funz` can be function handles or strings (see the Tips section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve  $\text{funx}(t)$ ,  $\text{funy}(t)$ , and  $\text{funz}(t)$  over the domain  $t_{\min} < t < t_{\max}$ .

`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 `myfunkt`:

```
function s = myfunkt(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```
ezplot3(@cos,@(t)myfunkt(t,1),@sqrt)
```

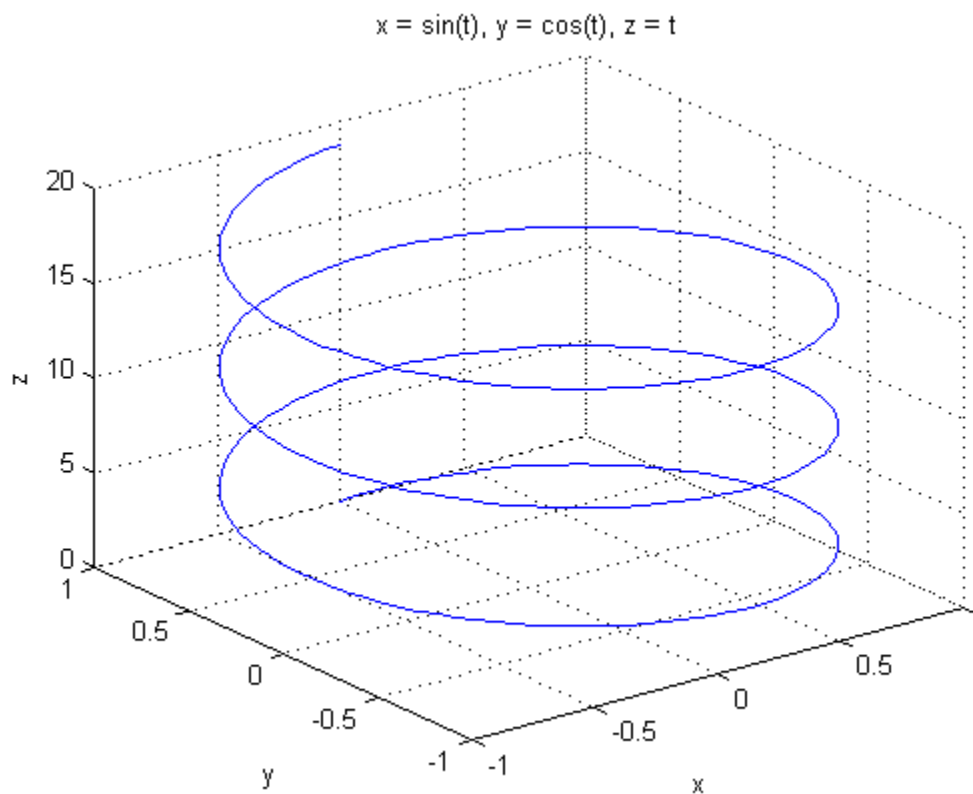
## Examples

This example plots the parametric curve

$$x = \sin t, \quad y = \cos t, \quad z = t$$

over the domain  $[0, 6\pi]$ :

```
ezplot3('sin(t)', 'cos(t)', 't', [0,6*pi])
```

**See Also**

[ezplot](#) | [ezpolar](#) | [function\\_handle](#) | [plot3](#)

**How To**

- [Anonymous Functions](#)

# ezpolar

---

**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  $\rho = \text{fun}(\theta)$  over the default domain  $0 < \theta < 2\pi$ .

`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))
```

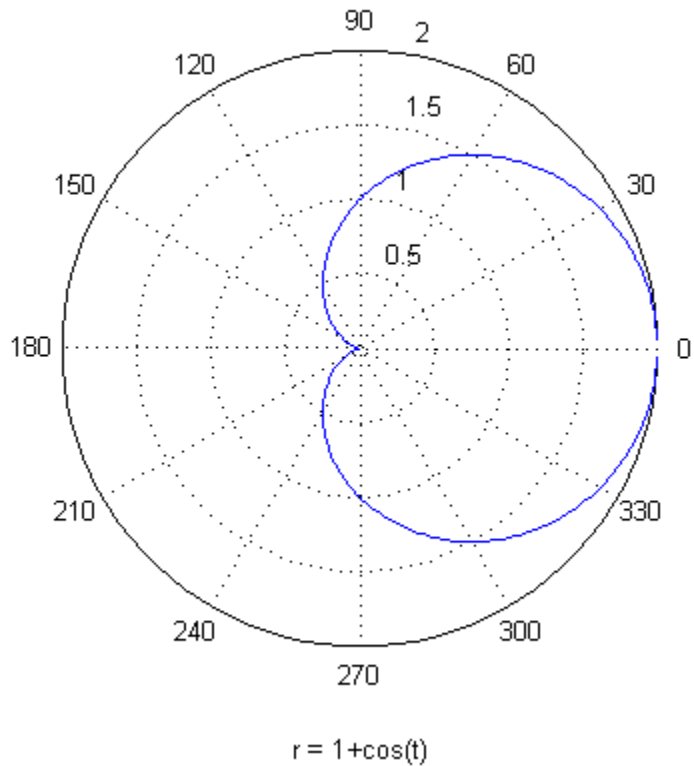
## Examples

This example creates a polar plot of the function

$$1 + \cos(t)$$

over the domain  $[0, 2\pi]$ :

```
ezpolar('1+cos(t)')
```



## See Also

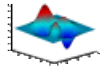
[ezplot](#) | [ezplot3](#) | [function\\_handle](#) | [plot](#) | [plot3](#) | [polar](#)

## How To

- [Anonymous Functions](#)

**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  $\text{fun}(x, y)$  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 2-1503 section for details on vector inputs vs axes limit outputs.

`ezsurf(funx, funy, funz)` plots the parametric surface  $\text{funx}(s, t)$ ,  $\text{funy}(s, t)$ , and  $\text{funz}(s, t)$  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 ezsurfz 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 ezsurf. 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  $u_{\min}$ ,  $u_{\max}$ ,  $v_{\min}$ , and  $v_{\max}$  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))
```

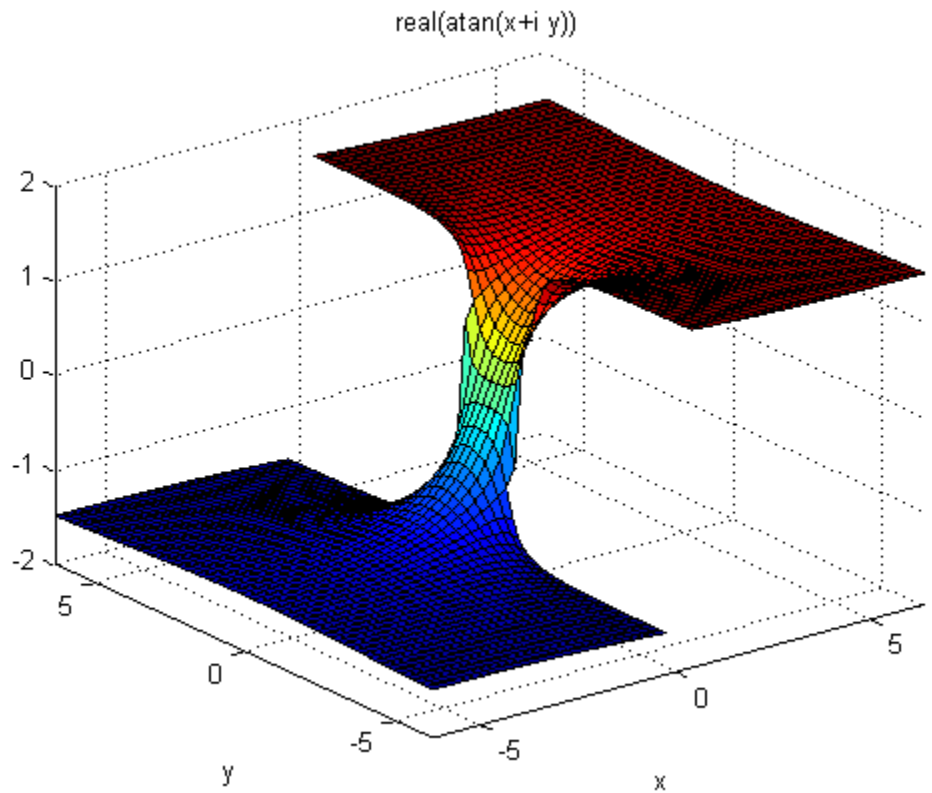
## Examples

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

$$f(x, y) = \text{real}(a \tan(x + iy))$$

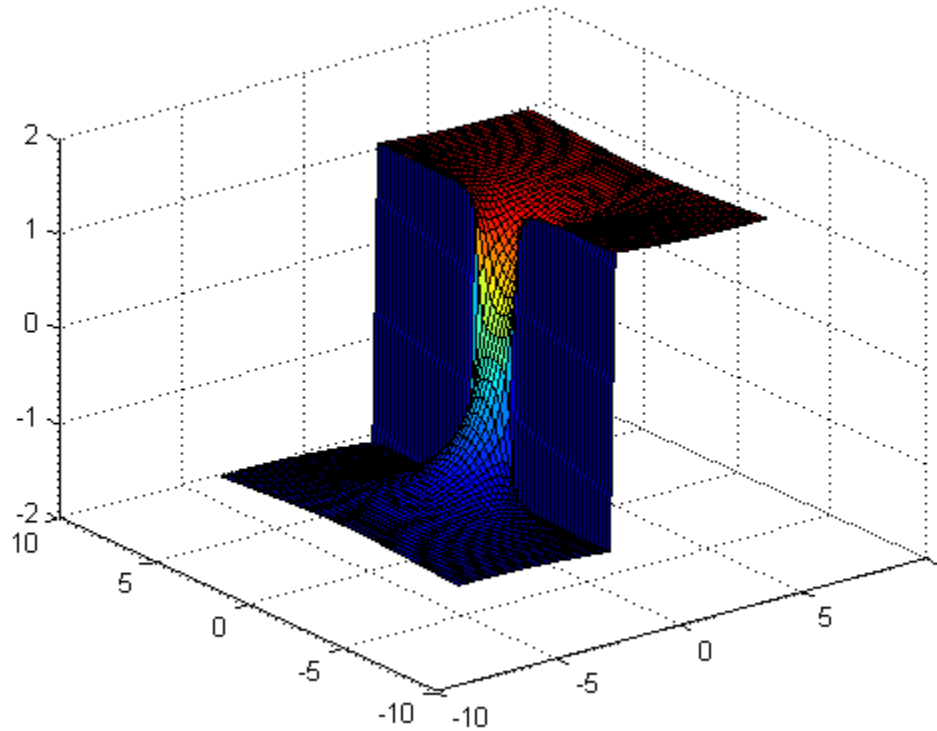
over the default domain  $-2\pi < x < 2\pi$ ,  $-2\pi < y < 2\pi$ :

```
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  $y$ -axis limits in different ways depending on how you input the domain (if at all). In the following table,  $R$  is the vector  $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$  and  $v$  is the manually entered domain vector.

Number of domain values specified:	Resulting domain vector:
<code>v = [ ];</code>	<code>R = [-2*pi, 2*pi, -2*pi, 2*pi];</code>
<code>v = [ v(1) ];</code>	<code>R = double([-abs(v),abs(v),-abs(v),abs(v)]);</code>
<code>v = [ v(1) v(2) ];</code>	<code>R = double([v(1),v(2),v(1),v(2)]);</code>
<code>v = [ v(1) v(2) v(3) ];</code>	<code>R = double([-v(1),v(2),-abs(v(3)),abs(v(3))]);</code>
<code>v = [ v(1) v(2) v(3) v(4) ];</code>	<code>R = double(v);</code>
<code>v = [ v(1)..v(n) ];</code> <code>n&gt;4</code>	<code>R = double([-abs(v(1)), abs(v(1)), -abs(v(1)), abs(v(1))]);</code>

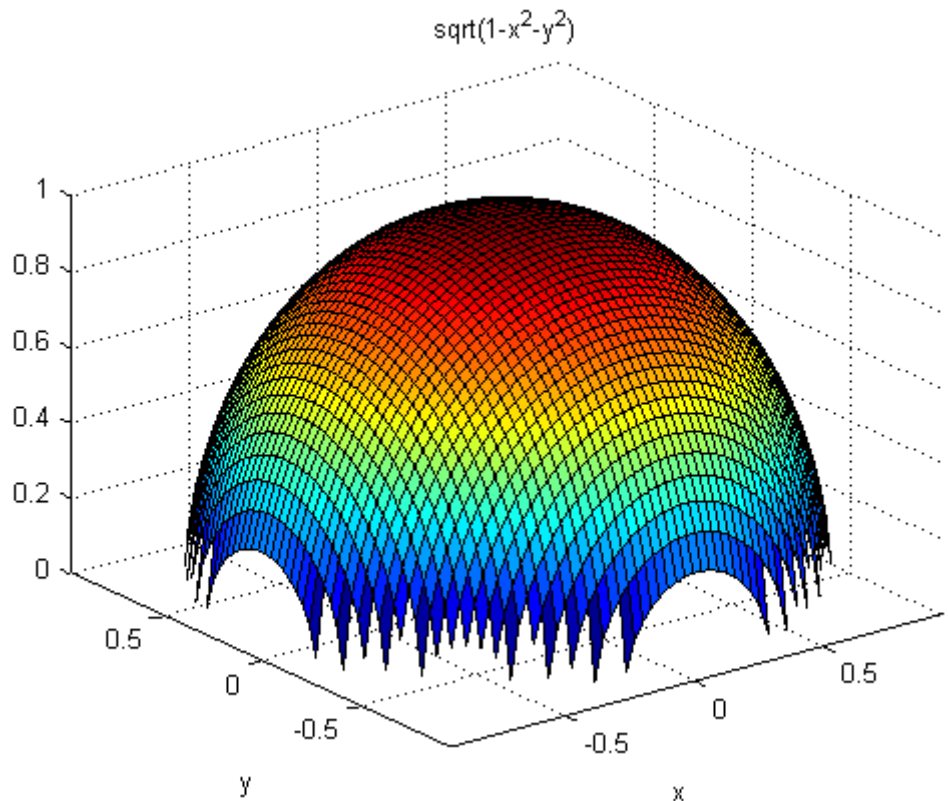
If you specify a single number in non-vector format (without square brackets, [ ]), ezsurf interprets it as the n, the number of points desired between the axes max and min values.

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  $x^2-y^2 \leq 1$ . The default graph of this function looks like this:

```
ezsurf('sqrt(1-x^2-y^2)')
```

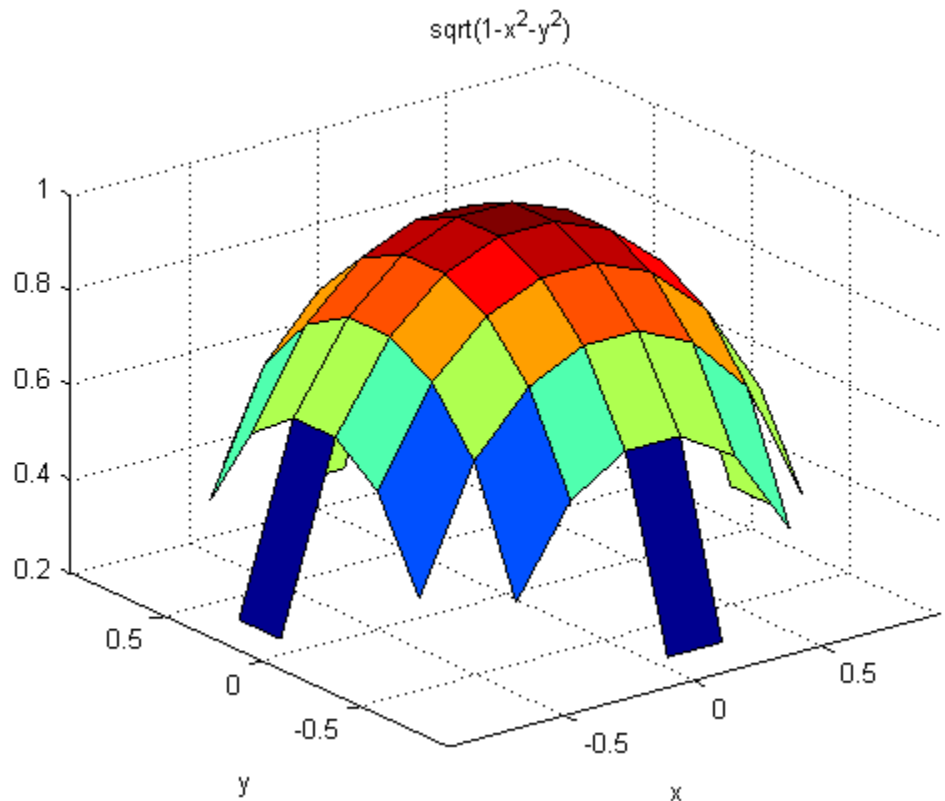




You can see that there are 60 points between the minimum and

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

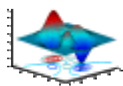
[ezmesh](#) | [ezsurf](#) | [function\\_handle](#) | [surf](#)

## How To

- [Anonymous Functions](#)

**Purpose**

Easy-to-use combination surface/contour 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  $\text{fun}(x,y)$  using the `surf` function. The 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` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]` (where  $\text{min} < x < \text{max}$ ,  $\text{min} < y < \text{max}$ ).

`ezsurf(funx,funy,funz)` plots the parametric surface  $\text{funx}(s,t)$ ,  $\text{funy}(s,t)$ , and  $\text{funz}(s,t)$  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  $f$  over the default domain using an  $n$ -by- $n$  grid. The default value for  $n$  is 60.

`ezsurf(...,'circ')` plots  $f$  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 handles to the graphics objects in `h`.

## Tips

`ezsurf` and `ezsurf` 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 `ezsurf`. For example, the MATLAB syntax for a surface/contour 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

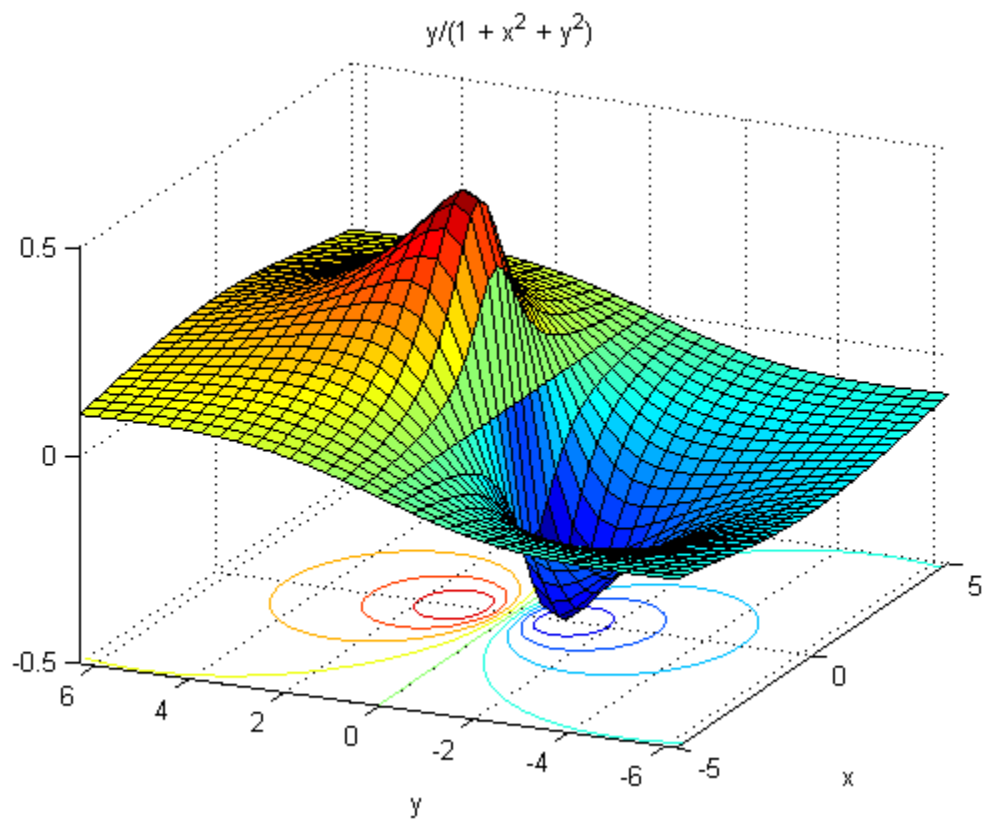
Create a surface/contour plot of the expression

$$f(x,y) = \frac{y}{1+x^2+y^2}$$

over the domain  $-5 < x < 5$ ,  $-2\pi < y < 2\pi$ , with a computational grid of size 35-by-35:

```
ezsurf('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](#) | [surf](#)

## How To

- [Anonymous Functions](#)

**Purpose** Unit normals to specified triangles

**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.  $m = \text{length}(\text{TI})$ , 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;  
theta = rand(numpts,1)*2*pi;  
phi = rand(numpts,1)*pi;  
x = cos(theta).*sin(phi);  
y = sin(theta).*sin(phi);  
z = cos(phi);  
dt = DelaunayTri(x,y,z);
```

## TriRep.faceNormals

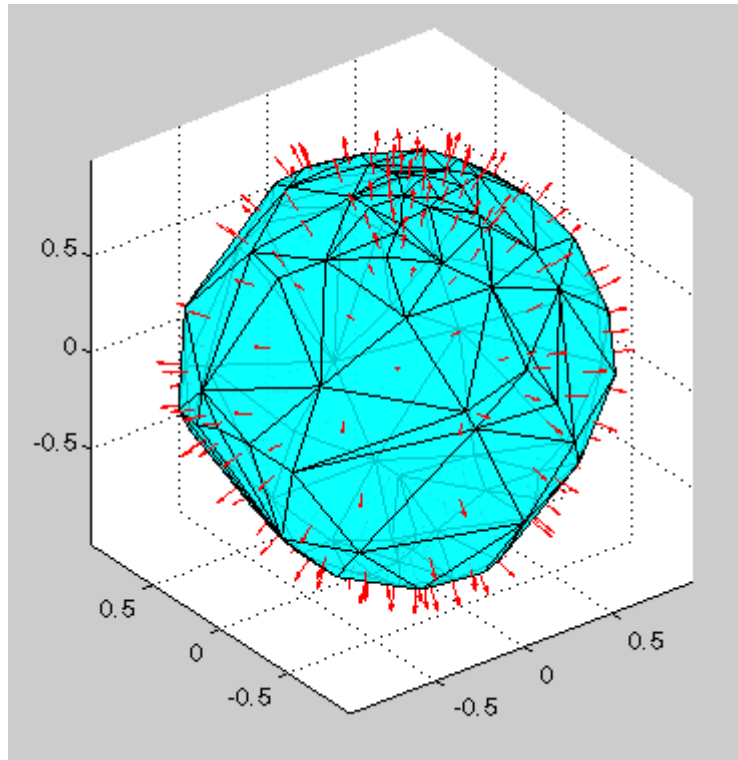
---

```
[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

[TriRep.freeBoundary](#) | [DelaunayTri](#)

# factor

---

<b>Purpose</b>	Prime factors
<b>Syntax</b>	<code>f = factor(n)</code>
<b>Description</b>	<code>f = factor(n)</code> returns a row vector containing the prime factors of <code>n</code> .
<b>Examples</b>	<pre>f = factor(123) f =      3     41</pre>
<b>See Also</b>	<code>isprime</code>   <code>primes</code>

**Purpose** Factorial function

**Syntax** `factorial(N)`

**Description** `factorial(N)`, for scalar `N`, is the product of all the integers from 1 to `N`, i.e. `prod(1:n)`. When `N` is an `N`-dimensional array, `factorial(N)` is the factorial for each element of `N`.

Since double precision numbers only have about 15 digits, the answer is only accurate for `n <= 21`. For larger `n`, the answer will have the right magnitude, and is accurate for the first 15 digits.

**See Also** `prod`

# false

---

**Purpose** Logical 0 (false)

**Syntax**  
`false`  
`false(n)`  
`false(m, n)`  
`false(m, n, p, ...)`  
`false(size(A))`

**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.

**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`.

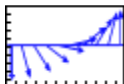
```
s = serial('COM1');  
fopen(s)  
fprintf(s, '*IDN?')  
idn = fscanf(s);  
fclose(s)
```

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(...,LineStyle)
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(...,LineStyle)` draws a feather plot using the line type, marker symbol, and color specified by `LineStyle`.

`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`.

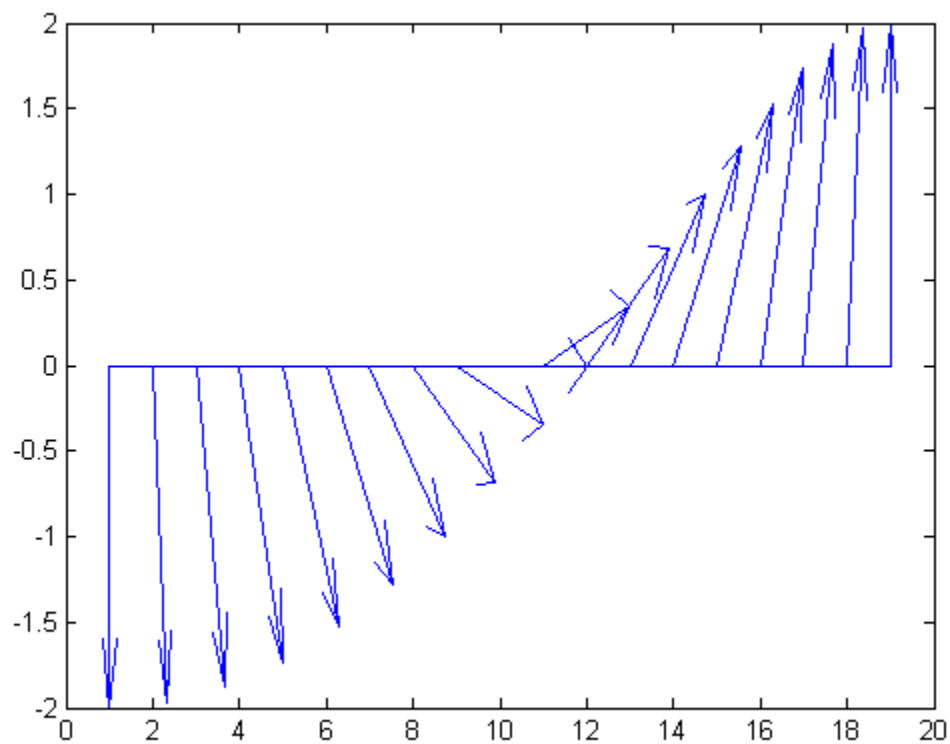
**Examples**

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);
```

# feather

---



## See Also

[compass](#) | [LineSpec](#) | [rose](#)



**Purpose** Sharp edges of surface triangulation

**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 $\pi$ 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(:,:);
```

# TriRep.featureEdges

---

Elevate the 2-D mesh to create a surface:

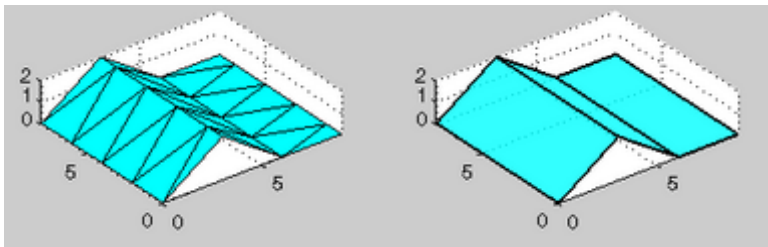
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0];
subplot(1,2,1);
trisurf(tri,x,y,z, 'FaceColor', 'cyan');
axis equal;
% TRISURF display of surface mesh
% showing mesh edges
```

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](#) | [DelaunayTri](#)

---

<b>Purpose</b>	Test for end-of-file
<b>Syntax</b>	<code>status = feof(fileID)</code>
<b>Description</b>	<p><code>status = feof(fileID)</code> returns 1 if a previous operation set the end-of-file indicator for the specified file. Otherwise, <code>feof</code> returns 0. <code>fileID</code> is an integer file identifier obtained from <code>fopen</code>.</p> <p>Opening an empty file does <i>not</i> set the end-of-file indicator. Read operations, and the <code>fseek</code> and <code>frewind</code> functions, move the file position indicator.</p>
<b>Examples</b>	<p>Read <code>bench.dat</code>, which contains MATLAB benchmark data, one character at a time:</p> <pre>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);</pre>
<b>See Also</b>	<code>fclose</code>   <code>ferror</code>   <code>fopen</code>   <code>frewind</code>   <code>fseek</code>   <code>ftell</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Testing for End of File (EOF)”</li></ul>

# ferror

---

**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`

**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);
```

# feval

---

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](#)

**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;

# Feval (COM)

Passing Mechanism	Description
	<pre>a = h.Feval('sin', 1, x);</pre>
Reference a server variable	<p>To reference a variable defined in the server, specify the variable name followed by an equals (=) sign:</p> <pre>h.PutWorkspaceData('x', 'base', -pi:0.01:pi); a = h.Feval('sin', 1, 'x=');</pre> <p>MATLAB does not reassign the server variable.</p>

## 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
```

## Feval (COM)

---

```
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**  
 $Y = \text{fft}(x)$   
 $Y = \text{fft}(X, n)$   
 $Y = \text{fft}(X, [], \text{dim})$   
 $Y = \text{fft}(X, n, \text{dim})$

**Definitions** The functions  $Y = \text{fft}(x)$  and  $y = \text{ifft}(X)$  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  $N$ th root of unity.

**Description**  $Y = \text{fft}(x)$  returns the discrete Fourier transform (DFT) of vector  $x$ , computed with a fast Fourier transform (FFT) algorithm.

If the input  $X$  is a matrix,  $Y = \text{fft}(X)$  returns the Fourier transform of each column of the matrix.

If the input  $X$  is a multidimensional array,  $\text{fft}$  operates on the first nonsingleton dimension.

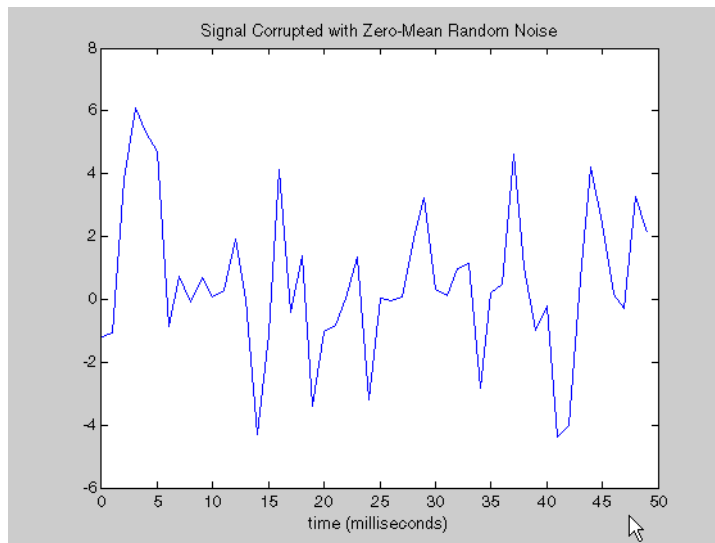
$Y = \text{fft}(X, n)$  returns the  $n$ -point DFT.  $\text{fft}(X)$  is equivalent to  $\text{fft}(X, n)$  where  $n$  is the size of  $X$  in the first nonsingleton dimension. If the length of  $X$  is less than  $n$ ,  $X$  is padded with trailing zeros to length  $n$ . If the length of  $X$  is greater than  $n$ , the sequence  $X$  is truncated. When  $X$  is a matrix, the length of the columns are adjusted in the same manner.

$Y = \text{fft}(X, [], \text{dim})$  and  $Y = \text{fft}(X, n, \text{dim})$  applies the FFT operation across the dimension  $\text{dim}$ .

## 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:

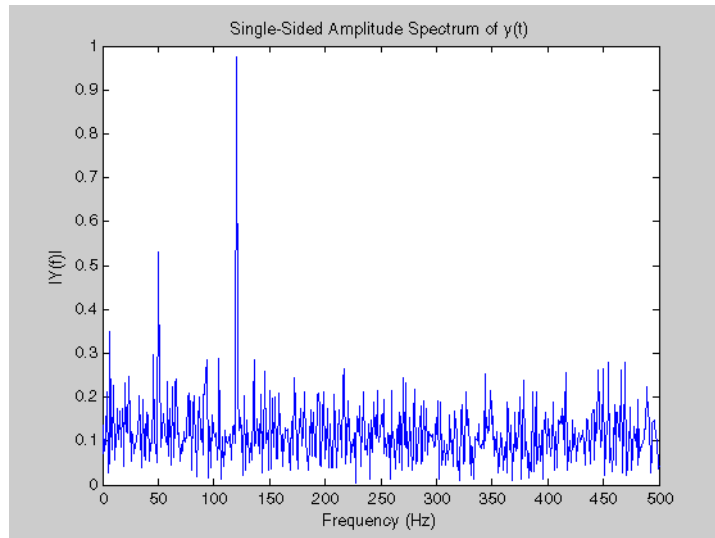
```
Fs = 1000; % Sampling frequency
T = 1/Fs; % Sample time
L = 1000; % Length of signal
t = (0:L-1)*T; % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t)); % Sinusoids plus noise
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted 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  $N = N_1 N_2$ ), 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. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.
- [6] Oppenheim, A. V. and R. W. Schaffer, *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`



---

<b>Purpose</b>	2-D fast Fourier transform
<b>Syntax</b>	<pre>Y = fft2(X) Y = fft2(X,m,n)</pre>
<b>Description</b>	<p><code>Y = fft2(X)</code> returns the two-dimensional discrete Fourier transform (DFT) of <code>X</code>. The DFT is computed with a fast Fourier transform (FFT) algorithm. The result, <code>Y</code>, is the same size as <code>X</code>.</p> <p>If the dimensionality of <code>X</code> is greater than 2, the <code>fft2</code> function returns the 2-D DFT for each higher dimensional slice of <code>X</code>. For example, if <code>size(X) = [100 100 3]</code>, then <code>fft2</code> computes the DFT of <code>X(:,:,1)</code>, <code>X(:,:,2)</code> and <code>X(:,:,3)</code>.</p> <p><code>Y = fft2(X,m,n)</code> truncates <code>X</code>, or pads <code>X</code> with zeros to create an <code>m</code>-by-<code>n</code> array before doing the transform. The result is <code>m</code>-by-<code>n</code>.</p>
<b>Algorithms</b>	<p><code>fft2(X)</code> can be simply computed as</p> <pre>fft(fft(X).').'</pre> <p>This computes the one-dimensional DFT of each column <code>X</code>, then of each row of the result. The execution time for <code>fft</code> 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.</p> <hr/> <p><b>Note</b> You might be able to increase the speed of <code>fft2</code> using the utility function <code>fftw</code>, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.</p> <hr/>
<b>Data Type Support</b>	<p><code>fft2</code> supports inputs of data types <code>double</code> and <code>single</code>. If you call <code>fft2</code> with the syntax <code>y = fft2(X, ...)</code>, the output <code>y</code> has the same data type as the input <code>X</code>.</p>

# fft2

---

## See Also

`fft` | `fftn` | `fftw` | `fftshift` | `ifft2`

**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`.

# fftn

---

## See Also

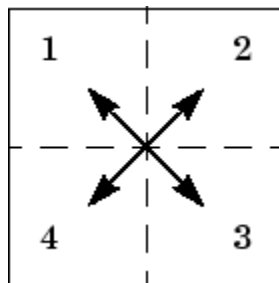
[fft](#) | [fft2](#) | [fftn](#) | [fftw](#) | [ifftn](#)

**Purpose** Shift zero-frequency component to center of spectrum

**Syntax**  $Y = \text{fftshift}(X)$   
 $Y = \text{fftshift}(X, \text{dim})$

**Description**  $Y = \text{fftshift}(X)$  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.



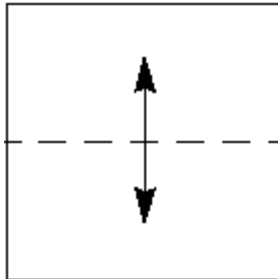
For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of  $X$  along each dimension.

$Y = \text{fftshift}(X, \text{dim})$  applies the `fftshift` operation along the dimension `dim`.

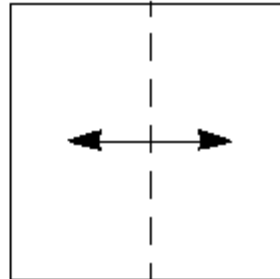
# fftshift

---

For dim = 1:



For dim = 2:



---

**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  $Y(1,1) = \text{sum}(\text{sum}(X))$ ; 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 2-1547 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 FFT 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](#)

---

<b>Purpose</b>	Read line from file, removing newline characters
<b>Syntax</b>	<code>tline = fgetl(fileID)</code>
<b>Description</b>	<p><code>tline = fgetl(fileID)</code> returns the next line of the specified file, removing the newline characters. <code>fileID</code> is an integer file identifier obtained from <code>fopen</code>. <code>tline</code> is a text string unless the line contains only the end-of-file marker. In this case, <code>tline</code> is the numeric value <code>-1</code>.</p> <p><code>fgetl</code> reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use <code>fopen</code>.</p>
<b>Examples</b>	<p>Read and display the file <code>fgetl.m</code> one line at a time:</p> <pre>fid = fopen('fgetl.m');  tline = fgetl(fid); while ischar(tline)     disp(tline)     tline = fgetl(fid); end  fclose(fid);</pre> <p>Compare these results to the <code>fgets</code> example, which replaces the calls to <code>fgetl</code> with <code>fgets</code>.</p>
<b>See Also</b>	<code>fclose</code>   <code>feof</code>   <code>ferror</code>   <code>fgets</code>   <code>fopen</code>   <code>fprintf</code>   <code>fread</code>   <code>fscanf</code>   <code>fwrite</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Testing for EOF with <code>fgetl</code> and <code>fgets</code>”</li></ul>

# fgetl (serial)

---

**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
```

# fgetl (serial)

---

## **See Also**

fgets | fopen | BytesAvailable | InputBufferSize | ReadAsyncMode  
| Status | Terminator | Timeout | ValuesReceived



---

<b>Purpose</b>	Read line from file, keeping newline characters
<b>Syntax</b>	<pre><i>tline</i> = fgets(<i>fileID</i>) <i>tline</i> = fgets(<i>fileID</i>, <i>nchar</i>)</pre>
<b>Description</b>	<p><i>tline</i> = fgets(<i>fileID</i>) reads the next line of the specified file, including the newline characters. <i>fileID</i> is an integer file identifier obtained from fopen. <i>tline</i> is a text string unless the line contains only the end-of-file marker. In this case, <i>tline</i> is the numeric value -1. fgets reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use fopen.</p> <p><i>tline</i> = fgets(<i>fileID</i>, <i>nchar</i>) returns at most <i>nchar</i> characters of the next line. <i>tline</i> does not include any characters after the newline characters or the end-of-file marker.</p>
<b>Examples</b>	<p>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:</p> <pre>fid = fopen('fgets.m');  tline = fgets(fid); while ischar(tline)     disp(tline)     tline = fgets(fid); end  fclose(fid);</pre> <p>Compare these results to the fgetl example, which replaces the calls to fgets with fgetl.</p>
<b>See Also</b>	<code>fclose</code>   <code>feof</code>   <code>ferror</code>   <code>fgetl</code>   <code>fopen</code>   <code>fprintf</code>   <code>fread</code>   <code>fscanf</code>   <code>fwrite</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Testing for EOF with fgetl and fgets”</li></ul>

# fgets (serial)

---

**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
```

# fgets (serial)

---

## **See Also**

fgetc | fopen | BytesAvailable | BytesAvailableFcn |  
InputBufferSize | Status | Terminator | Timeout | ValuesReceived

**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)
```

# fieldnames

---

```
names =  
    'MIN_VALUE'  
    'MAX_VALUE'  
    'TYPE'  
    'SIZE'
```

## See Also

[setfield](#) | [getfield](#) | [isfield](#) | [orderfields](#) | [rmfield](#)

## How To

- “Generate Field Names from Variables”

---

<b>Purpose</b>	Create figure graphics object
<b>Syntax</b>	<pre>figure figure('PropertyName',propertyvalue,...) figure(h) h = figure(...)</pre>
<b>Properties</b>	For a list of properties, see <a href="#">Figure Properties</a> .
<b>Description</b>	<p><code>figure</code> creates figure graphics objects. Figure objects are the individual windows on the screen in which the MATLAB software displays graphical output.</p> <p><code>figure</code> 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.</p> <p><code>figure('PropertyName',propertyvalue,...)</code> creates a new figure object using the values of the properties specified. For a description of the properties, see <a href="#">Figure Properties</a>. MATLAB uses default values for any properties that you do not explicitly define as arguments.</p> <p><code>figure(h)</code> does one of two things, depending on whether or not a figure with handle <code>h</code> exists. If <code>h</code> is the handle to an existing figure, <code>figure(h)</code> makes the figure identified by <code>h</code> 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 <code>h</code> is not the handle to an existing figure, but is an integer, <code>figure(h)</code> creates a figure and assigns it the handle <code>h</code>. <code>figure(h)</code> where <code>h</code> is not the handle to a figure, and is not an integer, is an error.</p> <p><code>h = figure(...)</code> returns the handle to the figure object.</p>
<b>Tips</b>	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 <a href="#">Figure Properties</a> for a description of these properties.

# figure

---

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).

# Figure Properties

---

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,evt)
    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:

- 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.
- The close operation executes the function defined by the figure `CloseRequestFcn`. The default function is named `closereq` and is predefined as:

```
function closereq
```

# Figure Properties

---

```
if isempty(gcf)
    if length(dbstack) == 1
        warning('MATLAB:closereq', ...
            'Calling closereq from the command line is now obsolete, use close instead');
    end
    close force
else
    if (isa(gcf,'ui.figure'))
        % Convert GBT1.5 figure to a double.
        delete(double(gcf));
    else
        delete(gcf);
    end
end
end
```

These statements unconditionally delete the current figure, destroying the window. `closereq` takes advantage of the fact that the `close` command makes all figures specified as arguments the current figure before calling the respective close request function.

Note that `closereq` honors the user's `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.

# Figure Properties

---

## 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.

`CreateFcn`The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcb0`.

## 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

# Figure Properties

---

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

# Figure Properties

---

`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

*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.

# Figure Properties

---

```
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,

# Figure Properties

---

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.

# Figure Properties

---

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 patches and surfaces 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., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, it contains 'command' when pressing the <b>command</b> 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**

# Figure Properties

---

**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.

## Figure Properties

---

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., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, it contains 'command' when releasing the <b>command</b> 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.

# Figure Properties

---

```
function key_releaseFcn
figure('KeyReleaseFcn',@cb)
function cb(src,evt)
if ~isempty(evt.Modifier)
for ii = 1:length(evt.Modifier)
out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
evt.Character,evt.Modifier{ii},evt.Key);
disp(out)
end
else
out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
evt.Character,'No modifier key',evt.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

# Figure Properties

---

*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

# Figure Properties

---

*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

# Figure Properties

---

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. The handle to the root is always 0.

## Pointer

`crosshair` | `{arrow}` | `watch` | `top1` |  
`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 Properties

---

*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 reenabte 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.

# Figure Properties

---

- 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.

# Figure Properties

---

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	<b>Shift</b> - click left mouse button or click both left and right mouse buttons.	<b>Shift</b> -click left mouse button or click middle mouse button.



Selection Type	Microsoft Windows	X-Windows
Alternate	<b>Control</b> - click left mouse button or click right mouse button.	<b>Control</b> -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')
```

# Figure Properties

---

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 `WindowState` of a window to `'modal'` hides both its toolbar and menu bar, if they exist. Changing `WindowState` from `'modal'` to `'normal'` or `'docked'` displays any toolbar or menu bar a figure has.

If you start MATLAB with the `-nojvm` option, figures do not display the toolbar because most tools require Java figures. This option is obsolete and no longer supported in MATLAB software.

## 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 =  $\frac{1}{72}$  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.

# Figure Properties

---

**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.

## Figure Properties

---

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., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, it contains 'command' when releasing the <b>command</b> 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., <b>control</b> , <b>alt</b> , <b>shift</b> ). On Macintosh computers, it contains 'command' when releasing the <b>command</b> 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 <code>One screen at a time</code> , <code>VerticalScrollAmount</code> 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

# Figure Properties

---

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 `uimenu`s 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 `uimenu`s are displayed.

## Docked WindowStyle

# Figure Properties

---

When `WindowState` is `docked`, the figure is docked in the desktop or a document window. When you issue the following command:

```
set(figure_handle, 'WindowState', 'docked')
```

MATLAB docks the figure identified by *figure\_handle* and sets the `DockControls` property to `on`, if it was `off`.

Note that if `WindowState` is `docked`, you cannot set the `DockControls` property to `off`.

The value of the `WindowState` 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.

# Figure Properties

---

- **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')
```

# Figure Properties

---

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  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  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  0  0  0  None
0x26 24  tc  0 24  0  r  .  .  8  8  8  8  0 24  8  0  0  0  0  0  0  0  0  0  None
0x27 24  tc  0 24  0  r  y  .  8  8  8  8  0  0  0  0  16 16 16  0  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`.

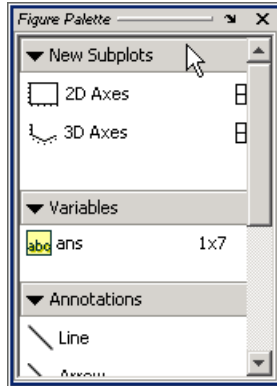
## See Also

`figure`

# figurepalette

---

**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

# fileattrib

---

**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.

# fileattrib

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 <code>fileattrib</code> 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

# fileattrib

---

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. <sup>2</sup>

## 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
```

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

```
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')
```

```
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 specified 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

---

```
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 specified 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')
```

# fileattrib

---

```
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 -w
```

```
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](#)

# filebrowser

---

<b>Purpose</b>	Open Current Folder browser, or select it if already open
<b>Syntax</b>	filebrowser
<b>Description</b>	filebrowser opens the Current Folder browser, or if it is already open, makes it the selected tool.
<b>See Also</b>	cd   copyfile   fileattrib   ls   mkdir   movefile   pwd   rmdir
<b>How To</b>	<ul style="list-style-type: none"><li>• “Working with Files and Folders”</li></ul>

- 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
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`

**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`.

# fileparts

---

## 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”

# fileread

---

**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`



**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
```

# filesep

---

Create a path to `iofun` on a UNIX<sup>3</sup> 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.

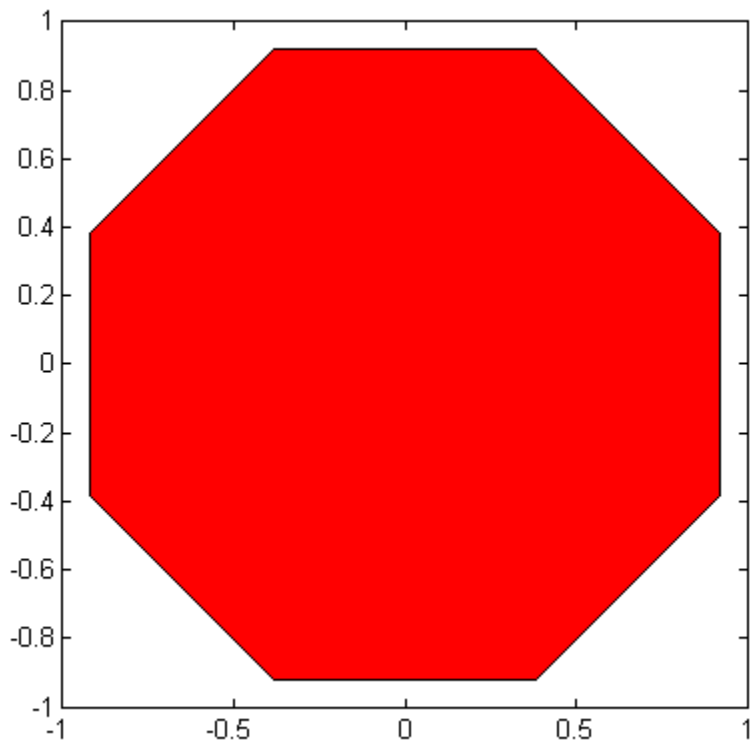
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the  $n$ 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](#)

# fill3

---

## 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.

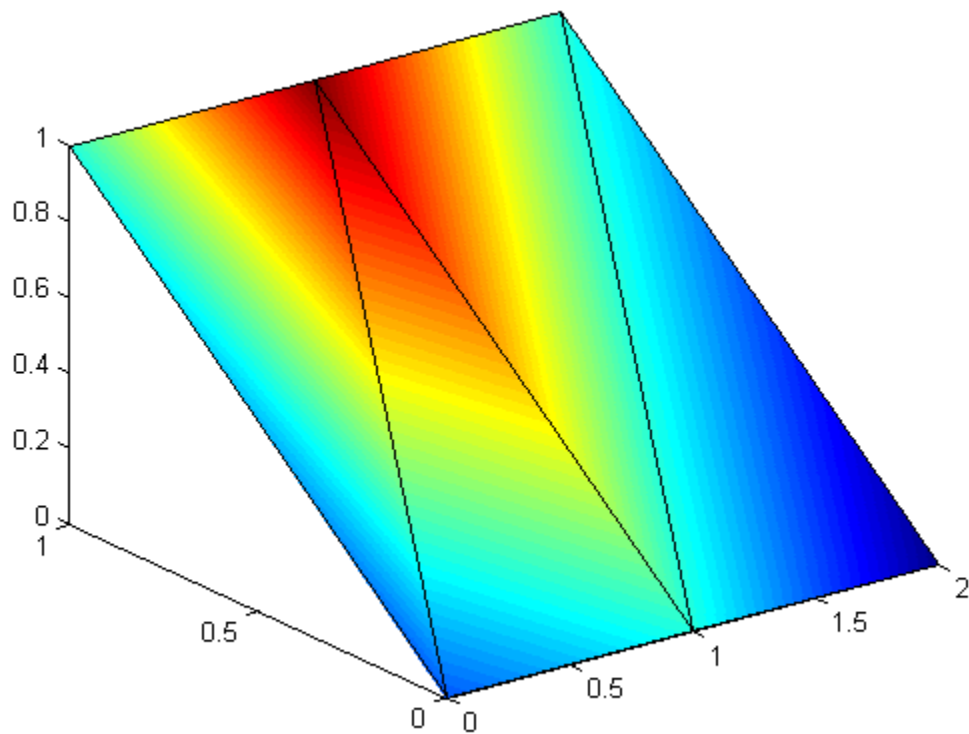
## Examples

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)
```

# fill3

---



## 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  $\max(\text{length}(a), \text{length}(b)) - 1$ . 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  $\max(\text{length}(a), \text{length}(b)) - 1$ , or an array with the leading dimension of size  $\max(\text{length}(a), \text{length}(b)) - 1$  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`.

# filter

## Examples

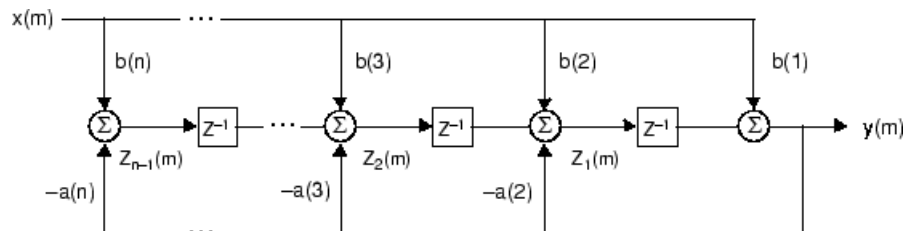
You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
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,





# 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

```

# find

---

## 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
```

# find

---

```
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](#)

---

<b>Purpose</b>	Find all graphics objects
<b>Syntax</b>	<pre>object_handles = findall(handle_list) object_handles = findall(handle_list, 'property', 'value', ...)</pre>
<b>Description</b>	<p><code>object_handles = findall(handle_list)</code> returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in <code>handle_list</code>.</p> <p><code>object_handles = findall(handle_list, 'property', 'value', ...)</code> returns the handles of all objects in the hierarchy under the objects identified in <code>handle_list</code> that have the specified properties set to the specified values.</p>
<b>Tips</b>	<code>findall</code> is similar to <code>findobj</code> , except that it finds objects even if their <code>HandleVisibility</code> is set to <code>off</code> .
<b>Examples</b>	<pre>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</pre>
<b>See Also</b>	<code>allchild</code>   <code>findobj</code>

# 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('-regex','PropertyName','regex',...)` matches objects using regular expressions as if the value of you passed the property `PropertyName` to the `regex` function as

```
regex(PropertyValue,'regex')
```

If a match occurs, `findobj` returns the object handle. See the `regex` 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])
```

# findobj

---

## 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 = findboj(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 = findboj(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  
end
```

## findobj (handle)

---

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  
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](#)

# 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` | `regex` | `regexp` | `regpi` | `regprep`

**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:

# finish

---

- Click the Close box  in the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX<sup>4</sup> 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

- “Exiting the MATLAB Program”

4. UNIX is a registered trademark of The Open Group in the United States and other countries.



---

<b>Purpose</b>	Display FITS metadata
<b>Syntax</b>	<code>fitsdisp(filename)</code> <code>fitsdisp(filename,Name,Value)</code>
<b>Description</b>	<p><code>fitsdisp(filename)</code> displays metadata for all the Header/Data Units (HDUs) found in the FITS file specified by <code>filename</code>.</p> <p><code>fitsdisp(filename,Name,Value)</code> displays metadata for all the Header/Data Units (HDUs) found in the FITS file with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>filename</b> Text string specifying the name of an existing FITS file.</p> <p><b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>Index</b> Positive scalar value or vector specifying the HDUs.</p> <p><b>Mode</b> One of the following strings:</p> <ul style="list-style-type: none"><li>• <code>standard</code> – Display standard keywords</li><li>• <code>min</code> – Display only HDU types and sizes</li><li>• <code>full</code> – Display all HDU keywords</li></ul> <p><b>Default:</b> <code>standard</code></p>

# fitsdisp

---

## 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 2-1676.

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: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Slope	Value, used with Intercept, to calculate actual pixel values from the array pixel values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	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: $\text{actual\_value} = \text{slope} * \text{array\_value} + \text{Intercept}$	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: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	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: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	Double
Slope	Value, used with Intercept, to calculate actual pixel values from the array pixel values, using the equation: $\text{actual\_value} = \text{Slope} * \text{array\_value} + \text{Intercept}$	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

# fitsinfo

---

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, <code>data</code> , is a 1-D cell array.
'binarytable'	Read data from the Binary Table extension. The return value, <code>data</code> , 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, 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`.

### 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');
```

# fitsread

---

```
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”



---

<b>Purpose</b>	Write image to FITS file
<b>Syntax</b>	<code>fitswrite(imagedata,filename)</code> <code>fitswrite(imagedata,filename,Name,Value)</code>
<b>Description</b>	<p><code>fitswrite(imagedata,filename)</code> writes <code>imagedata</code> to the FITS file specified by <code>filename</code>. If <code>filename</code> does not exist, <code>fitswrite</code> creates the file as a simple FITS file. If <code>filename</code> exists, <code>fitswrite</code> overwrites the file or appends the image to the end of the file, depending on the value of the <code>writemode</code> argument.</p> <p><code>fitswrite(imagedata,filename,Name,Value)</code> writes <code>imagedata</code> to the FITS file specified by <code>filename</code> with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>imagedata</b> Image array.</p> <p><b>filename</b> Text string specifying the name of an existing FITS file or the name you want to assign to a new FITS file.</p> <p><b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>WriteMode'</b> One of these strings:</p> <ul style="list-style-type: none"><li>• <code>overwrite</code></li><li>• <code>append</code></li></ul>

# fitswrite

---

**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` | `fitsreadmatlab.io.fits` |

**Purpose** Round toward zero

**Syntax**  $B = \text{fix}(A)$

**Description**  $B = \text{fix}(A)$  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`

# flipdim

---

**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`

**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`

# flipud

---

**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` | `flip1r` | `rot90`

# floor

---

**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`



---

<b>Purpose</b>	Simple function of three variables
<b>Syntax</b>	<pre>v = flow v = flow(n) v = flow(x,y,z) [x,y,z,v] = flow(...)</pre>
<b>Description</b>	<p><code>flow</code>, a function of three variables, generates fluid-flow data that is useful for demonstrating <code>slice</code>, <code>interp3</code>, and other functions that visualize scalar volume data.</p> <p><code>v = flow</code> produces a 50-by-25-by-25 array.</p> <p><code>v = flow(n)</code> produces a n-by-2n-by-n array.</p> <p><code>v = flow(x,y,z)</code> evaluates the speed profile at the points x, y, and z.</p> <p><code>[x,y,z,v] = flow(...)</code> returns the coordinates as well as the volume data.</p>
<b>See Also</b>	<code>slice</code>   <code>interp3</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Slicing Fluid Flow Data”</li></ul>

# fminbnd

---

**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  $x_1 < x < x_2$ . `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:

<code>Display</code>	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.
<code>FunValCheck</code>	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.
<code>MaxFunEvals</code>	Maximum number of function evaluations allowed.
<code>MaxIter</code>	Maximum number of iterations allowed.

<code>OutputFcn</code>	User-defined function that is called at each iteration. See “Output Functions” in MATLAB Mathematics for more information.
<code>PlotFcns</code>	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 ( <code>[]</code> ). <ul style="list-style-type: none"> <li>• <code>@optimplotx</code> plots the current point</li> <li>• <code>@optimplotfval</code> plots the function value</li> <li>• <code>@optimplotfunccount</code> plots the function count</li> </ul> See “Plot Functions” in MATLAB Mathematics for more information.
<code>TolX</code>	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	<code>fminbnd</code> converged to a solution $x$ based on <code>options.TolX</code> .
0	Maximum number of function evaluations or iterations was reached.
-1	Algorithm was terminated by the output function.
-2	Bounds are inconsistent ( $x_1 > x_2$ ).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization in the following fields:

# fminbnd

---

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  $\pi$  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  $\pi$  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 `fminbnd`. 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

# fminbnd

---

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 an interior point at a distance of no more than  $2 \cdot \text{TolX}$  from  $x_1$  or  $x_2$ , where `TolX` is the termination tolerance. 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 2-1706 and “Example 3” on page 2-1706 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.

# fminsearch

---

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 ( []). <ul style="list-style-type: none"><li>• @optimplotx plots the current point</li><li>• @optimplotfval plots the function value</li><li>• @optimplotfunccount plots the function count</li></ul> 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	<code>fminsearch</code> converged to a solution <code>x</code> .
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:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

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:

$$f(x) = 100(x_2 - x_1^2)^2 + (a - x_1)^2.$$

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.

# fminsearch

---

## 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

**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.

# fopen

---

## 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*.

# fopen

---

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`

**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.

## fopen (serial)

---

```
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.

# for

---

- 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

# format

---

**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 2-1725	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 <code>single</code> values.	
<code>shortE</code>	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
<code>longE</code>	Long scientific notation, with 15 digits after the decimal point for <code>double</code> values, and 7 digits after the decimal point for <code>single</code> values.  Integer-valued floating-point numbers with a maximum of 9 digits do not display in scientific notation.	3.141592653589793e+00
<code>shortG</code>	The more compact of short fixed decimal or scientific notation, with 5 digits.	3.1416
<code>longG</code>	The more compact of long fixed decimal or scientific notation, with 15 digits for <code>double</code> values, and 7 digits for <code>single</code> values.	3.14159265358979
<code>shortEng</code>	Short engineering notation, with 4 digits after the decimal point, and an exponent that is a multiple of 3.	3.1416e+000
<code>longEng</code>	Long engineering notation, with 15 significant digits, and an exponent that is a multiple of 3.	3.14159265358979e+000

# format

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 =  
  
    ffffffffffffffffff  
  
realmax  
  
ans =  
  
    7fefffffffffffffff
```

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

```

```

5.123456789000000e+000
51.23456789000000e+000
512.3456789000000e+000
5.123456789000000e+003
51.23456789000000e+003
512.3456789000000e+003
5.123456789000000e+006
51.23456789000000e+006
512.3456789000000e+006
5.123456789000000e+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.

# 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”

# fplot

---

**Purpose** Plot function between specified limits

**Syntax**

```
fplot(fun,limits)
fplot(fun,limits,LineStyle)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineStyle)
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  $[f_1(x), f_2(x), f_3(x)]$  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,LineStyle)` plots `fun` using the line specification `LineStyle`.



`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,LineStyle)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineStyle` for more information.

`fplot(fun,limits,n)` with  $n \geq 1$  plots the function with a minimum of  $n+1$  points. The default  $n$  is 1. The maximum step size is restricted to be  $(1/n) * (x_{\max} - x_{\min})$ .

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineStyle`, 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)`.

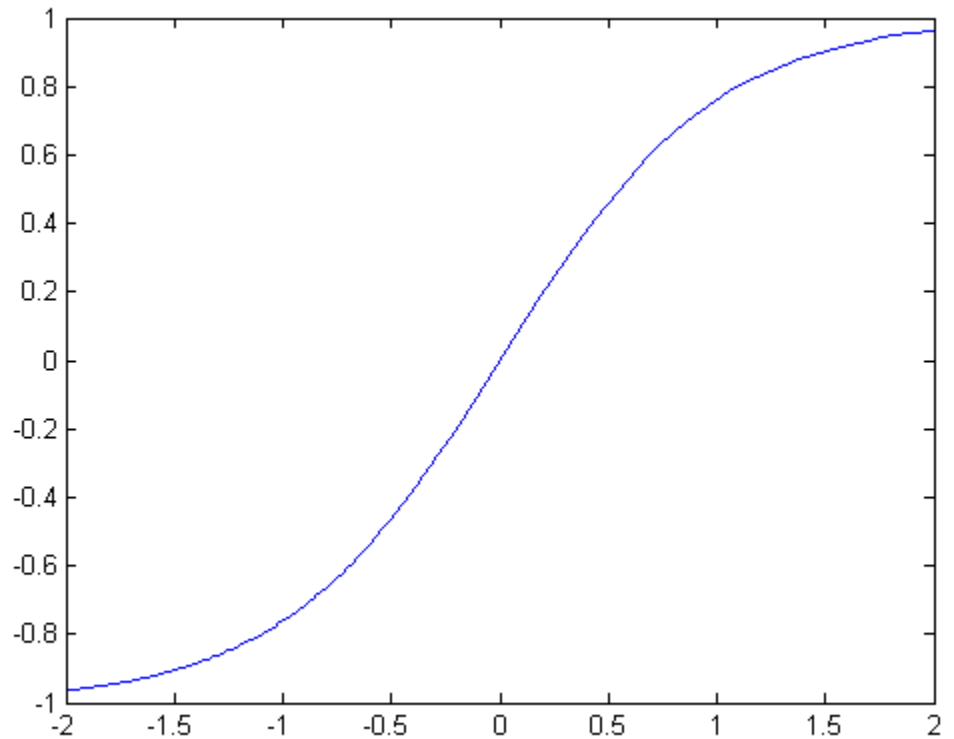
## Tips

`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:

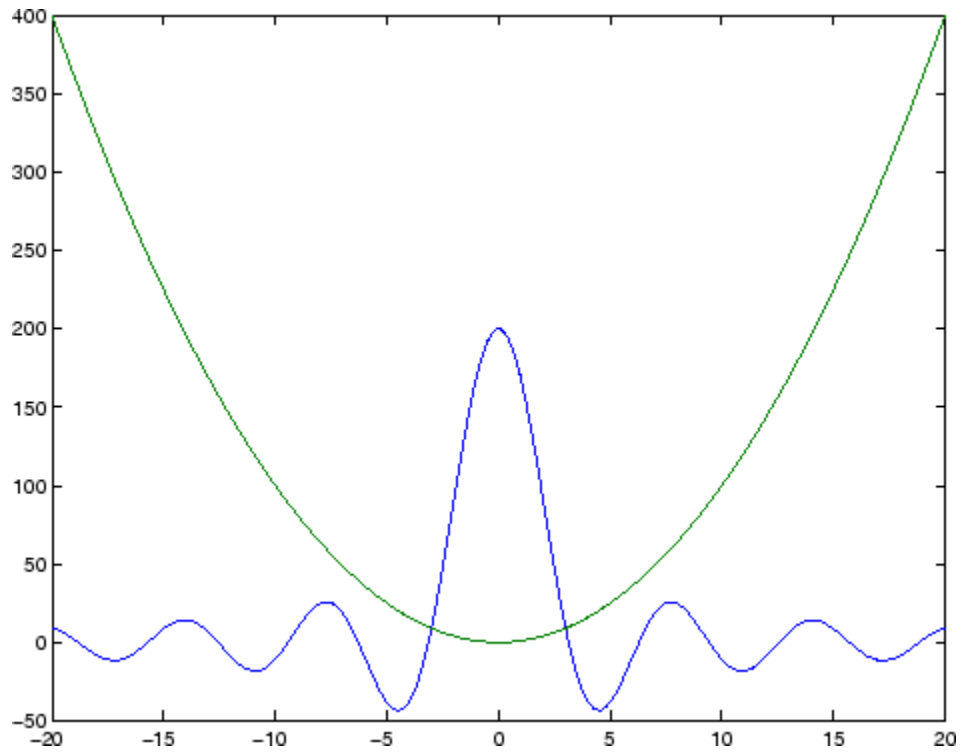
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to myfun:

```
fh = @myfun;
```

Plot the function with the statement

```
fplot(fh,[-20 20])
```



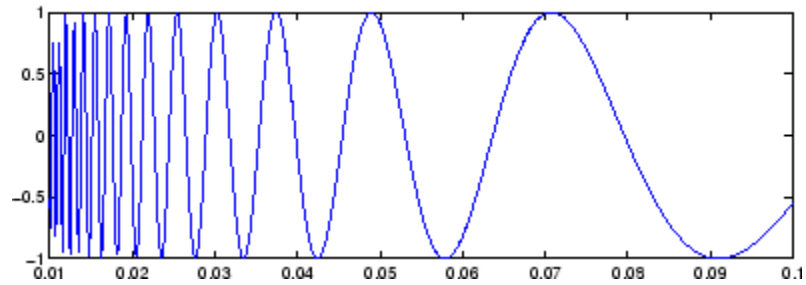
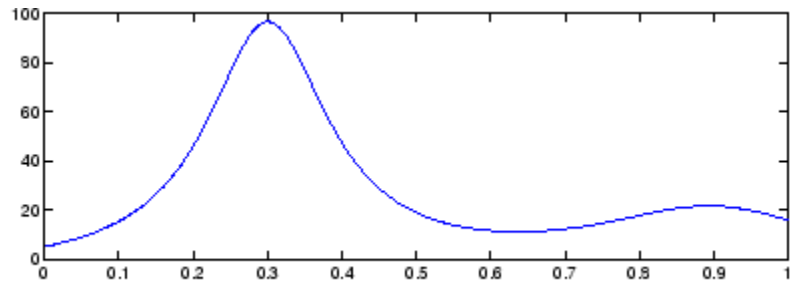
### Additional Example

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])
```

# fplot

---



## See Also

[eval](#) | [ezplot](#) | [feval](#) | [LineStyle](#) | [plot](#)

## How To

- [Anonymous Functions](#)

**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**

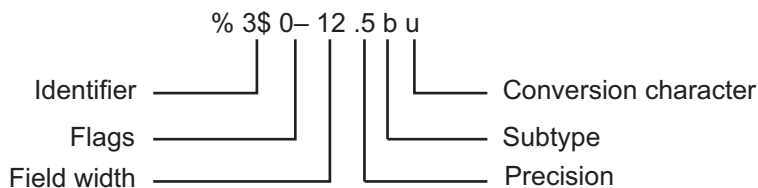
string

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.
- Escape characters, including:

' ' Single quotation mark

%% Percent character



<code>\\</code>	Backslash
<code>\a</code>	Alarm
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\xN</code>	Hexadecimal number, <i>N</i>
<code>\N</code>	Octal number, <i>N</i>

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.

# fprintf

---

## 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”

**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)

# fprintf (serial)

---

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](#)

# frame2im

---

**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



---

<b>Purpose</b>	Read data from binary file
<b>Syntax</b>	<pre>A = fread(<i>fileID</i>) A = fread(<i>fileID</i>, <i>sizeA</i>) A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>) A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>, <i>skip</i>) A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>, <i>skip</i>, <i>machineformat</i>) [A, <i>count</i>] = fread(...)</pre>
<b>Description</b>	<p><code>A = fread(<i>fileID</i>)</code> reads data from a binary file into column vector <i>A</i> and positions the file pointer at the end-of-file marker.</p> <p><code>A = fread(<i>fileID</i>, <i>sizeA</i>)</code> reads <i>sizeA</i> elements into <i>A</i> and positions the file pointer after the last element read. <i>sizeA</i> can be an integer, or can have the form <math>[m, n]</math>.</p> <p><code>A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>)</code> interprets values in the file according to the form and size described by the <i>precision</i>. The <i>sizeA</i> parameter is optional.</p> <p><code>A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>, <i>skip</i>)</code> skips <i>skip</i> bytes after reading each value. If <i>precision</i> is <i>bitn</i> or <i>ubitn</i>, specify <i>skip</i> in bits. The <i>sizeA</i> parameter is optional.</p> <p><code>A = fread(<i>fileID</i>, <i>sizeA</i>, <i>precision</i>, <i>skip</i>, <i>machineformat</i>)</code> reads data with the specified <i>machineformat</i>. The <i>sizeA</i> and <i>skip</i> parameters are optional.</p> <p><code>[A, <i>count</i>] = fread(...)</code> returns the number of elements <code>fread</code> reads into <i>A</i>.</p>
<b>Input Arguments</b>	<p><b>fileID</b></p> <p>An integer file identifier obtained from <code>fopen</code>.</p> <p><b>sizeA</b></p> <p>Dimensions of the output array <i>A</i>. Specify in one of the following forms:</p>

# fread

---

<code>inf</code>	Column vector with the number of elements in the file. (default)
<code>n</code>	Column vector with $n$ elements. If the file has fewer than $n$ elements, <code>fread</code> pads $A$ with zeros.
<code>[m,n]</code>	$m$ -by- $n$ matrix, filled in column order. If the file has fewer than $m*n$ elements, <code>fread</code> pads $A$ with zeros. $n$ can be <code>inf</code> , 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:

<code>'source'</code>	Specifies class of input values. Output matrix $A$ is class <code>double</code> . Example: <code>'int16'</code>
<code>'source=&gt;output'</code>	Specifies classes of input and output. Example: <code>'int8=&gt;char'</code>
<code>'*source'</code>	Output has the same class as input. Example: <code>'*uint8'</code> For <code>'bitn'</code> or <code>'ubitn'</code> precisions, output has the smallest class that can contain the input. Example: <code>'*ubit18'</code> is equivalent to <code>'ubit18=&gt;uint32'</code>
<code>'N*source'</code> OR <code>'N*source=&gt;output'</code>	Valid if you specify a <code>skip</code> parameter. Read $N$ values before each skip. Example: <code>'4*int8'</code>

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 <i>n</i>	$1 \leq n \leq 64$
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*3	32 (4)
	integer*4	64 (8)
	schar	8 (1)
	signed char	8 (1)
	short	16 (2)
	long	system-dependent
	bit <i>n</i>	$1 \leq n \leq 64$

# fread

---

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 *bitn* or *ubitn*, 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 *bitn* or *ubitn*, 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.164'	Big-endian ordering, 64-bit long data type
'a' or 'ieee-le.164'	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
```

# fread

---

```
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
```

# fread

---

```
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”



**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:

<code>n</code>	Read at most <code>n</code> values into a column vector.
<code>[m,n]</code>	Read at most <code>m</code> -by- <code>n</code> values filling an <code>m</code> -by- <code>n</code> 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.

# fread (serial)

---

## 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	<code>uchar</code>	8-bit unsigned character
	<code>schar</code>	8-bit signed character
	<code>char</code>	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

fgetc | fgets | fopen | fscanf | BytesAvailable | BytesAvailableFcn | InputBufferSize | Status | Terminator | ValuesReceived

# TriRep.freeBoundary

---

**Purpose** Facets referenced by only one simplex

**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:

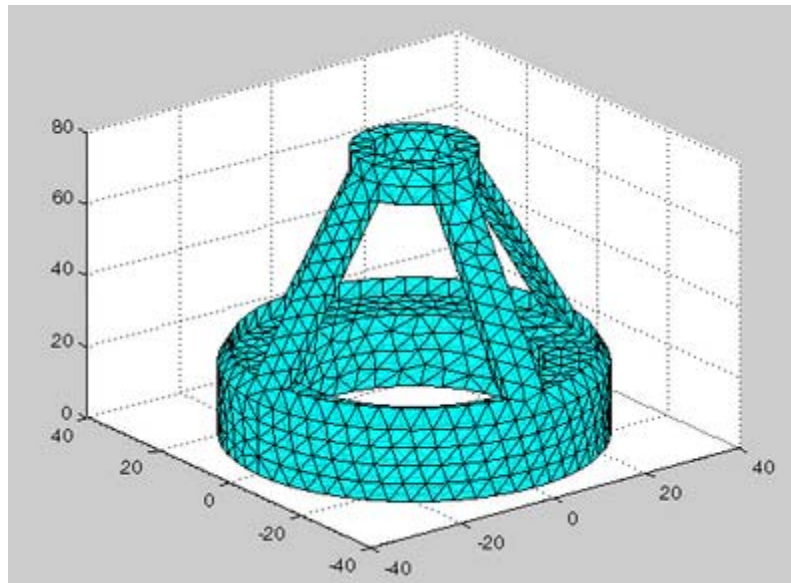
```
load tetmesh;  
trep = TriRep(tet, X);
```

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);
```

# TriRep.freeBoundary

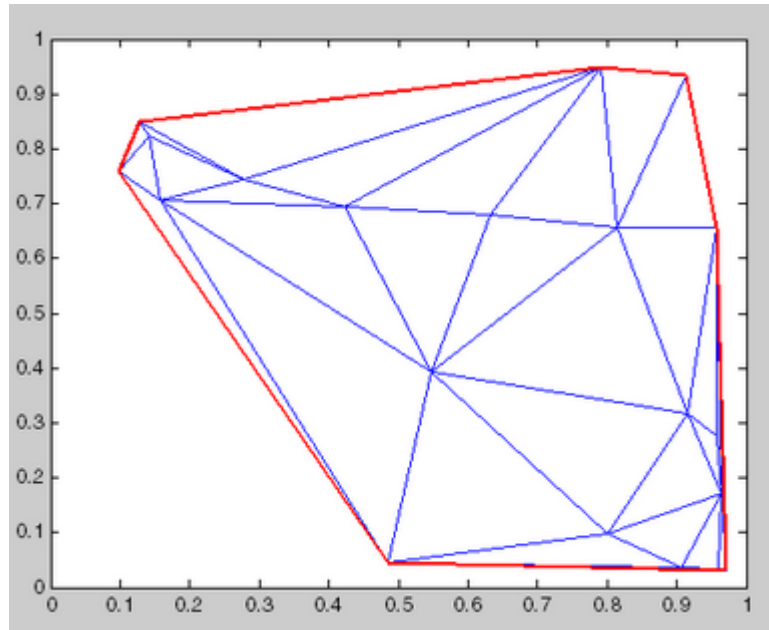
---

```
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

[DelaunayTri](#) | [DelaunayTri.convexHull](#) | [TriRep.featureEdges](#) | [TriRep.faceNormals](#)

**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`



---

<b>Purpose</b>	Read data from text file
<b>Syntax</b>	<pre>A = fscanf(<i>fileID</i>, <i>format</i>) A = fscanf(<i>fileID</i>, <i>format</i>, <i>sizeA</i>) [A, <i>count</i>] = fscanf(...)</pre>
<b>Description</b>	<p><code>A = fscanf(<i>fileID</i>, <i>format</i>)</code> reads and converts data from a text file into array <i>A</i> in column order. To convert, <code>fscanf</code> uses the <i>format</i> and the encoding scheme associated with the file. To set the encoding scheme, use <code>fopen</code>. The <code>fscanf</code> function reapplies the <i>format</i> throughout the entire file, and positions the file pointer at the end-of-file marker. If <code>fscanf</code> cannot match the <i>format</i> to the data, it reads only the portion that matches into <i>A</i> and stops processing.</p> <p><code>A = fscanf(<i>fileID</i>, <i>format</i>, <i>sizeA</i>)</code> reads <i>sizeA</i> elements into <i>A</i>, and positions the file pointer after the last element read. <i>sizeA</i> can be an integer, or can have the form <math>[m,n]</math>.</p> <p><code>[A, <i>count</i>] = fscanf(...)</code> returns the number of elements that <code>fscanf</code> successfully reads.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Format specifiers for the reading functions <code>sscanf</code> and <code>fscanf</code> differ from the formats for the writing functions <code>sprintf</code> and <code>fprintf</code>. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.</li></ul>
<b>Input Arguments</b>	<p><b>fileID</b> Integer file identifier obtained from <code>fopen</code>.</p> <p><b>format</b> String enclosed in single quotation marks that describes each type of element (field). Includes one or more of the following specifiers.</p>

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:

<code>inf</code>	Read to the end of the file. (default)
<code>n</code>	Read at most <i>n</i> elements.
<code>[m,n]</code>	Read at most <i>m*n</i> elements in column order. <i>n</i> can be <code>inf</code> , but <i>m</i> 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'`).

# fscanf

---

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:

<code>n</code>	Read at most <code>n</code> values into a column vector.
<code>[m,n]</code>	Read at most <code>m</code> -by- <code>n</code> values filling an <code>m</code> -by- <code>n</code> 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')
```

## fscanf (serial)

---

```
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`



---

<b>Purpose</b>	Move to specified position in file						
<b>Syntax</b>	<pre>fseek(<i>fileID</i>, <i>offset</i>, <i>origin</i>) status = fseek(<i>fileID</i>, <i>offset</i>, <i>origin</i>)</pre>						
<b>Description</b>	<p><code>fseek(<i>fileID</i>, <i>offset</i>, <i>origin</i>)</code> sets the file position indicator <i>offset</i> bytes from <i>origin</i> in the specified file.</p> <p><code>status = fseek(<i>fileID</i>, <i>offset</i>, <i>origin</i>)</code> returns 0 when the operation is successful. Otherwise, it returns -1.</p>						
<b>Input Arguments</b>	<p><b>fileID</b> Integer file identifier obtained from <code>fopen</code>.</p> <p><b>offset</b> Number of bytes to move from <code>origin</code>. Can be positive, negative, or zero. The <i>n</i> bytes of a given file are in positions 0 through <i>n</i>-1.</p> <p><b>origin</b> Starting location in the file:</p> <table><tr><td>'bof' or -1</td><td>Beginning of file</td></tr><tr><td>'cof' or 0</td><td>Current position in file</td></tr><tr><td>'eof' or 1</td><td>End of file</td></tr></table>	'bof' or -1	Beginning of file	'cof' or 0	Current position in file	'eof' or 1	End of file
'bof' or -1	Beginning of file						
'cof' or 0	Current position in file						
'eof' or 1	End of file						
<b>Examples</b>	<p>Copy 5 bytes from the file <code>test1.dat</code>, starting at the tenth byte, and append to the end of <code>test2.dat</code>:</p> <pre>% Create files test1.dat and test2.dat % Each character uses 8 bits (1 byte)  fid1 = fopen('test1.dat', 'w+'); fwrite(fid1, 'ABCDEFGHJKLMNOPQRSTUVWXYZ');</pre>						

# fseek

---

```
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`

# FTP

---

**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

<code>ascii</code>	Set FTP transfer type to ASCII
<code>binary</code>	Set FTP transfer type to binary
<code>cd</code>	Change or view current folder on FTP server
<code>close</code>	Close connection to FTP server
<code>delete</code>	Remove file on FTP server

---

<code>dir</code>	View contents of folder on FTP server
<code>mget</code>	Download files from FTP server
<code>mkdir</code>	Create folder on FTP server
<code>mput</code>	Upload file or folder to FTP server
<code>rename</code>	Rename file on FTP server
<code>rmdir</code>	Remove folder on FTP server

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals 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')
```

# FTP

---

**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](#)

# func2str

---

**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.

# function

---

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];  
[ave,stdev] = stat2(values)
```

```
ave =  
    47.3400  
stdev =  
    29.4124
```

## See Also

nargin | nargsout | pcode | return | varargin | vararginout |  
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};
```

# function\_handle (@)

---

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 `quad` function to compute its integral from zero to one:

```
quad(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;
```

## function\_handle (@)

---

```
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
<code>function</code>	Function name
<code>type</code>	Function type (e.g., simple, overloaded)
<code>file</code>	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);

finfo = functions(fh)
finfo =
    function: 'get_handles_nested/nestfun'
           type: 'nested'
           file: 'c:\matlab\get_handles_nested.m'
    workspace: [1x1 struct]

finfo.workspace

```

# functions

---

```
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
<code>exp</code>	<code>funm(A, @exp)</code>
<code>log</code>	<code>funm(A, @log)</code>
<code>sin</code>	<code>funm(A, @sin)</code>
<code>cos</code>	<code>funm(A, @cos)</code>
<code>sinh</code>	<code>funm(A, @sinh)</code>
<code>cosh</code>	<code>funm(A, @cosh)</code>

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.

$F = \text{funm}(A, \text{fun}, \text{options})$  sets the algorithm's parameters to the values in the structure `options`.

The following table lists the fields of options.

Field	Description	Values
<code>options.Display</code>	Level of display	'off' (default), 'on', 'verbose'
<code>options.TolBlk</code>	Tolerance for blocking Schur form	Positive scalar. The default is 0.1.
<code>options.TolTay</code>	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is <code>eps</code> .
<code>options.MaxTerms</code>	Maximum number of Taylor series terms	Positive integer. The default is 250.
<code>options.MaxSqrt</code>	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is 100.
<code>options.Ord</code>	Specifies the ordering of the Schur form $T$ .	A vector of length <code>length(A)</code> . <code>options.Ord(i)</code> is the index of the block into which $T(i, i)$ is placed. The default is <code>[]</code> .

$F = \text{funm}(A, \text{fun}, \text{options}, p1, p2, \dots)$  passes extra inputs `p1, p2, ...` to the function.

$[F, \text{exitflag}] = \text{funm}(\dots)$  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>i</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  $I = \text{eye}(\text{size}(X))$ .

## 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`

# fwrite

---

## 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 <i>n</i>	$1 \leq n \leq 64$
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*3	32 (4)
	integer*4	64 (8)
	schar	8 (1)
	signed char	8 (1)
	short	16 (2)
	long	system-dependent
	bit <i>n</i>	$1 \leq n \leq 64$

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 `bitn` or `ubitn`, 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 `bitn` or `ubitn`, 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.164'	Big-endian ordering, 64-bit long data type
'a' or 'ieee-le.164'	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 it is ready to receive data. Check your 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](#).

# fwrite (serial)

---

## 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	<code>uchar</code>	8-bit unsigned character
	<code>schar</code>	8-bit signed character
	<code>char</code>	8-bit signed or unsigned character
Integer	<code>int8</code>	8-bit integer
	<code>int16</code>	16-bit integer
	<code>int32</code>	32-bit integer
	<code>uint8</code>	8-bit unsigned integer
	<code>uint16</code>	16-bit unsigned integer
	<code>uint32</code>	32-bit unsigned integer
	<code>short</code>	16-bit integer
	<code>int</code>	32-bit integer
	<code>long</code>	32- or 64-bit integer
	<code>ushort</code>	16-bit unsigned integer
	<code>uint</code>	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** Find root of continuous function of one variable

**Syntax**

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)
[x,fval] = fzero(...)
[x,fval,exitflag] = fzero(...)
[x,fval,exitflag,output] = fzero(...)
```

**Description** `x = fzero(fun,x0)` tries to find a zero of `fun` near `x0`, if `x0` is a scalar. `fun` is a function handle. The value `x` returned by `fzero` is near a point where `fun` changes sign, or NaN if the search fails. In this case, the search terminates when the search interval is expanded until an Inf, NaN, or complex value is found.

“Parameterizing Functions” explains how to pass additional parameters to your objective function `fun`. See also “Example 2” on page 2-1807 and “Example 3” on page 2-1807 below.

If `x0` is a vector of length two, `fzero` assumes `x0` is an interval where the sign of `fun(x0(1))` differs from the sign of `fun(x0(2))`. An error occurs if this is not true. Calling `fzero` with such an interval guarantees `fzero` will return a value near a point where `fun` changes sign.

`x = fzero(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `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. 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' (the default) displays no error.
OutputFcn	User-defined function that is called at each iteration. See “Output Functions” in MATLAB Mathematics for more information.
PlotFns	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 ( []). <ul style="list-style-type: none"> <li>• @optimplotx plots the current point</li> <li>• @optimplotfval plots the function value</li> </ul> See “Plot Functions” in MATLAB Mathematics for more information.
TolX	Termination tolerance on x

`[x,fval] = fzero(...)` returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fzero(...)` returns a value `exitflag` that describes the exit condition of `fzero`:

- 1           Function converged to a solution `x`.
- 1           Algorithm was terminated by the output function.
- 3           NaN or Inf function value was encountered during search for an interval containing a sign change.

- 4 Complex function value was encountered during search for an interval containing a sign change.
- 5 fzero might have converged to a singular point.
- 6 fzero can not detect a change in sign of the function.

[x,fval,exitflag,output] = fzero(...) returns a structure output that contains information about the optimization in the following fields:

algorithm	'bisection, interpolation'
funcCount	Number of function evaluations
intervaliterations	Number of iterations to find an interval where fun changes sign
iterations	Number of zero-finding iterations
message	Exit message

---

**Note** For the purposes of this command, zeros are considered to be points where the function actually crosses, not just touches, the  $x$ -axis.

---

## Arguments

fun is the function whose zero is to be computed. It 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

```
x = fzero(@myfun,x0);
```

where myfun is a function such as

```
function f = myfun(x)
f = ...           % Compute function value at x
```

or as a function handle for an anonymous function:

```
x = fzero(@(x)sin(x*x),x0);
```

Other arguments are described in the syntax descriptions above.

## Examples

### Example 1

Calculate  $\pi$  by finding the zero of the sine function near 3.

```
x = fzero(@sin,3)
x =
    3.1416
```

### Example 2

To find the zero of cosine between 1 and 2

```
x = fzero(@cos,[1 2])
x =
    1.5708
```

Note that  $\cos(1)$  and  $\cos(2)$  differ in sign.

### Example 3

To find a zero of the function  $f(x) = x^3 - 2x - 5$ , write an anonymous function `f`:

```
f = @(x)x.^3-2*x-5;
```

Then find the zero near 2:

```
z = fzero(f,2)
z =
    2.0946
```

Because this function is a polynomial, the statement `roots([1 0 -2 -5])` finds the same real zero, and a complex conjugate pair of zeros.

```
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

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 = cos(a*x);
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fzero`. To optimize for a specific value of `a`, such as `a = 2`.

**1** Assign the value to `a`.

```
a = 2; % define parameter first
```

**2** Call `fzero` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

```
x = fzero(@(x) myfun(x,a),0.1)
```

## Algorithms

The `fzero` command is a function file. The algorithm, which was originated 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 the `fzero` function file is based, is in [2].

## Limitations

The `fzero` command finds a point where the function changes sign. If the function is *continuous*, this is also a point where the function has a value near zero. If the function is not continuous, `fzero` may return values that are discontinuous points instead of zeros. For example, `fzero(@tan,1)` returns 1.5708, a discontinuous point in `tan`.

Furthermore, the `fzero` command defines a *zero* as a point where the function crosses the  $x$ -axis. Points where the function touches, but does not cross, the  $x$ -axis are not valid zeros. For example,  $y = x.^2$  is a parabola that touches the  $x$ -axis at 0. Because the function never crosses the  $x$ -axis, however, no zero is found. For functions with no valid zeros, `fzero` executes until `Inf`, `NaN`, or a complex value is detected.



**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**

roots | fminbnd | optimset | function\_handle

**How To**

- “Anonymous 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`

# gallery

---

- 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**

$A = \text{gallery}('binomial', n)$  returns an  $n$ -by- $n$  matrix, with integer entries such that  $A^2 = 2^{n-1} \cdot \text{eye}(n)$ .

Thus,  $B = A \cdot 2^{-(n-1)/2}$  is involutory, that is,  $B^2 = \text{eye}(n)$ .

### **cauchy – Cauchy matrix**

$C = \text{gallery}('cauchy', x, y)$  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  $1:y$ .

$C = \text{gallery}('cauchy', x)$  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  $0 < x(1) < \dots < x(n)$  and  $0 < y(1) < \dots < y(n)$ .

### **chebspec – Chebyshev spectral differentiation matrix**

$C = \text{gallery}('chebspec', n, \text{switch})$  returns a Chebyshev spectral differentiation matrix of order  $n$ . Argument  $\text{switch}$  is a variable that determines the character of the output matrix. By default,  $\text{switch} = 0$ .

For  $\text{switch} = 0$  (“no boundary conditions”),  $C$  is nilpotent ( $C^n = 0$ ) and has the null vector  $\text{ones}(n, 1)$ . The matrix  $C$  is similar to a Jordan block of size  $n$  with eigenvalue zero.

For  $\text{switch} = 1$ ,  $C$  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  $A = H(\alpha) + \delta \cdot \text{eye}(n)$ , 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  $4 \cdot \alpha \cdot \cos(k \cdot \pi / (n+2))^2$ , `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  $w = [1 \ t \ t^2 \ \dots \ t^{(n-1)}]$  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  $n-1$ ,  $n-3$ ,  $n-5$ ,  $\dots$ ,  $(1$  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  $y = \text{zeros}(n,1)$ . 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  $\text{diag}(B) - \text{tril}(B,-1) - \text{triu}(B,1)$ , where  $B = \text{abs}(A)$ . `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 $\theta$ )
$k = 4$	$n \geq 4$	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**

$A = \text{gallery}('cycol', [m \ n], k)$  returns an  $m$ -by- $n$  matrix with cyclically repeating columns, where one “cycle” consists of  $\text{randn}(m, k)$ . Thus, the rank of matrix  $A$  cannot exceed  $k$ , and  $k$  must be a scalar.

Argument  $k$  defaults to  $\text{round}(n/4)$ , and need not evenly divide  $n$ .

$A = \text{gallery}('cycol', n, k)$ , where  $n$  is a scalar, is the same as  $\text{gallery}('cycol', [n \ n], k)$ .

### **dorr – Diagonally dominant, ill-conditioned, tridiagonal matrix**

$[c, d, e] = \text{gallery}('dorr', n, \theta)$  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  $\text{gallery}('tridiag', c, d, e)$ .

$A = \text{gallery}('dorr', n, \theta)$  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  $\mu(A) = \text{norm}(\text{inv}(A), 'fro')$  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  $\text{abs}(\det(A)) = 1$ , 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  $\text{abs}(n(i) - n(j))$ . 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  $\text{inv}(A)$  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  $\text{inv}(A)$  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  $\sqrt{\epsilon}$  and 0, respectively.

The characteristic polynomial of  $A$  is given by:

$$\det(A-tI) = (\lambda-t)^N - \alpha(-1)^n.$$

## **frank – Matrix with ill-conditioned eigenvalues**

$F = \text{gallery}('frank', n, k)$  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  $(1, n) \text{ — } (n, 1)$ . 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  $\text{floor}(n/2)$  ill-conditioned eigenvalues — the smaller ones.

## **gcdmat – Greatest common divisor matrix**

$A = \text{gallery}('gcdmat', n)$  returns the  $n$ -by- $n$  matrix with  $(i, j)$  entry  $\text{gcd}(i, j)$ . Matrix  $A$  is symmetric positive definite, and  $A.^r$  is symmetric positive semidefinite for all nonnegative  $r$ .

## **gearmat – Gear matrix**

$A = \text{gallery}('gearmat', n, i, j)$  returns the  $n$ -by- $n$  matrix with ones on the sub- and super-diagonals,  $\text{sign}(i)$  in the  $(1, \text{abs}(i))$  position,  $\text{sign}(j)$  in the  $(n, n+1-\text{abs}(j))$  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  $[\sin(w+a), \sin(w+2a), \dots, \sin(w+n*a)]$ , 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**

$A = \text{gallery}('grcar', n, k)$  returns an  $n$ -by- $n$  Toeplitz matrix with  $-1$ s on the subdiagonal,  $1$ s on the diagonal, and  $k$  superdiagonals of  $1$ s. 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  $d \pm k*i$ , for  $1 \leq k \leq m$ . 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  $x = \alpha*e1$  ( $\alpha \geq 0$ ) 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  $e_1$  is the first column of  $\text{eye}(n, n)$ . Note that if  $x$  is complex, then  $\text{sign}(x) = \exp(i \cdot \arg(x))$  (which equals  $x ./ \text{abs}(x)$  when  $x$  is nonzero).

If  $x = 0$ , then  $v = 0$  and  $\text{beta} = 1$ .

## **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  $1:\text{imax}$ .  $j$  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  $\text{imax}$ , size vector and  $j$  inputs will always return the same array.

In any call to `gallery('integerdata', ...)` you can substitute individual inputs  $m, n, \dots$  for the size vector input  $[m, n, \dots]$ . 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  $\text{imin}:\text{imax}$ .

`[A, B, ...] = gallery('integerdata', [imin imax], [m, n, ...], j)` returns multiple  $m$ -by- $n$ -by-... arrays  $A, B, \dots$ , 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  $\text{ones}(n, 1) * x'$  and whose strict upper triangle agrees with that of  $[1 \ y] * \text{ones}(1, n)$ .

The matrix is nonsingular if  $x(1) \neq 0$  and  $x(i+1) \neq y(i)$  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 ( $A^*A = \text{eye}(n)$ ) and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

$B = (\text{eye}(n) - A)/2$  and  $B = (\text{eye}(n) + A)/2$  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  $A(i,j) = \rho^{(|i-j|)}$ , 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  $L = \text{inv}(\text{gallery('tril',n,-rho,1)})'$ , and  $D(i,i) = (1 - \text{abs}(\rho)^2)^{\text{eye}(n)}$ , except  $D(1,1) = 1$ .
- Positive definite if and only if  $0 < \text{abs}(\rho) < 1$ .
- The inverse  $\text{inv}(A)$  is tridiagonal.

## **krylov – Krylov matrix**

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

$[x, Ax, A^2x, \dots, A^{(j-1)}x]$

where  $A$  is an  $n$ -by- $n$  matrix and  $x$  is a length  $n$  vector. The defaults are  $x = \text{ones}(n,1)$ , 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

$[\text{ones}(1,n); \mu * \text{eye}(n)]$

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  $\text{sqrt}(\text{eps})$ .

## **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  $j \geq i$ .

The Lehmer matrix  $A$  has these properties:

- A is totally nonnegative.
- The inverse  $\text{inv}(A)$  is tridiagonal and explicitly known.
- The order  $n \leq \text{cond}(A) \leq 4*n*n$ .

### **leslie – Matrix of birth numbers and survival rates**

$L = \text{gallery}('leslie', a, b)$  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.,  $0 < b(i) \leq 1$ .

$L = \text{gallery}('leslie', n)$  generates the Leslie matrix with  $a = \text{ones}(n, 1)$ ,  $b = \text{ones}(n-1, 1)$ .

### **lesp – Tridiagonal matrix with real, sensitive eigenvalues**

$A = \text{gallery}('lesp', n)$  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  $D = \text{diag}(1!, 2!, \dots, n!)$ .

### **lotkin – Lotkin matrix**

$A = \text{gallery}('lotkin', n)$  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**

$A = \text{gallery}('minij', n)$  returns the  $n$ -by- $n$  symmetric positive definite matrix with  $A(i, j) = \min(i, j)$ .

The  $\text{minij}$  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  $r=1:n$ .
- $(n+1)*\text{ones}(\text{size}(A))-A$  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  $A, B, \dots$ , 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  $k$ th 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:

$$k = 1 \quad Q(i,j) = \sqrt{2/(n+1)} * \sin(i*j*\pi/(n+1))$$

Symmetric eigenvector matrix for second difference matrix. This is the default.

$$k = 2 \quad Q(i,j) = 2/(\sqrt{2*n+1}) * \sin(2*i*j*\pi/(2*n+1))$$

Symmetric.

$$k = 3 \quad Q(r,s) = \exp(2*\pi*i*(r-1)*(s-1)/n) / \sqrt{n}$$

Unitary, the Fourier matrix.  $Q^4$  is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)!`

$$k = 4 \quad \text{Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is ones}(1:n)/\sqrt{n}.$$

$k = 5$       $Q(i, j) = \sin(2\pi(i-1)(j-1)/n) + \cos(2\pi(i-1)(j-1)/n)$

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 $\pi$

$C = \text{gallery}('parter', n)$  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

$A = \text{gallery}('pei', n, \alpha)$ , where  $\alpha$  is a scalar, returns the symmetric matrix  $\alpha \text{eye}(n) + \text{ones}(n)$ . 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)

$A = \text{gallery}('poisson', n)$  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

$A = \text{gallery}('prolate', n, w)$  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**

$A = \text{gallery}('randcolu', n)$  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  $\text{gallery}('randcorr', n)$ .

$\text{gallery}('randcolu', x)$  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  $n$ .

$\text{gallery}('randcolu', x, m)$  where  $m \geq n$ , produces an  $m$ -by- $n$  matrix.

$\text{gallery}('randcolu', x, m, k)$  provides a further option:

- |         |                                                                                                                                                        |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| $k = 0$ | $\text{diag}(x)$ 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 1s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector  $x$ , where  $\text{length}(x) > 1$ . The vector  $x$  must have nonnegative elements summing to  $\text{length}(x)$ .

`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, k1, ku)` returns a banded (multidiagonal) random matrix of order  $n$  with  $\text{cond}(A) = \text{kappa}$  and singular values from the distribution `mode`. If  $n$  is a two-element vector,  $A$  is  $n(1)$ -by- $n(2)$ .

Arguments `k1` 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 `k1` is present, `ku` defaults to `k1`.

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  $\sqrt{1/\text{eps}}$ . In the special case where  $\text{kappa} < 0$ , A is a random, full, symmetric, positive definite matrix with  $\text{cond}(A) = -\text{kappa}$  and eigenvalues distributed according to mode. Arguments k1 and ku, if present, are ignored.

`A = gallery('randsvd', n, kappa, mode, k1, 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:

- $(n - \text{floor}(\log_2(n))) - 1$  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  $\varepsilon > 0$ .

Barrett and Jarvis conjecture that “the small eigenvalues all lie inside the unit circle  $\text{abs}(Z) = 1$ ,” 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  $\text{abs}(e(i)) \leq m^{-1}/m$ , where  $m = n+1$ .
- $i \leq e(i) \leq i+1$  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  $i \neq j$  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  $\text{ones}(n,1)$ , respectively.

The eigenvalues are ill-conditioned.  $A$  has the property that  $A(i, j) + A(j, i) = 1$  for  $i \neq j$ .

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,

$$T = w(1)*T(\theta(1)) + \dots + w(m)*T(\theta(m))$$

where  $T(\theta(k))$  has  $(i, j)$  element  $\cos(2*\pi*\theta(k)*(i-j))$ .

By default:  $m = n$ ,  $w = \text{rand}(m, 1)$ , and  $\theta = \text{rand}(m, 1)$ .

### **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  $\text{length}(d) - 1$ .

`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

$$\text{cond}(\text{gallery}('triw',n,2)) = \cot(\pi/(4*n))^2,$$

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  $n = 3 \cdot nx \cdot ny + 2 \cdot nx + 2 \cdot ny + 1$ .

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

$$0.25 \leq \text{eig}(\text{inv}(D) \cdot A) \leq 4.5$$

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`.

<code>n = 3</code>	Upper triangular system $Ux=b$ illustrating inaccurate solution.
<code>n = 4</code>	Lower triangular system $Lx=b$ , ill-conditioned.
<code>n = 5</code>	<code>hilb(6)(1:5,2:6)*1.8144</code> . A symmetric positive definite matrix.
<code>n = 21</code>	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  $n$ :

$$\text{gamma}(n+1) = n! = \text{prod}(1:n)$$

**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` | `gammaaln` | `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  $A \geq 0$ , `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  $\text{abs}(X) > A+1$ . 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` | `gamma1n` | `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:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{(a-1)} dt$$

The upper incomplete gamma function is defined as:

$$\text{gammainc}(x,a) = \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{(a-1)} dt$$

`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 | gamma1n | psi

# gammaln

---

**Purpose**            Logarithm of gamma function

**Syntax**            `Y = gammaln(A)`

**Description**        `Y = gammaln(A)` returns the logarithm of the gamma function,  $\text{gammaln}(A) = \log(\text{gamma}(A))$ . The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using  $\log(\text{gamma}(A))$ .

**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

---

<b>Purpose</b>	Handle of figure containing object whose callback is executing
<b>Syntax</b>	<code>fig = gcbf</code>
<b>Description</b>	<p><code>fig = gcbf</code> 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, <code>gcbf</code> returns the figure's handle.</p> <p>When no callback is executing, <code>gcbf</code> returns the empty matrix, <code>[]</code>.</p> <p>The value returned by <code>gcbf</code> is identical to the figure output argument returned by <code>gcbo</code>.</p>
<b>See Also</b>	<code>gcbo</code>   <code>gco</code>   <code>gcf</code>   <code>gca</code>

---

<b>Purpose</b>	Handle of object whose callback is executing
<b>Syntax</b>	<pre>h = gcbo [h,figure] = gcbo</pre>
<b>Description</b>	<p><code>h = gcbo</code> returns the handle of the graphics object whose callback is executing.</p> <p><code>[h,figure] = gcbo</code> returns the handle of the current callback object and the handle of the figure containing this object.</p>
<b>Tips</b>	<p>The MATLAB software stores the handle of the object whose callback is executing in the root <code>CallbackObject</code> property. If a callback interrupts another callback, MATLAB replaces the <code>CallbackObject</code> 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.</p> <p>The root <code>CallbackObject</code> property is read only, so its value is always valid at any time during callback execution. The root <code>CurrentFigure</code> property, and the figure <code>CurrentAxes</code> and <code>CurrentObject</code> properties (returned by <code>gcf</code>, <code>gca</code>, and <code>gco</code>, 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.</p> <p>When you write callback routines for the <code>CreateFcn</code> and <code>DeleteFcn</code> of any object and the figure <code>ResizeFcn</code>, you must use <code>gcbo</code> since those callbacks do not update the root's <code>CurrentFigure</code> property, or the figure's <code>CurrentObject</code> or <code>CurrentAxes</code> properties; they only update the root's <code>CurrentFigure</code> property.</p> <p>When no callbacks are executing, <code>gcbo</code> returns <code>[]</code> (an empty matrix).</p>
<b>See Also</b>	<code>gca</code>   <code>gcf</code>   <code>gco</code>   <code>rootobject</code>

**Purpose** Greatest common divisor

**Syntax**  $G = \text{gcd}(A,B)$   
 $[G,C,D] = \text{gcd}(A,B)$

**Description**  $G = \text{gcd}(A,B)$  returns an array containing the greatest common divisors of the corresponding elements of integer arrays A and B. By convention,  $\text{gcd}(0,0)$  returns a value of 0; all other inputs return positive integers for G.

$[G,C,D] = \text{gcd}(A,B)$  returns both the greatest common divisor array G, and the arrays C and D, which satisfy the equation:  $A(i) \cdot C(i) + B(i) \cdot D(i) = G(i)$ . These are useful for solving Diophantine equations and computing elementary Hermite transformations.

**Examples** The first example involves elementary Hermite transformations.

For any two integers a and b there is a 2-by-2 matrix E with integer entries and determinant = 1 (a *unimodular* matrix) such that:

$$E * [a;b] = [g,0],$$

where g is the greatest common divisor of a and b as returned by the command  $[g,c,d] = \text{gcd}(a,b)$ .

The matrix E equals:

$$\begin{array}{cc} c & d \\ -b/g & a/g \end{array}$$

In the case where  $a = 2$  and  $b = 4$ :

$$\begin{array}{l} [g,c,d] = \text{gcd}(2,4) \\ g = \\ \quad 2 \\ c = \\ \quad 1 \\ d = \\ \quad 0 \end{array}$$

So that

$$E = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$$

In the next example, we solve for  $x$  and  $y$  in the Diophantine equation  $30x + 56y = 8$ .

$$[g, c, d] = \text{gcd}(30, 56)$$

$$g = 2$$

$$c = -13$$

$$d = 7$$

By the definition, for scalars  $c$  and  $d$ :

$$30(-13) + 56(7) = 2,$$

Multiplying through by  $8/2$ :

$$30(-13*4) + 56(7*4) = 8$$

Comparing this to the original equation, a solution can be read by inspection:

$$x = (-13*4) = -52; \quad y = (7*4) = 28$$

## References

[1] Knuth, Donald, *The Art of Computer Programming*, Vol. 2, Addison-Wesley: Reading MA, 1973. Section 4.5.2, Algorithm X.

## See Also

lcm

**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`



---

<b>Purpose</b>	Handle of current object
<b>Syntax</b>	<pre>h = gco h = gco(figure_handle)</pre>
<b>Description</b>	<p><code>h = gco</code> returns the handle of the current object.</p> <p><code>h = gco(figure_handle)</code> returns the value of the current object for the figure specified by <code>figure_handle</code>.</p>
<b>Tips</b>	<p>The current object is the last object clicked on, excluding <code>uimenu</code>. 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 <code>CurrentObject</code> property.</p> <p>The <code>CurrentObject</code> of the <code>CurrentFigure</code> does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the <code>CurrentObject</code> or even the <code>CurrentFigure</code>. Some callbacks, such as <code>CreateFcn</code> and <code>DeleteFcn</code>, and <code>uimenu Callback</code>, intentionally do not update <code>CurrentFigure</code> or <code>CurrentObject</code>.</p> <p><code>gcbo</code> 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.</p>
<b>Examples</b>	<p>This statement returns the handle to the current object in figure window 2:</p> <pre>h = gco(2)</pre>
<b>See Also</b>	<code>gca</code>   <code>gcbo</code>   <code>gcf</code>

**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**  
genpath  
genpath folderName  
p = genpath('folderName')

**Description** genpath returns a path string that includes all the folders and subfolders below *matlabroot*/toolbox, including empty subfolders.

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* or folders that begin with the @ character (class folders) or the + character (package folders).

p = genpath('folderName') returns the path string to variable, p.

**Examples** Generate a path that includes *matlabroot*/toolbox/images and all folders below it:

```
p = genpath(fullfile(matlabroot,'toolbox','images'))
```

```
p =
```

```
C:\Program Files\MATLAB\R20nn\toolbox\images;C:\Program Files\MATLAB\R20nn\toolbox\images\images;C:\Program Files\MATLAB\R20nn\toolbox\images\images\ja;C:\Program Files\MATLAB\R20nn\toolbox\images\imdemos;C:\Program Files\MATLAB\R20nn\toolbox\images\imdemos\ja;
```

R20nn represents the folder for the MATLAB release, for example, R2009b.

Use genpath in conjunction with addpath to add a folder and its subfolders to the search path. Add myfiles and its subfolders to the search path:

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

**See Also** addpath | path | pathsep | pathtool | rehash | restoredefaultpath | rmpath | savepath

**How To**

- “Using 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 2-1856, 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 2-1857, 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 2-1856, 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 2-1855, below.

If the `str` input contains any whitespace characters, `genvarname` removes them 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`

# get

---

**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  $m = \text{length}(H)$  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';
```

# get

---

```
'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](#) | [Handle Graphics Properties](#)

<b>Purpose</b>	Query property values for audioplayer object
<b>Syntax</b>	<pre>Value = get(obj, Name) Values = get(obj, {Name1, ..., NameN}) Values = get(obj) get(obj)</pre>
<b>Description</b>	<p><i>Value</i> = <code>get(obj, Name)</code> returns the value of the specified property for object <i>obj</i>.</p> <p><i>Values</i> = <code>get(obj, {Name1, ..., NameN})</code> returns the values of the specified properties in a 1-by-<i>N</i> cell array.</p> <p><i>Values</i> = <code>get(obj)</code> returns a scalar structure that contains the values of all properties of <i>obj</i>. Each field name corresponds to a property name.</p> <p><code>get(obj)</code> displays all property names and their current values.</p>
<b>Examples</b>	<p>Create an audioplayer object from the example file <code>handel.mat</code> and query the object properties:</p> <pre>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'});</pre>
<b>Alternatives</b>	<p>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 <code>TotalSamples</code> property for an object named <code>handelObj</code> (as created in the Example):</p>

# audioplayer.get

---

```
numSamples = handelObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(handelObj, 'TotalSamples');
```

## **See Also**

[audioplayer | set](#)

<b>Purpose</b>	Query property values for audiorecorder object
<b>Syntax</b>	<pre>Value = get(obj,Name) Values = get(obj,{Name1,...,NameN}) Values = get(obj) get(obj)</pre>
<b>Description</b>	<p><i>Value</i> = <code>get(obj,Name)</code> returns the value of the specified property for object <i>obj</i>.</p> <p><i>Values</i> = <code>get(obj,{Name1,...,NameN})</code> returns the values of the specified properties in a 1-by-<i>N</i> cell array.</p> <p><i>Values</i> = <code>get(obj)</code> returns a scalar structure that contains the values of all properties of <i>obj</i>. Each field name corresponds to a property name.</p> <p><code>get(obj)</code> displays all property names and their current values.</p>
<b>Examples</b>	<p>Create an audiorecorder object and query the object properties:</p> <pre>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'});</pre>
<b>Alternatives</b>	<p>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 <code>TotalSamples</code> property for an object named <code>recorderObj</code> (as created in the Example):</p> <pre>numSamples = recorderObj.TotalSamples;</pre>

# audiorecorder.get

---

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 =
```

## get (COM)

---

```
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 structs.

`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'
```

## get (memmapfile)

---

**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**

# VideoReader.get

---

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)
```

# getabstime (tscollection)

---

```
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, retval]`

**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` are `base` 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.

<b>Purpose</b>	Override to change command window display
<b>Syntax</b>	getdisp(H)
<b>Description</b>	getdisp(H) called by <code>get</code> when <code>get</code> is called with no output arguments and a single input argument that is a handle array. Override this <code>hgsetget</code> class method in a subclass to change how property information is displayed in the command window.
<b>See Also</b>	<code>hgsetget</code>   <code>get</code> ( <code>hgsetget</code> )
<b>How To</b>	<ul style="list-style-type: none"><li>• “Implementing a Set/Get Interface for Properties”</li></ul>

# 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`



<b>Purpose</b>	Field of structure array
<b>Syntax</b>	<pre>value = getfield(struct, 'field') value = getfield(struct, {sIndx1, ..., sIndxM}, 'field', {fIndx1, ..., fIndxN})</pre>
<b>Description</b>	<p><code>value = getfield(struct, 'field')</code>, where <code>struct</code> is a 1-by-1 structure, returns the contents of the specified field, equivalent to <code>value = struct.field</code>. Pass field references as strings.</p> <pre>value = getfield(struct, {sIndx1, ..., sIndxM}, 'field', {fIndx1, ..., fIndxN})</pre> <p>returns the contents of the specified field, equivalent to <code>value = struct(sIndx1, ..., sIndxM).field(fIndx1, ..., fIndxN)</code>. The <code>getfield</code> function supports multiple sets of <code>field</code> and <code>fIndx</code> inputs, and all <code>Indx</code> inputs are optional. If structure <code>struct</code> or any of the fields is a nonscalar structure, and you do not specify an <code>Indx</code>, the <code>getfield</code> function returns the values associated with the first index. If you specify a single colon operator for an <code>fIndx</code> input, enclose it in single quotation marks: <code>' : '</code>.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• For most cases, retrieve data from a structure array by indexing rather than using the <code>getfield</code> function. For more information, see “Access Data in a Structure Array” and “Generate Field Names from Variables”.</li><li>• Call <code>getfield</code> to simplify references to structure arrays with nested fields, or to avoid creating unnecessary temporary variables, as shown in the Examples section.</li></ul>
<b>Examples</b>	<p>The what function returns a structure array that describes the MATLAB files in the current folder. Find the files with the <code>.m</code> extension:</p> <pre>files = getfield(what, 'm');</pre> <p>To perform the same task by indexing requires that you create a temporary variable:</p>

```
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:

<code>Extension</code>	File extension.
<code>Description</code>	Text description of the file format.
<code>ContainsVideo</code>	Whether VideoReader can read video from this format.
<code>ContainsAudio</code>	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:

# VideoReader.getFileFormats

---

```
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.

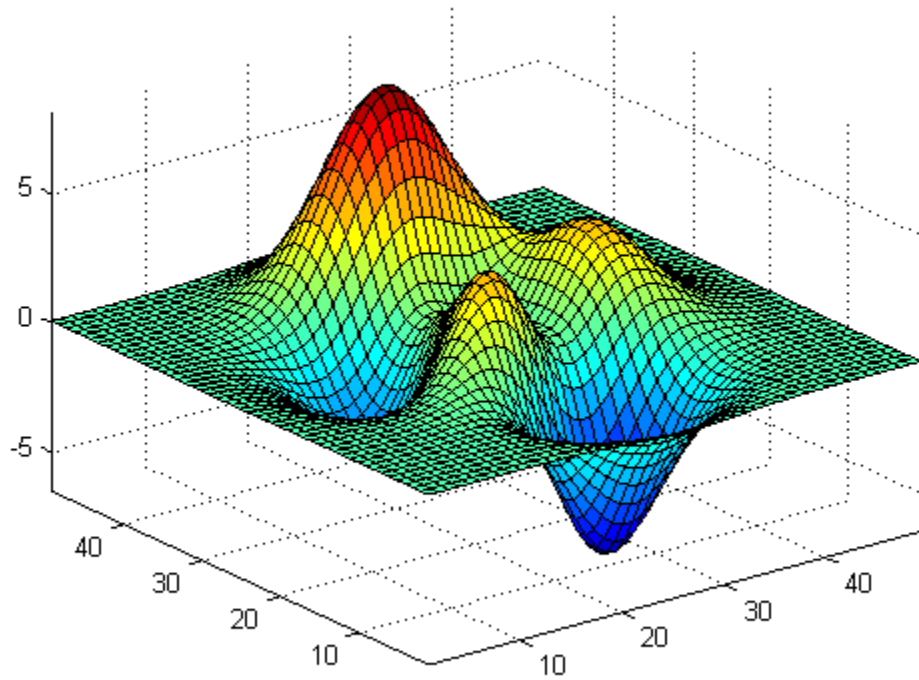
## Examples

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)
```

# GetFullMatrix

---

**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”

**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 (uipanel, or uibuttongroups). Results are not reliable for children of axes or other Handle Graphics objects.

**Examples**

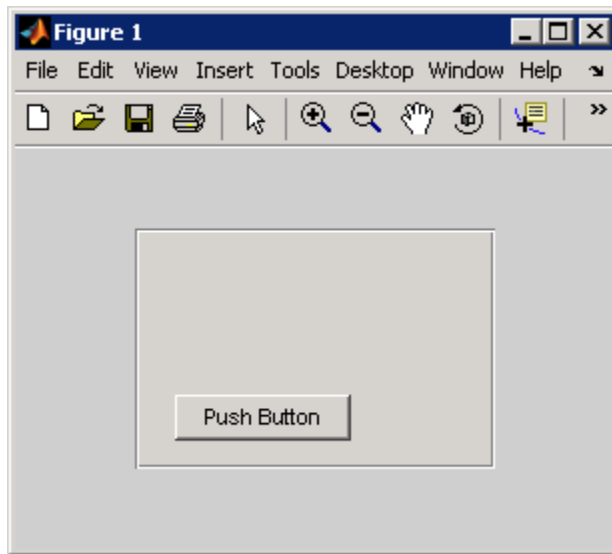
This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the panel.

```
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
```

# getpixelposition

---



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.

# getpref

---

---

**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:

<code>Name</code>	String profile name, such as 'Uncompressed AVI'.
<code>Description</code>	String description of the profile.
<code>FileExtensions</code>	Cell array of strings containing file extensions supported by the file format.
<code>ColorChannels</code>	Number of color channels in each output video frame.
<code>CompressionRatio</code>	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.
<code>FrameRate</code>	Rate of playback for the video in frames per second. Default: 30.

# VideoWriter.getProfiles

---

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 2-1909 , 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 identifier

The extended option displays the error message and error call stack:

```
test_getreport('filethatdoesnotexist.txt', 'extended')
ans =
```

## getReport (MException)

---

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)
```

# getsampleusingtime (tscollection)

---

**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

---

<b>Purpose</b>	Cell array of names of timeseries objects in tscollection object
<b>Syntax</b>	<code>names = gettimeseriesnames(tsc)</code>
<b>Description</b>	<code>names = gettimeseriesnames(tsc)</code> returns names of timeseries objects in a tscollection object <code>tsc</code> . <code>names</code> is a cell array of strings.
<b>Examples</b>	<p><b>1</b> Create timeseries objects <code>a</code> and <code>b</code>.</p> <pre>a = timeseries(rand(1000,1),'name','position'); b = timeseries(rand(1000,1),'name','response');</pre> <p><b>2</b> Create a tscollection object that includes these two time series.</p> <pre>tsc = tscollection({a,b});</pre> <p><b>3</b> Get the names of the timeseries objects in <code>tsc</code>.</p> <pre>names = gettimeseriesnames(tsc)  names =      'position'    'response'</pre>
<b>See Also</b>	<code>timeseries</code>   <code>tscollection</code>

**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 = tttsafterevent(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 = tttsafterevent(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`

**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:

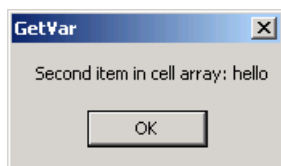
```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create variable C1 in the base workspace of the server
h.PutWorkspaceData('C1', 'base', {25.72, 'hello', rand(4)});
%The client reads the data
C2 = h.GetVariable('C1','base')
```

# GetVariable

---

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.

**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.

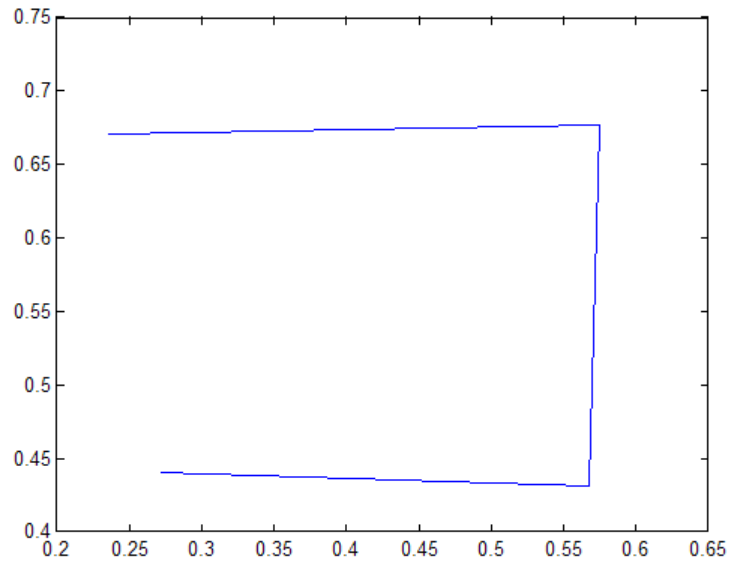
```
[x,y] = ginput(4)
```

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 the file that opens. 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:

```
function global_demo1(num)
    global globalvar

    if nargin > 0
        globalvar = num;
    end

    fprintf('Global variable in function %s is %d\n', ...
           mfilename, globalvar);
```

Create another function `global_demo2` just like it. These two functions have separate function workspaces, but share a common global workspace:

```
copyfile global_demo1 global_demo2
```

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 nargin < 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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $\text{inv}(M)*A*x = \text{inv}(M)*b$  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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where  $0 \leq \text{iter}(1) \leq \text{maxit}$  and  $0 \leq \text{iter}(2) \leq \text{restart}$ .

`[x,flag,relres,iter,resvec] = gmres(A,b,...)` also returns a vector of the residual norms at each inner iteration, including  $\text{norm}(b-A*x0)$ .

## 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,f10,rr0,it0,rv0] = gmres(A,b,[],tol,maxit);
```

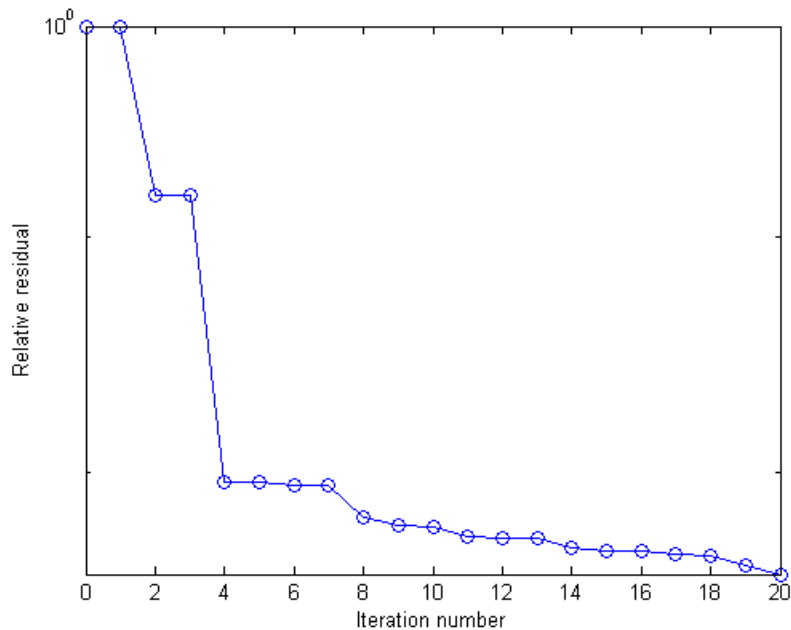


`fl0` is 1 because `gmres` does not converge to the requested tolerance  $1e-12$  within the requested 20 iterations. The best approximate solution that `gmres` returns is the last one (as indicated by `it0(2) = 20`). MATLAB stores the residual history in `rv0`.

#### 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:

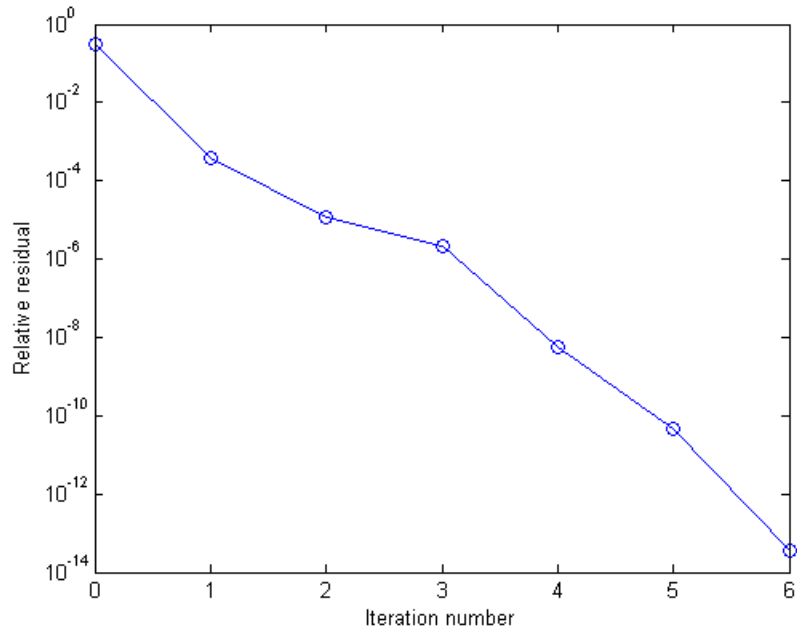
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));  
[x1,f11,rr1,it1,rv1] = gmres(A,b,[],tol,maxit,L,U);
```

`f11` is 0 because `gmres` drives the relative residual to  $1.1196e-013$  (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(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

## 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));
```

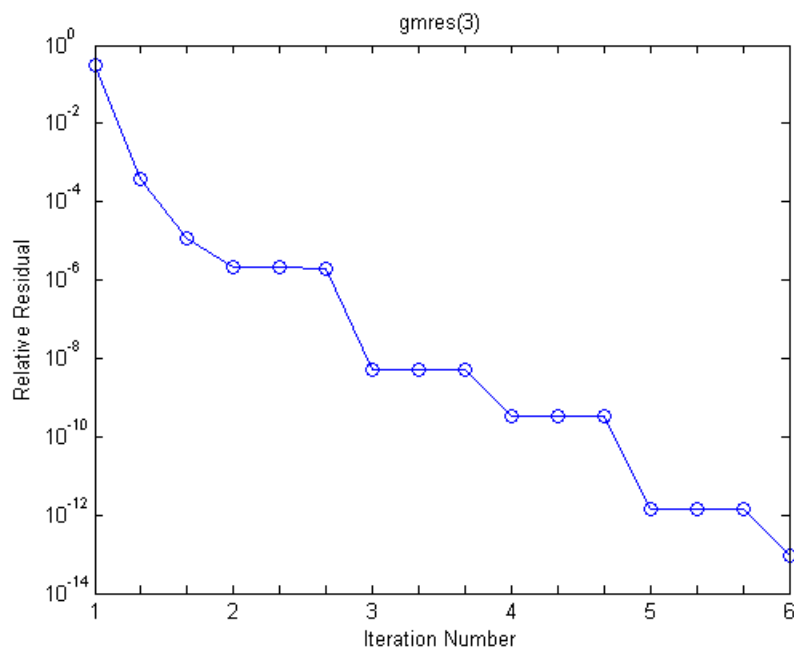
The benefit to using restarted gmres is to limit the amount of memory required to execute the method. Without restart, gmres requires `maxit` vectors of storage to keep the basis of the Krylov subspace. Also, gmres must orthogonalize against all of the previous vectors at each step. Restarting limits 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)`:

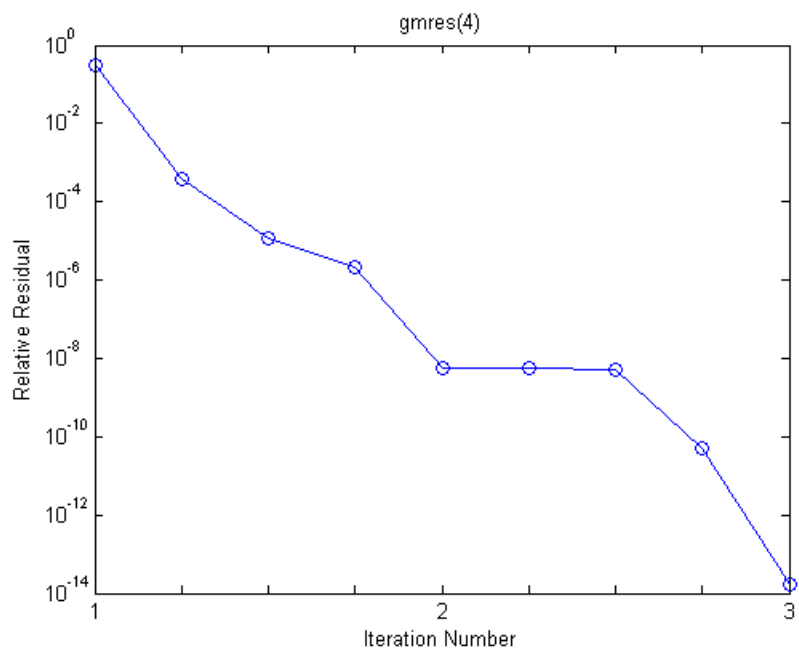
```
tol = 1e-12; maxit = 20;
re3 = 3;
[x3,f13,rr3,it3,rv3] = gmres(A,b,re3,tol,maxit,L,U);
re4 = 4;
[x4,f14,rr4,it4,rv4] = gmres(A,b,re4,tol,maxit,L,U);
re5 = 5;
[x5,f15,rr5,it5,rv5] = gmres(A,b,re5,tol,maxit,L,U);
```

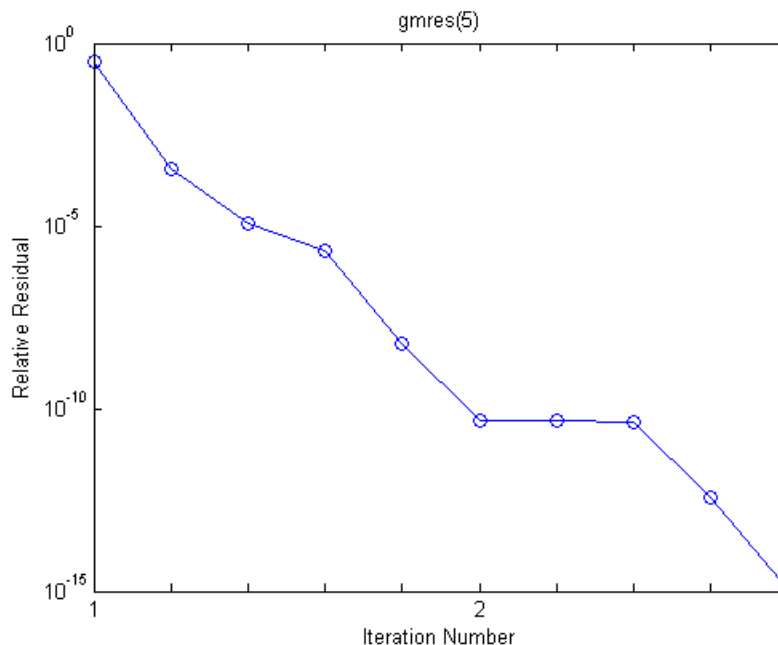
`f13`, `f14`, and `f15` are all 0 because in each case restarted gmres drives the relative residual to less than the prescribed tolerance of  $1e-12$ . The following plots show the convergence histories of each restarted gmres method. `gmres(3)` converges at outer iteration 5, inner iteration 3 (`it3 = [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

---





## 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](#)

# gplot

---

## Purpose

Plot nodes and links representing adjacency matrix

## Syntax

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineStyle)
```

## 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,LineStyle)` plots the nodes using the line type, marker symbol, and color specified by `LineStyle`.

## Tips

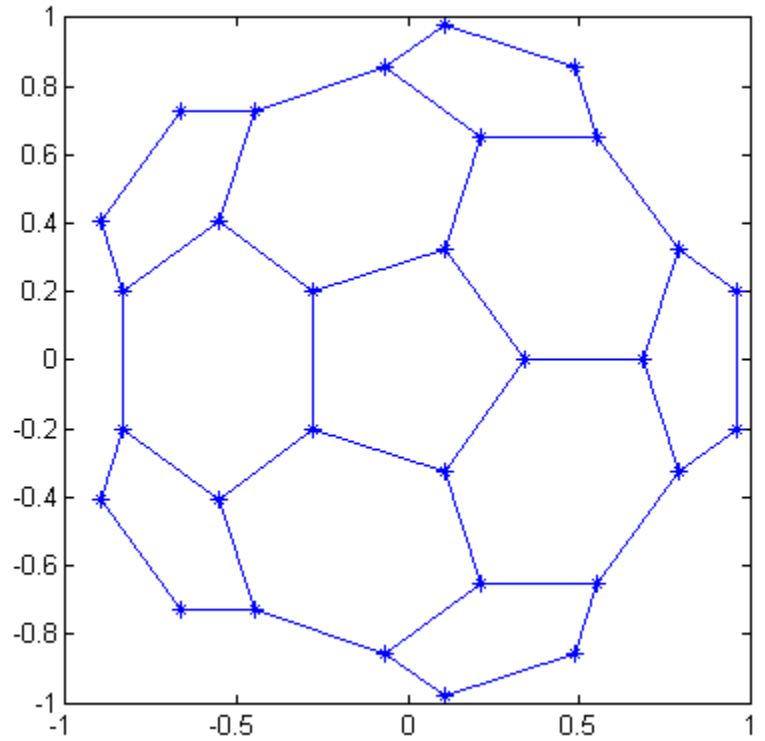
For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node  $i$ , and `Coordinates(j,:) = [x(j) y(j)]` denotes node  $j$ . If node  $i$  and node  $j$  are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

## 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](#)

# grabcode

---

**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
```

# gradient

---

**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,  $F(x,y,z, \dots)$ ,

$$\nabla F = \frac{\partial F}{\partial x} \hat{i} + \frac{\partial F}{\partial y} \hat{j} + \frac{\partial F}{\partial z} \hat{k} + \dots$$

**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 ( $h_1, h_2, \dots$ ) 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.

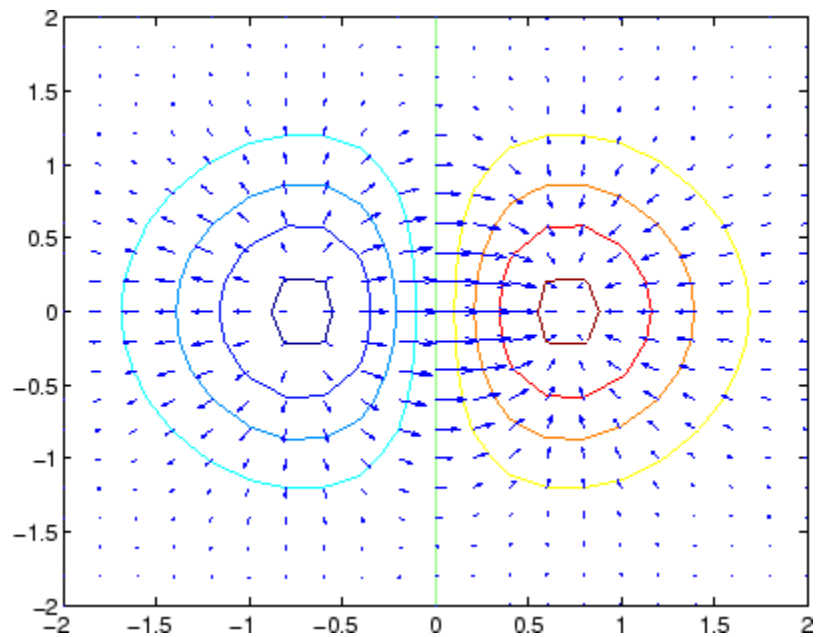
## Examples

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

# gradient



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](#)

**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`

# grid

---

**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')
```



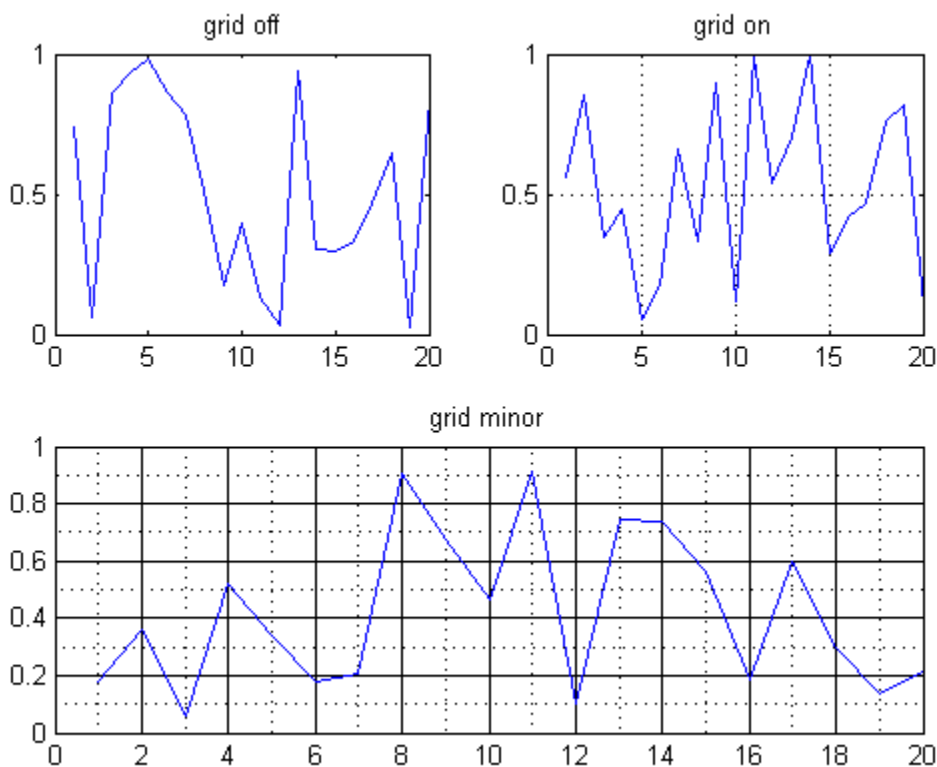
To customize the locations of ticks along an axis, set the axes `XTick`, `YTick`, and `ZTick` properties. For minor ticks, use the `XMinorTick`, `YMinorTick`, and `ZTick` properties.

## 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')
```


# grid



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

**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);
```

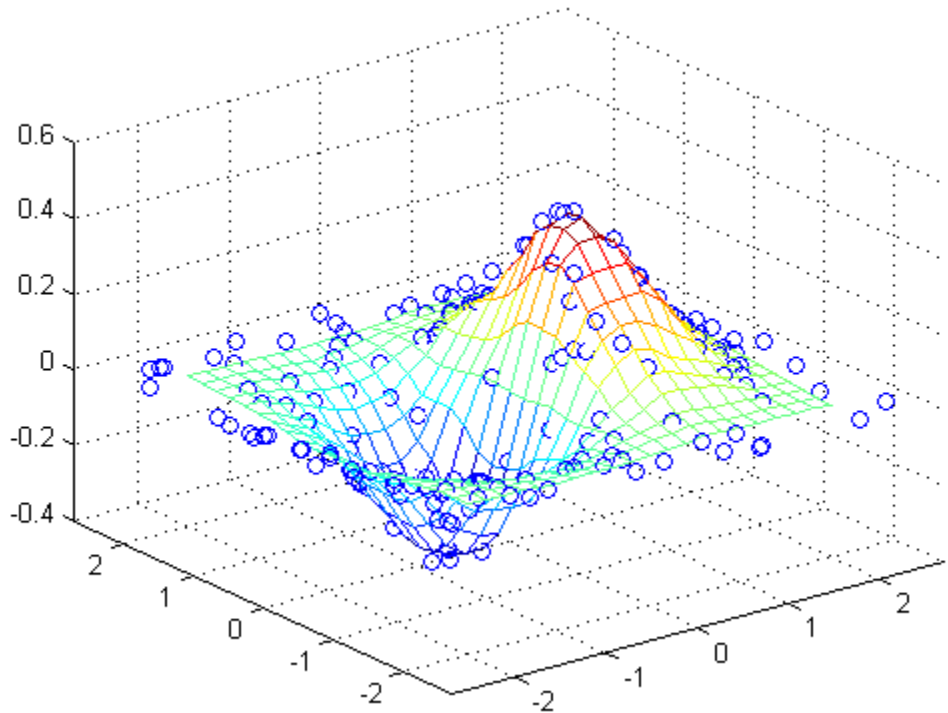
$x$ ,  $y$ , and  $v$  are vectors containing scattered (nonuniform) sample points and data.

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].

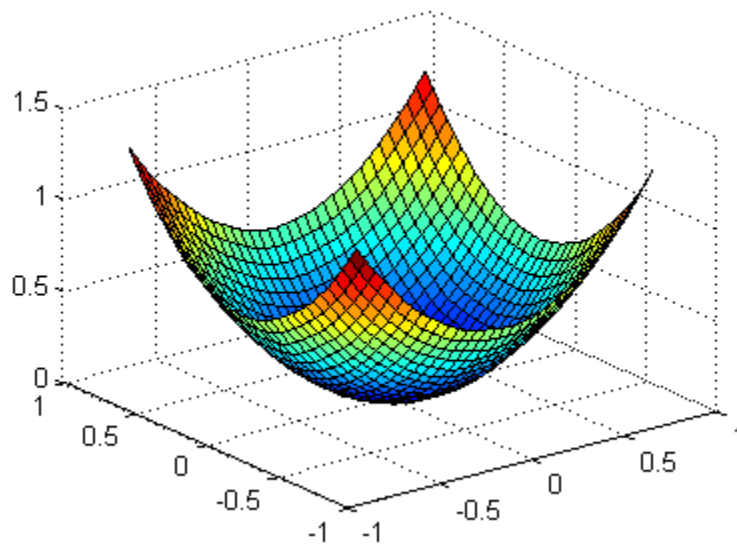
```
d = -0.8:0.05:0.8;
[xq,yq,zq] = meshgrid(d,d,0);
```

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]);
```

# griddata

---



## See Also

`TriScatteredInterp` | `delaunay` | `griddatan` | `interp` | `interp` | `meshgrid` | `ndgrid`



**Purpose** Data gridding and hypersurface fitting for 3-D data

---

**Note** `griddata3` has been removed. Use `griddata` instead.

---

**Syntax**

```
w = griddata3(x,y,z,v,xi,yi,zi)
w = griddata3(x,y,z,v,xi,yi,zi,method)
```

**Description** `w = griddata3(x,y,z,v,xi,yi,zi)` fits a hypersurface of the form  $w = f(x,y,z)$  to the data in the (usually) nonuniformly spaced vectors  $(x,y,z,v)$ . `griddata3` interpolates this hypersurface at the points specified by  $(xi,yi,zi)$  to produce  $w$ .  $w$  is the same size as  $xi$ ,  $yi$ , and  $zi$ .  $(xi,yi,zi)$  is usually a uniform grid (as produced by `meshgrid`) and is where `griddata3` gets its name.

`w = griddata3(x,y,z,v,xi,yi,zi,method)` defines the type of surface that is fit to the data, where `method` is either:

'linear'	Tessellation-based linear interpolation (default)
'nearest'	Nearest neighbor interpolation

If `method` is `[]`, the default 'linear' method is used.

`w = griddata3(...,options)` specifies a cell array of strings `options` that were previously in `Qhull`. `Qhull`-specific options are no longer required and are currently ignored.

**Examples** Create vectors `x`, `y`, and `z` containing nonuniformly sampled data:

```
x = gallery('uniformdata',[5000 1],0)-1;
y = gallery('uniformdata',[5000 1],1)-1;
z = gallery('uniformdata',[5000 1],2)-1;
v = x.^2 + y.^2 + z.^2;
```

Define a regular grid, and grid the data to it:

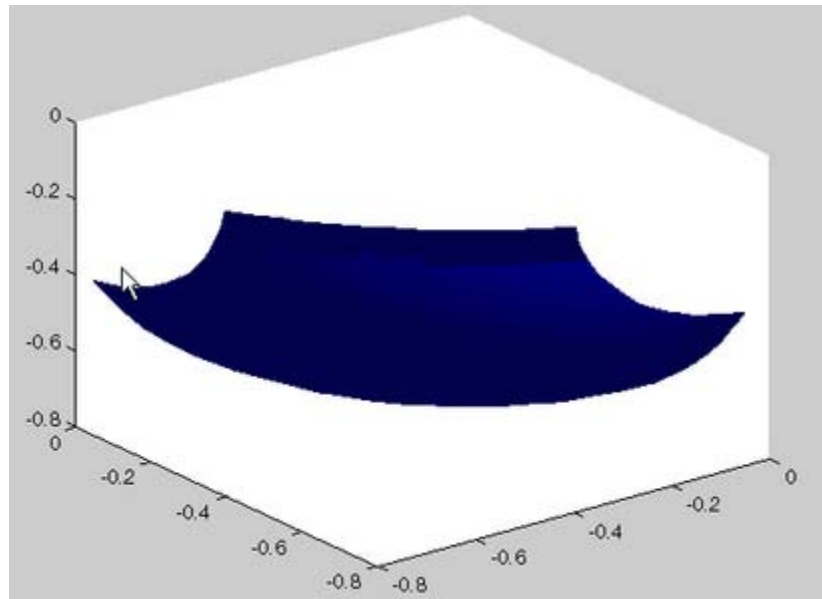
## griddata3

---

```
d = -0.8:0.05:0.8;  
[xi,yi,zi] = meshgrid(d,d,d);  
w = griddata3(x,y,z,v,xi,yi,zi);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8:

```
p = patch(isosurface(xi,yi,zi,w,0.8));  
isonormals(xi,yi,zi,w,p);  
set(p,'FaceColor','blue','EdgeColor','none');  
view(3), axis equal, axis off, camlight, lighting phong
```



**Purpose**

Data gridding and hypersurface fitting (dimension  $\geq 2$ )

**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;
```

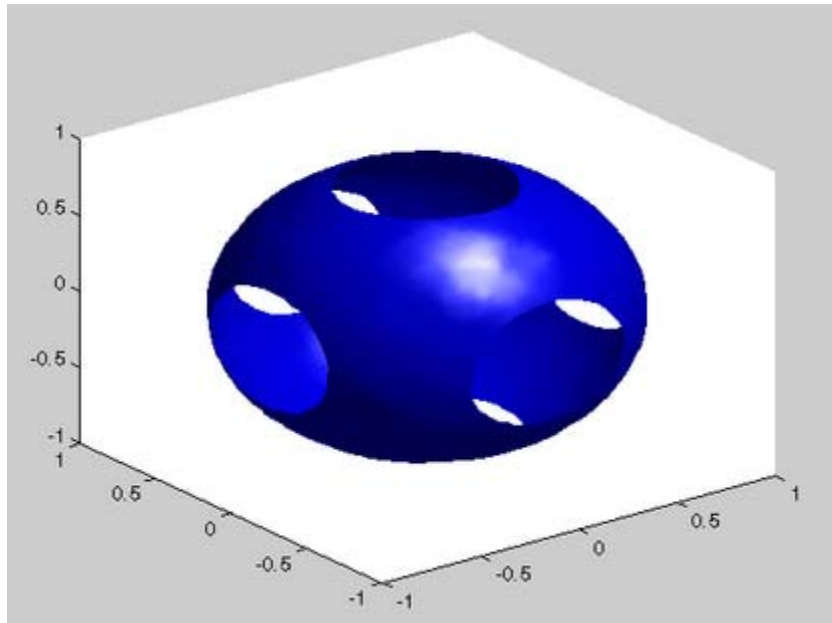
# griddatan

---

```
[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

[delauayn](#) | [griddata](#) | [griddata3](#) | [meshgrid](#)

<b>Purpose</b>	Interpolant for gridded data
<b>Description</b>	The <code>griddedInterpolant</code> class constructs an interpolant $F$ from a gridded data set. The <i>interpolant</i> $F$ is an interpolating function that can be evaluated at query locations.
<b>Construction</b>	<p><code>F = griddedInterpolant(x,v)</code> takes vectors <math>x</math> and <math>v</math> of equal length and creates an interpolant <math>F</math>. The first input argument, <math>x</math>, is a vector of sample points. <math>v</math> is a vector of corresponding values such that <math>v = f(x)</math>. The interpolant <math>F</math> interpolates a vector of query locations <math>xq</math> using <math>vq = F(xq)</math> to return the interpolated values <math>vq</math>.</p> <p><code>F = griddedInterpolant(X1,X2,...,Xn,V)</code> specifies a grid as <math>n</math> <math>n</math>-dimensional arrays <math>Xi</math>. The grid <math>(X1, X2, \dots, Xn)</math> is created from grid vectors <math>xgi</math> by <math>[X1, X2, \dots, Xn] = \text{ndgrid}(xg1, xg2, \dots, xgn)</math>.</p> <p><code>F = griddedInterpolant(V)</code> creates an interpolant <math>F</math> through the <math>n</math>-dimensional array of values <math>V</math> using the default grid formed by grid vectors <math>xgi = 1:\text{size}(V, i)</math>.</p> <p><code>F = griddedInterpolant({xg1,xg2,...,xgn},V)</code> specifies the grid in compact form in terms of grid vectors <math>xgi</math>, where <math>\text{length}(xgi)</math> is <math>\text{size}(V, i)</math>.</p> <p><code>F = griddedInterpolant(...,Method)</code> specifies the interpolation method. The values for 'Method' are nearest, linear (default), spline, pchip, or cubic.</p> <p>The interpolant <math>F</math> can be evaluated on scattered or gridded query locations.</p>

## Input Arguments

**x**

Vector of input coordinates the same size as  $v$ .

**v**

Vector of input values the same size as  $x$ .

**X1, X2, ..., Xn**

# griddedInterpolant

---

n n-dimensional arrays  $X_i$  that specify a grid. `size(Xi)` is `size(V)`.

## **V**

n-dimensional array of values associated with each grid point.

## **{xg1,xg2,...,xgn}**

Cell array of grid vectors  $x_{gi}$  that specify the grid in compact form.

## **Method**

Interpolation method. Possible values are:

`nearest`

Nearest neighbor interpolation.

`linear` (default)

Linear, bilinear, trilinear,... interpolation.

`spline`

Spline interpolation.

`pchip`

Shape-preserving piecewise cubic interpolation (1-D only).

`cubic`

Cubic, bicubic, tricubic,... For uniformly spaced data only.

## **Properties**

### **GridVectors**

Cell array containing grid vectors.

### **Values**

Array of values associated with each grid point.

### **Method**

Method used to interpolate data. Possible values are the same as for the `Method` input: `nearest`, `linear` (default), `spline`, `pchip`, or `cubic`.

## Evaluation

The interpolant  $F$  can be evaluated on scattered or gridded query locations as follows:

- $Vq = F(Xq)$  evaluates  $F$  at the query locations  $Xq$  to produce the corresponding values  $Vq$ . The  $m$ -by- $n$  matrix  $Xq$  represents  $m$  points scattered in  $n$ -dimensional space.  $Vq$  is  $m$ -by-1.
- $Vq = F(xq1, xq2, \dots, xqn)$  specifies the query locations  $xqi$  as column vectors of length  $m$  representing  $m$  points scattered in  $n$ -dimensional space.  $Vq$  is  $m$ -by-1.
- $Vq = F(Xq1, Xq2, \dots, Xqn)$  specifies the query locations as  $n$   $n$ -dimensional arrays  $Xqi$  of equal size specifying the full grid.  $Vq$  is the same size as  $Xqi$ .
- $Vq = F(\{xgq1, xgq2, \dots, xgqn\})$  specifies the query locations as a grid formed by the grid vectors  $xgqi$ .  $Vq$  is the size of grid defined by  $xgqi$ ,  $\text{length}(xgq1)$ -by- $\text{length}(xgq2)$ -by-...-by- $\text{length}(xgqn)$ .

## Definitions

A *gridded data set* is an axis-aligned, ordered set of points.

A *scattered data set* consists of a set of points where the points have no structure among their relative locations.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals 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

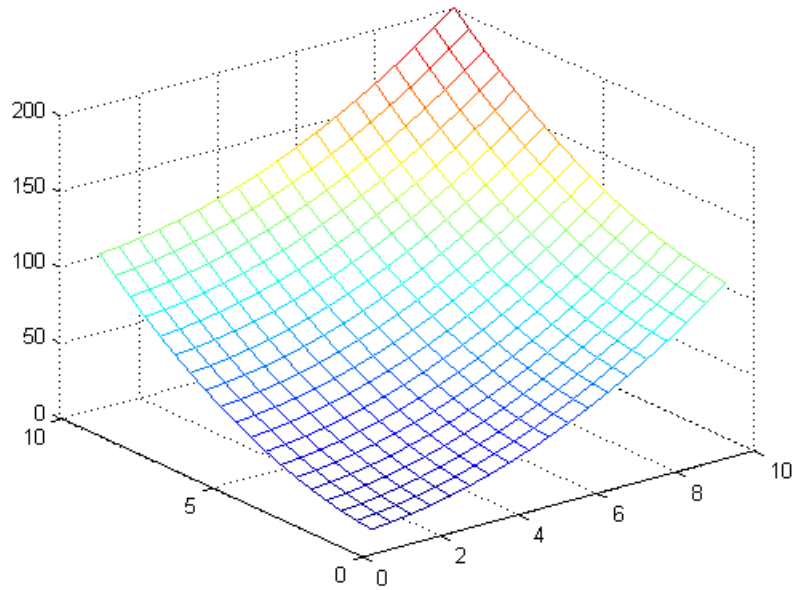
Interpolate to refine a grid-based sampling:

```
[X1, X2] = ndgrid(1:10,1:10);  
V = X1.^2 + X2.^2;  
F = griddedInterpolant(X1,X2,V, 'cubic');  
[Xq1, Xq2] = ndgrid(1:0.5:10,1:0.5:10);
```

# griddedInterpolant

---

```
Vq = F(Xq1,Xq2);  
mesh(Xq1,Xq2,Vq);
```



## See Also

[TriScatteredInterp](#) | [interp1](#) | [interp2](#) | [interp3](#) | [interpN](#) | [ndgrid](#) | [meshgrid](#)

## How To

- [Class Attributes](#)
- [Property Attributes](#)
- [“Interpolating Gridded Data”](#)



**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

$$\begin{aligned} A &= U * C * X' \\ B &= V * S * X' \\ C' * C + S' * S &= I \end{aligned}$$

$A$  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,  $\sqrt{\text{diag}(C' * C) ./ \text{diag}(S' * S)}$ .

The nonzero elements of  $S$  are always on its main diagonal. If  $m \geq p$  the nonzero elements of  $C$  are also on its main diagonal. But if  $m < p$ , the nonzero diagonal of  $C$  is  $\text{diag}(C, p-m)$ . 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  $n \geq p$ , 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  $\text{diag}(C) ./ \text{diag}(S)$ .

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  $\text{null}(A)$  and  $\text{null}(B)$  do not overlap and replace  $X$  by  $\text{inv}(X)$  or  $\text{inv}(X')$ .

Note, however, that when  $\text{null}(A)$  and  $\text{null}(B)$  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
--------	---	---

$$S = \begin{bmatrix} 0 & 0.3155 & 0 \\ 0 & 0 & 0.9807 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1.0000 & 0 & 0 \\ 0 & 0.9489 & 0 \\ 0 & 0 & 0.1957 \end{bmatrix}$$

Since A is rank deficient, the first diagonal element of C is zero.

The economy sized decomposition,

$$[U, V, X, C, S] = \text{gsvd}(A, B, 0)$$

produces a 5-by-3 matrix U and a 3-by-3 matrix C.

$$U = \begin{bmatrix} 0.5700 & -0.6457 & -0.4279 \\ -0.7455 & -0.3296 & -0.4375 \\ -0.1702 & -0.0135 & -0.4470 \\ 0.2966 & 0.3026 & -0.4566 \\ 0.0490 & 0.6187 & -0.4661 \end{bmatrix}$$

$$C = \begin{bmatrix} 0.0000 & 0 & 0 \\ 0 & 0.3155 & 0 \\ 0 & 0 & 0.9807 \end{bmatrix}$$

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.

$$\begin{aligned} \text{sigma} &= \text{gsvd}(A, B) \\ \text{sigma} &= \begin{bmatrix} 0.0000 \\ 0.3325 \end{bmatrix} \end{aligned}$$

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
```

# gt

---

1	0	1	1	0	0
0	1	1	1	0	1

## See Also

lt | ge | le | ne | eq

## How To

- “Relational Operators”



**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](#)

# guidata

---

**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 `gcb0`), 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](#)

# guide

---

**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”

**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`

# gunzip

---

**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`

# gzip

---

**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`

<b>Purpose</b>	Create HDF5 data set
<b>Syntax</b>	<code>h5create(filename,datasetname,size,Name,Value)</code>
<b>Description</b>	<code>h5create(filename,datasetname,size,Name,Value)</code> creates an HDF5 data set in the file specified by <code>filename</code> .
<b>Input Arguments</b>	<p><b>filename</b></p> <p>Text string specifying the name of an HDF5 file. If <code>filename</code> does not already exist, <code>h5create</code> creates it, with additional options specified by one or more <code>Name,Value</code> pair arguments.</p> <p><b>datasetname</b></p> <p>Text string specifying the name of the data set you want to create. If <code>datasetname</code> is a full path name, <code>h5create</code> creates all intermediate groups, if they don't already exist.</p> <p><b>size</b></p> <p>Array specifying the extents of the dataset. To specify an unlimited extent, set the corresponding element of <code>size</code> to <code>Inf</code>.</p>
<b>Name-Value Pair Arguments</b>	<p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>Datatype</b></p> <p>Any of the following MATLAB datatypes.</p>

double	uint64	uint32	uint16	uint8
single	int64	int32	int16	int8

# h5create

---

**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

## 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”

# h5info

---

**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 '/ '.



<b>Location Type</b>	<b>Field</b>	<b>Description</b>
	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		

# h5info

---

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”

**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**

# h5read

---

Numeric value specifying the interval 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”

<b>Purpose</b>	Read attribute from HDF5 file
<b>Syntax</b>	<code>attval = h5readatt(filename,location,attr)</code>
<b>Description</b>	<code>attval = h5readatt(filename,location,attr)</code> retrieves the value for the named attribute <code>attr</code> from the given location in the HDF5 file <code>filename</code> .
<b>Input Arguments</b>	<p><b>filename</b> Text string specifying the name of an HDF5 file.</p> <p><b>location</b> Text string specifying the full path of the attribute in an HDF5 file. <code>location</code> can refer to either a group or a data set.</p> <p><b>attr</b> Text string specifying the name of an attribute in an HDF5 file.</p>
<b>Output Arguments</b>	<p><b>attval</b> Value of the attribute.</p>
<b>Examples</b>	<p>Read a group attribute.</p> <pre>attval = h5readatt('example.h5','/', 'attr2');</pre> <hr/> <p>Read a data set attribute.</p> <pre>attval = h5readatt('example.h5','/g4/lon', 'units');</pre>
<b>See Also</b>	<code>h5writeatt</code>   <code>h5info</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>“Using the High-Level HDF5 Functions to Import Data”</li></ul>

# h5write

---

## 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
```

# h5write

---

```
h5disp('myfile.h5');
```

## See Also

[h5read](#) | [h5create](#) | [h5writeatt](#) | [h5disp](#)

## Tutorials

- “Exporting to Hierarchical Data Format (HDF5) Files”



<b>Purpose</b>	Write HDF5 attribute
<b>Syntax</b>	<code>h5writeatt(filename,location,attname,attvalue)</code>
<b>Description</b>	<code>h5writeatt(filename,location,attname,attvalue)</code> writes the attribute named <code>attname</code> with the value <code>attvalue</code> to the HDF5 file <code>filename</code> . The parent object <code>location</code> can be either a group or variable. <code>location</code> is the complete path name of the group or variable to which you want to associate the attribute.
<b>Input Arguments</b>	<p><b>filename</b> Text string specifying the name of an HDF5 file.</p> <p><b>location</b> Text string specifying the full path identifying a group or variable in an HDF5 file.</p> <p><b>attname</b> Text string specifying the name of an attribute in an HDF5 file. If the attribute does not exist, <code>h5writeatt</code> creates the attribute with the name specified.  If the specified attribute already exists but does not have a datatype or dataspace consistent with <code>attvalue</code>, <code>h5writeatt</code> deletes the attribute and recreates it. String attributes are created with a scalar dataspace.</p> <p><b>attvalue</b> Value to be written to the attribute in an HDF5 file.</p>
<b>Examples</b>	Create a root group attribute whose value is the current time. <pre>srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5'); copyfile(srcFile,'myfile.h5'); fileattrib('myfile.h5','+w'); h5writeatt('myfile.h5','/', 'creation_date',datestr(now));</pre>

# h5writeatt

---

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”

**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`

# handle

---

**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
<code>addlistener</code>	Creates a listener for the specified event and assigns a callback function to execute when the event occurs.
<code>notify</code>	Broadcast a notice that a specific event is occurring on a specified handle object or array of handle objects.
<code>delete</code>	Handle object destructor method that is called when the object's lifecycle ends.
<code>findobj</code>	Finds objects matching the specified conditions from the input array of handle objects.
<code>findprop</code>	Returns a <code>meta.property</code> objects associated with the specified property name.
<code>isvalid</code>	Returns a logical array in which elements are true if the corresponding elements in the input array are valid handles. This method is <b>Sealed</b> so you cannot override it in a handle subclass.
<code>eq</code> <code>ne</code> <code>lt</code> <code>le</code> <code>gt</code> <code>ge</code> <code>relationaloperator</code>	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`

# handle

---

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.

**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** For information about MATLAB HDF5 capabilities, which is a completely separate, incompatible format, see `hdf5`.

---

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.

<b>Application Programming Interface</b>	<b>Description</b>	<b>MATLAB Function</b>
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	<code>hdfan</code>
External Files	Manipulates external file elements.	<code>hdfhx</code>



<b>Application Programming Interface</b>	<b>Description</b>	<b>MATLAB Function</b>
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes.  Note: Use the MATLAB functions <code>imread</code> and <code>imwrite</code> with HDF raster image formats.	<code>hdfdf24</code> , <code>hdfdfr8</code>
HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	<code>matlab.io.hdfeos.gd</code> , <code>hdfpt</code> , <code>matlab.io.hdfeos.sw</code>
HDF Utilities	Provides functions to open and close HDF files and handle errors.	<code>hdfh</code> , <code>hdfhd</code> , <code>hdfhe</code>
MATLAB HDF Utilities	Provides utility functions that help you work with HDF files in the MATLAB environment.	<code>hdfml</code>
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	<code>matlab.io.hdf4.sd</code>
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	<code>hdfv</code>
V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	<code>hdfvf</code> , <code>hdfvh</code> , <code>hdfvs</code>

For copyright information, see the `hdf4copyright.txt` file.

# hdf

---

hdfinfo | hdfread | hdf5tool | hdf5

**Purpose**

Summary of MATLAB HDF5 capabilities

**Description**

MATLAB provides both high- and low-level access to Hierarchical Data Format (HDF5) files.

- High-level access functions make it easy to read a data set from an HDF5 file or write a variable from the MATLAB workspace into an HDF5 file.
- Low-level access functions provide direct access to the more than 300 functions in the HDF5 library.

---

**Note** For information about MATLAB HDF4 capabilities, which is a completely separate and incompatible format, see `hdf`.

---

**High-level Access Functions**

These functions provide high-level access to HDF5 files:

- `h5create` — Create HDF5 data set
- `h5disp` — Display contents of HDF5 file
- `h5info` — Return information about HDF5 file
- `h5read` — Read data from HDF5 data set
- `h5readatt` — Read value of attribute from HDF5 group or data set
- `h5write` — Write data to HDF5 data set
- `h5writeatt` — Write data to HDF5 attribute

---

**Note** To create or write to nonnumeric datasets or attributes, you must use the low-level interface.

---

## Low-level Access Functions

MATLAB provides direct access to dozens of functions in the HDF5 library. Using these functions, you can read and write complex datatypes, use HDF5 data subsetting capabilities, and take advantage of other features present in the HDF5 library. To use these MATLAB functions, you must be familiar with HDF5 programming concepts and, when using the low-level functions, details about the functions in the library. To get this information, see <http://www.hdfgroup.org>.

The HDF5 library organizes the routines in the library into collections called *interfaces*. MATLAB organizes the corresponding MATLAB functions into packages that match these HDF5 library interfaces. For example, the MATLAB functions for the HDF5 Attribute Interface are in the H5A package.

The following table lists the HDF5 library interfaces alphabetically with their associated MATLAB package.

<b>HDF5 Library Interface</b>	<b>MATLAB Package Name</b>	<b>Description</b>
Library	H5	General-purpose functions for use with the entire HDF5 library, such as initialization
Attribute	H5A	Manipulate metadata associated with data sets or groups
Dataset	H5D	Manipulate multidimensional arrays of data elements, together with supporting metadata
Dimension Scale	H5DS	Manipulate dimension scale associated with dataset dimensions
Error	H5E	Handle errors
File	H5F	Access files

<b>HDF5 Library Interface</b>	<b>MATLAB Package Name</b>	<b>Description</b>
Group	H5G	Organize objects in a file; analogous to a folder structure
Identifier	H5I	Manipulate HDF5 object identifiers
Link	H5L	Manipulate links in a file
MATLAB	H5ML	MATLAB utility functions that are not part of the HDF5 library itself
Object	H5O	Manipulate objects in a file
Property	H5P	Manipulate object property lists
Reference	H5R	Manipulate HDF5 references, which are like UNIX links or Windows shortcuts
Dataspace	H5S	Define and work with data spaces, which describe the dimensionality of a data set
Datatype	H5T	Define the type of variable that is stored in a dataset
Filters and Compression	H5Z	Create inline data filters and data compression

In most cases, the syntax of the MATLAB function matches the syntax of the HDF5 library function. To get detailed information about the MATLAB syntax of an HDF5 library function, view the help for the individual MATLAB function, as follows:

```
help H5F.open
```

To view a list of all the MATLAB HDF5 functions in a particular interface, type:

```
help H5F
```

# hdf5

---

## **See Also**

`hdf` | `h5create` | `h5disp` | `h5info` | `h5read` | `h5readatt` | `h5write` | `h5writeatt`

**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 dimensions. 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.

# hdf5info

---

---

**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`



**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',B00L)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', B00L)
```

**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. For more information about the HDF5 data type objects, see the `hdf5` reference page.

---

**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.

---

# hdf5read

---

`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

[hdf5](#) | [hdf5info](#) | [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

---

`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 <code>uint8</code> , the <code>hdf5write</code> 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, <code>hdf5write</code> uses the same data type when writing to the file. For all HDF5 objects, except <code>HDF5.h5enum</code> objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For <code>HDF5.h5enum</code> 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)))
```

# hdf5write

---

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

[hdf5](#) | [hdf5read](#) | [hdf5info](#)

**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 `hdinfo` 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: []
```

# hdfinfo

---

```
    Dims: [2x1 struct]
    Label: {}
    Description: {}
    Index: 0
```

## See Also

[hdfread](#) | [hdf](#)

**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 `structurehinfo`, 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 2-2039 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.



Parameter	Description
'Index'	<p>Three-element cell array, {start, stride, edge}, specifying the location, range, and values to be read from the data set</p> <ul style="list-style-type: none"> <li>• <b>start</b> — 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.</li> <li>• <b>stride</b> — A 1-based array specifying the interval between the values to read Default: 1, read every element of the data set.</li> <li>• <b>edge</b> — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions</li> </ul>

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', {[1],[2 1],[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

# hdfread

---

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
<b>Required Parameter</b>	
'Fields'	String specifying the field to be read. You can specify only one field name for a Grid data set.
<b>Mutually Exclusive Optional Parameters</b>	
'Index'	Three-element cell array, { <code>start</code> , <code>stride</code> , <code>edge</code> }, specifying the location, range, and values to be read from the data set  <code>start</code> — 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.  <code>stride</code> — An array specifying the interval between the values to read Default: 1, read every element of the data set.  <code>edge</code> — 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
<b>Optional Parameters</b>	
'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.

# hdfread

For example,

```
data = hdfread('grid.hdf','PolarGrid','Fields','ice_temp','Index', {[5 10],[1],[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
<b>Required Parameters</b>	
'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
<b>Mutually Exclusive Optional Parameters</b>	
'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
<b>Required Parameter</b>	
'Fields'	String naming the data set field to be read. You can specify only one field name for a Swath data set.
<b>Mutually Exclusive Optional Parameters</b>	
'Index'	<p>Three-element cell array, {<code>start</code>, <code>stride</code>, <code>edge</code>}, specifying the location, range, and values to be read from the data set</p> <ul style="list-style-type: none"> <li>• <code>start</code> — 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.</li> <li>• <code>stride</code> — An array specifying the interval between the values to read            Default: 1, read every element of the data set.</li> <li>• <code>edge</code> — An array specifying the length of each dimension to read            Default: An array containing the lengths of the corresponding dimensions</li> </ul>
'Time'	<p>Three-element cell array, {<code>start</code>, <code>stop</code>, <code>mode</code>}, where <code>start</code> and <code>stop</code> specify the beginning and the endpoint for a period of time, and <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:</p> <ul style="list-style-type: none"> <li>• Its midpoint is within the box (<code>mode='midpoint'</code>).</li> <li>• Either endpoint is within the box (<code>mode='endpoint'</code>).</li> </ul>

# hdfread

Parameter	Description
<b>Optional Parameters</b>	
'Box'	<p>Three-element cell array, {longitude, latitude, mode} specifying the longitude and latitude coordinates that define a region. longitude and latitude are two-element vectors that specify longitude and latitude coordinates. 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:</p> <ul style="list-style-type: none"><li>• Its midpoint is within the box (mode='midpoint').</li><li>• Either endpoint is within the box (mode='endpoint').</li><li>• Any point is within the box (mode='anypoint').</li></ul>
'Vertical'	<p>Two-element cell array, {dimension, range}</p> <ul style="list-style-type: none"><li>• dimension is a string specifying either a dimension name or field name to subset the data by.</li><li>• range is a two-element vector 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.</li></ul> <p>'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.</p>

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: {}
```

# hdfread

---

```
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](#)

# hdftool

---

**Purpose** Browse and import data from HDF4 or HDF-EOS files

**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` | `hdftool` | `hdfinfo` | `hdfread` | `uiimport`

**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, .

## 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”

**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

# helpdlg

---

**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 `WindowState` 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.

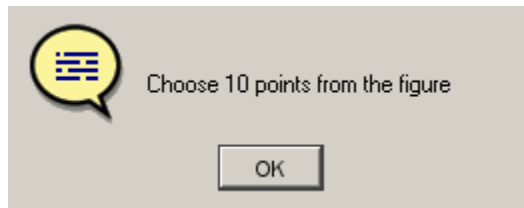
**Tips** MATLAB wraps the text in 'helpstring' to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

**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](#)

# helpwin

---

**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

**Purpose** Hessenberg form of matrix

**Syntax**  
 $H = \text{hess}(A)$   
 $[P, H] = \text{hess}(A)$   
 $[AA, BB, Q, Z] = \text{hess}(A, B)$

**Description**  $H = \text{hess}(A)$  finds H, the Hessenberg form of matrix A.  
 $[P, H] = \text{hess}(A)$  produces a Hessenberg matrix H and a unitary matrix P so that  $A = P*H*P'$  and  $P'*P = \text{eye}(\text{size}(A))$ .  
 $[AA, BB, Q, Z] = \text{hess}(A, B)$  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

# matlab.mixin.Heterogeneous

---

**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.

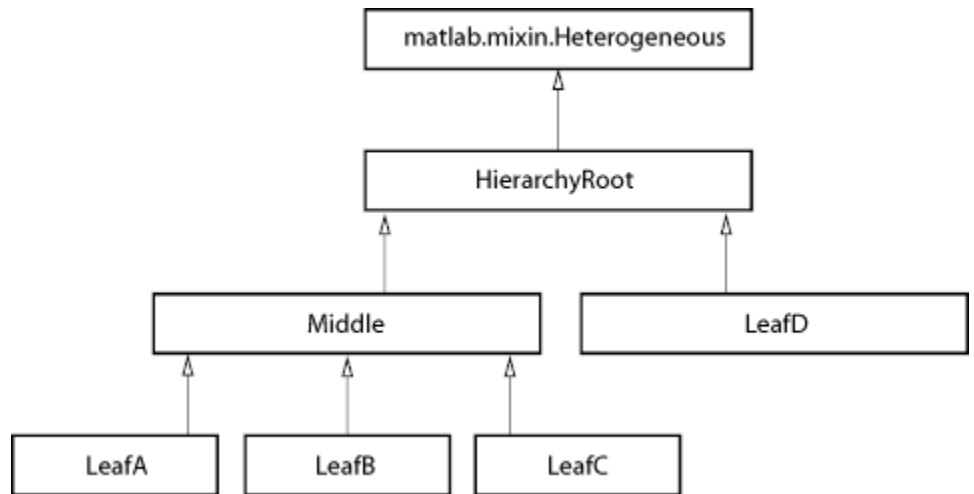
The following class definition enables the formation of heterogeneous arrays that combine instances of any classes derived from `HierarchyRoot`.

```
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 another 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 nargin == 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]
```

# matlab.mixin.Heterogeneous

---

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',objectToC
    ...
    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

<code>cat</code>	Concatenation for heterogeneous arrays
<code>getDefaultScalarElement</code>	Return default object for heterogeneous array operations
<code>horzcat</code>	Horizontal concatenation for heterogeneous arrays
<code>vertcat</code>	Vertical concatenation for heterogeneous arrays

## See Also

`handle`

## How To

- Class Attributes
- Property Attributes
- “Creating Subclasses — Syntax and Techniques”

# matlab.mixin.Heterogeneous.cat

---

**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 2-2065 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 2-2065.

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 2-2065 is not suitable as a default object.
  - Override `getDefaultScalarElement` only in the “Root Class” on page 2-2065 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.



# matlab.mixin.Heterogeneous.getDefaultScalarElement

---

## 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

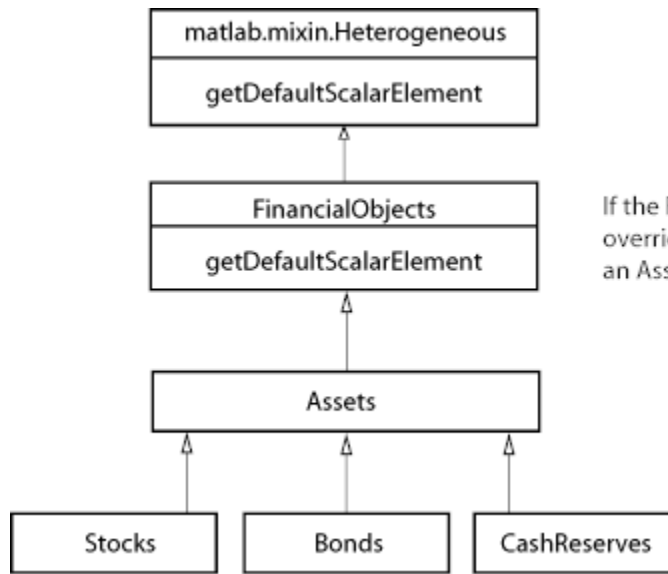
Static	true
Access	Protected
Sealed	true not required

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.

# matlab.mixin.Heterogeneous.getDefaultScalarElement



If the FinancialObjects class is abstract, override getDefaultScalarElement to return an Assets object 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

**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.

# matlab.mixin.Heterogeneous.horzcat

---

## 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`

**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.

# matlab.mixin.Heterogeneous.vertcat

---

## 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`

**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`

# hex2num

---

**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.

NaNs, infinities and denorms are handled correctly.

**Examples** `hex2num('400921fb54442d18')`

returns `Pi`.

`hex2num('bff')`

returns

`ans =`

`-1`

**See Also** `num2hex` | `hex2dec` | `sprintf` | `format`



**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. For details, see [How to Print or Export](#).

**See Also** print

# hggroup

---

## 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.

---

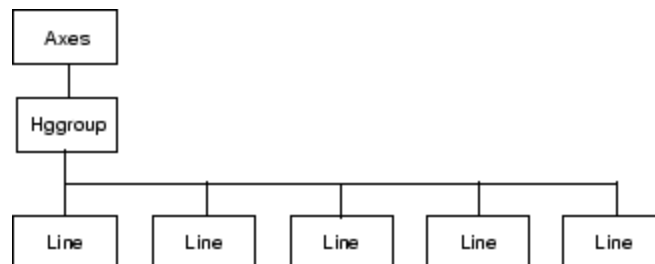
```
function doc_hgroup
hg = hgroup('ButtonDownFcn',@set_lines);
hl = line(randn(5),randn(5),'HitTest','off','Parent',hg);

function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hgroup 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.



# hggroup

---

## 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”

## Purpose

Hgggroup properties

## Creating Hgggroup Objects

Use `hggroup` to create hgggroup 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.

## Hgggroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

### Annotation

`hg.Annotation` object (read-only)

*Control the display of hgggroup objects in legends.* Specifies whether this hgggroup 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 hgggroup object is displayed in a figure legend.

# Hgroup Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
on	Include the hgroup object in a legend as one entry, but not its children objects
off	Do not include the hgroup or its children in a legend (default)
children	Include only the children of the hgroup 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-to-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 hgggroup 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.

# Hgroup Properties

---

## Children

array of graphics object handles

*Children of the hgroup object.* An array containing the handles of all objects parented to the hgroup 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 hgroup `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 hgroup object. You must define this property as a default value for hgroup objects or in a call to the hgroup function to create a new hgroup object. For example, the statement:

```
set(0, 'DefaultHgroupCreateFcn', @myCreateFcn)
```

defines a default value on the root level that applies to every hgroup object created in that MATLAB session. Whenever you create an hgroup object, the function associated with the function handle `@myCreateFcn` executes.



MATLAB executes the callback after setting all the hgggroup object's properties. Setting the `CreateFcn` property on an existing hgggroup 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 hgggroup object is deleted (for example, this might happen when you issue a `delete` command on the hgggroup 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

# Hgggroup Properties

---

*String used by legend.* The `legend` function uses the `DisplayName` property to label the `hgggroup` 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 `hgggroup` 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.

# Hgroup Properties

---

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 hgroup 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.

# Hgroup Properties

---

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 hgroup object.* This property contains the handle of the hgroup object's parent object. The parent of an hgroup object is the axes, hgroup, 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 hgroup 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 hgroup 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 hgroup object and set the Tag property:

```
t = hgroup('Tag', 'group1')
```

When you want to access the object, use findobj to find its handle. For example:

# Hgroup Properties

---

```
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 hgroup objects, Type is 'hgroup'. The following statement finds all the hgroup objects in the current axes.

```
t = findobj(gca,'Type','hgroup');
```

## UIContextMenu

handle of uicontextmenu object

*Associate a context menu with the hgroup object.* Assign this property the handle of a uicontextmenu object created in the hgroup object's figure. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click the hgroup object.

## UserData

array

*User-specified data.* Data you want to associate with the hgroup 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 hgroup object and its children.* By default, hgroup object visibility is on. This means all children of the hgroup are visible unless the child object's Visible property is off. Setting an hgroup object's Visible property to off also makes its children invisible.



**See Also**      `hgroup`

# hgload

---

## 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`

# hgsave

---

**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. For details, see [How to Print or Export](#).

## See Also

[hgload](#) | [open](#) | [save](#)

# hgsetget

---

**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
<code>set</code>	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
<code>get</code>	Returns value of specified property or a struct with all property values.
<code>setdisp</code>	Called when <code>set</code> is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.
<code>getdisp</code>	Called when <code>get</code> 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”

<b>Purpose</b>	Create hgtransform graphics object
<b>Syntax</b>	<pre>h = hgtransform h = hgtransform('PropertyName',propertyvalue,...)</pre>
<b>Properties</b>	For a list of properties, see Hgtransform Properties.
<b>Description</b>	<p><code>h = hgtransform</code> creates an hgtransform object and returns its handle.</p> <p><code>h = hgtransform('PropertyName',propertyvalue,...)</code> creates an hgtransform object with the property value settings specified in the argument list. For a description of the properties, see Hgtransform Properties.</p> <p>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):</p> <pre>h = hgtransform; surface('Parent',h,...)</pre> <p>The primary advantage of parenting objects to an hgtransform object is that you can perform <i>transforms</i> (for example, translation, scaling, rotation, etc.) on the child objects in unison.</p> <p>The parent of an hgtransform object is either an axes object or another hgtransform.</p> <p>Although you cannot see an hgtransform object, setting its Visible property to off makes all its children invisible as well.</p>
<b>Exceptions and Limitations</b>	<ul style="list-style-type: none"><li>• An hgtransform object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.</li><li>• hgtransform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.</li></ul>

- 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.

---

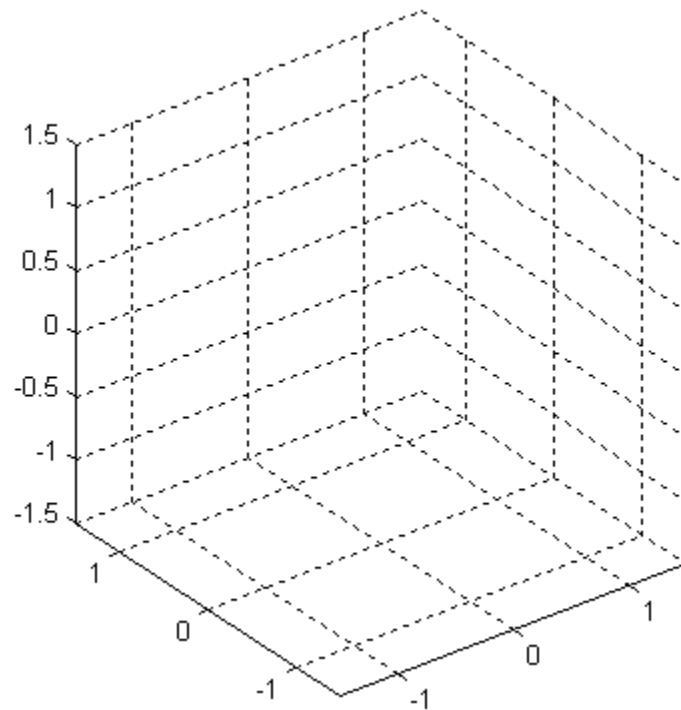
**Tip** If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

- 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
```



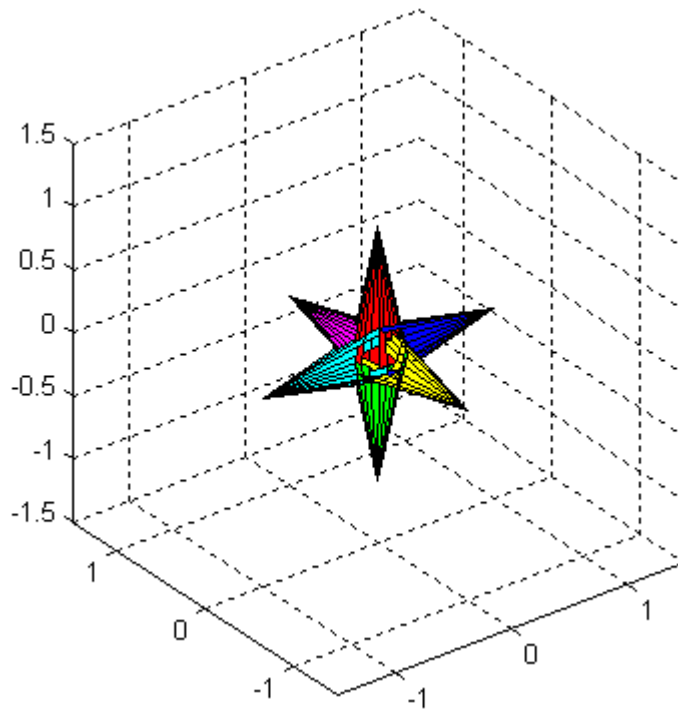


**2** Create the objects you want to parent to the hgtransform object.

```
[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');
```

# hgtransform

---

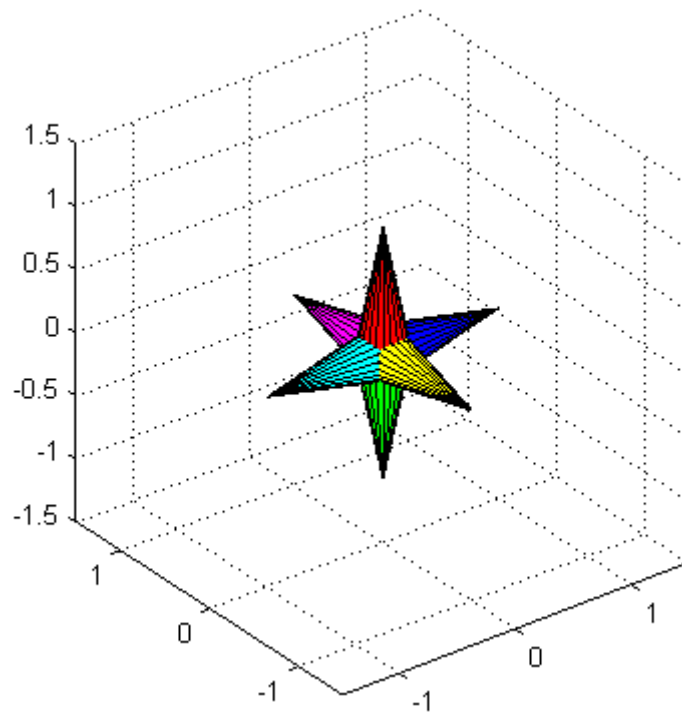


- 3** Create an hgtransform object and parent the surface objects to it. The figure should not change from the image above.

```
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;
```

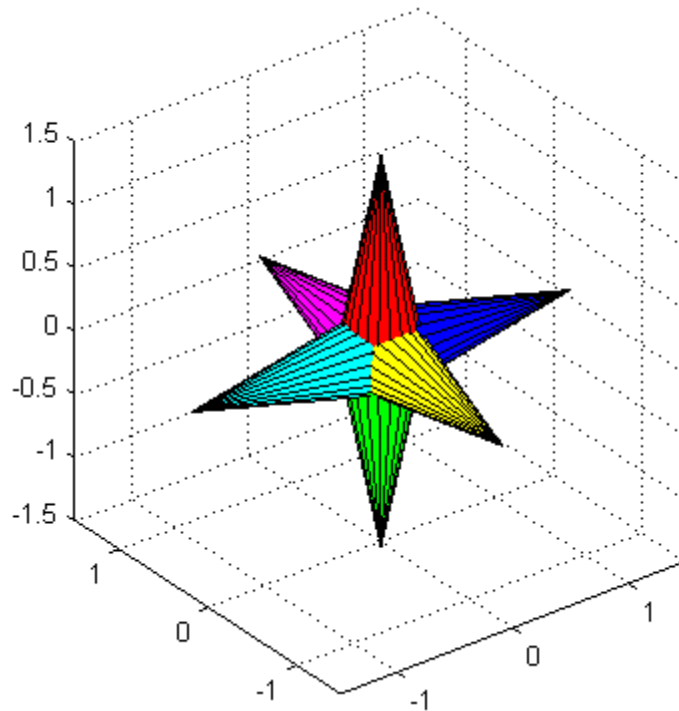
- 6 Form the  $z$ -axis rotation matrix and the scaling matrix. Rotate 360 degrees ( $2\pi$  radians) and scale by using the increasing values of  $r$ .

```
for r = 1:.1:2*pi  
    % Z-axis rotation matrix  
    Rz = makehgtform('zrotate',r);  
    % Scaling matrix  
    Sxy = makehgtform('scale',r/4);
```

# hgtransform

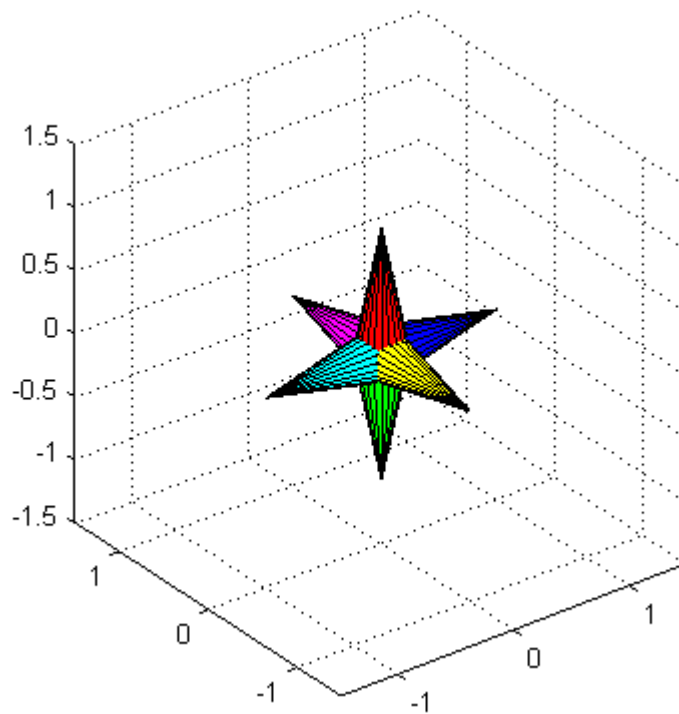
---

```
% 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.

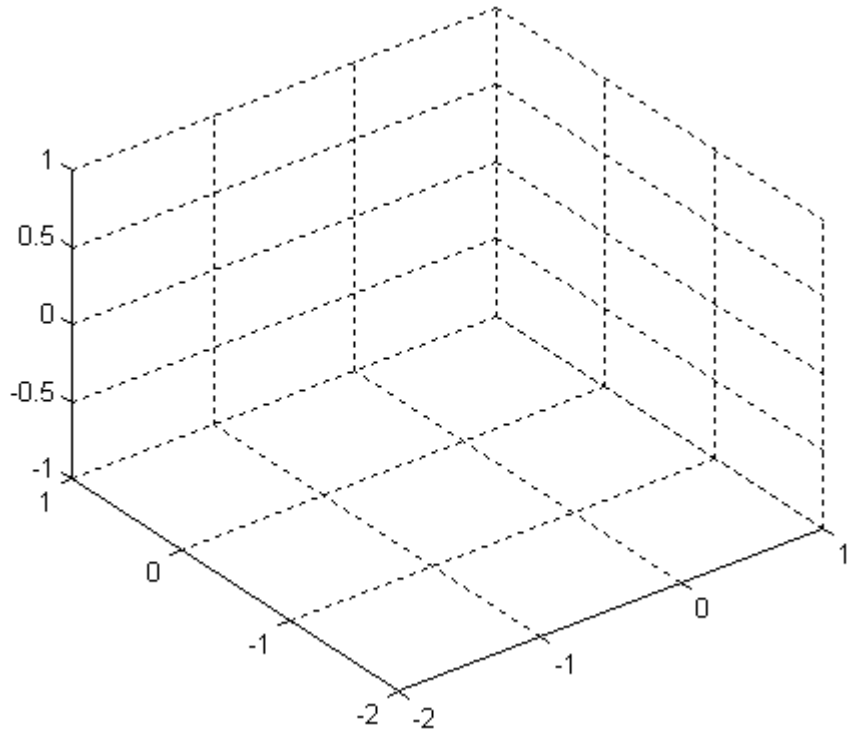
---

- 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.

# hgtransform

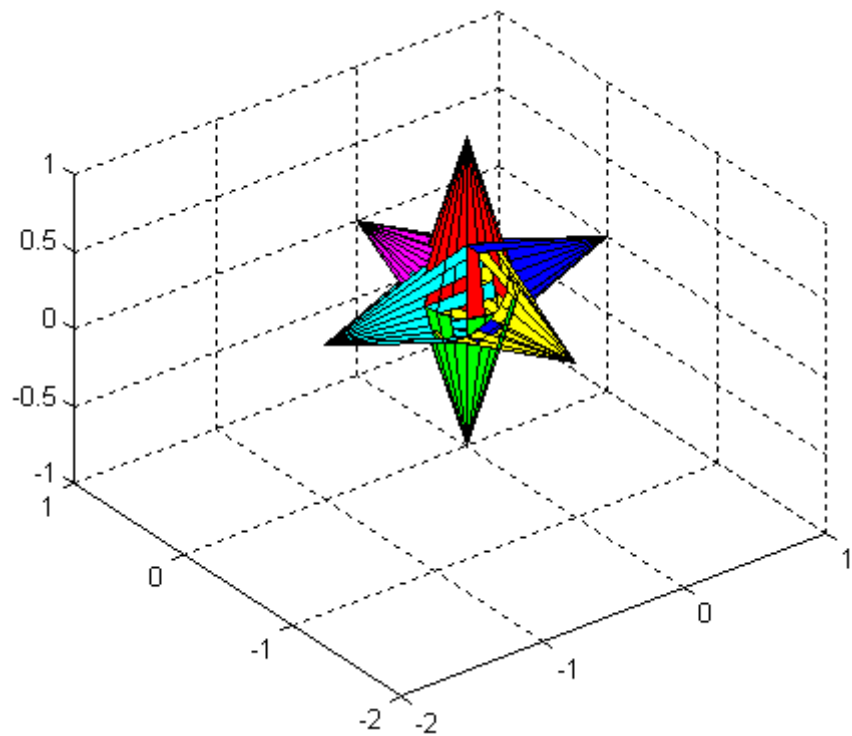
---

```
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.

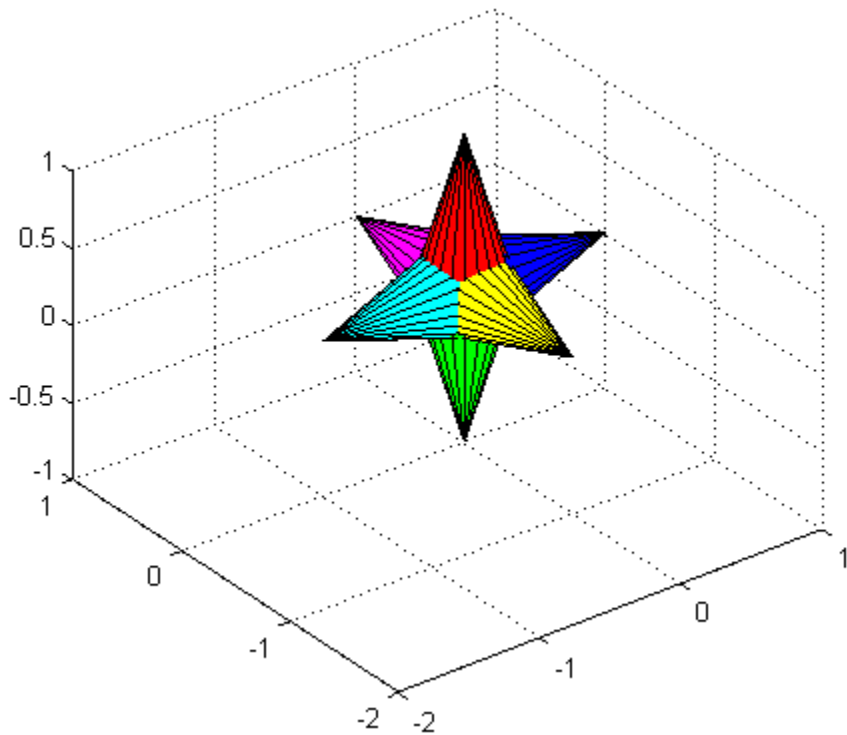
```
t1 = hgtransform('Parent',ax);  
t2 = hgtransform('Parent',ax);
```

- 4** Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```

# hgtransform

---



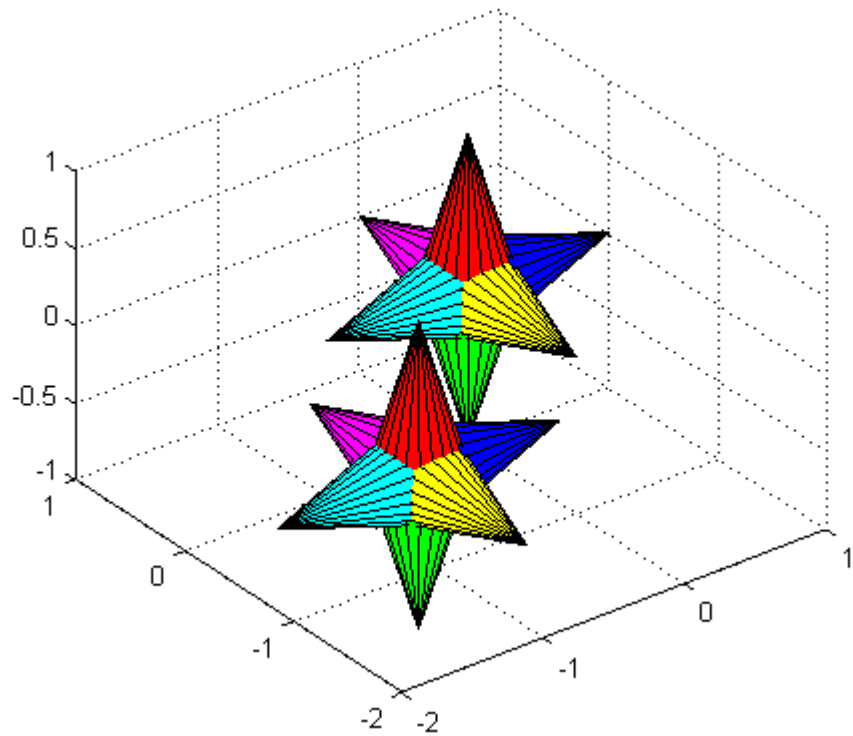
- 5** Parent the surfaces to hgtransform t1, then copy the surface objects and parent the copies to hgtransform t2. This figure should not change.

```
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
```

# hgtransform

---

## 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”

## 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.

# Hgtransform Properties

---

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
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

# Hgtransform Properties

---

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.

# Hgtransform Properties

---

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.

# Hgtransform Properties

---

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

# Hgtransform Properties

---

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, hgroup, 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

# Hgtransform Properties

---

*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

# hidden

---

**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.

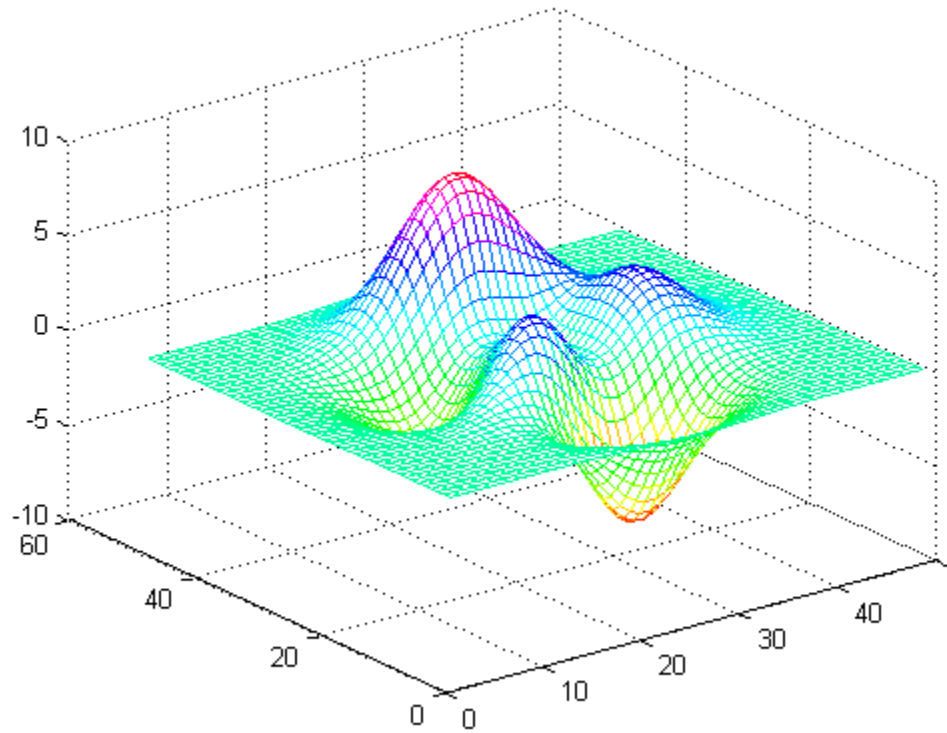
**Algorithms** When a surface graphics object has a uniform `FaceColor` matching the `Color` property of the axes, `hidden off` sets the `FaceColor` of the surface object to 'none'.

`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
```



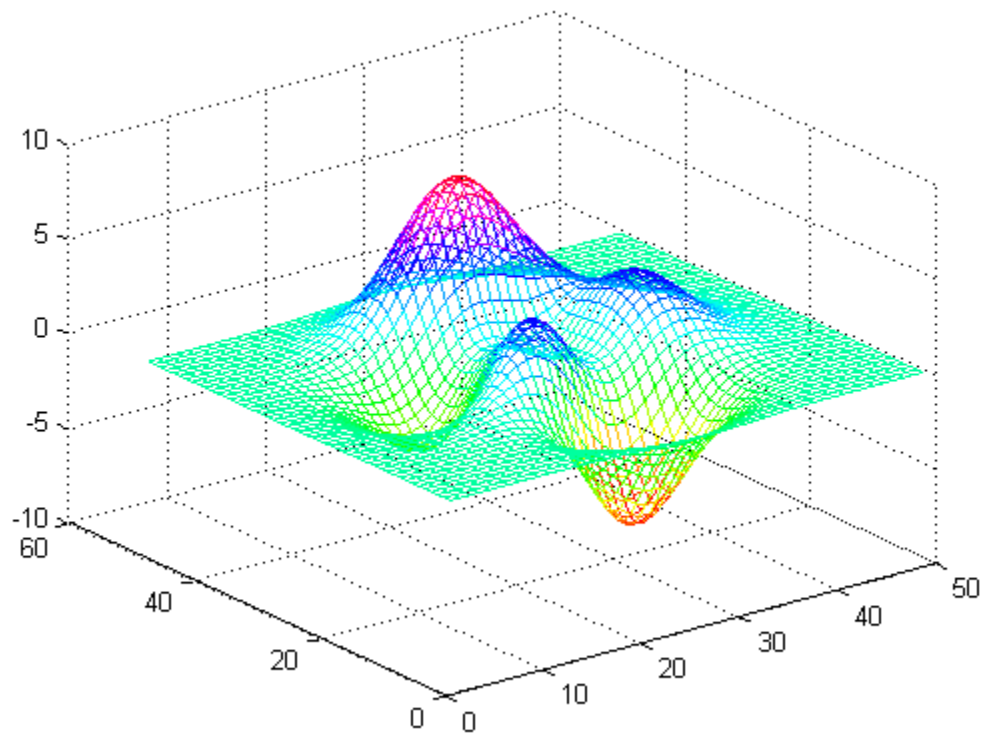


Set hidden line removal off to show the obscured parts of the mesh.

```
hidden off
```

# hidden

---



## See Also

[shading](#) | [mesh](#) | [EdgeColor](#) | [FaceColor](#)

---

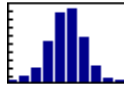
<b>Purpose</b>	Hilbert matrix
<b>Syntax</b>	<code>H = hilb(n)</code>
<b>Description</b>	<code>H = hilb(n)</code> returns the Hilbert matrix of order <code>n</code> .
<b>Definitions</b>	The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i,j) = 1/(i + j - 1)$ .
<b>Examples</b>	Even the fourth-order Hilbert matrix shows signs of poor conditioning.  <code>cond(hilb(4)) =</code> <code>1.5514e+04</code>
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.
<b>See Also</b>	<code>invhilb</code>

# hist

---

## Purpose

Histogram plot



## Syntax

```
n = hist(Y)
n = hist(Y,x)
n = hist(Y,nbins)
[n,xout] = hist(...)
hist(...)
hist(axes_handle,...)
```

## Description

A histogram shows the distribution of data values.

`n = hist(Y)` bins the elements in vector `Y` into 10 equally spaced containers and returns the number of elements in each container as a row vector. If `Y` is an `m`-by-`p` matrix, `hist` treats the columns of `Y` as vectors and returns a 10-by-`p` matrix `n`. Each column of `n` contains the results for the corresponding column of `Y`. No elements of `Y` can be complex or of type `integer`.

`n = hist(Y,x)` where `x` is a vector, returns the distribution of `Y` among `length(x)` bins with centers specified by `x`. For example, if `x` is a 5-element vector, `hist` distributes the elements of `Y` into five bins centered on the `x`-axis at the elements in `x`, none of which can be complex. Note: use `histc` if it is more natural to specify bin edges instead of centers.

`n = hist(Y,nbins)` where `nbins` is a scalar, uses `nbins` number of bins.

`[n,xout] = hist(...)` returns vectors `n` and `xout` containing the frequency counts and the bin locations. You can use `bar(xout,n)` to plot the histogram.

`hist(...)` without output arguments produces a histogram plot of the output described above. `hist` distributes the bins along the `x`-axis between the minimum and maximum values of `Y`.

---

`hist(axes_handle, ...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

## Tips

All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin. If `Y` is a matrix, `hist` scales the current colormap and uses each column number (`1:n`) to assign each a unique color.

The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from 0 to the greatest number of elements deposited in any bin. The *x*-range of the leftmost and rightmost bins extends to include the entire data range in the case when the user-specified range does not cover the data range; this often results in "boxes" at either or both edges of the distribution. If you want a plot in which this does not happen (that is, all bins have equal width), you can create a histogram-like display using the `bar` command.

Histograms bins are created as patch objects and always plotted with a face color that maps to the first color in the current colormap (by default, blue) and with black edges. To change colors or other patch properties, use code similar to that given in the example.

The `hist` function accepts data that contains `inf` values, but does not consider them while populating the bins because the bin centers are all finite values. So, you should filter non-finite values before passing them to `hist`.

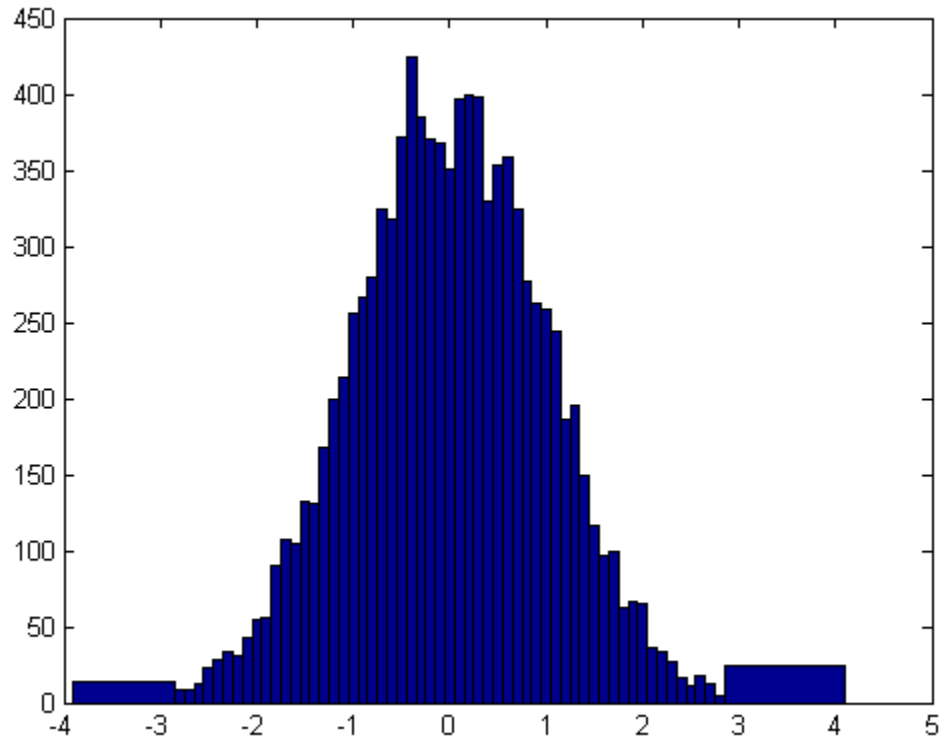
## Examples

Generate a bell-curve histogram from Gaussian data.

```
x = -4:0.1:4;  
y = randn(10000,1);  
hist(y,x)
```

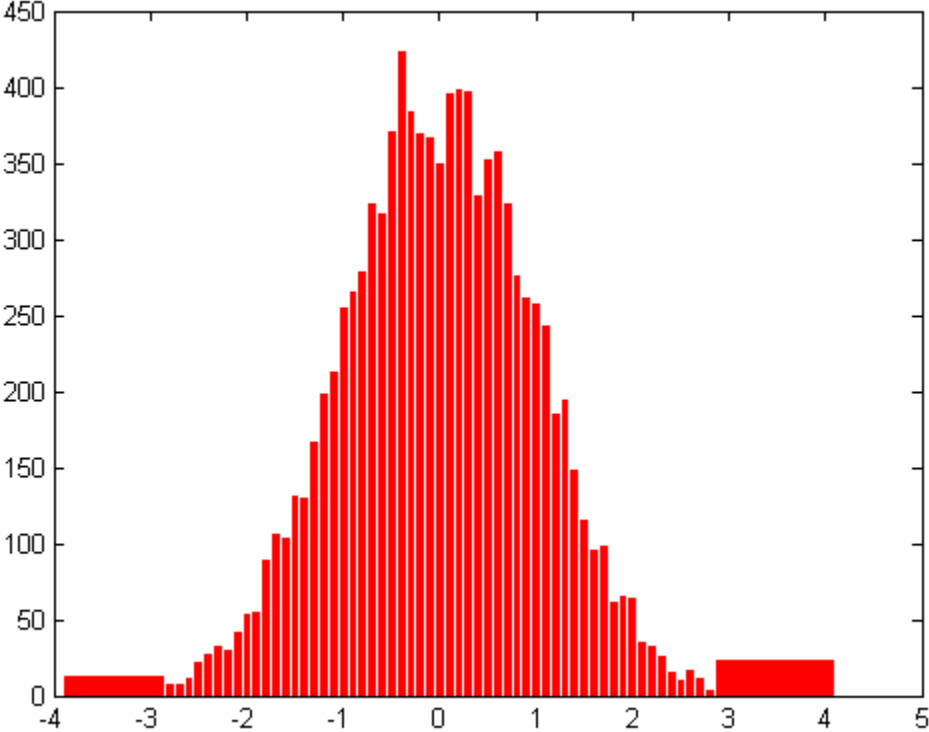
# hist

---



Change the color of the graph so that the bins are red and the edges of the bins are white.

```
h = findobj(gca,'Type','patch');  
set(h,'FaceColor','r','EdgeColor','w')
```



**See Also**

bar | ColorSpec | histc | mode | patch | rose | stairs

**How To**

- “Histograms”

# histc

---

**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
```

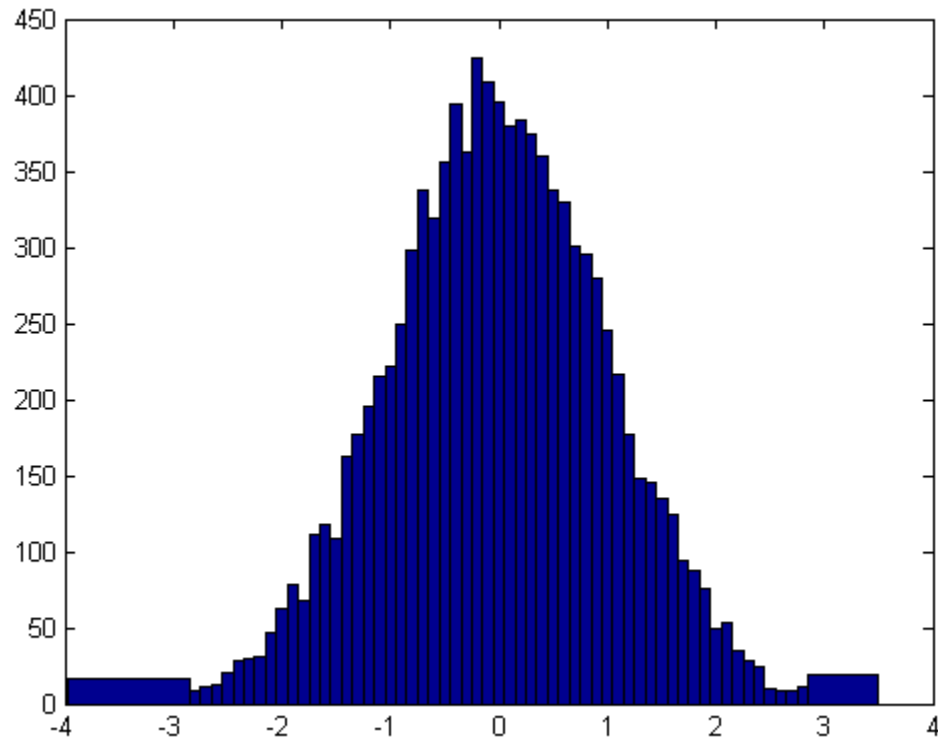
To plot the histogram, use the `bar` command.

**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

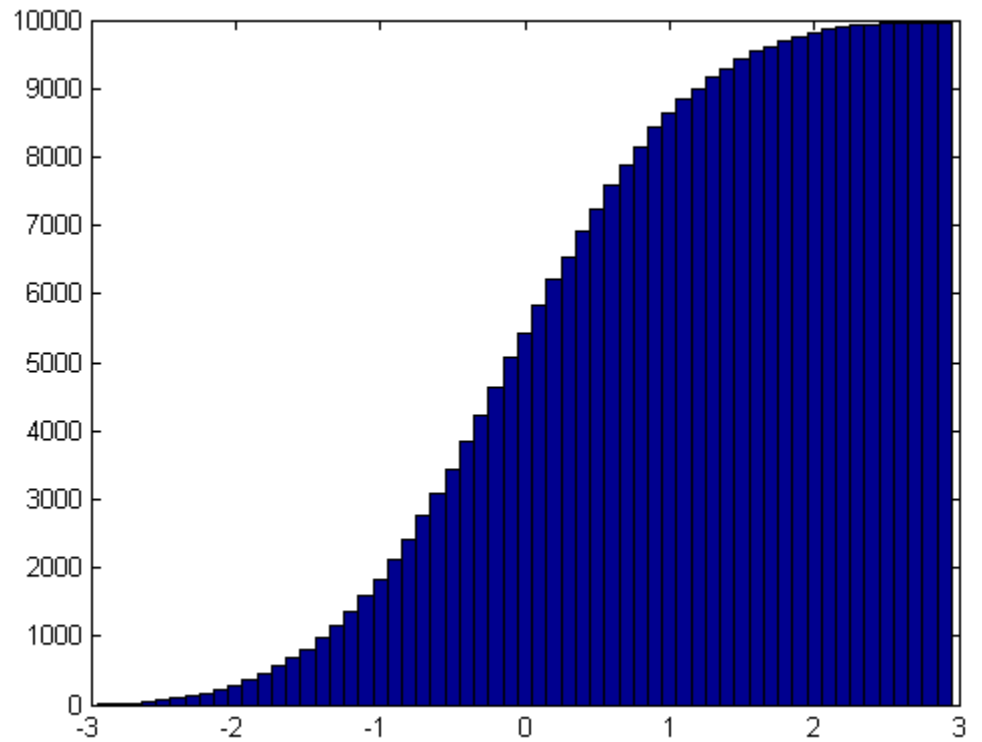
```
c_elements = cumsum(n_elements)
```

Plot the cumulative distribution like a histogram using bar:

# histc

---

```
figure(2)  
bar(x,c_elements,'BarWidth',1)
```



## See Also

[bar](#) | [hist](#) | [mode](#)

---

<b>Purpose</b>	Retain current graph when adding new graphs
<b>Syntax</b>	<code>hold on</code> <code>hold off</code> <code>hold all</code> <code>hold</code> <code>hold(axes_handle,...)</code>
<b>Description</b>	<p>The <code>hold</code> 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.</p> <p><code>hold on</code> 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.</p> <p><code>hold off</code> 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.</p> <p><code>hold all</code> holds the graph and the current line color and line style so that subsequent plotting commands do not reset the <code>ColorOrder</code> and <code>LineStyleOrder</code> 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.</p> <p><code>hold</code> toggles the hold state between adding to the graph and replacing the graph.</p> <p><code>hold(axes_handle,...)</code> applies the hold to the axes identified by the handle <code>axes_handle</code>. If several axes objects exist in a figure window, each axes has its own hold state. <code>hold</code> also creates an axes if one does not exist.</p> <p>Test the hold state using the <code>ishold</code> function.</p>
<b>Examples</b>	<p>Create a graph, set <code>hold</code> to on, then create another graph in the same axes:</p> <pre>x = -pi:pi/20:pi;</pre>

# hold

---

```
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”

**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

# horzcat

---

**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”

# horzcat (tscollection)

---

**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`)



**Purpose** Server host identification number

**Syntax** `id = hostid`

# hostid

---

## Description

`id = hostid` usually returns a single element cell array containing the MATLAB server host identifier as a string. On UNIX<sup>5</sup> 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:

$$c = \text{sqrt}(\text{abs}(a).^2 + \text{abs}(b).^2)$$

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

$$c = \text{sqrt}(\text{abs}(w).^2 + \text{abs}(x).^2 + \text{abs}(y).^2 + \text{abs}(z).^2)$$

- 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  $\sqrt{2} * \text{abs}(a)$ , as shown in this example:

```
x = 1.271161e308;
```

# hypot

---

```
y = x * sqrt(2)
y =
  1.7977e+308
```

```
y = hypot(x,x)
y =
  1.7977e+308
```

## See Also

[sqrt](#) | [abs](#) | [norm](#)

---

<b>Purpose</b>	Imaginary unit
<b>Syntax</b>	<code>i</code> <code>a+bi</code> <code>x+i*y</code>
<b>Description</b>	<p>As the basic imaginary unit <code>sqrt(-1)</code>, <code>i</code> is used to enter complex numbers. You can also use the character <code>j</code> as the imaginary unit. If you want to create a complex number without using <code>i</code> and <code>j</code>, you can use the <code>complex</code> function.</p> <p>Since <code>i</code> is a function, it can be overridden and used as a variable. However, it is best to avoid using <code>i</code> and <code>j</code> for variable names if you intend to use them in complex arithmetic.</p> <p>If desired, you can use the character <code>i</code> without a multiplication sign as a suffix in forming a complex numerical constant.</p> <p>For speed and improved robustness, you can replace complex <code>i</code> and <code>j</code> by <code>1i</code>. For example, instead of using</p> <pre>a = i;</pre> <p>use</p> <pre>a = 1i;</pre>
<b>Examples</b>	<pre>Z = 2+3i Z = x+i*y Z = r*exp(i*theta)</pre>
<b>See Also</b>	<code>conj</code>   <code>imag</code>   <code>j</code>   <code>real</code>

# ichol

---

**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 <i>incomplete Cholesky with zero-fill</i> (IC(0)). The 'ict' variant performs <i>incomplete Cholesky with threshold</i>



Field Name	Summary	Description
droptol	Drop tolerance when type is 'ict'	<p><i>dropping</i> (ICT). The default value is 'nofill'.</p> <p>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 <math>j</math> of the factorization is <math>\text{norm}(A(j:\text{end}, j), 1) * \text{droptol}</math>. 'droptol' is ignored if 'type' is 'nofill'. The default value is 0.</p>
michol	Indicates whether to perform modified incomplete Cholesky	<p>Indicates whether or not <i>modified incomplete Cholesky</i> (MIC) is performed. The field may be 'on' or 'off'. When performing MIC, the diagonal is compensated for dropped elements to enforce the relationship <math>A * e = L * L' * e</math> where <math>e = \text{ones}(\text{size}(A, 2), 1)</math>. The default value is 'off'.</p>

<b>Field Name</b>	<b>Summary</b>	<b>Description</b>
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 $A + \alpha \cdot \text{diag}(\text{diag}(A))$ 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));
```

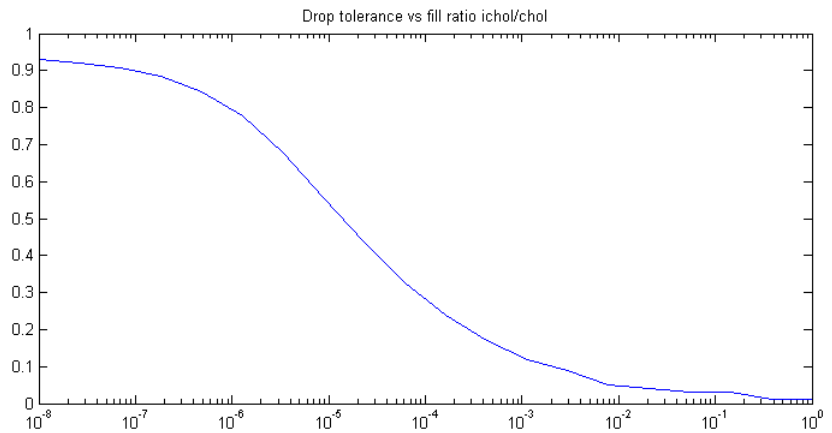
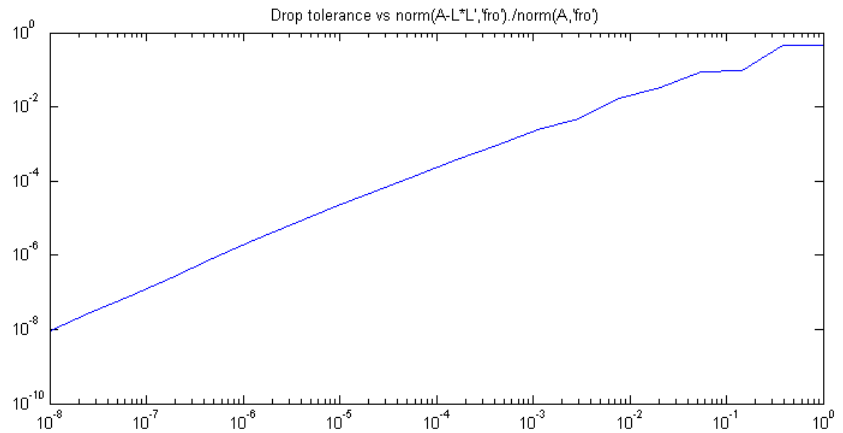
A is the two-dimensional, five-point discrete negative Laplacian on a 100-by-100 square grid with Dirichlet boundary conditions. The size of A is  $98 \times 98 = 9604$  (not 10000 as the borders of the grid are used to impose the Dirichlet conditions).

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,f10,rr0,it0,rv0] = pcg(A,b,tol,maxit);
```

Note that `f10 = 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,f11,rr1,it1,rv1] = pcg(A,b,tol,maxit,L1,L1');
```

`f11 = 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,  $\text{norm}(A*e - L*(L'*e))$  will be approximately zero for  $e = \text{ones}(\text{size}(A,2),1)$  even though  $\text{norm}(A-L*L', 'fro')/\text{norm}(A, 'fro')$  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.

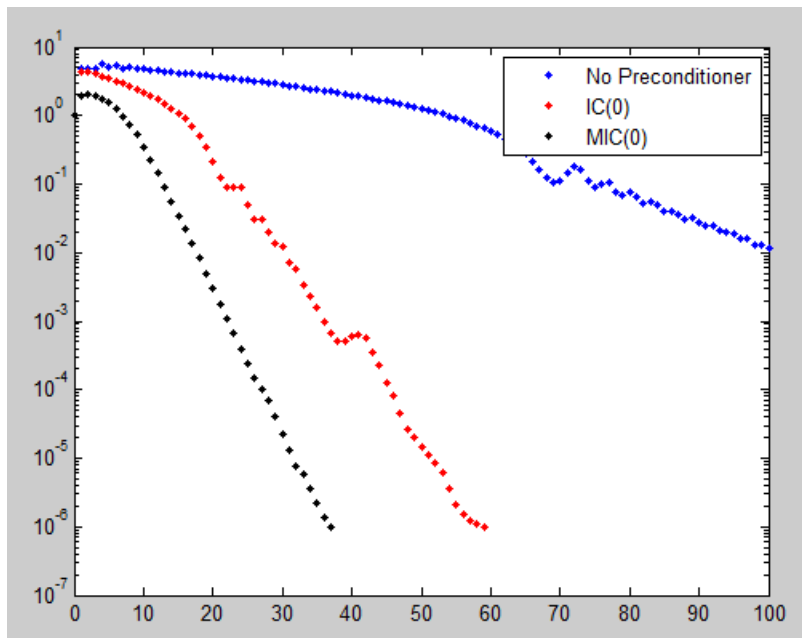
---

```
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,f12,rr2,it2,rv2] = pcg(A,b,tol,maxit,L2,L2');
```

pcg converges ( $f12 = 0$ ) 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.

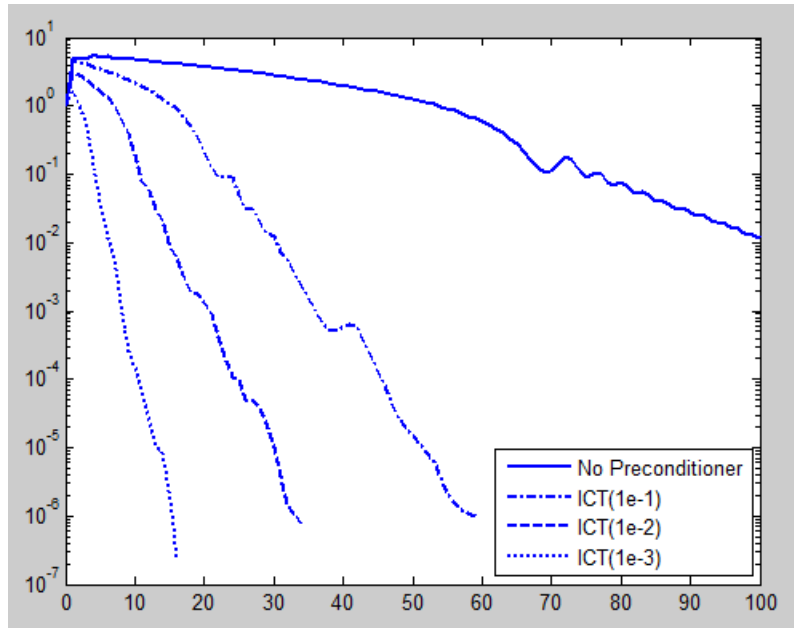
You can also try incomplete Cholesky factorizations with threshold dropping. The following plot shows convergence of pcg with preconditioners constructed with various drop tolerances.

```
L3 = ichol(A, struct('type','ict','droptol',1e-1));
[x3,f13,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  $\text{norm}(A \cdot e - L \cdot (L' \cdot e))$  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,f10,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  $L \cdot L'$  approximates  $A$ , `ichol` with diagonal compensation constructs  $L$  such that  $L \cdot L'$  approximates  $M = A + \alpha \cdot \text{diag}(\text{diag}(A))$  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,f11,rr1,it1,rv1] = pcg(A,b,1e-6,100,L1,L1');
```

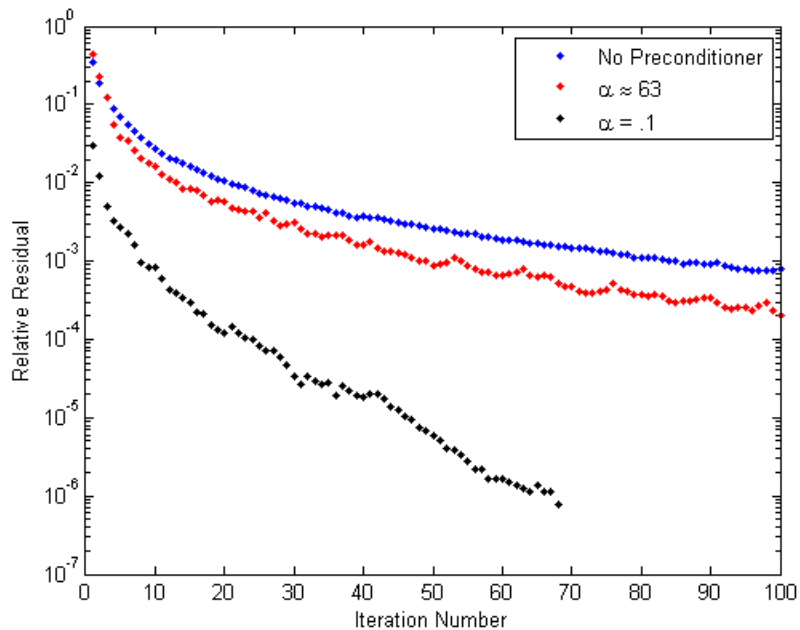


Here, `pcg` still fails to converge to the desired tolerance within the desired number of iterations, but as the plot below shows, convergence is better for `pcg` with this preconditioner than with no preconditioner. Choosing a smaller `alpha` may help. With some experimentation, we can settle on an appropriate value for `alpha`:

```
alpha = .1;
L2 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x2,f12,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

---

```
idivide(a,b,'round')    % Returns [-1 1]
```

## **See Also**

```
ldivide | rdivide | mldivide | mrdivide
```

## 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.

# if/elseif/else

---

## 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  $x(i) = \text{conj}(x(\text{mod}(N-i+1,N)+1))$  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  $X(i,j) = \text{conj}(X(\text{mod}(M-i+1, M) + 1, \text{mod}(N-j+1, N) + 1))$  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 `iff2` 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

`iff2` supports inputs of data types `double` and `single`. If you call `iff2` with the syntax `y = iff2(X, ...)`, the output `y` has the same data type as the input `X`.

### See Also

`dftmtx` | `freqz` | `fft2` | `fftw` | `fftshift` | `ifft` | `ifftn` | `ifftshift`

**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

$$X(i_1, i_2, \dots, i_k) = \text{conj}(X(\text{mod}(N_1 - i_1 + 1, N_1) + 1, \text{mod}(N_2 - i_2 + 1, N_2) + 1, \dots, \text{mod}(N_k - i_k + 1, N_k) + 1))$$

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
<code>type</code>	<p>Type of factorization. Values for <code>type</code> include:</p> <ul style="list-style-type: none"> <li>'nofill'—Performs ILU factorization with 0 level of fill in, known as ILU(0). With <code>type</code> set to 'nofill', only the <code>milu</code> setup option is used; all other fields are ignored.</li> <li>'crout'—Performs the Crout version of ILU factorization, known as ILUC. With <code>type</code> set to 'crout', only the <code>droptol</code> and <code>milu</code> setup options are used; all other fields are ignored.</li> <li>'ilutp' (default)—Performs ILU factorization with threshold and pivoting.</li> </ul> <p>If <code>type</code> is not specified, the ILU factorization with pivoting ILUTP is performed. Pivoting is never performed with <code>type</code> set to 'nofill' or 'crout'.</p>
<code>droptol</code>	<p>Drop tolerance of the incomplete LU factorization. <code>droptol</code> is a non-negative scalar. The default value is 0, which produces the complete LU factorization.</p>

Field Name	Description
	<p>The nonzero entries of U satisfy</p> $\text{abs}(U(i,j)) \geq \text{droptol} * \text{norm}(A(:,j)),$ <p>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</p> $\text{abs}(L(i,j)) \geq \text{droptol} * \text{norm}(A(:,j)) / U(j,j).$
milu	<p>Modified incomplete LU factorization. Values for milu include:</p> <ul style="list-style-type: none"> <li>• '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, <math>A * e = L * U * e</math>, where e is the vector of ones.</li> <li>• '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, <math>e' * A = e' * L * U</math>.</li> <li>• 'off' (default)—No modified incomplete LU factorization is produced.</li> </ul>
udiag	<p>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.</p>
thresh	<p>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.</p>



`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`.

---

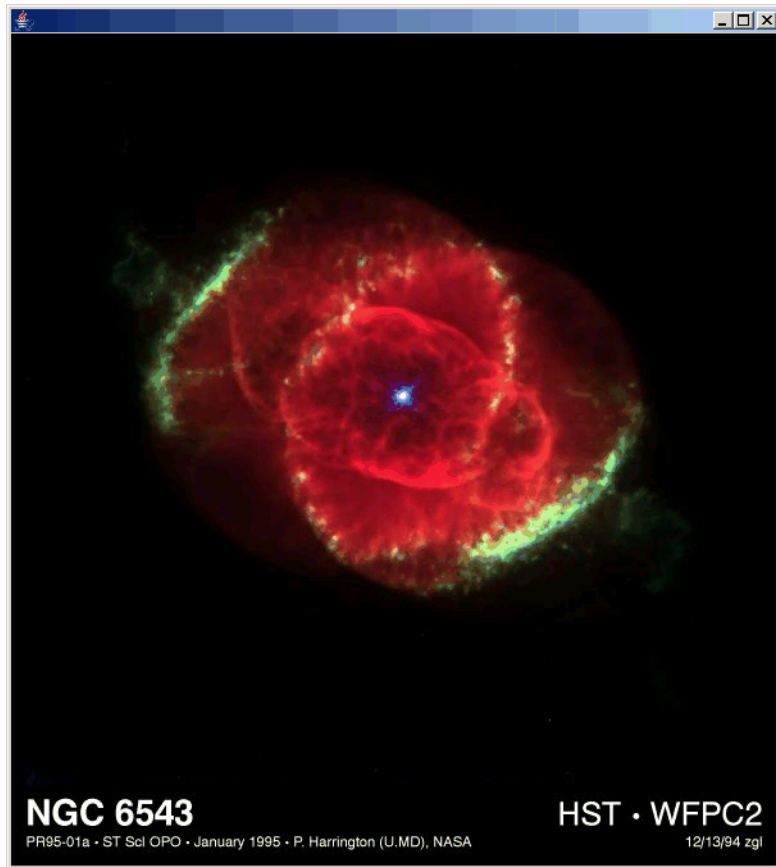
**Note** Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or 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**             $Y = \text{imag}(Z)$

**Description**        $Y = \text{imag}(Z)$  returns the imaginary part of the elements of array  $Z$ .

**Examples**            $\text{imag}(2+3i)$

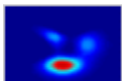
ans =

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

---

`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.

<b>Image Type</b>	<b>Double-Precision Data (double Array)</b>	<b>8-Bit Data (uint8 Array) 16-Bit Data (uint16 Array)</b>
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
```

## image

---

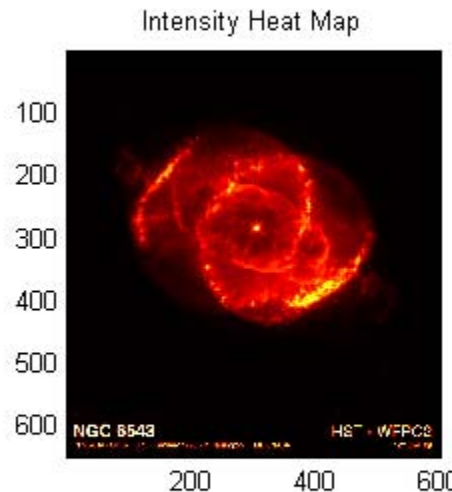
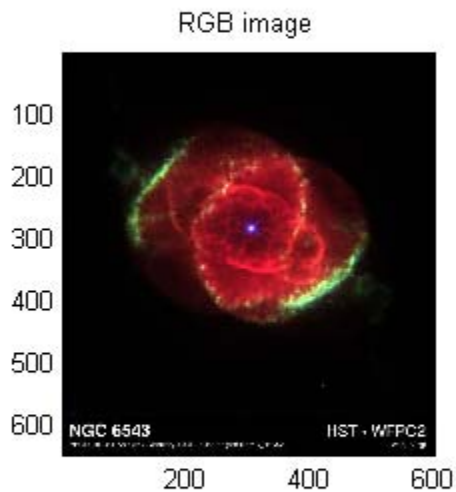


---

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')
```



# 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`)

# Image Properties

---

- 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:

<b>IconDisplayStyle Value</b>	<b>Purpose</b>
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

# Image Properties

---

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)

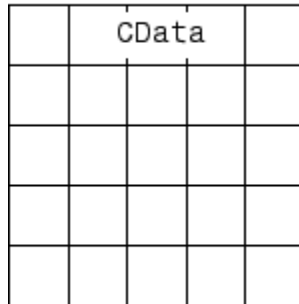
Note that the behavior of NaNs in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

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.

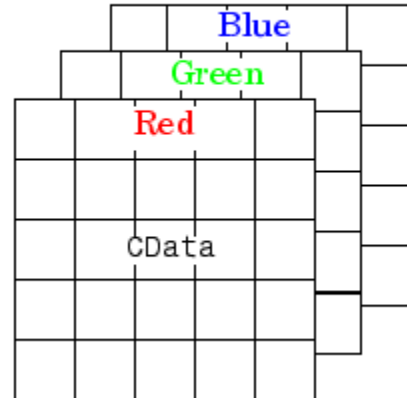
# Image Properties

---

Indexed Colors



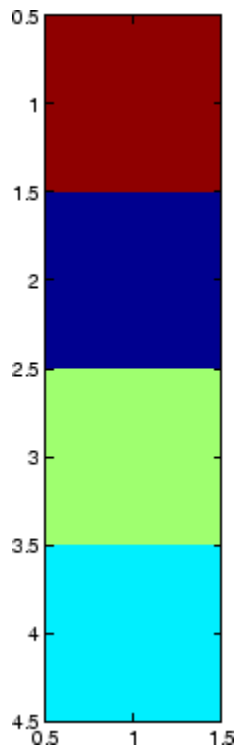
True Colors



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.

# Image Properties

---

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.

# Image Properties

---

- 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.

# Image Properties

---

## 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.

# Image Properties

---

- 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, hgroup, or hgtransform

*Parent of object.* Handle of the object’s parent. The parent is normally the axes, hgroup, 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)

# Image Properties

---

*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:

$$(XData(2) - XData(1)) / (\text{size}(CData, 2) - 1)$$

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:

$$(YData(2) - YData(1)) / (\text{size}(CData, 1) - 1)$$

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.

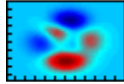
## See Also

`image`

# imagesc

---

**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 2-2201 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  $\text{length}(x) > 2$  or  $\text{length}(y) > 2$ , `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.

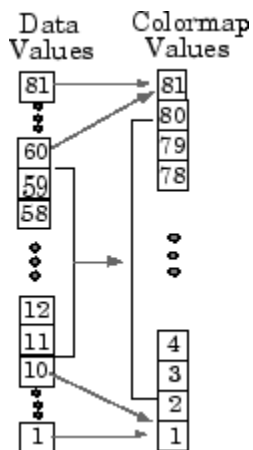
By default, `imagesc` plots the  $y$ -axis from lowest to highest value, top to bottom. To reverse this, type `set(gca, 'YDir', 'normal')`. This will reverse both the  $y$ -axis and the image.

## 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:



# imagesc

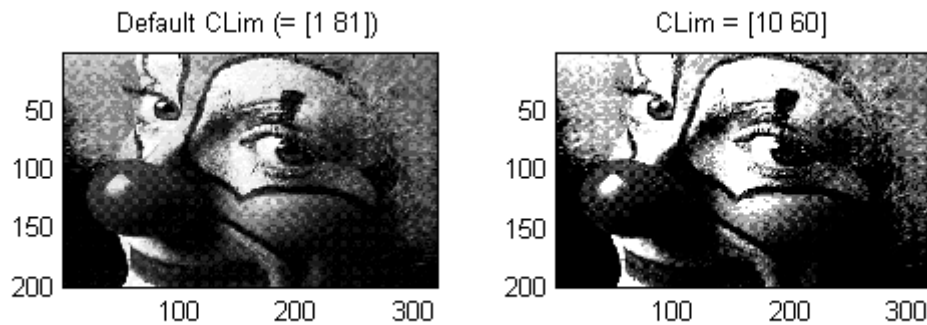
---

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]')
```

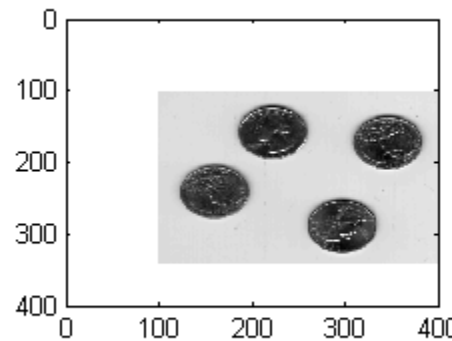
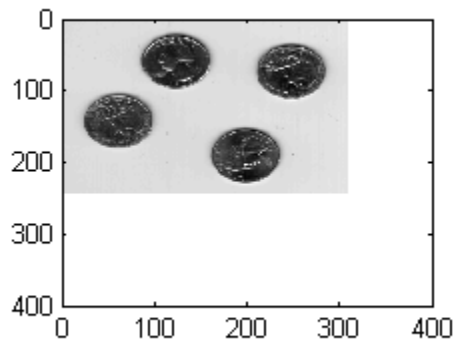


This example shows how to shift the image starting from origin to a position (100, 100),

```
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:



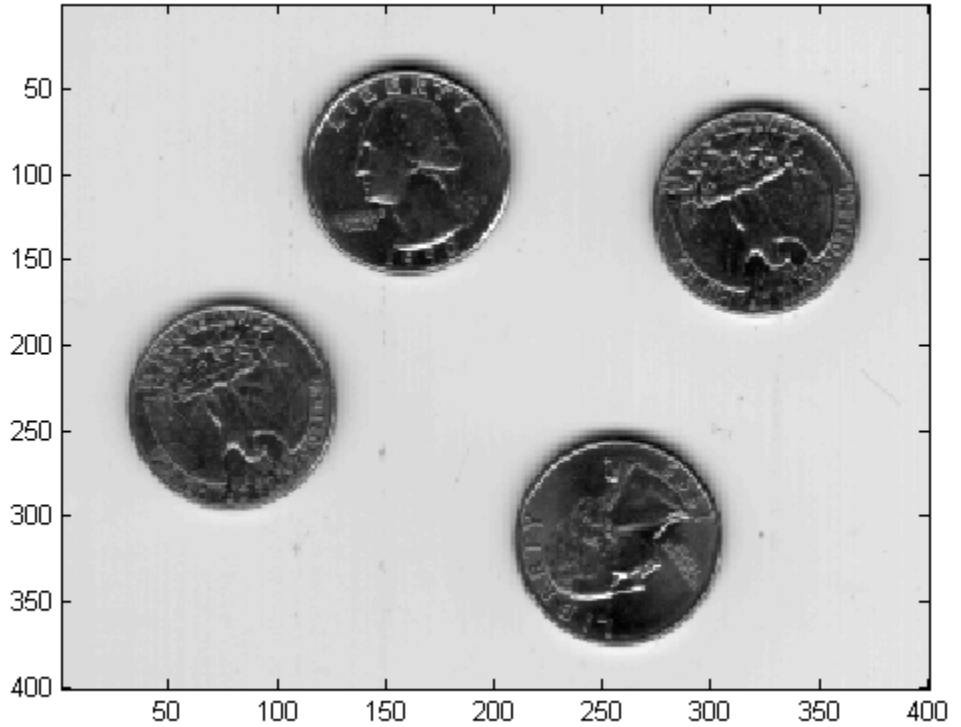
The top right corner of the image is now starting from (100,100) instead of the origin.

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);
```

# imagesc

---



## 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  $(\text{floor}(1/\text{tol})+1)^3$  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
```

# imapprox

---

```
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,fmt)
info = imfinfo(filename)
info = imfinfo(URL,...)
```

**Description** `info = imfinfo(filename,fmt)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current folder or in a folder on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`. The possible values for `fmt` are contained in the MATLAB file format registry. To view a list of these formats, run the `imformats` command.

If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, `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)` attempts to infer the format of the file from its contents.

`info = imfinfo(URL,...)` reads the image from the specified Internet URL. The URL must include the protocol type (e.g., `http://`)

### Information Returned

The set of fields in `info` depends on the individual file and its format. However, the first nine fields are always the same. This table lists these common fields, in the order they appear in the structure, and describes their values. “Format-Specific Notes” on page 2-2210 contains information about some fields returned by certain formats.

Field	Value
Filename	A string containing the name of the file or the Internet URL specified; if the file is not in the current folder, the string contains the full pathname of the file
FileModDate	A string containing the date when the file was last modified
FileSize	An integer indicating the size of the file in bytes
Format	A string containing the file format, as specified by <i>fmt</i> ; for formats with more than one possible extension (e.g. JPEG and TIFF files), <code>imfinfo</code> returns the first variant in the file format registry.
FormatVersion	A string or number describing the file format version
Width	An integer indicating the width of the image in pixels
Height	An integer indicating the height of the image in pixels
BitDepth	An integer indicating the number of bits per pixel
ColorType	A string indicating the type of image; this can include, but is not limited to, 'truecolor' for a truecolor (RGB) image, 'grayscale' for a grayscale intensity image, or 'indexed' for an indexed image

## Format-Specific Notes

- **JPEG and TIFF only** — If `filename` contains Exchangeable Image File Format (EXIF) tags, the `info` structure returned by `imfinfo` 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.

**Examples**

```
info = imfinfo('ngc6543a.jpg')

info =

    Filename: [1x95 char]
  FileModDate: '01-Oct-1996 17:19:44'
    FileSize: 27387
      Format: 'jpg'
  FormatVersion: ''
        Width: 600
        Height: 650
    BitDepth: 24
    ColorType: 'truecolor'
  FormatSignature: ''
  NumberOfSamples: 3
    CodingMethod: 'Huffman'
  CodingProcess: 'Sequential'
        Comment: {[1x69 char]}
```

**See Also**

```
imformats | imread | imwrite
```

# imformats

---

**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
<code>ext</code>	A cell array of strings that specify filename extensions that are valid for this format
<code>isa</code>	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
<code>info</code>	A string specifying the name of the function that reads information about a file. This can also be a function handle.
<code>read</code>	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
<code>write</code>	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.
<code>alpha</code>	Returns 1 if the format has an alpha channel, 0 otherwise
<code>description</code>	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'}
```

# imformats

---

```
isa: @isbmp
info: @imbmpinfo
read: @readbmp
write: @writebmp
alpha: 0
description: 'Windows Bitmap (BMP)'
```

## See Also

[imfinfo](#) | [imread](#) | [imwrite](#) | [path](#)

**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.

# import

---

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.*
```

`import` 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');
```

# import

---

```
printer = getPrinterInfo;
```

## **See Also**

`clear` | `load` | `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`

# importdata

---

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'`

## **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:** `' '`

## **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	<code>data</code> <code>textdata</code> <code>colheaders</code> <code>rowheaders</code>	For ASCII files, <code>data</code> contains a double array. Other fields contain cell arrays of strings. <code>textdata</code> includes row and column headers. For spreadsheets, each field contains a struct, with one field for each worksheet.
Images	<code>cdata</code> <code>colormap</code> <code>alpha</code>	See <code>imread</code> .
Audio files	<code>data</code> <code>fs</code>	See <code>auread</code> or <code>wavread</code> .

The MATLAB helper functions for all 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
```

# importdata

---

```
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”

# imread

---

**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 2-2227 for a discussion of bitdepths, and see “Format-Specific Information” on page 2-2227 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 2-2227.

## 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 2-2228	“JPEG — Joint Photographic Experts Group” on page 2-2230	“PNG — Portable Network Graphics” on page 2-2233
“CUR — Cursor File” on page 2-2228	“JPEG 2000 — Joint Photographic Experts Group 2000” on page 2-2231	“PPM — Portable Pixmap” on page 2-2234
“GIF — Graphics Interchange Format” on page 2-2229	“PBM — Portable Bitmap” on page 2-2232	“RAS — Sun Raster” on page 2-2234

“HDF4 — Hierarchical Data Format” on page 2-2230	“PCX — Windows Paintbrush” on page 2-2232	“TIFF — Tagged Image File Format” on page 2-2234
“ICO — Icon File” on page 2-2230	“PGM — Portable Graymap” on page 2-2232	“XWD — X Window Dump” on page 2-2237

## 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	No Compression	Compression	Output Class
1-bit	y	–	logical
2-bit to 8-bit	y	–	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 2-2228

## 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 Bitdepths (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** Only 1- and 3-sample images are supported. Indexed JPEG 2000 images are not supported.

Supported Bitdepths (Bits-per-sample)	Lossy Compression	Lossless Compression	Output Class	Notes
1-bit	y	y	logical	Grayscale only
2- to 8-bit	y	y	uint8	Grayscale or RGB
9- to 16-bit	y	y	uint16	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 <code>imfinfo</code> function.
'PixelRegion'	{ROWS, COLS} — The <code>imread</code> function returns the sub-image specified by the boundaries in ROWS and

# imread

Parameter	Value
	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.

## 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
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

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

The following table lists the supported bit/sample and corresponding output classes for TIFF files.

Supported Bitdepths	Compression							Output Class	Notes
	None	Packbits	CCITT	RGB	ICCLAB	CIELAB	CMYK		
1-bit	y	y	y	-	-	-	-	logical	
8-bit	y	y	-	-	-	-	-	uint8	
12-bit	-	-	-	-	-	-	-	uint16	Grayscale or Indexed
16-bit	-	-	-	-	-	-	-	uint16	Grayscale or Indexed
24-bit	y	y	-	y	y	y	-	uint8	3 samples
32-bit	-	-	-	-	-	-	y	uint8	4 samples
36-bit	-	-	-	y	-	-	-	uint16	3 samples
48-bit	-	-	-	y	y	y	-	uint16	3 samples
64-bit	-	-	-	-	-	-	y	double	4 samples

---

**Note** `imread` supports 8-bit integral and 32-bit floating point tiled TIFF images, with any compression and color space combination listed above, and 32-bit IEEE floating point images.

---

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, <code>imread</code> reads the third image in the file. If you omit this argument, <code>imread</code> reads the first image in the file.
'Info'	Structure array returned by <code>imfinfo</code> .  Note: When reading images from a multi-image TIFF file, passing the output of <code>imfinfo</code> as the value of the 'Info' argument helps <code>imread</code> 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 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	XPixmaps	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 <code>imread</code>
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');
```

# imread

---

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`

# imwrite

---

## 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 2-2241.

*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 2-2242. 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 2-2242.



`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 2-2244.

## 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 2-2244.

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 <hr/> <b>Note</b> <code>imwrite</code> converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images. <hr/>

<b>Format</b>	<b>Full Name</b>	<b>Variants</b>
'jp2' or 'jpx'	“JPEG 2000 — Joint Photographic Experts Group 2000” on page 2-2247	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 ZPixmap

## Format-Specific GIF – Graphics Interchange Format Parameters

Format-specific parameters.

Parameter	Values
'BackgroundColor'	A scalar integer that specifies which index in the colormap should be treated as the transparent color for the image and is used for certain disposal methods in animated GIFs. If X is <code>uint8</code> or <code>logical</code> , indexing starts at 0. If X is <code>double</code> , 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'	<p>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.</p> <p>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.</p>
'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 <i>quality</i> (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 2-2247.

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. <code>imwrite</code> 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 2-2246.

Parameter	Values	Default
'Comment'	A column vector cell array of strings or a character matrix. <code>imwrite</code> 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
'ReductionLevel'	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 <code>uint8</code> , <code>uint16</code> , or <code>double</code> , in which case the values should be in the range <code>[0,1]</code> .
'Author'	A string
'Background'	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range <code>[1,P]</code> , where <code>P</code> is the colormap length. For grayscale images: a scalar in the range <code>[0,1]</code> . For truecolor images: a three-element vector in the range <code>[0,1]</code> .
'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, <code>imwrite</code> uses 8 bits per pixel, if image is <code>double</code> or <code>uint8</code> ; 16 bits per pixel if image is <code>uint16</code> ; 1 bit per pixel if image is <code>logical</code> .

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 <code>datenum</code> function) or a string convertible to a date vector via the <code>datevec</code> function. Values should be in Coordinated Universal Time (UTC).
'InterlaceType'	Either 'none' (the default) or 'adam7'
'ResolutionUnit'	Either 'unknown' or 'meter'
'SignificantBits'	<p>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].</p> <p>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.</p>
'Software'	A string
'Source'	A string

Parameter	Values
'Transparency'	<p>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.)</p> <p>For indexed images: a <math>Q</math>-element vector in the range <math>[0,1]</math>, where <math>Q</math> is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, <math>Q = 1</math>.</p> <p>For grayscale images: a scalar in the range <math>[0,1]</math>. The value indicates the grayscale color to be considered transparent.</p> <p>For truecolor images: a three-element vector in the range <math>[0,1]</math>. The value indicates the truecolor color to be considered transparent.</p> <hr/> <p><b>Note</b> You cannot specify 'Transparency' and 'Alpha' at the same time.</p> <hr/>
'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 <code>uint8</code> , <code>uint16</code> , or <code>double</code> . 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' <sup>1</sup> (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'

<sup>1</sup>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`

# TriRep.incenters

---

<b>Purpose</b>	Incenters of specified simplices	
<b>Syntax</b>	IC = inceters(TR,SI) [IC RIC] = inceters(TR, SI)	
<b>Description</b>	IC = inceters(TR,SI) returns the coordinates of the incenter of each specified simplex SI.  [IC RIC] = inceters(TR, SI) returns the incenters and the corresponding radius of the inscribed circle/sphere.	
<b>Input Arguments</b>	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.
<b>Output Arguments</b>	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.
<b>Definitions</b>	A simplex is a triangle/tetrahedron or higher-dimensional equivalent.	
<b>Examples</b>	<b>Example 1</b>	
	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.

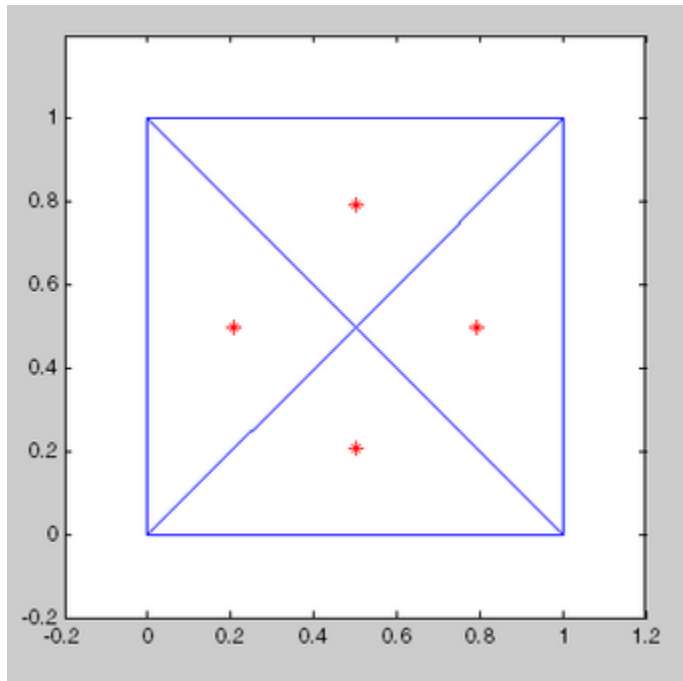
```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

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

`DelaunayTri` | `circumcenters`

**Purpose** Status of triangles in 2-D constrained Delaunay triangulation

**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];
```

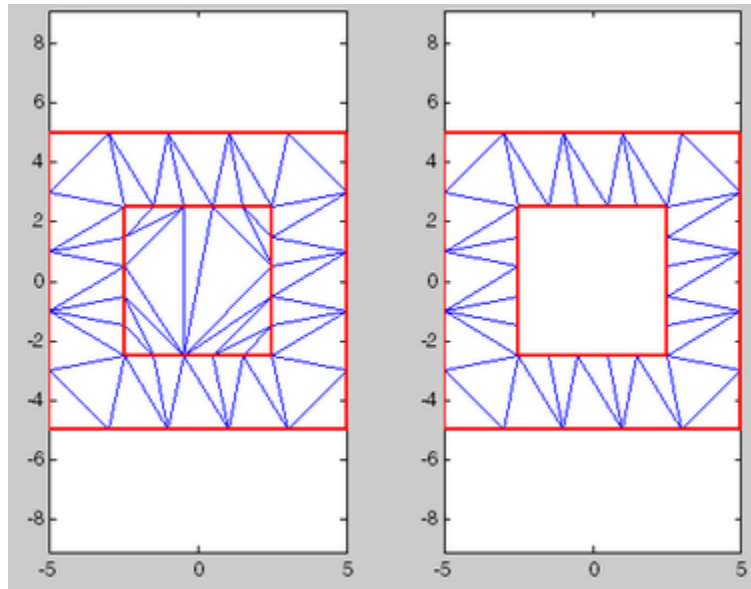
## DelaunayTri.inOutStatus

---

```
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;
```



# ind2rgb

---

<b>Purpose</b>	Convert indexed image to RGB image
<b>Syntax</b>	<code>RGB = ind2rgb(X,map)</code>
<b>Description</b>	<code>RGB = ind2rgb(X,map)</code> converts the matrix <code>X</code> and corresponding colormap <code>map</code> to RGB (truecolor) format.
<b>Class Support</b>	<code>X</code> can be of class <code>uint8</code> , <code>uint16</code> , <code>single</code> , or <code>double</code> . <code>RGB</code> is an <code>m-by-n-by-3</code> array of class <code>double</code> .
<b>See Also</b>	<code>image</code>   <code>imread</code>

**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)`.

---

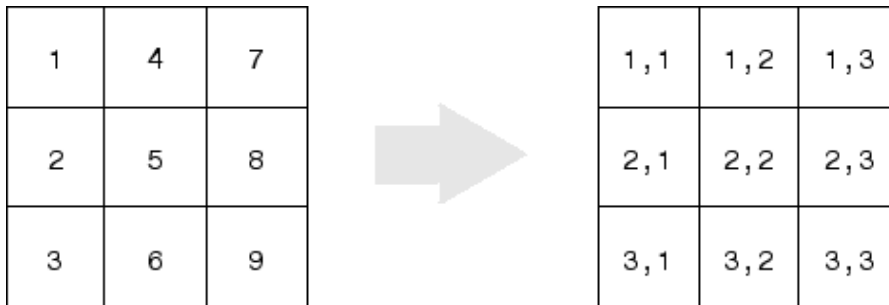
`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1,I2,...,In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

**Examples****Example 1 – Two-Dimensional Matrices**

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

# ind2sub

---



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)
```

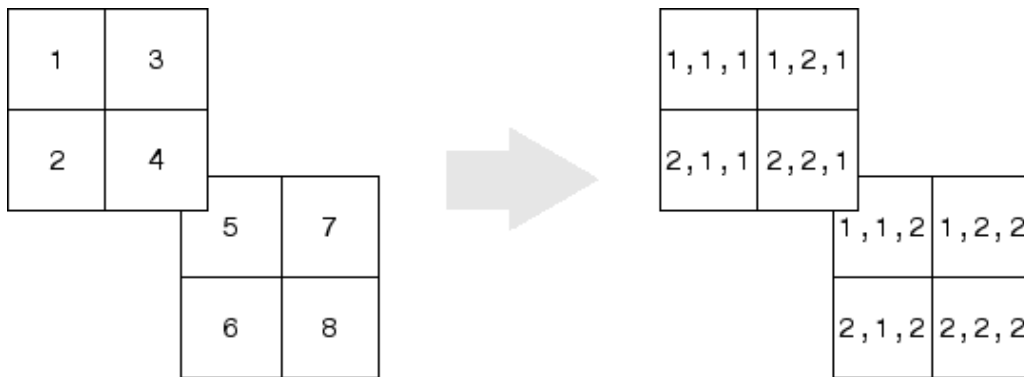
```
I =
     3     1     2     3
```

```
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.

# ind2sub

---

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](#)

**Purpose**

Infinity

**Syntax**

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
```

**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 Infs.

Inf(m,n) or inf([m,n]) is an m-by-n matrix of Infs.

Inf(m,n,p,...) or Inf([m,n,p,...]) is an m-by-n-by-p-by-... array of Infs.

---

**Note** The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

---

Inf(...,classname) is an array of Infs of class specified by classname. classname must be either 'single' or 'double'.

**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).

# Inf

---

## See Also

`isinf` | `NaN`

**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 determine 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`

# info

---

**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

---

<b>Purpose</b>	Construct inline object
<b>Syntax</b>	<code>inline(expr)</code> <code>inline(expr, arg1, arg2, ...)</code> <code>inline(expr, n)</code>
<b>Description</b>	<p><code>inline(expr)</code> constructs an inline function object from the MATLAB expression contained in the string <code>expr</code>. The input argument to the inline function is automatically determined by searching <code>expr</code> for an isolated lower case alphabetic character, other than <code>i</code> or <code>j</code>, that is not part of a word formed from several alphabetic characters. If no such character exists, <code>x</code> is used. If the character is not unique, the one closest to <code>x</code> is used. If two characters are found, the one later in the alphabet is chosen.</p> <p><code>inline(expr, arg1, arg2, ...)</code> constructs an inline function whose input arguments are specified by the strings <code>arg1, arg2, ...</code>. Multicharacter symbol names may be used.</p> <p><code>inline(expr, n)</code> where <code>n</code> is a scalar, constructs an inline function whose input arguments are <code>x, P1, P2, ...</code>.</p>
<b>Tips</b>	<p>Three commands related to <code>inline</code> allow you to examine an inline function object and determine how it was created.</p> <p><code>char(fun)</code> converts the inline function into a character array. This is identical to <code>formula(fun)</code>.</p> <p><code>argnames(fun)</code> returns the names of the input arguments of the inline object <code>fun</code> as a cell array of strings.</p> <p><code>formula(fun)</code> returns the formula for the inline object <code>fun</code>.</p> <p>A fourth command <code>vectorize(fun)</code> inserts a <code>.</code> before any <code>^</code>, <code>*</code> or <code>/</code> in the formula for <code>fun</code>. The result is a vectorized version of the inline function.</p>
<b>Examples</b>	<b>Example 1</b> 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)
```

# inmem

---

**Purpose** Names of functions, MEX-files, Java classes in memory

**Syntax**

```
M = inmem
[M, X] = inmem
[M, X, J] = 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, J] = inmem` also returns a cell array `J` containing the names of the Java 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 Java classes.

## Examples

Example 1

This example lists the functions that are required to run `erf`.

```
clear all;           % Clear the workspace
erf(0.5);
```

```
M = inmem
M =
    'erf'
```

### Example 2

Generate a plot, and then find the function and MEX-files that had been loaded to perform this operation:

```
clear all
surf(peaks)
```

```
[m x] = inmem('-completenames');  
  
m{1:5}  
ans =  
    F:\matlab\toolbox\matlab\general\usejava.m  
ans =  
    F:\matlab\toolbox\matlab\graph3d\private\surfchk.m  
ans =  
    F:\matlab\toolbox\matlab\graphics\gcf.m  
ans =  
    F:\matlab\toolbox\matlab\datatypes\@opaque\char.m  
ans =  
    F:\matlab\toolbox\matlab\graphics\findall.m  
  
x  
x =  
    Empty cell array: 0-by-1
```

**See Also**

```
clear
```

# inpolygon

---

**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

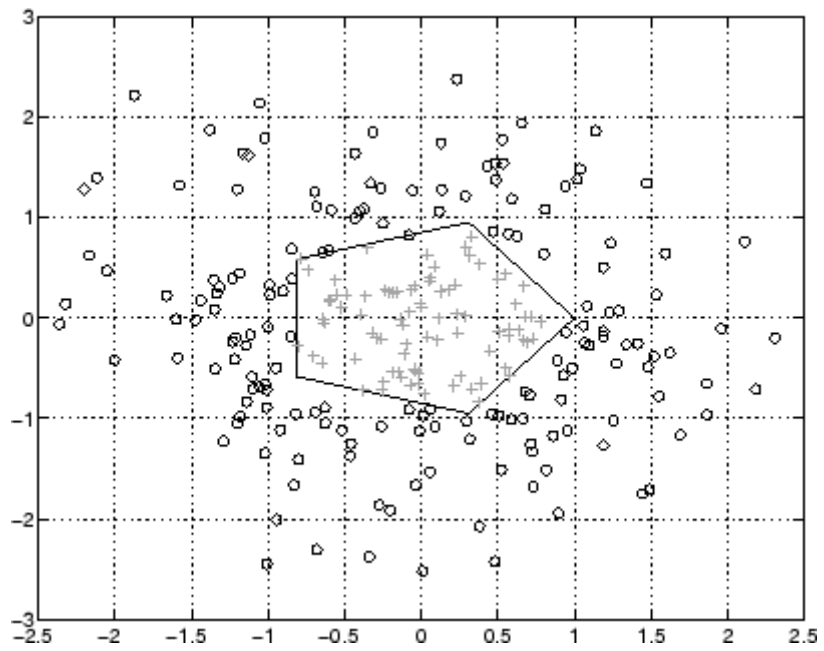
`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point  $(X(p,q), Y(p,q))$  is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

$ON(p,q) = 1$  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')
```



# input

---

**Purpose** Request user input

**Syntax** `evalResponse = input(prompt)`  
`strResponse = input(prompt, 's')`

**Description** `evalResponse = input(prompt)` displays the *prompt* string on the screen, waits for input from the keyboard, evaluates any expressions in the input, and returns the value in *evalResponse*. To evaluate expressions, the `input` function accesses variables in the current workspace.

`strResponse = input(prompt, 's')` returns the entered text as a MATLAB string, without evaluating expressions.

- Tips**
- If you press the **Return** key without entering anything, `input` returns an empty matrix.
  - To create a prompt that spans several lines, use `'\n'` to indicate each new line. To include a backslash (`'\'`) in the prompt, use `'\\'`.
  - If you enter an invalid expression at the prompt, MATLAB displays the relevant error message and then redisplay the prompt.

**Examples** Request a text response. Assign a default value ('Y') by checking for an empty matrix.

```
reply = input('Do you want more? Y/N [Y]: ', 's');  
if isempty(reply)  
    reply = 'Y';  
end
```

**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 `WindowState` 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 matrix.

- 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 a matrix, it should 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.

# inputdlg

---

```
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:

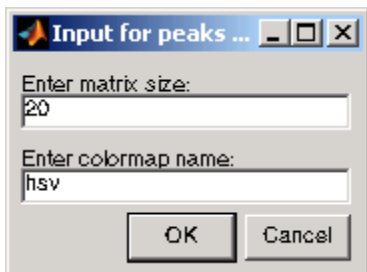
```
val = str2num(answer{1})
val =
    1.0000    2.0000    3.0000
    4.0000   -5.0000    6.0000 + 7.0000i
```

## 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 for peaks function';
num_lines = 1;
def = {'20', 'hsv'};
answer = inputdlg(prompt, dlg_title, num_lines, def);
```



### Example 2

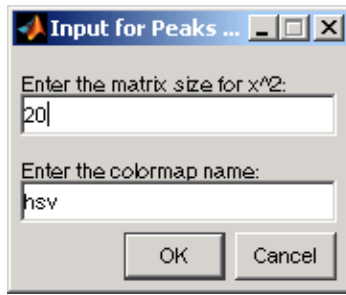
Create a dialog box using the default options. Then use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:', ...
        'Enter the colormap name:'};
name='Input for Peaks function';
numlines=1;
```

# inputdlg

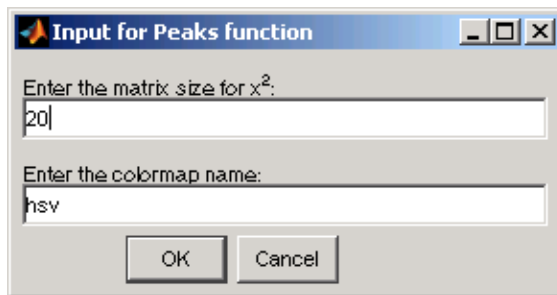
---

```
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` | `nargchk`

# inputParser

---

**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:

- |                        |                                                   |
|------------------------|---------------------------------------------------|
| <code>false (0)</code> | Names are not sensitive to case: 'a' matches 'A'. |
| <code>true (1)</code>  | 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:

- |                        |                                                                                                                                                                 |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>false (0)</code> | Throw an error whenever inputs are not in the scheme.                                                                                                           |
| <code>true (1)</code>  | Store the parameter names and values of unmatched inputs in the <code>Unmatched</code> property of the <code>inputParser</code> 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.

# inputParser

---

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.

<code>true</code> (1)	Expand structures into separate inputs. Each field name corresponds to an input parameter name.
<code>false</code> (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**

<code>addOptional</code>	Add optional argument to Input Parser scheme
<code>addParamValue</code>	Add parameter name and value argument to Input Parser scheme

addRequired	Add required argument to Input Parser scheme
createCopy	Create copy of inputParser object
parse	Parse function inputs

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

The createCopy method allows you to create another scheme with the same properties as an existing inputParser object.

## 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. @

# inputParser

---

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
```

# inputParser

---

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](#) | [nargchk](#) | [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.

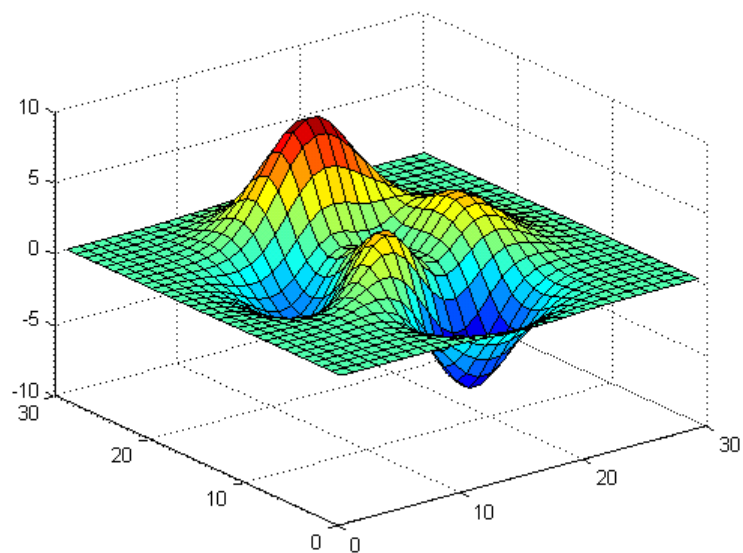
---

## Examples

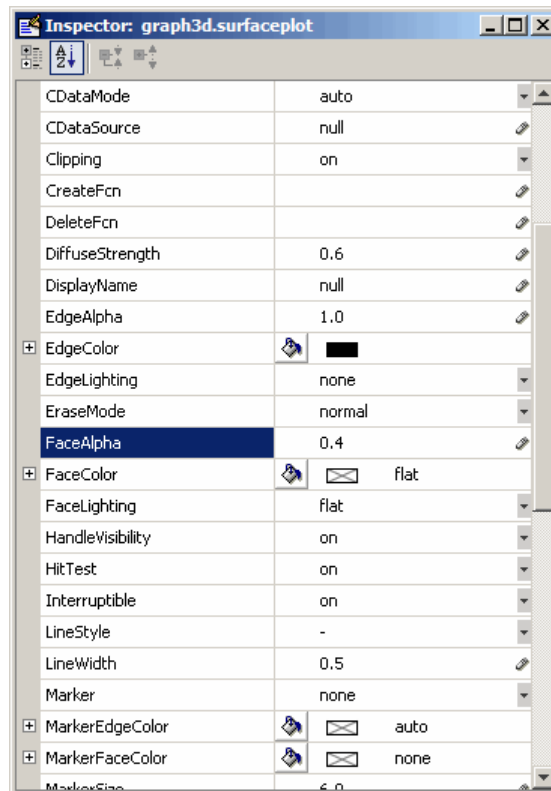
### Example 1

Create a surface mesh plot and view its properties with the Property Inspector:

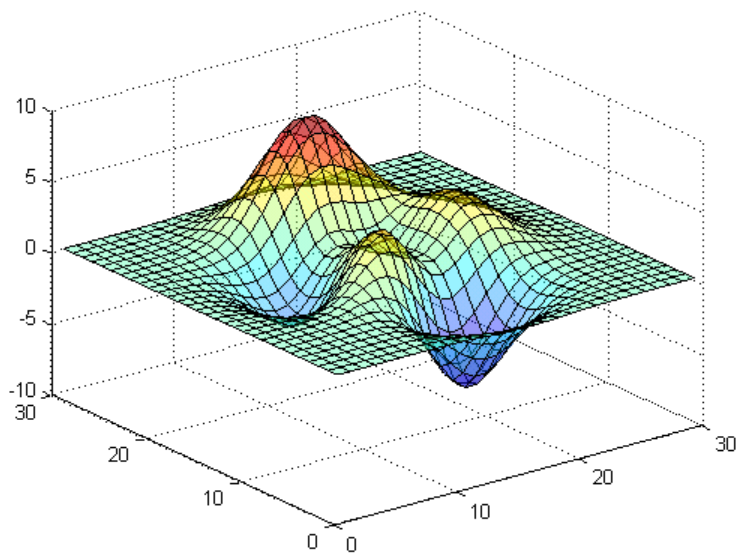
```
Z = peaks(30);  
h = surf(Z)  
inspect(h)
```



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.



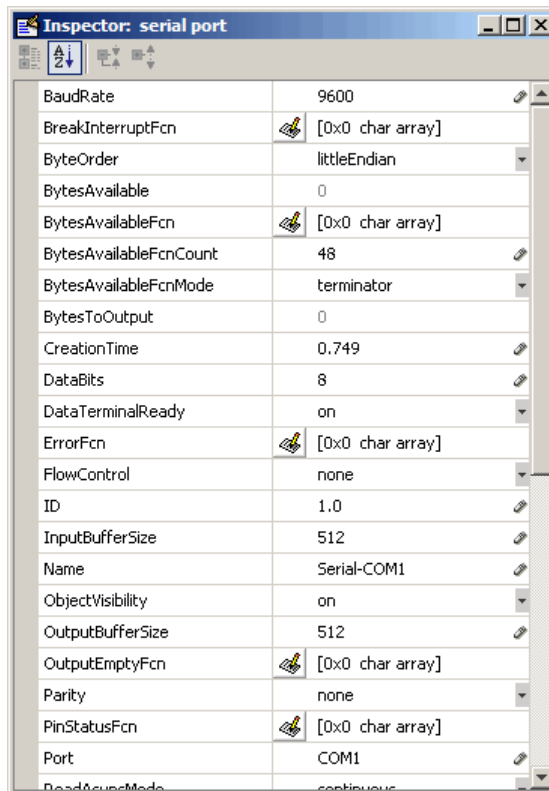
When you press **Enter** or click a different field, the FaceAlpha property of the surface object is updated:



### 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

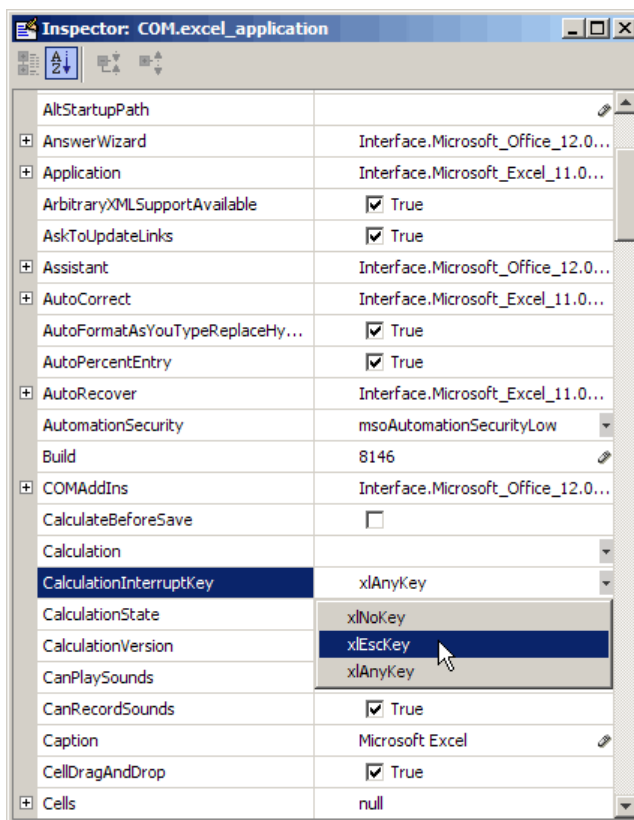
Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');  
inspect(h)
```

Scroll down until you see the `CalculationInterruptKey` property, which by default is `x1AnyKey`. 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
```

# inspect

---

## **See Also**

`get` | `set` | `isprop` | `guide` | `addproperty` | `deletproperty`

<b>Purpose</b>	Event information when event occurs
<b>Syntax</b>	<code>instrcallback(obj,event)</code>
<b>Description</b>	<p><code>instrcallback(obj,event)</code> displays a message that contains the event type, event, the time the event occurred, and the name of the serial port object, <code>obj</code>, that caused the event to occur.</p> <p>For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.</p>
<b>Tips</b>	You should use <code>instrcallback</code> as a template from which you create callback functions that suit your specific application needs.
<b>Examples</b>	<p>The following example creates the serial port objects <code>s</code>, on a Windows platform. It configures <code>s</code> to execute <code>instrcallback</code> when an output-empty event occurs. The event occurs after the <code>*IDN?</code> command is written to the instrument.</p> <pre>s = serial('COM1'); set(s, 'OutputEmptyFcn', @instrcallback) fopen(s) fprintf(s, '*IDN?', 'async')</pre> <p>The resulting display from <code>instrcallback</code> is shown below.</p> <pre>OutputEmpty event occurred at 08:37:49 for the object: Serial-COM1.</pre> <p>Read the identification information from the input buffer and end the serial port session.</p> <pre>idn = fscanf(s); fclose(s) delete(s) clear s</pre>

# 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`



<b>Purpose</b>	Convert to 8-bit signed integer
<b>Syntax</b>	<code>intArray = int8(array)</code>
<b>Description</b>	<code>intArray = int8(array)</code> converts the elements of an array into signed 8-bit (1-byte) integers of class <code>int8</code> .
<b>Input Arguments</b>	<p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If <code>array</code> is already of class <code>int8</code>, the <code>int8</code> function has no effect.</p>
<b>Output Arguments</b>	<p><b>intArray</b></p> <p>Array of class <code>int8</code>. Values range from <math>-2^7</math> to <math>2^7 - 1</math>.</p> <p>The <code>int8</code> function maps any values in <code>array</code> that are outside the limit to the nearest endpoint. For example,</p> <pre>int8(2^7)    % 2^7 = 128</pre> <p>returns</p> <pre>ans =     127</pre>
<b>Examples</b>	<p>Convert a double array to <code>int8</code>:</p> <pre>mydata = int8(magic(10));</pre>
<b>Alternatives</b>	<p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = int8(zeros(100));    % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'int8'); % Preferred</pre>

# int8

---

## See Also

[double](#) | [single](#) | [int16](#) | [int32](#) | [int64](#) | [uint8](#) | [uint16](#) | [uint32](#) | [uint64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)

<b>Purpose</b>	Convert to 16-bit signed integer
<b>Syntax</b>	<code>intArray = int16(array)</code>
<b>Description</b>	<code>intArray = int16(array)</code> converts the elements of an array into signed 16-bit (2-byte) integers of class <code>int16</code> .
<b>Input Arguments</b>	<p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If <code>array</code> is already of class <code>int16</code>, the <code>int16</code> function has no effect.</p>
<b>Output Arguments</b>	<p><b>intArray</b></p> <p>Array of class <code>int16</code>. Values range from <math>-2^{15}</math> to <math>2^{15} - 1</math>.</p> <p>The <code>int16</code> function maps any values in <code>array</code> that are outside the limit to the nearest endpoint. For example,</p> <pre>int16(2^15)    % 2^15 = 32768</pre> <p>returns</p> <pre>ans =     32767</pre>
<b>Examples</b>	<p>Convert a double array to <code>int16</code>:</p> <pre>mydata = int16(magic(100));</pre>
<b>Alternatives</b>	<p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = int16(zeros(100));    % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'int16'); % Preferred</pre>

# int16

---

## See Also

[double](#) | [single](#) | [int8](#) | [int32](#) | [int64](#) | [uint8](#) | [uint16](#) | [uint32](#) | [uint64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)

<b>Purpose</b>	Convert to 32-bit signed integer
<b>Syntax</b>	<code>intArray = int32(array)</code>
<b>Description</b>	<code>intArray = int32(array)</code> converts the elements of an array into signed 32-bit (4-byte) integers of class <code>int32</code> .
<b>Input Arguments</b>	<p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If array is already of class <code>int32</code>, the <code>int32</code> function has no effect.</p>
<b>Output Arguments</b>	<p><b>intArray</b></p> <p>Array of class <code>int32</code>. Values range from <math>-2^{31}</math> to <math>2^{31} - 1</math>.</p> <p>The <code>int32</code> function maps any values in array that are outside the limit to the nearest endpoint. For example,</p> <pre>int32(2^31)    % 2^31 = 2147483648</pre> <p>returns</p> <pre>ans =     2147483647</pre>
<b>Examples</b>	<p>Convert a double array to <code>int32</code>:</p> <pre>mydata = int32(magic(100));</pre>
<b>Alternatives</b>	<p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = int32(zeros(100));    % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'int32'); % Preferred</pre>

# int32

---

## See Also

[double](#) | [single](#) | [int8](#) | [int16](#) | [int64](#) | [uint8](#) | [uint16](#) | [uint32](#) | [uint64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)

<b>Purpose</b>	Convert to 64-bit signed integer
<b>Syntax</b>	<code>intArray = int64(array)</code>
<b>Description</b>	<code>intArray = int64(array)</code> converts the elements of an array into signed 64-bit (8-byte) integers of class <code>int64</code> .
<b>Tips</b>	<p>Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, <code>double</code> values cannot represent all integers greater than <math>2^{53}</math> correctly. Before performing arithmetic operations on values larger than <math>2^{53}</math> in magnitude, convert the values to 64-bit integers. For example,</p> <pre>x = int64(2^53+1);    % Floating-point arithmetic, loses precision</pre> <p>is not as accurate as the 64-bit integer arithmetic operation:</p> <pre>x = int64(2^53) + 1;    % Preferred</pre>
<b>Input Arguments</b>	<p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If <code>array</code> is already of class <code>int64</code>, the <code>int64</code> function has no effect.</p>
<b>Output Arguments</b>	<p><b>intArray</b></p> <p>Array of class <code>int64</code>. Values range from <math>-2^{63}</math> to <math>2^{63} - 1</math>.</p> <p>The <code>int64</code> function maps any values in <code>array</code> that are outside the limit to the nearest endpoint. For example,</p> <pre>int64(2^63)    % 2^63 = 9223372036854775808</pre> <p>returns</p> <pre>ans =     9223372036854775807</pre>

# int64

---

## 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, 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`.

**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 2-2316 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:

$$\text{abs}(q - Q) \leq \max(\text{AbsTol}, \text{RelTol} * \text{abs}(q))$$

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  $z = \text{fun}(x,y)$  over the planar region  $x_{\min} \leq x \leq x_{\max}$  and  $y_{\min}(x) \leq y \leq y_{\max}(x)$ .

`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  $x_{\min} \leq x \leq x_{\max}$  and  $y_{\min}(x) \leq y \leq y_{\max}(x)$ . 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, 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`.

**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 2-2324 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 2-2324 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, <code>integral2</code> uses the 'tiled' method. It uses the 'iterated' method when any of the integration limits are infinite. This is the default method.
'tiled'	<code>integral2</code> 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'	<code>integral2</code> calls <code>integral</code> to perform an iterated integral. The outer integral is evaluated over $x_{\min} \leq x \leq x_{\max}$ . The inner integral is evaluated over $y_{\min}(x) \leq y \leq y_{\max}(x)$ . The integration limits can be infinite.

**Example:** 'Method', 'tiled' specifies the tiled integration method.

## Output Arguments

### q - Computed integral

Computed integral of  $\text{fun}(x,y)$  over the specified region, returned as a numeric value.

# integral2

---

## Tips

- The `integral2` function attempts to satisfy:

$$\text{abs}(q - Q) \leq \max(\text{AbsTol}, \text{RelTol} * \text{abs}(q))$$

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  $\text{abs}(q)$  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 integrand 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 parameterizing 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  $0 \leq x \leq 1$  and  $0 \leq y \leq 1 - x$ .

```
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  $0 \leq \theta \leq \pi/2$  and  $0 \leq r \leq rmax$ .

```
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  $0 \leq x \leq 5$  and  $-5 \leq y \leq 0$ . 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  $z = \text{fun}(x,y,z)$  over the region  $x_{\min} \leq x \leq x_{\max}$ ,  $y_{\min}(x) \leq y \leq y_{\max}(x)$  and  $z_{\min}(x,y) \leq z \leq z_{\max}(x,y)$ .

`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  $x_{\min} \leq x \leq x_{\max}$ ,  $y_{\min}(x) \leq y \leq y_{\max}(x)$ , and  $z_{\min}(x,y) \leq z \leq z_{\max}(x,y)$ . 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, 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`.

**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 2-2331 section.

---

**Example:** `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

# integral3

---

## 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 2-2331 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, <code>integral3</code> uses the 'tiled' method. It uses the 'iterated' method when any of the integration limits are infinite. This is the default method.
'tiled'	<code>integral3</code> calls <code>integral</code> to integrate over $x_{\min} \leq x \leq x_{\max}$ . It calls <code>integral2</code> with the 'tiled' method to evaluate the double integral over $y_{\min}(x) \leq y \leq y_{\max}(x)$ and $z_{\min}(x,y) \leq z \leq z_{\max}(x,y)$ .
'iterated'	<code>integral3</code> calls <code>integral</code> to integrate over $x_{\min} \leq x \leq x_{\max}$ . It calls <code>integral2</code> with the 'iterated' method to evaluate the double integral over $y_{\min}(x) \leq y \leq y_{\max}(x)$ and $z_{\min}(x,y) \leq z \leq z_{\max}(x,y)$ . 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:

$$\text{abs}(q - Q) \leq \max(\text{AbsTol}, \text{RelTol} * \text{abs}(q))$$

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.

# integral3

---

- 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 integrand 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 parameterizing 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  $0 \leq x \leq \pi$ ,  $0 \leq y \leq 1$ , and  $-1 \leq z \leq 1$ .

```
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  $-\infty \leq x \leq 0$ ,  $-100 \leq y \leq 0$ , and  $-100 \leq z \leq 0$ .

```
format long
q1 = integral3(f,-Inf,0,-100,0,-100,0)
```

```
q1 =
```

```
2.734244598320928e+03
```

# integral3

---

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”

# interp1

---

**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(x, Y, ...)` where `x` is a vector and `Y` is an array representing multiple value sets.
- `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  $[n, d1, d2, \dots, dk]$ , where  $n$  is the length of  $x$ . The interpolation is performed for each  $d1$ -by- $d2$ -by-...- $dk$  value in  $Y$ . The sizes of  $x_i$  and  $y_i$  are related as follows:
  - If  $x_i$  is a scalar or vector,  $\text{size}(y_i)$  equals  $[\text{length}(x_i), d1, d2, \dots, dk]$ .
  - If  $x_i$  is an array of size  $[m1, m2, \dots, mj]$ ,  $y_i$  has size  $[m1, m2, \dots, mj, d1, d2, \dots, dk]$ .

$y_i = \text{interp1}(Y, x_i)$  assumes that  $x = 1:N$ , where  $N$  is the length of  $Y$  for vector  $Y$ , or  $\text{size}(Y, 1)$  for matrix  $Y$ .

$y_i = \text{interp1}(x, Y, x_i, \text{method})$  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,  $\text{interp1}(x, Y, x_i, \text{method})$  returns NaN for any element of  $x_i$  that is outside the interval spanned by  $x$ . For all other methods,  $\text{interp1}$  performs extrapolation for out of range values.

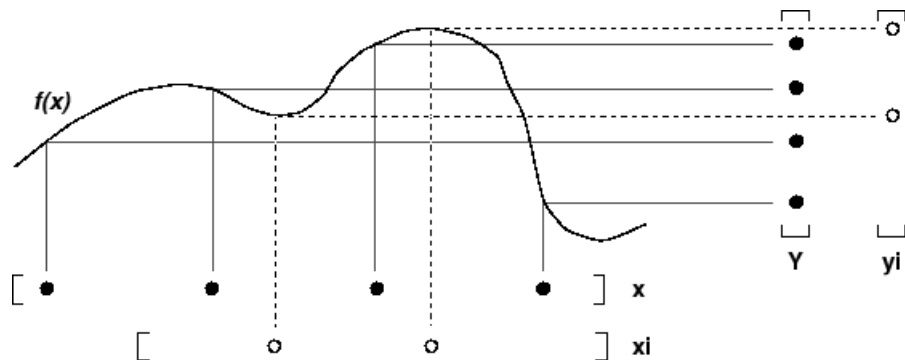
$y_i = \text{interp1}(x, Y, x_i, \text{method}, \text{'extrap'})$  uses the specified interpolation algorithm specified by  $\text{method}$  to perform extrapolation for out of range values.

$y_i = \text{interp1}(x, Y, x_i, \text{method}, \text{extrapval})$  returns the scalar  $\text{extrapval}$  for out of range values. NaN and 0 are often used for  $\text{extrapval}$ .

# interp1

`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`.



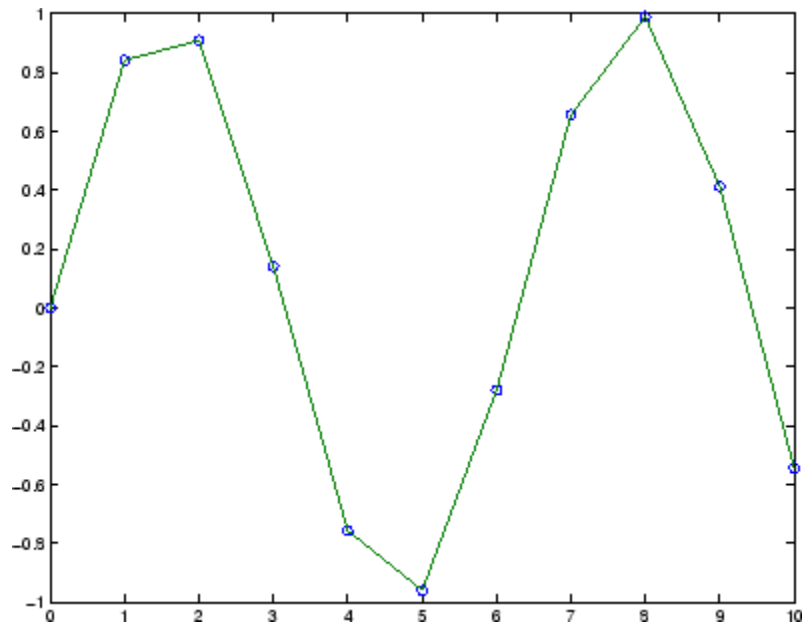
Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is `[x,Y]` and `interp1` *looks up* the elements of `xi` in `x`, and, based upon their locations, returns values `yi` interpolated within the elements of `Y`.

## 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

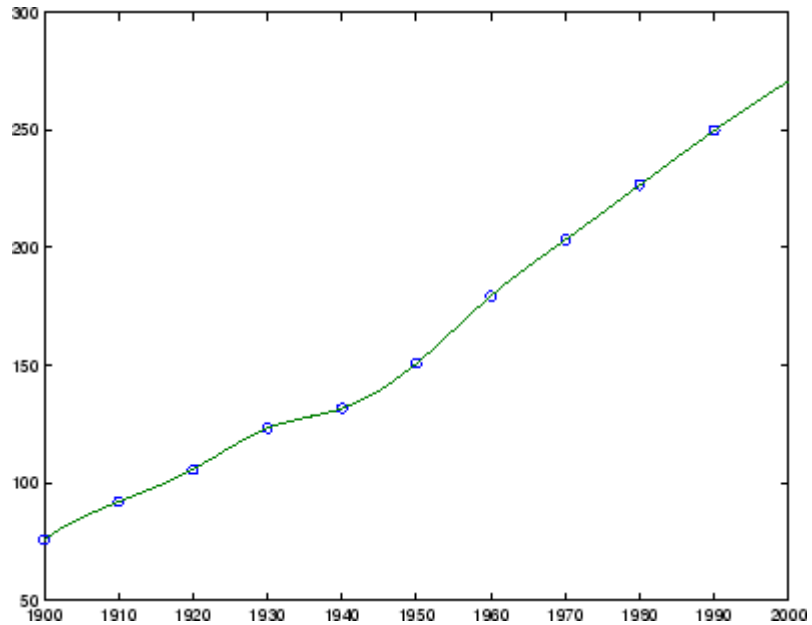
```
ans =
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```
x = 1900:1:2000;
```

# interp1

```
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.

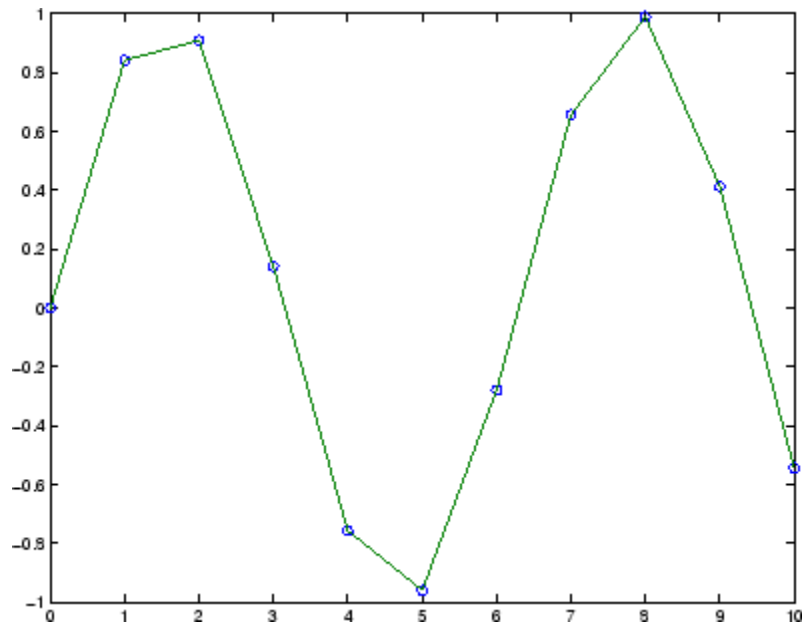
For `interp1q` to work properly,

- `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](#)

# interp2

---

**Purpose** 2-D data interpolation (table lookup)

---

**Note** In a future release, `interp2` will not accept mixed combinations of row and column vectors for the sample and query grids. In addition, the following syntaxes will be removed or changed in a future release:

- `interp2(..., 'cubic')`
- `interp2(..., *METHOD)`

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 `NaNs`.

`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
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. <code>interp2</code> switches to 'spline' and issues a warning if the data is not equally spaced.

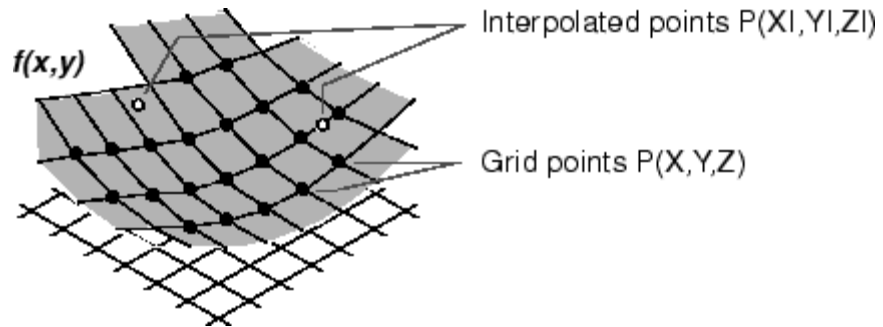
All interpolation methods require that `X` and `Y` be monotonic, and have the same format (“plaid”) as if they were produced by `meshgrid`. If you provide two monotonic vectors, `interp2` changes them to a plaid internally. Variable spacing is handled by mapping the given values in `X`, `Y`, `XI`, and `YI` to an equally spaced domain before interpolating. For faster interpolation when `X` and `Y` are equally spaced and monotonic, use the methods `'*linear'`, `'*cubic'`, `'*spline'`, or `'*nearest'`.

`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.

# interp2



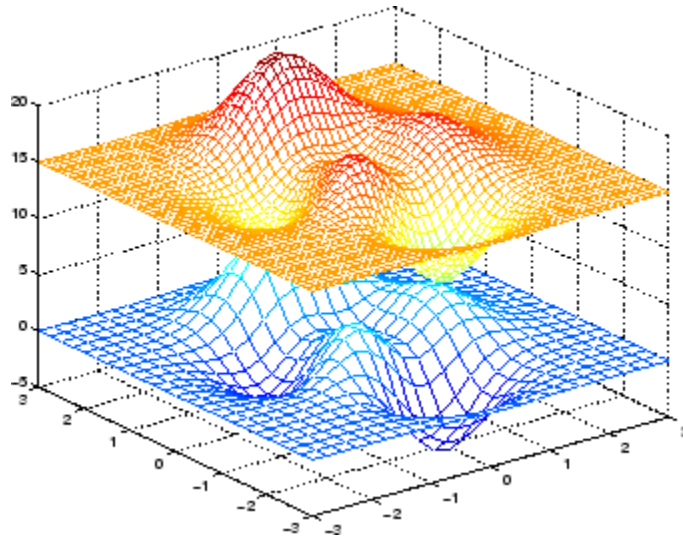
Interpolation is the same operation as table lookup. Described in table lookup terms, the table is  $\text{tab} = [\text{NaN}, Y; X, Z]$  and `interp2` looks up the elements of  $XI$  in  $X$ ,  $YI$  in  $Y$ , and, based upon their location, returns values  $ZI$  interpolated within the elements of  $Z$ .

## 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** 3-D data interpolation (table lookup)

---

**Note** In a future release, `interp3` will not accept mixed combinations of row and column vectors for the sample and query grids. In addition, the following syntaxes will be removed or changed in a future release:

- `interp3(..., 'cubic')`
- `interp3(..., *METHOD)`

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
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. <code>interp3</code> 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. For faster interpolation when `X`, `Y`, and `Z` are equally spaced and monotonic, use the methods '\*linear', '\*cubic', or '\*nearest'.

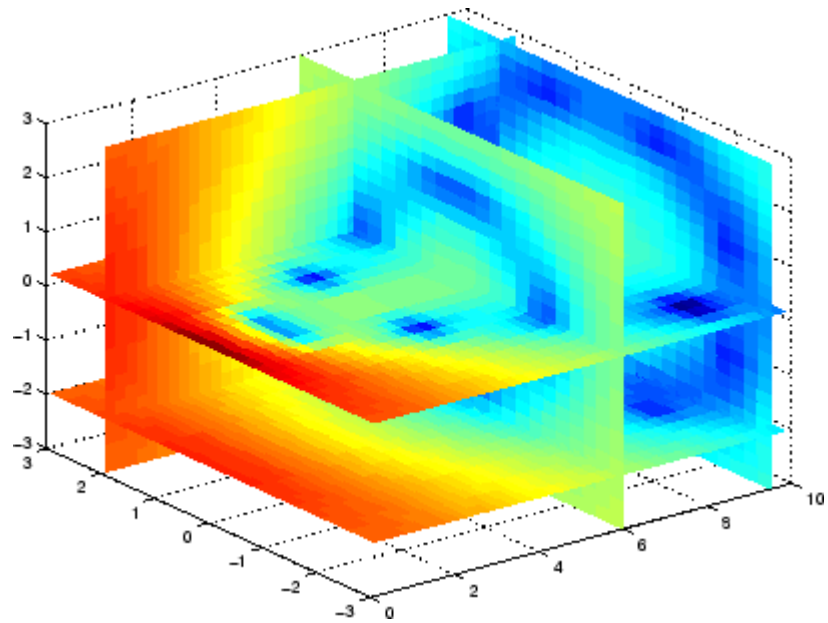
## Examples

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
```

# interp3

---



## See Also

`interp1` | `interp2` | `interpn` | `meshgrid`

<b>Purpose</b>	1-D interpolation using FFT method
<b>Syntax</b>	<pre>y = interpft(x,n) y = interpft(x,n,dim)</pre>
<b>Description</b>	<p><code>y = interpft(x,n)</code> returns the vector <code>y</code> that contains the value of the periodic function <code>x</code> resampled to <code>n</code> equally spaced points.</p> <p>If <code>length(x) = m</code>, and <code>x</code> has sample interval <code>dx</code>, then the new sample interval for <code>y</code> is <code>dy = dx*m/n</code>. Note that <code>n</code> cannot be smaller than <code>m</code>.</p> <p>If <code>X</code> is a matrix, <code>interpft</code> operates on the columns of <code>X</code>, returning a matrix <code>Y</code> with the same number of columns as <code>X</code>, but with <code>n</code> rows.</p> <p><code>y = interpft(x,n,dim)</code> operates along the specified dimension.</p>
<b>Algorithms</b>	The <code>interpft</code> command uses the FFT method. The original vector <code>x</code> is transformed to the Fourier domain using <code>fft</code> and then transformed back with more points.
<b>Examples</b>	<p>Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:</p> <pre>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);</pre> <p>Perform the interpolation:</p> <pre>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')</pre>
<b>See Also</b>	<code>interp1</code>

# interp

---

**Purpose** N-D data interpolation (table lookup)

---

**Note** In a future release, `interp` will not accept mixed combinations of row and column vectors for the sample and query grids. In addition, the following syntaxes will be removed or changed in a future release:

- `interp(..., 'cubic')`
- `interp(..., *METHOD)`

For more information, and recommendations for updating your code, see “Interpolation and Computational Geometry Functionality Being Removed or Changed”.

---

**Syntax**

```
VI = interp(X1,X2,X3,...,V,Y1,Y2,Y3,...)
VI = interp(V,Y1,Y2,Y3,...)
VI = interp(V,ntimes)
VI = interp(...,method)
VI = interp(...,method,extrapval)
```

**Description** `VI = interp(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`, `interp` 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 NaNs. `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. `interp` works for all `n`-dimensional arrays with 2 or more dimensions.

`VI = interp(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 = interp(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interp(V)` is the same as `interp(V,1)`.



`VI = interpn(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. <code>interp</code> 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'.

`interp` 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. For faster interpolation when `X1, X2, X3, etc.` are equally spaced and monotonic, use the methods '\*linear', '\*cubic', or '\*nearest'.

## 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);
```

# interp

---

Now construct a finer grid:

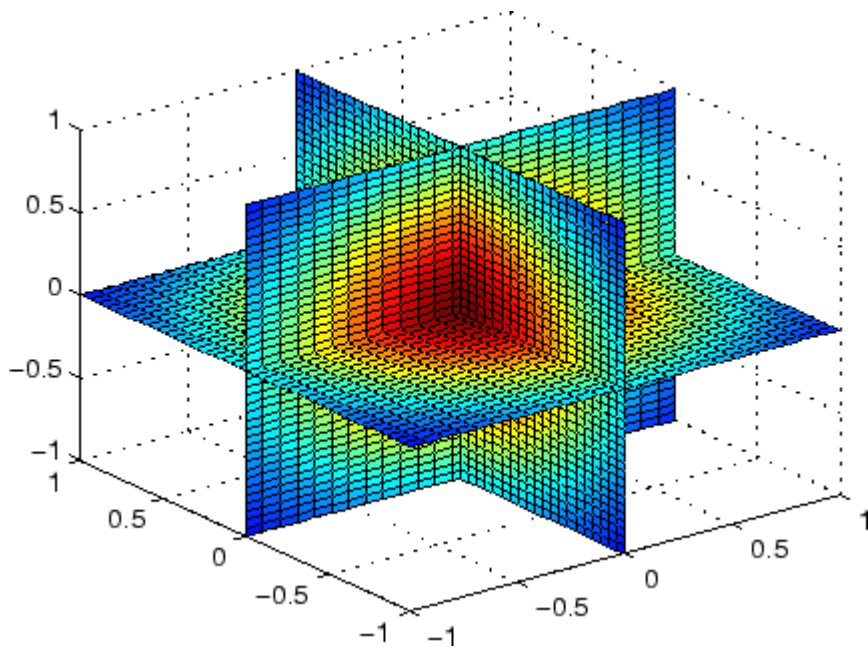
```
[xi,yi,zi,ti] = ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1, ...  
                      0:0.5:10);
```

Compute the spline interpolation at xi, yi, zi, and ti:

```
vi = interp(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

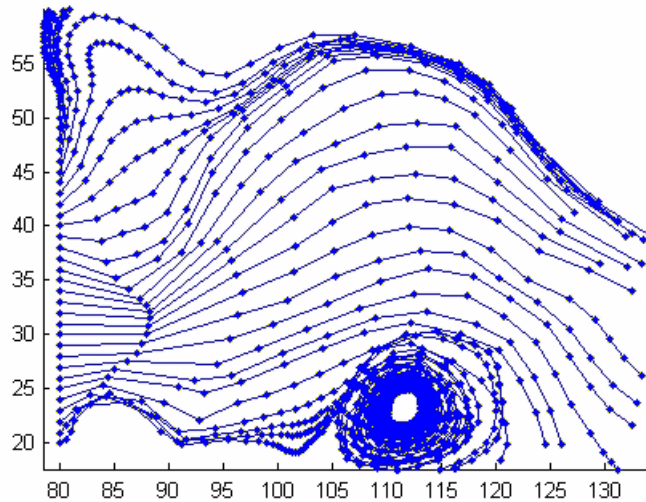
This example draws streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices 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);
```

# interpstreamspeed

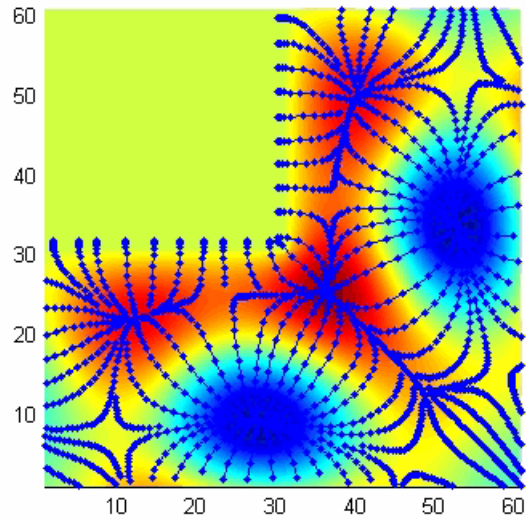
---

```
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** Find set intersection of two arrays

**Compatibility** In a future release, the behavior of `intersect` will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assessing the Impact of Forthcoming Changes to `intersect`” on page 2-2366. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.

**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(__,'R2012a')

[C,ia,ib] = intersect(__,'legacy')

[C,ia,ib] = intersect(A,B,setOrder)

[C,ia,ib] = intersect(A,B,'rows',setOrder)
```

**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')` returns the rows common to both A and B. The rows of the matrix C are in sorted order.

`[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( __ , 'R2012a' )` adopts the future behavior of the `intersect` function. `[C,ia,ib] = intersect( __ , 'legacy' )` preserves the current behavior. You can specify either flag, 'R2012a' or 'legacy', as the final input argument in any previous syntax.

`[C,ia,ib] = intersect(A,B,setOrder)` and `[C,ia,ib] = intersect(A,B, '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`, then `B`.

## Input Arguments

### A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays of which to find the intersection, specified as `logical`, `char`, a cell array of strings, or any numeric class. `A` and `B` also can be any object with the class methods: `sort` (or `sortrows` for the 'rows' option), `eq`, and `ne`. This includes 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`

**Complex Number Support:** Yes

### 'rows' - Rows flag

Rows flag, specified as 'rows', identifies the rows of `A` and `B` as individual elements. When you specify this flag, `intersect` returns the rows common to both `A` and `B`. This option does not support cell arrays.

# intersect

---

## setOrder - Order flag

'sorted' | '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, are returned in sorted order. For example: <code>C = intersect([7 0 5],[7 1 5],'sorted')</code> returns <code>C = [5 7]</code> .
'stable'	The values (or rows) in C, are returned in the same order as they appear in A and B. For example: <code>C = intersect([7 0 5],[7 1 5],'stable')</code> returns <code>C = [7 5]</code> .

## 'legacy' - Legacy behavior flag

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

## 'R2012a' - Future behavior flag

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior has on your existing code.

## Output Arguments

### C - Values common to A and B

vector | matrix

Values common to A and B, returned as a vector or matrix. C is a vector unless you specify the 'rows' flag. If 'rows' is specified, C is a matrix containing the rows common to both A and B.

## ia - Index to A

vector

Index to A, returned as a vector, identifies the elements in A that are common to B. If there is a repeated value (or row) in A, then ia contains the index to the last occurrence of the value (or row).

If you call `intersect` with the `setOrder` or 'R2012a' argument, ia contains the index to the first occurrence of any repeated value (or row) in A.

## ib - Index to B

vector

Index to B, returned as a vector, identifies the elements in B that are common to A. If there is a repeated value (or row) in B, then ib contains the index to the last occurrence of the value (or row).

If you call `intersect` with the `setOrder` or 'R2012a' argument, ib contains the index to the first occurrence of any repeated value (or row) in B.

## Examples

### Intersection of Two Vectors

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
C = intersect(A,B)
```

```
C =
```

```
4 7
```

### Find the Intersection of Two Vectors and Their Indices

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
[C,ia,ib] = intersect(A,B)
```

```
C =
```

```
4 7
```

# intersect

---

```
ia =
```

```
5 4
```

```
ib =
```

```
4 1
```

## Intersection of Rows in Two Matrices

```
A = [2 2 2; 0 0 1; 1 2 3; 1 1 1];
```

```
B = [1 2 3; 2 2 2; 2 2 0];
```

```
[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 =
```

# intersect

---

```
3  
1
```

`intersect` adopts the forthcoming behavior when you specify the `setOrder` argument. If you omit the `setOrder` argument, the output is sorted, but the size and content of `C`, `ia`, and `ib` conforms to the current behavior.

## Assessing the Impact of Forthcoming Changes to `intersect`

Use the 'R2012a' flag to assess the impact of the forthcoming behavior changes. Use the 'legacy' flag to preserve the current behavior of your existing code.

Find the intersection of A and B with the current default behavior.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];  
[C1,ia1,ib1] = intersect(A,B)
```

```
C1 =
```

```
4 7
```

```
ia1 =
```

```
5 4
```

```
ib1 =
```

```
4 1
```

Find the intersection of A and B, and opt into the forthcoming behavior. In the future, this behavior will be the default.

```
[C2,ia2,ib2] = intersect(A,B,'R2012a')
```

```
C2 =
     4     7
```

```
ia2 =
     5
     1
```

```
ib2 =
     3
     1
```

Notice that `ia1` and `ia2` have different shapes and content. The same is also true for `ib1` and `ib2`.

Find the unique elements of `A`, and preserve the current behavior.

```
[C3,ia3,ib3] = intersect(A,B,'legacy')
```

```
C3 =
     4     7
```

```
ia3 =
     5     4
```

```
ib3 =
     4     1
```

Notice that `C3`, `ia3`, and `ib3` match `C1`, `ia1`, and `ib1` respectively.

# intersect

---

## Intersection of Vectors Containing NaNs

```
A = [5 NaN NaN]; B = [5 NaN NaN];  
C = intersect(A,B)
```

```
C =
```

```
    5
```

intersect treats NaN values as distinct.

## See Also

[unique](#) | [union](#) | [ismember](#) | [issorted](#) | [setdiff](#) | [setxor](#) | [sort](#)



**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`

# intmin

---

**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**  $Y = \text{inv}(X)$

**Description**  $Y = \text{inv}(X)$  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  $x = \text{inv}(A) * b$ . A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator  $x = A \backslash b$ . 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  $\text{inv}(A) * b$  and solving it directly with  $A \backslash b$ . A random matrix  $A$  of order 500 is constructed so that its condition number,  $\text{cond}(A)$ , is  $1.e10$ , and its norm,  $\text{norm}(A)$ , 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 `\`, `/`

---

<b>Purpose</b>	Inverse of Hilbert matrix																
<b>Syntax</b>	$H = \text{invhilb}(n)$																
<b>Description</b>	$H = \text{invhilb}(n)$ generates the exact inverse of the exact Hilbert matrix for $n$ less than about 15. For larger $n$ , $\text{invhilb}(n)$ generates an approximation to the inverse Hilbert matrix.																
<b>Limitations</b>	<p>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, <math>n</math>, is less than 15.</p> <p>Comparing <math>\text{invhilb}(n)</math> with <math>\text{inv}(\text{hilb}(n))</math> involves the effects of two or three sets of roundoff errors:</p> <ul style="list-style-type: none"><li>• The errors caused by representing <math>\text{hilb}(n)</math></li><li>• The errors in the matrix inversion process</li><li>• The errors, if any, in representing <math>\text{invhilb}(n)</math></li></ul> <p>It turns out that the first of these, which involves representing fractions like <math>1/3</math> and <math>1/5</math> in floating-point, is the most significant.</p>																
<b>Examples</b>	<p><math>\text{invhilb}(4)</math> is</p> <table><tr><td>16</td><td>-120</td><td>240</td><td>-140</td></tr><tr><td>-120</td><td>1200</td><td>-2700</td><td>1680</td></tr><tr><td>240</td><td>-2700</td><td>6480</td><td>-4200</td></tr><tr><td>-140</td><td>1680</td><td>-4200</td><td>2800</td></tr></table>	16	-120	240	-140	-120	1200	-2700	1680	240	-2700	6480	-4200	-140	1680	-4200	2800
16	-120	240	-140														
-120	1200	-2700	1680														
240	-2700	6480	-4200														
-140	1680	-4200	2800														
<b>References</b>	[1] Forsythe, G. E. and C. B. Moler, <i>Computer Solution of Linear Algebraic Systems</i> , Prentice-Hall, 1967, Chapter 19.																
<b>See Also</b>	hilb																

# invoke

---

**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
```

# invoke

---

```
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”



**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 NaNs 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

## is\*

---

isunix	Determine if running MATLAB for UNIX <sup>6</sup> 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**

<code>logical</code>	Logical array of true and false values
<code>char</code>	Character array
<code>single</code>	Single-precision floating-point array
<code>double</code>	Double-precision floating-point array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>int64</code>	64-bit signed integer array
<code>uint64</code>	64-bit unsigned integer array
<code>cell</code>	Cell array

`struct`            Structure 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

<code>numeric</code>	Integer or floating-point array ( <code>double</code> , <code>single</code> , <code>int8</code> , <code>uint8</code> , <code>int16</code> , <code>uint16</code> , <code>int32</code> , <code>uint32</code> , <code>int64</code> , <code>uint64</code> )
<code>float</code>	Single- or double-precision floating-point array ( <code>double</code> , <code>single</code> )
<code>integer</code>	Signed or unsigned integer array ( <code>int8</code> , <code>uint8</code> , <code>int16</code> , <code>uint16</code> , <code>int32</code> , <code>uint32</code> , <code>int64</code> , <code>uint64</code> )

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`

# isappdata

---

<b>Purpose</b>	True if application-defined data exists
<b>Syntax</b>	<code>isappdata(h,name)</code>
<b>Description</b>	<code>isappdata(h,name)</code> returns 1 if application-defined data with the specified name exists on the object specified by handle <code>h</code> , and returns 0 otherwise.
<b>Tips</b>	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 <code>AppData</code> property. Only Handle Graphics MATLAB objects use this property.
<b>See Also</b>	<code>getappdata</code>   <code>rmappdata</code>   <code>setappdata</code>



**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*`

# iscellstr

---

**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*`

**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*`

# iscolumn

---

**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`

---

<b>Purpose</b>	Determine whether input is COM or ActiveX object
<b>Syntax</b>	<pre>tf = h.iscom tf = iscom(h)</pre>
<b>Description</b>	<p><code>tf = h.iscom</code> returns logical 1 (true) if handle <code>h</code> is a COM or Microsoft ActiveX object. Otherwise, returns logical 0 (false).</p> <p><code>tf = iscom(h)</code> is an alternate syntax.</p> <p>COM functions are available on Microsoft Windows systems only.</p>
<b>Examples</b>	<p>Test an instance of a Microsoft Excel application:</p> <pre>h = actxserver('Excel.Application'); h.iscom</pre> <p>MATLAB displays <code>true</code>, indicating object <code>h</code> is a COM object.</p> <hr/> <p>Test an Excel interface:</p> <pre>h = actxserver('Excel.Application'); %Create a workbooks object w = h.get('workbooks'); w.iscom</pre> <p>MATLAB displays <code>false</code>, indicating object <code>w</code> is not a COM object.</p>
<b>How To</b>	<ul style="list-style-type: none"><li>• “MATLAB COM Integration”</li></ul>

# isdir

---

**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*`

<b>Purpose</b>	Test if vertices are joined by edge	
<b>Syntax</b>	TF = isEdge(TR, V1, V2) TF = isEdge(TR, EDGE)	
<b>Description</b>	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.	
<b>Input Arguments</b>	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.
<b>Output Arguments</b>	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.
<b>Examples</b>	<b>Example 1</b>	
		Load a 2-D triangulation and use TriRep to query the presence of an edge between pairs of points.
		<pre>load trimesh2d trep = TriRep(tri, x,y);</pre>
		Test if vertices 3 and 117 are connected by an edge
		<pre>isEdge(trep, 3, 117)</pre>

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**

DelaunayTri



<b>Purpose</b>	Determine whether array is empty
<b>Syntax</b>	<code>TF = isempty(A)</code>
<b>Description</b>	<code>TF = isempty(A)</code> returns logical 1 (true) if <code>A</code> 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.
<b>Examples</b>	<pre>B = rand(2,2,2); B(:,:,:) = [];  isempty(B)  ans = 1</pre>
<b>See Also</b>	<code>is*</code>

# 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)

**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.

NaNs (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 NaNs 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

---

`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](#)

# isequaln

---

## 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  $A_1, A_2, \dots, A_n$  do not have to match. If  $A_1, A_2, \dots, A_n$  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);
```

# isequaln

---

```
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

# isequal (MException)

---

**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)

**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 NaNs are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider NaNs 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
```

# isequalwithequalnans

---

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](#)

**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 `Db1Click`:

```
f = figure ('position', [100 200 200 200]);  
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);  
h.isevent('Db1Click')
```

MATLAB displays `true`, indicating `Db1Click` 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;
```

# isevent

---

## 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
```

# isfield

---

## 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



**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\*

# isfloat

---

**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`

**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 2-2411 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
```

# isglobal

---

```
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*
```

**Purpose** Test for valid graphics object handle

**Syntax** `ishandle(H)`

**Description** `ishandle(H)` returns an array whose elements are 1 where the elements of `H` are valid graphics or Java object handles, and 0 where they are not.

#### Non-Graphics 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.

**Examples** Ensure that an input argument is the handle to a graphics object.

```
function updateLine(h,x,y)

    if ishandle(h)
        set(h,'XData',x,'YData',y)
        ...
    else
        line('XData',x,'YData',y);
        ...
    end
end
```

**See Also** `findobj` | `gca` | `gcf` | `gco` | `isa` | `set`

**How To** • “Accessing Object Handles”

# 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”

**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”

# isinf

---

**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\*



**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**

<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>int64</code>	64-bit signed integer array
<code>uint64</code>	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)
```

# isinteger

---

```
ans =  
    0  
% isinteger returns a logical value  
isinteger(isinteger(uint8(3)))  
ans =  
    0
```

## See Also

[isa](#) | [isnumeric](#) | [isfloat](#)

<b>Purpose</b>	Determine whether input is COM interface
<b>Syntax</b>	<pre>tf = h.isinterface tf = isinterface(h)</pre>
<b>Description</b>	<p><code>tf = h.isinterface</code> returns logical 1 (true) if handle <code>h</code> is a COM interface. Otherwise, returns logical 0 (false).</p> <p><code>tf = isinterface(h)</code> is an alternate syntax.</p> <p>COM functions are available on Microsoft Windows systems only.</p>
<b>Examples</b>	<p>Test an instance of a Microsoft Excel application:</p> <pre>h = actxserver('Excel.Application'); h.isinterface</pre> <hr/> <p>MATLAB displays <code>false</code>, indicating object <code>h</code> is not an interface.</p> <p>Test a workbooks object:</p> <pre>w = h.get('workbooks'); w.isinterface</pre> <p>MATLAB displays <code>true</code>, indicating object <code>w</code> is an interface.</p>
<b>See Also</b>	<code>iscom</code>   <code>interfaces</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Exploring Interfaces”</li></ul>

# isjava

---

**Purpose** Determine whether input is Java object

**Syntax** `tf = isjava(A)`

**Description** `tf = isjava(A)` returns logical 1 (true) if A is a Java object, and logical 0 (false) otherwise.

**Examples** Create an instance of the Java Date class and `isjava` indicates that it is a Java object.

```
myDate = java.util.Date;  
isjava(myDate)
```

The MATLAB software displays:

```
ans =
```

```
1
```

Note that `isobject`, which tests for MATLAB objects, returns logical 0 (false). Type:

```
isobject(myDate)
```

MATLAB displays:

```
ans =
```

```
0
```

**See Also** `isobject` | `javaArray` | `javaMethod` | `javaObject` | `isa` | `is*`

<b>Purpose</b>	Determine if containers.Map object contains key
<b>Syntax</b>	<code>tf = isKey(mapObj, keySet)</code>
<b>Description</b>	<code>tf = isKey(mapObj, keySet)</code> looks for the specified keys in <code>mapObj</code> , and returns logical <code>true</code> (1) for the keys that it finds, and logical <code>false</code> (0) for those it does not. <code>keySet</code> is a scalar key or a cell array of keys.
<b>Input Arguments</b>	<p><b>mapObj</b> Object of class <code>containers.Map</code>.</p> <p><b>keySet</b> Scalar value, string, or cell array that specifies keys to find in <code>mapObj</code>.</p>
<b>Output Arguments</b>	<p><b>tf</b> Array of logical values. If <code>keySet</code> is a scalar or a string, <code>tf</code> is a scalar. Otherwise, <code>tf</code> has the same size and dimensions as <code>keySet</code>.</p>
<b>Examples</b>	<p><b>Find Keys in a Map</b></p> <p>Construct a map that contains rainfall data for several months:</p> <pre>months = {'Jan', 'Feb', 'Mar', 'Apr'}; rainfall = [327.2, 368.2, 197.6, 178.4]; mapObj = containers.Map(months, rainfall);</pre> <p>Determine if keys Apr, May, and Jun are in the map:</p> <pre>keySet = {'Apr', 'May', 'Jun'}; tf = isKey(mapObj, keySet)</pre> <p>This code returns 1-by-3 vector <code>tf</code>:</p> <pre>tf =     1     0     0</pre>

# containers.Map.isKey

---

## 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](#)

**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'
```

# iskeyword

---

```
'spmd'  
'switch'  
'try'  
'while'
```

## See Also

isvarname | genvarname | is\*



**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*`

# islogical

---

**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) & islogical(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*`

<b>Purpose</b>	Determine if version is for Mac OS X platform
<b>Syntax</b>	<code>tf = ismac</code>
<b>Description</b>	<code>tf = ismac</code> returns logical 1 (true) if the version of MATLAB software is for the Apple Mac OS X platform, and returns logical 0 (false) otherwise.
<b>See Also</b>	<code>isunix</code>   <code>ispc</code>   <code>computer</code>   <code>isstudent</code>   <code>is*</code>

# ismatrix

---

**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`

<b>Purpose</b>	Array elements that are members of set
<b>Compatibility</b>	In a future release, the behavior of <code>ismember</code> will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assess the Impact of the Forthcoming Changes to <code>ismember</code> ” on page 2-2433. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.
<b>Syntax</b>	<pre>Lia = ismember(A,B) Lia = ismember(A,B,'rows') [Lia,Locb] = ismember(A,B) [Lia,Locb] = ismember(A,B,'rows')  [Lia,Locb] = ismember( __ , 'R2012a' )  [Lia,Locb] = ismember( __ , 'legacy' )</pre>
<b>Description</b>	<p><code>Lia = ismember(A,B)</code> returns an array the same size as A, containing 1 (true) where the elements of A are found in B, and 0 (false) elsewhere.</p> <p><code>Lia = ismember(A,B,'rows')</code> returns a vector containing 1 (true) where the rows of matrix A are also rows of B, and 0 (false) elsewhere.</p> <p><code>[Lia,Locb] = ismember(A,B)</code> also returns an array, <code>Locb</code>, containing the highest index in B for each value in A that is a member of B. The output array, <code>Locb</code>, contains 0 wherever A is not a member of B.</p> <p><code>[Lia,Locb] = ismember(A,B,'rows')</code> also returns a vector, <code>Locb</code>, containing the highest index in B for each row in A that is also a row in B. The output vector, <code>Locb</code>, contains 0 wherever A is not a row of B.</p>

# ismember

---

`[Lia,Locb] = ismember( ___, 'R2012a' )` adopts the future behavior of the `ismember` function. `[Lia,Locb] = ismember( ___, 'legacy' )` preserves the current behavior. You can specify either flag, `'R2012a'` or `'legacy'`, as the final input argument in any previous syntax.

## 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 B. The query array, A, can be `logical`, `char`, a cell array of strings, or any numeric class. A also can be any object with the class methods: `sort` (or `sortrows` for the `'rows'` option), `eq`, and `ne`. This includes 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`

**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 A. The set array, B, can be `logical`, `char`, a cell array of strings, or any numeric class. B also can be any object with the class methods: `sort` (or `sortrows` for the `'rows'` option), `eq`, and `ne`. This includes 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

**Complex Number Support:** Yes

### 'rows' - Rows flag

Rows flag, specified as 'rows', identifies the rows of A and B as individual elements. When you specify this flag, `ismember` returns a vector containing 1 where the rows of A are also rows of B, and 0 elsewhere. A and B must have the same number of columns when you use the rows flag. This option does not support cell arrays.

### 'legacy' - Legacy behavior flag

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

### 'R2012a' - Future behavior flag

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior has on your existing code.

## Output Arguments

### Lia - Logical results array

vector | matrix | N-D array

Logical results array, returned as a vector, matrix, or N-D array containing 1 (true) wherever the values (or rows) in A are members of B, and 0 (false) elsewhere.

# ismember

---

Lia is an array of the same size as A, unless you specify the 'rows' flag. If 'rows' is specified, Lia is a column vector.

## Locb - Locations in B

vector | matrix

Locations in B, returned as a vector or matrix containing the highest indices to the values (or rows) in B that are found in A.

If you call ismember with the setOrder or 'R2012a' argument, Locb contains the lowest indices to the values (or rows) in B that are found in A.

Locb is an array of the same size as A unless you specify the 'rows' flag. If 'rows' is specified, Locb is a column vector. Furthermore, Locb contains 0 wherever A is not a member of B.

## Examples

### Find Values That Are Members of a Set

```
A = [5 3 4 2]; B = [2 4 4 4 6 8];  
Lia = ismember(A,B)
```

```
Lia =
```

```
    0    0    1    1
```

### Find Members of a Set and Return Indices to Set Values

```
A = [5 3 4 2]; B = [2 4 4 4 6 8];  
[Lia,Locb] = ismember(A,B)
```

```
Lia =
```

```
    0    0    1    1
```

```
Locb =
```

```
    0    0    4    1
```



**Find Rows That Belong to a Set**

```
A = [1 3 5 6; 2 4 6 8];
B = [2 4 6 8; 1 3 5 7; 2 4 6 8];
[Lia, Locb] = ismember(A,B, 'rows')
```

```
Lia =
```

```
0
1
```

```
Locb =
```

```
0
3
```

**Assess the Impact of the Forthcoming Changes to ismember**

Use the 'R2012a' flag to assess the impact of the forthcoming behavior changes. Use the 'legacy' flag to preserve the current behavior of your existing code.

Find the members of B with the current default 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    4    1
```

Find the members of B, and adopt the forthcoming behavior. In the future, this behavior will be the default.

# ismember

---

```
[Lia2,Locb2] = ismember(A,B,'R2012a')
```

```
Lia2 =
```

```
    0    0    1    1
```

```
Locb2 =
```

```
    0    0    2    1
```

Locb1 and Locb2 contain different indices.

Find the members of B, and preserve the current behavior.

```
[Lia3,Locb3] = ismember(A,B,'legacy')
```

```
Lia3 =
```

```
    0    0    1    1
```

```
Locb3 =
```

```
    0    0    4    1
```

Locb1 and Locb3 are the same.

## Search Members of Set Containing NaN Values

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =
```

```
    1    0    0
```

```
Locb =
```

```
1 0 0
```

ismember treats NaN values as distinct.

## 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”

**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 NaNs 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**

<code>single</code>	Single-precision floating-point array
<code>double</code>	Double-precision floating-point array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>int64</code>	64-bit signed integer array
<code>uint64</code>	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 =
```

# isnumeric

---

```
    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\\*](#)



<b>Purpose</b>	Determine if input is MATLAB object
<b>Syntax</b>	<code>tf = isobject(A)</code>
<b>Description</b>	<p><code>tf = isobject(A)</code> returns true if A is an object of a MATLAB class. Otherwise, it returns false.</p> <p>Handle Graphics objects return false. Use <code>ishandle</code> to test for Handle Graphics objects.</p> <p>Instances of MATLAB fundamental classes return false. Use <code>isa</code> to test for any of these types. See “Fundamental MATLAB Classes” for more on these classes.</p>
<b>Examples</b>	<p>These examples show the values <code>isobject</code> returns when passed specific types:</p> <p>Define the following MATLAB class:</p> <pre>classdef button &lt; handle     properties         UiHandle     end     methods         function obj = button(pos)             obj.UiHandle = uicontrol('Position',pos,...                 'Style','pushbutton');         end     end end</pre> <p>Determine which objects are MATLAB objects. For example:</p> <pre>h = button([20 20 60 60]); isobject(h) ans =      1 isobject(h.UiHandle)</pre>

# isobject

---

```
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`.

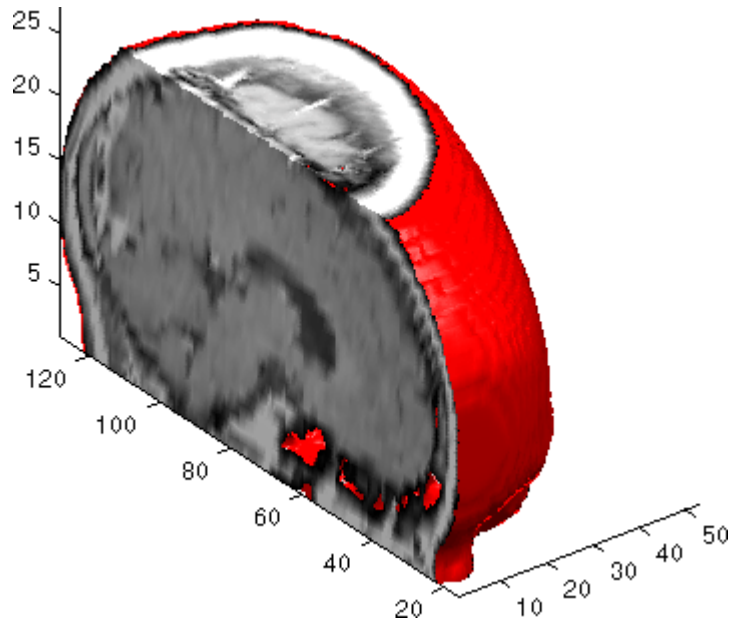
`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

**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` 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**

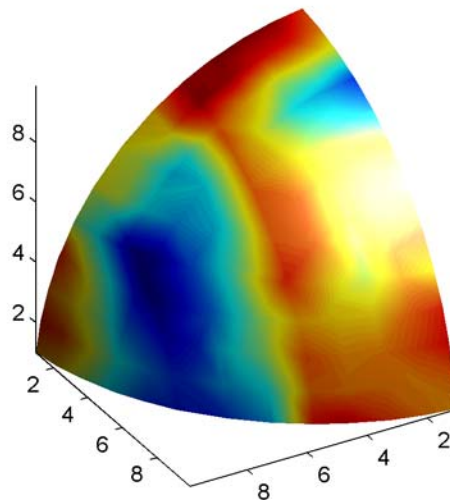
This example displays an isosurface and colors it with random data using indexed color. (See “Interpolating in Indexed Color Versus Truicolor” for information on how patch objects interpret color data.)

```
[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.

```

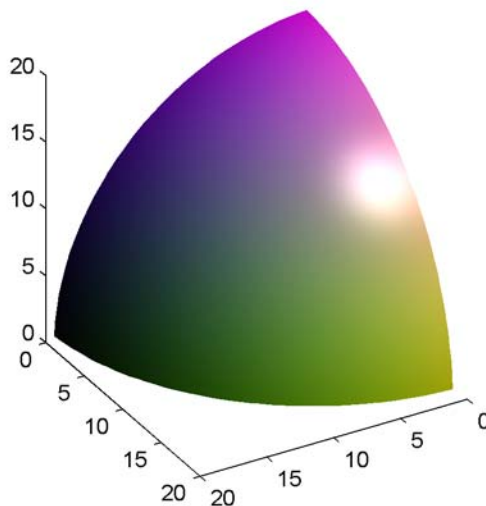
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);

```

# isocolors

---

```
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;
```



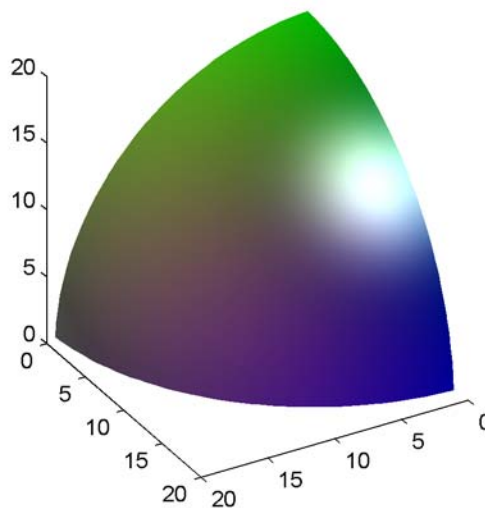
## Modified True Color Data

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](#) | [reducephatch](#) | [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:

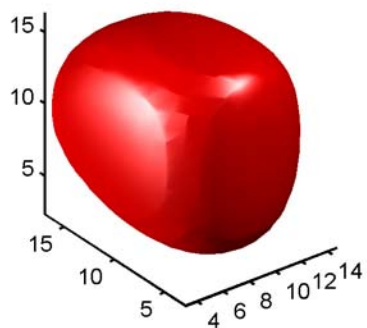
```
subplot(1,2,2)
p2 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3); daspect([1 1 1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Data Normals')
```

These isosurfaces illustrate the difference between triangle and data normals:

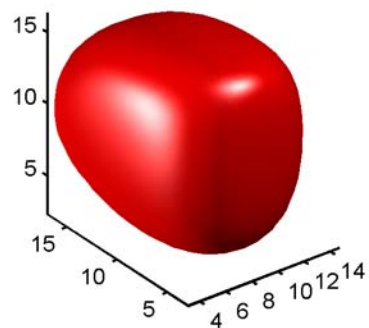
# isonormals

---

Triangle Normals



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).

# 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

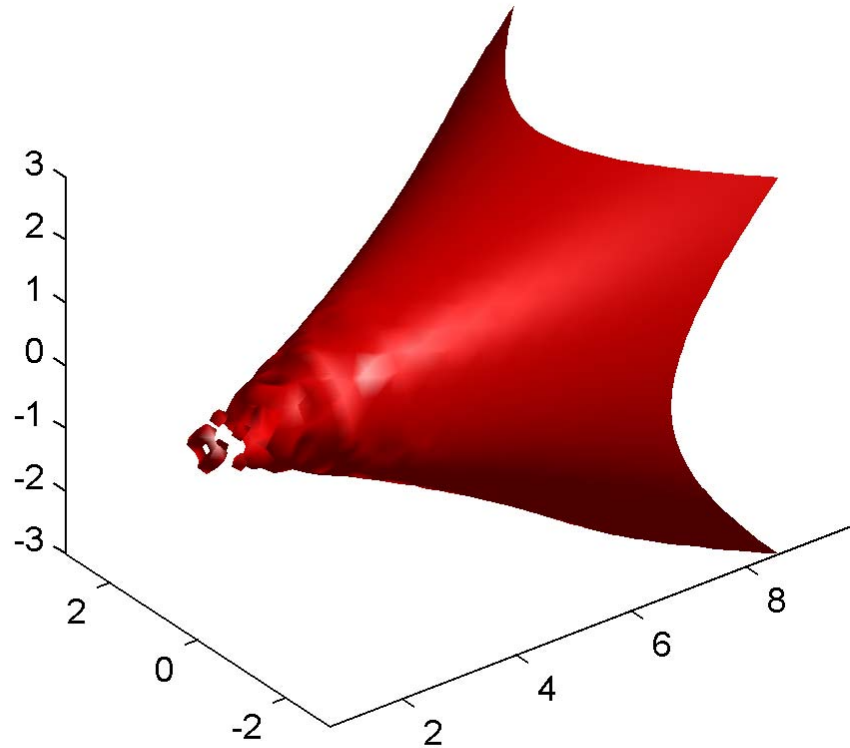
This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). The isosurface is drawn at the data value of -3. The statements that follow the `patch` command prepare the isosurface for lighting by

- 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
```

# isosurface

---



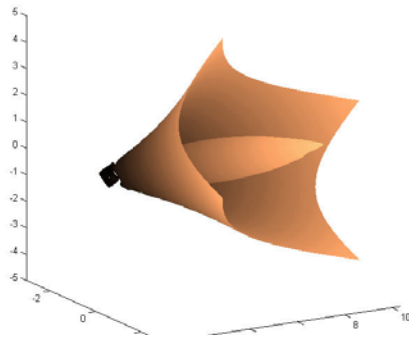
## Example 2

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*`

---

<b>Purpose</b>	Test for existence of preference
<b>Syntax</b>	<pre>ispref('group','pref') ispref('group') ispref('group',{'pref1','pref2',... 'prefn'})</pre>
<b>Description</b>	<p><code>ispref('group','pref')</code> returns 1 if the preference specified by <code>group</code> and <code>pref</code> exists, and 0 otherwise.</p> <p><code>ispref('group')</code> returns 1 if the <code>GROUP</code> exists, and 0 otherwise.</p> <p><code>ispref('group',{'pref1','pref2',... 'prefn'})</code> returns a logical array the same length as the cell array of preference names, containing 1 where each preference exists, and 0 elsewhere.</p>
<b>Examples</b>	<pre>addpref('mytoolbox','version','1.0') ispref('mytoolbox','version')  ans =      1</pre>
<b>See Also</b>	<code>addpref</code>   <code>getpref</code>   <code>rmpref</code>   <code>setpref</code>   <code>uigetpref</code>   <code>uisetpref</code>

# isprime

---

**Purpose** Array elements that are prime numbers

**Syntax** TF = isprime(A)

**Description** TF = isprime(A) returns an array the same size as A containing logical 1 (true) for the elements of A which are prime, and logical 0 (false) otherwise. A must contain only positive integers.

**Examples**

```
c = [2 3 0 6 10]

c =
     2     3     0     6    10

isprime(c)

ans =
     1     1     0     0     0
```

**See Also** is\*

<b>Purpose</b>	Determine if property of object
<b>Syntax</b>	<code>tf = isprop(obj,'PropertyName')</code>
<b>Description</b>	<p><code>tf = isprop(obj,'PropertyName')</code> returns logical 1 (true) if the specified <code>PropertyName</code> is a property of object <code>obj</code>. Otherwise, returns logical 0 (false).</p> <p>Note that classes can control access to property values by defining property attributes. Property access can be defined as:</p> <ul style="list-style-type: none"><li>• Readable and writable</li><li>• Read only</li><li>• Write only</li><li>• Accessible only to certain class methods</li></ul> <p>See “Property Attributes” for information on restrictions to property access.</p> <p>See “Getting Information About Properties” for information about obtaining specific information about the properties of a class.</p>
<b>Examples</b>	<p>This example uses <code>isprop</code> to determine if <code>XDataSource</code> is a property of object <code>h</code> before attempting to set the property value:</p> <pre>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</pre>
<b>See Also</b>	<code>properties</code>   <code>ismethod</code>

## Tutorials

- “Properties”

**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”

# isreal

---

**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,

# isreal

---

```
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] [5]
```

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\\*](#)

---

<b>Purpose</b>	Determine whether input is row vector
<b>Syntax</b>	<code>isrow(V)</code>
<b>Description</b>	<code>isrow(V)</code> returns logical 1 (true) if <code>size(V)</code> returns <code>[1 n]</code> with a nonnegative integer value <code>n</code> , and logical 0 (false) otherwise.
<b>Examples</b>	<p>Determine if a vector is a row. This example is a column so <code>isrow</code> returns 0:</p> <pre>V = rand(5,1); isrow(V) ans =     0</pre> <p>Transpose the vector to make it a row. <code>isrow</code> returns 1:</p> <pre>V1 = V'; isrow(V1) ans =     1</pre>
<b>See Also</b>	<code>iscolumn</code>   <code>ismatrix</code>   <code>isscalar</code>   <code>isvector</code>

# isscalar

---

**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*`

**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
```

# issorted

---

```
     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\*

# isspace

---

**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*`



**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

# isstr

---

**Purpose** Determine whether input is character array

---

**Note** `isstr` is not recommended. Use `ischar` instead.

---

**See Also** `ischar` | `isa` | `is*`

**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 <code>str</code> that are alphabetic
alphanum	True for those elements of <code>str</code> that are alphanumeric
cntrl	True for those elements of <code>str</code> that are control characters (for example, <code>char(0:20)</code> )
digit	True for those elements of <code>str</code> that are numeric digits
graphic	True for those elements of <code>str</code> 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 <code>str</code> that are lowercase letters
print	True for those elements of <code>str</code> that are graphic characters, plus <code>char(32)</code>

Category	Description
punct	True for those elements of <code>str</code> that are punctuation characters
wspace	True for those elements of <code>str</code> 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 <code>str</code> that are uppercase letters
xdigit	True for those elements of <code>str</code> 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 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
```

# isstrprop

---

## See Also

strings | ischar | isletter | isspace | iscellstr | isnumeric  
| isa | is\*

**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

# isstudent

---

**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*`



**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 stripped 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”

# isunix

---

**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<sup>7</sup> platform, and returns logical 0 (false) otherwise.

**See Also** `ispc` | `ismac` | `computer` | `isstudent` | `is*`

7. UNIX is a registered trademark of The Open Group in the United States and other countries

**Purpose** Is object valid handle class object

**Syntax** H1 = isvalid(Hobj)

**Description** H1 = 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 `ishandle` function.

---

**See Also** delete (handle) | handle

**How To** • “Testing Handle Validity”

# isvalid (serial)

---

**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`

**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)`

# isvarname

---

**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*`

**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`



**Purpose** Add entries to dynamic Java class path

**Syntax**  
`javaaddpath(dpath1,...,dpathN)`  
`javaaddpath(dpath1,...,dpathN,'-end')`

**Description**  
`javaaddpath(dpath1,...,dpathN)` adds one or more folders or JAR files to the beginning of the current dynamic Java class path.  
`javaaddpath(dpath1,...,dpathN,'-end')` adds files or folders to the end of the path.

**Tips**  
 The Java path consists of two segments: a static path (read only at startup) and a dynamic path. MATLAB always searches the static path before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file <code>javaclasspath.txt</code> . 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 <code>javaclasspath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic path (using <code>javaclasspath</code> ), modify the path (using <code>javaaddpath</code> and <code>javarmpath</code> ), and refresh the Java class definitions for all classes on the dynamic path (using <code>clear java</code> ) without restarting MATLAB.

# javaaddpath

---

## Input Arguments

### **dpath1,...,dpathN**

String or cell array of strings specifying the folder or JAR file.

## Examples

This example demonstrates modifying the dynamic Java class path.

- 1 Create a function to set your initial dynamic class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
    'C:\Work\JavaTest\timer.jar', ...
    'C:\Work\JavaTest\patch.jar'});
% end of file
```

- 2 Call this function to set up your dynamic class path and then display all current static and dynamic paths:

```
setdynpath;
```

```
javaclasspath
```

```
        STATIC JAVA PATH
```

```
D:\Sys0\Java\util.jar
D:\Sys0\Java\widgets.jar
D:\Sys0\Java\beans.jar
```

```
      .
      .
```

```
        DYNAMIC JAVA PATH
```

```
C:\Work\Java\ClassFiles
C:\Work\JavaTest\curvefit.jar
C:\Work\JavaTest\timer.jar
C:\Work\JavaTest\patch.jar
```

- 3 Add the following entries to the dynamic path. One entry specifies a folder and the other a Java Archive (JAR) file. When you add a folder to the path, MATLAB includes all files in that folder as part of the path:

```
javaaddpath({  
    'C:\Work\Java\Curvefit\Test', ...  
    'C:\Work\Java\mywidgets.jar'});
```

- 4 Display the dynamic path:

```
p = javaclasspath  
p =  
    'C:\Work\Java\ClassFiles'  
    'C:\Work\JavaTest\curvefit.jar'  
    'C:\Work\JavaTest\timer.jar'  
    'C:\Work\JavaTest\patch.jar'  
    'C:\Work\Java\Curvefit\Test'  
    'C:\Work\Java\mywidgets.jar'
```

- 5 Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

- 6 If you modify one or more classes defined on the dynamic path, you must clear the former definition for those classes from MATLAB memory. To clear all dynamic class definitions, type:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

- 7 Remove `mywidgets.jar` from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

# javaaddpath

---

---

Add a JAR file from an internet URL to the Java path:

```
javaaddpath http://www.example.com/my.jar
```

---

Add the current folder:

```
javaaddpath(pwd)
```

## See Also

```
javaclasspath | javarmpath | clear
```

## How To

- “Bringing Java Classes into MATLAB Workspace”

<b>Purpose</b>	Construct Java array
<b>Syntax</b>	<code>objArray = javaArray(PackageName.ClassName,x1,...,xn)</code>
<b>Description</b>	<code>objArray = javaArray(PackageName.ClassName,x1,...,xn)</code> constructs an empty Java array for objects of the specified Java class.
<b>Tips</b>	<ul style="list-style-type: none"> <li>• The array created by <code>javaArray</code> is equivalent to the array created by the following Java code:  <code>A = new class_name[x1]...[xn];</code></li> <li>• 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”.</li> </ul>
<b>Input Arguments</b>	<p><b>PackageName.ClassName</b> String specifying Java class. You must include the package name when specifying the class.</p> <p><b>x1,...,xn</b> Dimensions of the array.</p>
<b>Output Arguments</b>	<p><b>objArray</b> Java array with dimensions <code>x1, ..., xn</code>.</p>
<b>Examples</b>	<p>This example constructs and populates a 4-by-5 array of <code>java.lang.Double</code> objects.</p> <pre>dblArray = javaArray ('java.lang.Double', 4, 5); for m = 1:4     for n = 1:5         dblArray(m,n) = java.lang.Double((m*10) + n);     end end</pre>

# javaArray

---

```
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 | methodsvew | isjava
```

**Purpose** Generate error message based on Java feature support

**Syntax**  
javachk(feature)  
javachk(feature, component)

**Description** javachk(feature) returns a generic error message if the specified Java feature is not available in the current MATLAB session. If it is available, javachk returns an error structure with empty message and identifier fields.

The following table shows the valid feature arguments.

Feature	Description
'awt'	Java GUI components in the Abstract Window Toolkit 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.

javachk(feature, component) works the same as the above syntax, except that the specified component is also named in the error message.

**See Also** usejava

# javaclasspath

---

**Purpose** Get and set Java class 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 path.

javaclasspath('-dynamic') displays the dynamic Java path.

javaclasspath('-static') displays the static Java path.

dpath = javaclasspath returns the dynamic segment of the Java path in cell array, dpath. If no dynamic paths are defined, javaclasspath returns an empty cell array.

spath = javaclasspath('-static') returns the static segment of the Java path in cell array, spath. No path information is displayed unless you specify an output variable. If no static paths are defined, javaclasspath returns an empty cell array.

jpath = javaclasspath('-all') returns the entire Java path in cell array, jpath. The returned cell array contains first the static segment of the path, and then the dynamic segment. No path information is displayed unless you specify an output variable. If no dynamic paths are defined, javaclasspath returns an empty cell array.

javaclasspath(dpath) changes the dynamic Java path to dpath, where dpath can be a string or cell array of strings representing path entries. Relative paths are converted to absolute paths. Uses the clear java command to refresh the classes defined on the dynamic Java path.



`javaclasspath(dpath1, dpath2)` changes the dynamic Java path to the concatenation of the two paths `dpath1` and `dpath2`, where `dpath1` and `dpath2` can be a string or cell array of strings representing path entries. Relative paths are converted to absolute paths. Uses the `clear java` command to refresh the classes defined on the dynamic Java path.

`javaclasspath(statusmsg)` enables or disables the display of status messages from the `javaclasspath`, `javaaddpath`, and `javarmppath` functions. Values for the `statusmsg` argument are shown in the following table:

statusmsg	Description
'-v1'	Display status messages while loading the Java path from the file system.
'-v0'	Do not display status messages. This is the default.

## Tips

The Java path consists of two segments: a static path (read only at startup) and a dynamic path. MATLAB always searches the static path before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file <code>javaclasspath.txt</code> . 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 <code>javaclasspath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic

# javaclasspath

---

Path Type	Description
	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

This examples demonstrates modifying the dynamic Java class path.

- 1 Create a function to set your initial dynamic class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
    'C:\Work\JavaTest\timer.jar', ...
    'C:\Work\JavaTest\patch.jar'});
% end of file
```

- 2 Call this function to set up your dynamic class path and then display all current static and dynamic paths:

```
setdynpath;

javaclasspath

          STATIC JAVA PATH

D:\Sys0\Java\util.jar
D:\Sys0\Java\widgets.jar
D:\Sys0\Java\beans.jar
      .
      .

          DYNAMIC JAVA PATH

C:\Work\Java\ClassFiles
```

```
C:\Work\JavaTest\curvefit.jar
C:\Work\JavaTest\timer.jar
C:\Work\JavaTest\patch.jar
```

- 3 Add the following entries to the dynamic path. One entry specifies a folder and the other a Java Archive (JAR) file. When you add a folder to the path, MATLAB includes all files in that folder as part of the path:

```
javaaddpath({
    'C:\Work\Java\Curvefit\Test', ...
    'C:\Work\Java\mywidgets.jar'});
```

- 4 Display the dynamic path:

```
p = javaclasspath
p =
    'C:\Work\Java\ClassFiles'
    'C:\Work\JavaTest\curvefit.jar'
    'C:\Work\JavaTest\timer.jar'
    'C:\Work\JavaTest\patch.jar'
    'C:\Work\Java\Curvefit\Test'
    'C:\Work\Java\mywidgets.jar'
```

- 5 Create an instance of the `mywidgets` class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

- 6 If you modify one or more classes defined on the dynamic path, you must clear the former definition for those classes from MATLAB memory. To clear all dynamic class definitions, type:

```
clear java
```

If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

# javaclasspath

---

**7** Remove mywidgets.jar from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

## **See Also**

javaaddpath | javarmpath | clear

<b>Purpose</b>	Capture error information for Java exception
<b>Description</b>	Process information from a <code>matlab.exception.JavaException</code> object to handle Java errors thrown from Java methods called from MATLAB. This class is derived from <code>MException</code> .
<b>Construction</b>	<code>e = matlab.exception.JavaException(msgID, errMsg, excObj)</code> constructs instance <code>e</code> of <code>matlab.exception.JavaException</code> class.
	<b>Input Arguments</b>
	<b>msgID</b> message identifier
	<b>errMsg</b> error message string
	<b>excObj</b> <code>java.lang.Throwable</code> object that caused the exception
	<b>Output Arguments</b>
	<b>e</b> Instance of <code>matlab.exception.JavaException</code> class
<b>Properties</b>	<b>ExceptionObject</b> Java exception object that caused the error.
<b>Tips</b>	<ul style="list-style-type: none"><li>You do not typically construct a <code>matlab.exception.JavaException</code> object explicitly. MATLAB automatically constructs a <code>JavaException</code> object whenever Java throws an exception. The <code>JavaException</code> object wraps the original Java exception.</li></ul>
<b>Copy Semantics</b>	Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

# matlab.exception.JavaException

---

## 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”

**Purpose** Invoke Java method

**Syntax** `javaMethod(MethodName, ClassName, x1, ..., xn)`  
`javaMethod(MethodName, JavaObj, x1, ..., xn)`

**Description** `javaMethod(MethodName, ClassName, x1, ..., xn)` calls the static method `MethodName` in the class `ClassName` with the signature matching the arguments `x1, ..., xn`. `MethodName` and `ClassName` are strings.

`javaMethod(MethodName, JavaObj, x1, ..., xn)` calls the nonstatic method `MethodName` on the Java object array `JavaObj`.

**Tips** Use the `javaMethod` function to:

- Use methods having names longer than 31 characters.
- Specify the method you want to invoke at run-time, for example, as input from an application user.

The `javaMethod` function enables you to use methods having names longer than 31 characters. This is the only way you can invoke such a method in the MATLAB software. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions', T);
```

With `javaMethod`, you can also specify the method to be invoked at run-time. In this situation, your code calls `javaMethod` with a string variable in place of the method name argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

---

**Note** Typically, you do not need to use `javaMethod`. Use the default MATLAB syntax for invoking a Java methods instead. Use `javaMethod` for the cases described above.

---

# javaMethod

---

## Examples

To invoke the static Java method `isNaN` on class `java.lang.Double` type:

```
javaMethod('isNaN', 'java.lang.Double', 2.2)
```

The following example invokes the nonstatic method `setMonth`, where `myDate` is a `java.util.Date` object.

```
myDate = java.util.Date;  
javaMethod('setMonth', myDate, 3);
```

## See Also

[javaArray](#) | [javaObject](#) | [import](#) | [methods](#) | [isjava](#) | [javaMethodEDT](#)



**Purpose**

Invoke Java method from Event Dispatch Thread (EDT)

**Syntax**

```
javaMethodEDT(MethodName,ClassName,x1,...,xn)  
javaMethodEDT(MethodName,JavaObj,x1,...,xn)
```

**Description**

javaMethodEDT(MethodName,ClassName,x1,...,xn) invokes the Java class method with the specified input arguments from the Event Dispatch Thread (EDT).

javaMethodEDT(MethodName,JavaObj,x1,...,xn) invokes a nonstatic method on a Java object array.

**Input Arguments****MethodName**

String representing method in Java ClassName class.

**ClassName**

String representing Java class containing MethodName.

**x1,...,xn**

Input arguments matching MethodName signature.

**JavaObj**

Java object array.

**Examples**

Invoke the static method setDefaultLookAndFeelDecorated in class javax.swing.JFrame:

```
javaMethodEDT('setDefaultLookAndFeelDecorated','javax.swing.JFrame',
```

---

Create a vector v on the MATLAB thread and invoke the add method from the EDT.

```
v = java.util.Vector;  
javaMethodEDT('add',v,'string');
```

# javaMethodEDT

---

## See Also

[javaMethod](#) | [javaObjectEDT](#) | [import](#) | [methods](#) | [isjava](#)

<b>Purpose</b>	Invoke Java constructor
<b>Syntax</b>	<code>javaObject(ClassName, x1, ..., xn)</code>
<b>Description</b>	<p><code>javaObject(ClassName, x1, ..., xn)</code> calls the Java constructor for class <code>ClassName</code> with the argument list that matches <code>x1, ..., xn</code>, to return a new object. <code>ClassName</code> is a string.</p> <p>If there is no constructor that matches the class name and argument list passed to <code>javaObject</code>, an error occurs.</p>
<b>Tips</b>	For more information, see “Using the <code>javaObjectEDT</code> Function”.
<b>Examples</b>	<p>The following example constructs and returns a Java object of class <code>java.lang.String</code>:</p> <pre>strObj = javaObject('java.lang.String','hello')</pre>
<b>See Also</b>	<code>javaArray</code>   <code>javaMethod</code>   <code>import</code>   <code>methods</code>   <code>fieldnames</code>   <code>isjava</code>   <code>javaObjectEDT</code>

# javaObjectEDT

---

- Purpose** Java object constructor on Event Dispatch Thread (EDT)
- Syntax** `jObj = javaObjectEDT(ClassName,x1,...,xn)`
- Description** `jObj = javaObjectEDT(ClassName,x1,...,xn)` invokes the Java constructor for class `ClassName` with the signature matching the arguments `x1, ..., xn` from the EDT and returns a Java object array.
- Tips**
- Static methods on the specified class or Java object run on the MATLAB thread unless invoked using the `javaMethodEDT` function.

- Input Arguments**
- ClassName**  
Java class name.
- x1,...,xn**  
Optional input arguments for class constructor. If none, MATLAB calls the no-argument constructor.

- Output Arguments**
- jObj**  
Java object array. MATLAB invokes subsequent methods on this object from the EDT.

- Examples** Construct and return a `javax.swing.JFrame` object array from the EDT.

```
f = javaObjectEDT('javax.swing.JFrame', 'New Title')
```

---

Create a `JOptionPane` pane on the EDT and call the `createDialog` method.

```
% Create a JOptionPane pane on the EDT
optPane = javaObjectEDT('javax.swing.JOptionPane');
% Call the createDialog method - automatically done on the EDT
dlg = optPane.createDialog([], 'Sample Dialog');
% Tell MATLAB to dispatch methods on dlg from the EDT
```

```
javaObjectEDT(dlg);
```

**See Also**

```
javaMethodEDT | javaObject | import | methods | isjava
```

# javarmpath

---

**Purpose** Remove entries from dynamic Java class path

**Syntax**  
`javarmpath(dpath)`  
`javarmpath(dpath1,...,dpathN)`  
`javarmpath(v1,...,vN)`

**Description** `javarmpath(dpath)` removes specified file or folder from the current dynamic Java path.  
`javarmpath(dpath1,...,dpathN)` removes multiple files or folders.  
`javarmpath(v1,...,vN)` removes files or folders specified as MATLAB variables.

**Tips** The Java path consists of two segments: a static path (read only at startup) and a dynamic path. MATLAB always searches the static path before the dynamic path. Java classes on the static path should not have dependencies on classes on the dynamic path. Use `javaclasspath` to see the current static and dynamic Java paths.

MATLAB calls the `clear java` command whenever you change the dynamic path.

Path Type	Description
Static	Loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file <code>javaclasspath.txt</code> . 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 <code>javaclasspath.txt</code> and restart MATLAB.
Dynamic	Loaded at any time during a MATLAB session using the <code>javaclasspath</code> function. You can define the dynamic path (using <code>javaclasspath</code> ), modify the path (using <code>javaaddpath</code> and <code>javarmpath</code> ), and refresh the Java class definitions for all classes on the dynamic path (using <code>clear java</code> ) without restarting MATLAB.

**Input Arguments****dpath**

String specifying a file or folder.

**dpath1,...,dpathN**

Strings specifying multiple files or folders.

**v1,...,vN**

One or more MATLAB variables containing file or folder names.

**Examples**

This example demonstrates modifying the dynamic Java class path.

- 1 Create a function to set your initial dynamic class path:

```
function setdynpath
javaclasspath({
    'C:\Work\Java\ClassFiles', ...
    'C:\Work\JavaTest\curvefit.jar', ...
    'C:\Work\JavaTest\timer.jar', ...
    'C:\Work\JavaTest\patch.jar'});
% end of file
```

- 2 Call this function to set up your dynamic class path and then display all current static and dynamic paths:

```
setdynpath;

javaclasspath

        STATIC JAVA PATH

D:\Sys0\Java\util.jar
D:\Sys0\Java\widgets.jar
D:\Sys0\Java\beans.jar
.
.
```

## DYNAMIC JAVA PATH

```
C:\Work\Java\ClassFiles
C:\Work\JavaTest\curvefit.jar
C:\Work\JavaTest\timer.jar
C:\Work\JavaTest\patch.jar
```

- 3 Add the following entries to the dynamic path. One entry specifies a folder and the other a Java Archive (JAR) file. When you add a folder to the path, MATLAB includes all files in that folder as part of the path:

```
javaaddpath({
    'C:\Work\Java\Curvefit\Test', ...
    'C:\Work\Java\mywidgets.jar'});
```

- 4 Display the dynamic path:

```
p = javaclasspath
p =
    'C:\Work\Java\ClassFiles'
    'C:\Work\JavaTest\curvefit.jar'
    'C:\Work\JavaTest\timer.jar'
    'C:\Work\JavaTest\patch.jar'
    'C:\Work\Java\Curvefit\Test'
    'C:\Work\Java\mywidgets.jar'
```

- 5 Create an instance of the mywidgets class that is defined on the dynamic path:

```
h = mywidgets.calendar;
```

- 6 If you modify one or more classes defined on the dynamic path, you must clear the former definition for those classes from MATLAB memory. To clear all dynamic class definitions, type:

```
clear java
```



If you then create a new instance of one of these classes, MATLAB uses the latest definition of the class to create the object.

**7** Remove `mywidgets.jar` from the current dynamic class path:

```
javarmpath('C:\Work\Java\mywidgets.jar');
```

## See Also

`javaclasspath` | `javaaddpath` | `clear`

# 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

<b>Purpose</b>	Identify keys of containers.Map object
<b>Syntax</b>	<code>keySet = keys(mapObj)</code>
<b>Description</b>	<code>keySet = keys(mapObj)</code> returns cell array <code>keySet</code> , which contains all of the keys in <code>mapObj</code> .
<b>Input Arguments</b>	<b>mapObj</b> Object of class <code>containers.Map</code> .
<b>Output Arguments</b>	<b>keySet</b> 1-by-n cell array, where n is the number of keys in <code>mapObj</code> .
<b>Examples</b>	<b>Get the Keys in a Map</b> Create a map, and view the keys in the map: <pre>myKeys = {'a','b','c'}; myValues = [1,2,3]; mapObj = containers.Map(myKeys,myValues);  keySet = keys(mapObj)</pre> <p>This code returns 1-by-3 cell array <code>keySet</code>:</p> <pre>keySet =     'a'    'b'    'c'</pre>
<b>See Also</b>	<code>containers.Map</code>   <code>isKey</code>   <code>values</code>   <code>remove</code>

# kron

---

**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`.

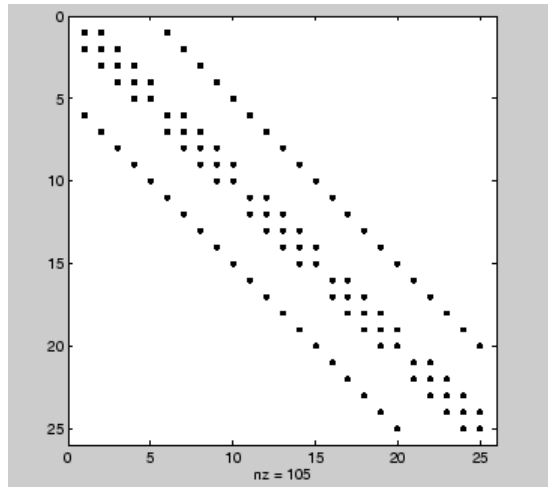
**Examples** If `X` is 2-by-3, then `kron(X,Y)` is

```
[ 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 retrieved 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
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 <code>identifier</code> field is an empty character array.
stack	Structure providing information on the location of the error. The structure has fields <code>file</code> , <code>name</code> , and <code>line</code> , and is the same as the structure returned by the <code>dbstack</code> function. If <code>lasterror</code> returns no stack information, <code>stack</code> 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
<code>file</code>	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
<code>name</code>	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, <code>name</code> is set to the file name.
<code>line</code>	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
```

# lasterror

---

## 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`:

# lastwarn

---

```
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 multiple of corresponding elements of arrays `A` and `B`. Inputs `A` and `B` must contain positive integer elements and must be the same size (or either can be scalar).

**Examples**

```
lcm(8,40)

ans =

    40

lcm(pascal(3),magic(3))

ans =

     8     1     6
     3    10    21
     4     9     6
```

**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,  $[L,D,P] = \text{ldl}(A, 'lower')$  gives the default behavior.

$[U,D,p] = \text{ldl}(A, 'upper', 'vector')$  returns the permutation information as a vector,  $p$ , as does  $[L,D,p] = \text{ldl}(A, 'lower', 'vector')$ . A must be a full matrix.

$[L,D,P,S] = \text{ldl}(A)$  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  $A$ .

$[L,D,P,S] = \text{LDL}(A, \text{THRESH})$  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.

$[U,D,p,S] = \text{LDL}(A, \text{THRESH}, 'upper', 'vector')$  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 2-2534
- “Example 2 — Three Output Form of `ldl`” on page 2-2534
- “Example 3 — The Structure of  $D$ ” on page 2-2535
- “Example 4 — Using the `'vector'` Option” on page 2-2535
- “Example 5 — Using the `'upper'` Option” on page 2-2536
- “Example 6 — `linsolve` and the Hermitian indefinite solver” on page 2-2536

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:

```

M1 = tril(M);
[Lm1, Dm1, Pm1] = ldl(M1, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
        'The difference between Lm1 and Lm is %g\n', norm(Lm1 - 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

# ldivide

---

**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

# legend

---

## 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 `DisplayNames` 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.

# legend

---

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                |

# legend

---

| 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`

# legend

---

- TextColor
- Interpreter
- String

## Tips

### Using Keywords as Legend Labels

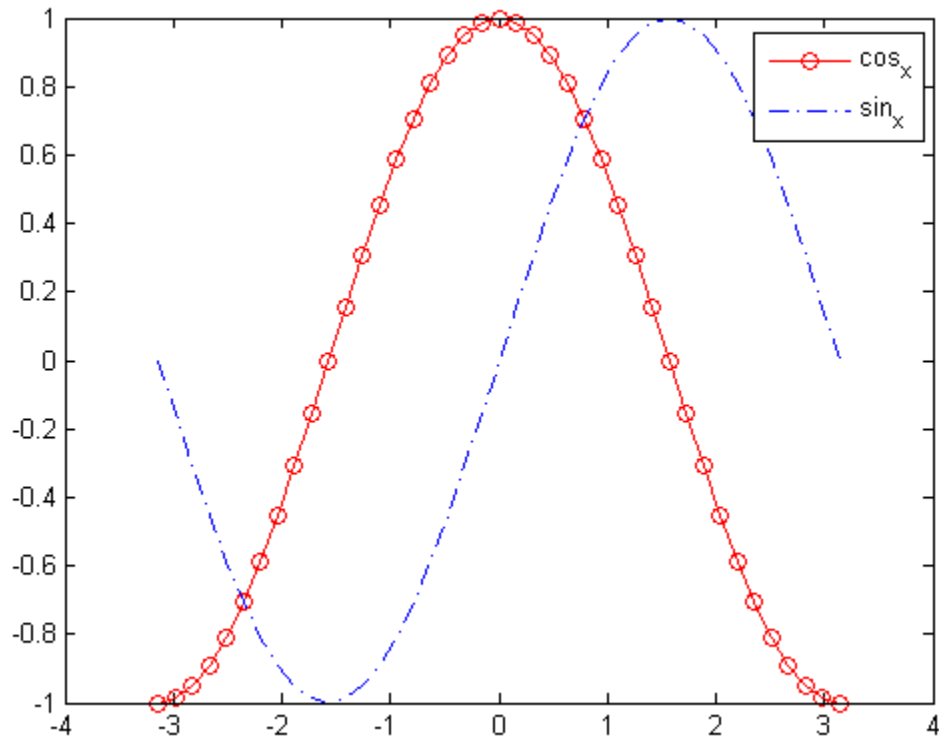
To use legend keywords like 'Location', 'Orientation', 'Off', 'Hide', or property names as legend labels, pass the string in a cell array to the legend function as,

```
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');
```



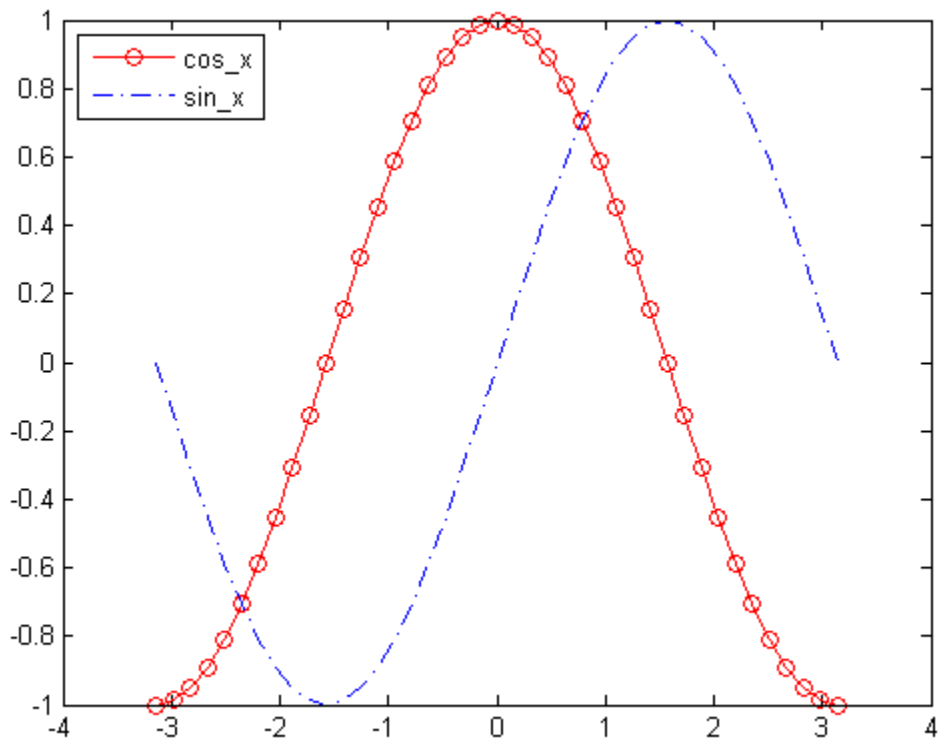
The legend reflects that `plot` specified a solid, red line ('-ro') for the cosine function and a dash-dot, blue line ('-.b') for the sine function.

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')
```

## legend

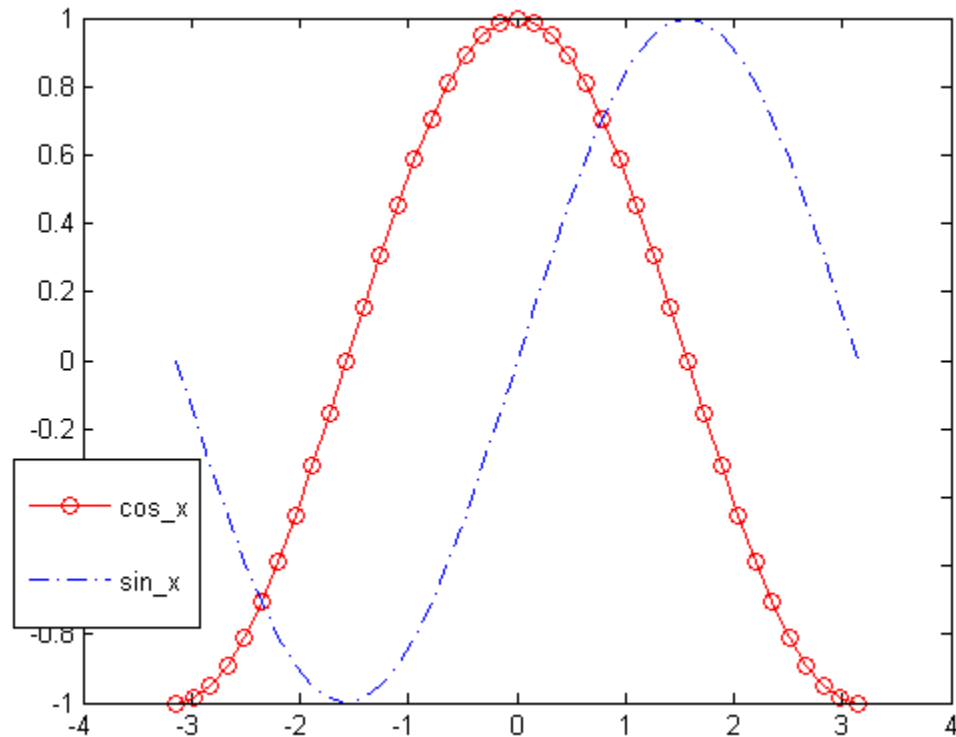
---



---

Setting the position of the legend at a particular position.

```
set(hleg1, 'Position', [.1,.2,.1,.2]);
```



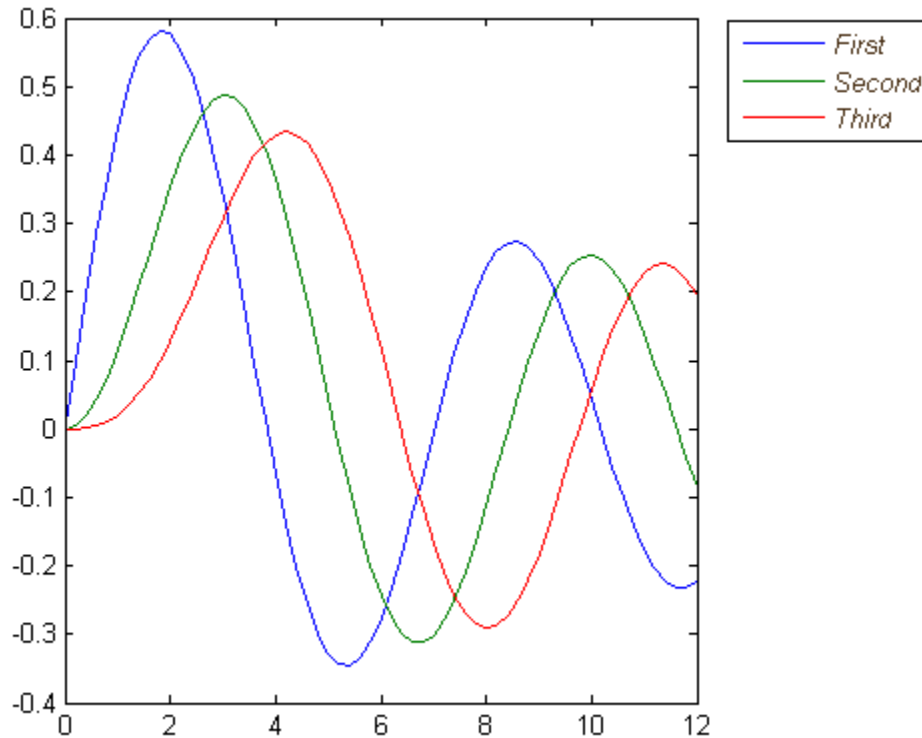
Use `besselj` to plot Bessel functions for orders 1, 2, and 3. Add a legend for the lines at the upper right, outside the axes.

```
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
```

# legend

---

```
set(hleg,'FontAngle','italic','TextColor',[.3,.2,.1])
```



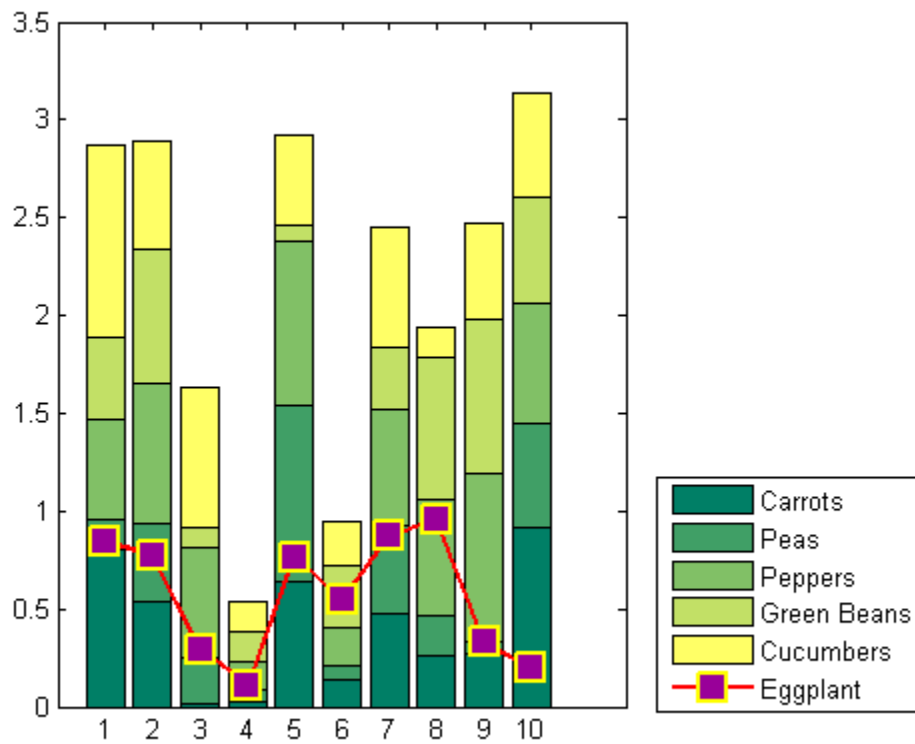
---

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')
```



# legend

---

## 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

[LineStyle](#) | [plot](#) | [Text Properties](#)

## How To

- “Adding a Legend to a Graph”

**Purpose** Associated Legendre functions

**Syntax**  
 $P = \text{legendre}(n, X)$   
 $S = \text{legendre}(n, X, 'sch')$   
 $N = \text{legendre}(n, X, 'norm')$

## 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

# legendre

---

$$\int_{-1}^1 (N_n^m(x))^2 dx = 1$$

and are related to the unnormalized associated Legendre functions

$P_n^m(x)$  by

$$N_n^m = (-1)^m \sqrt{\frac{(n + \frac{1}{2})(n - m)!}{(n + m)!}} P_n^m(x)$$

## Description

`P = legendre(n,X)` computes the associated Legendre functions

$P_n^m(x)$  of degree  $n$  and order  $m = 0, 1, \dots, n$ , 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  $q = \text{length}(X)$ . 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  $P(m+1, i, j, k, \dots)$  contains the associated Legendre function of degree  $n$  and order  $m$  evaluated at  $X(i, j, k, \dots)$ . 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 
$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

`N = legendre(n,X,'norm')` computes the fully normalized associated

Legendre functions 
$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

## 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
     2.49501
```

## Algorithms

legendre uses a three-term backward recursion relationship in  $m$ . This recursion is on a version of the Schmidt seminormalized

# Legendre

---

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

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x), \text{ by}$$

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

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)
```

# length

---

```
S =  
    f1: 'Name:'  
    f2: 'Charlie'  
    f3: 'DOB:'  
    f4: 1917  
  
structfun(@(field)length(field), S)  
ans =  
     5  
     7  
     4  
     1
```

## See Also

[numel](#) | [size](#) | [ndims](#)



|                         |                                                                                                                                                                                                                                                                                                          |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Length of containers.Map object                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>           | <code>mapLength = length(mapObj)</code>                                                                                                                                                                                                                                                                  |
| <b>Description</b>      | <code>mapLength = length(mapObj)</code> returns the number of key-value pairs in <code>mapObj</code> .                                                                                                                                                                                                   |
| <b>Input Arguments</b>  | <b>mapObj</b><br>Object of class <code>containers.Map</code> .                                                                                                                                                                                                                                           |
| <b>Output Arguments</b> | <b>mapLength</b><br>Scalar numeric value that indicates the number of key-value pairs in <code>mapObj</code> . This number is equivalent to <code>size(mapObj,1)</code> and to <code>mapObj.Count</code> .                                                                                               |
| <b>Examples</b>         | <b>Determine the Length of a Map</b><br>Create a map, and determine the number of key-value pairs:<br><pre>myKeys = {'a','b','c'};<br/>myValues = [1,2,3];<br/>mapObj = containers.Map(myKeys,myValues);<br/><br/>mapLength = length(mapObj)</pre> <p>This code returns</p> <pre>mapLength =<br/>3</pre> |
| <b>See Also</b>         | <code>containers.Map</code>   <code>isKey</code>   <code>keys</code>   <code>size</code>   <code>values</code>                                                                                                                                                                                           |

# length (serial)

---

**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

|                    |                                                                                                                                                       |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Length of time vector                                                                                                                                 |
| <b>Syntax</b>      | <code>length(tsc)</code>                                                                                                                              |
| <b>Description</b> | <code>length(tsc)</code> returns an integer that represents the length of the time vector for the <code>tscollection</code> object <code>tsc</code> . |
| <b>See Also</b>    | <code>tscollection</code>   <code>isempty (tscollection)</code>   <code>size (tscollection)</code>                                                    |

# libfunctions

---

**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`

# libfunctionsview

---

**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.

**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`

**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.

**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`

# libpointer

---

**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')`

### **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”

# 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.

**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)
```

# libstruct

---

```
ans =  
    1177
```

To clean up, first clear the `libstruct` object, and then unload the library.

```
clear sc  
unloadlibrary shrlibsampl
```

## 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

# license

---

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'
```

# license

---

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
```



---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Create light object                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Syntax</b>      | <pre>light('PropertyName',propertyvalue,...) handle = light(...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Properties</b>  | For a list of properties, see Light Properties.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b> | <p><code>light</code> creates a light object in the current axes. Lights affect only patch and surface objects.</p> <p><code>light('PropertyName',propertyvalue,...)</code> 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.</p> <p><code>handle = light(...)</code> returns the handle of the light object created.</p>                                                                                                                                                                                                                                                                                                       |
| <b>Tips</b>        | <p>You cannot see a light object <i>per se</i>, 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.</p> <p>You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see <code>set</code> and <code>get</code> for examples of how to specify these data types).</p> <p>See also the patch and surface <code>AmbientStrength</code>, <code>DiffuseStrength</code>, <code>SpecularStrength</code>, <code>SpecularExponent</code>, <code>SpecularColorReflectance</code>, and <code>VertexNormals</code> properties. Also see the lighting and material commands.</p> |
| <b>Examples</b>    | <p>Light the peaks surface plot with a light source located at infinity and oriented along the direction defined by the vector <code>[1 0 0]</code>, that is, along the <i>x</i>-axis.</p> <pre>h = surf(peaks);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

# light

---

```
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}

# Light Properties

---

## *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(gcf,'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`.

# Light Properties

---

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).

# Light 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*-,

# Light Properties

---

*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,e1)
light_handle = lightangle(az,e1)
lightangle(light_handle,az,e1)
[az,e1] = lightangle(light_handle)
```

**Description** `lightangle(az,e1)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `e1` 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,e1)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,e1)` sets the position of the light specified by `light_handle`.

`[az,e1] = 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)
end
drawnow
end
```

**See Also** `light` | `camlight` | `view`

**How To**

- “Lighting Overview”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Specify lighting algorithm                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>      | <code>lighting flat</code><br><code>lighting gouraud</code><br><code>lighting phong</code><br><code>lighting none</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b> | <p><code>lighting</code> 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 <code>lighting</code> command to have any effects, however, you must create a lighting object by using the <code>light</code> function.</p> <p><code>lighting flat</code> produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.</p> <p><code>lighting gouraud</code> calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.</p> <p><code>lighting phong</code> 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.</p> <p><code>lighting none</code> turns off lighting.</p> |
| <b>Tips</b>        | The <code>surf</code> , <code>mesh</code> , <code>pcolor</code> , <code>fill</code> , <code>fill3</code> , <code>surface</code> , and <code>patch</code> functions create graphics objects that are affected by light sources. The <code>lighting</code> command sets the <code>FaceLighting</code> and <code>EdgeLighting</code> properties of surfaces and patches appropriately for the graphics object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>See Also</b>    | <code>fill</code>   <code>fill3</code>   <code>light</code>   <code>material</code>   <code>mesh</code>   <code>patch</code>   <code>pcolor</code>   <code>shading</code>   <code>surface</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Lighting Overview”</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

# lin2mu

---

**Purpose** Convert linear audio signal to mu-law

**Syntax** `mu = lin2mu(y)`

**Description** `mu = lin2mu(y)` converts linear audio signal amplitudes in the range  $-1 \leq Y \leq 1$  to mu-law encoded “flints” in the range  $0 \leq u \leq 255$ .

**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

---

`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`:

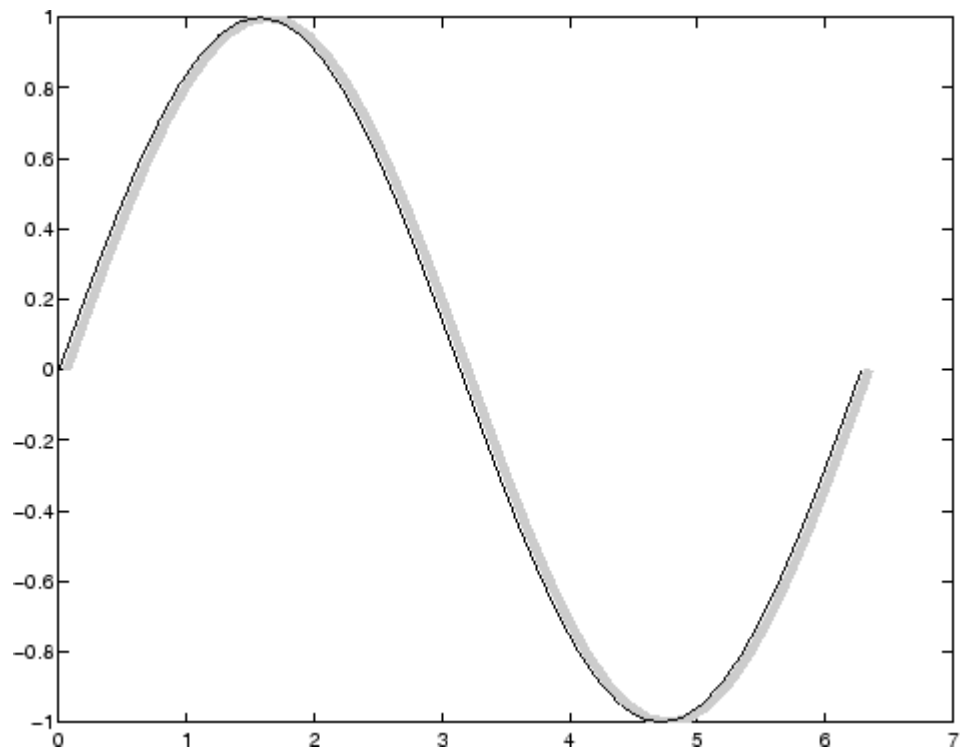
```
hline2 = line(t+.06,sin(t),'LineWidth',4,'Color',[.8 .8 .8]);
```

Finally, pull the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```

# line

---



## 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,...)
```

# line

---

```
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:

# Line Properties

---

| 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

# Line Properties

---

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(gcf, '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','-.-')
```

# Line Properties

---

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.

# Line Properties

---

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.

# Line Properties

---

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.

# Line Properties

---

---

**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}

# Line Properties

---

*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` (the factory default for axes objects), 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, `hgggroup`, 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)

# Line Properties

---

*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:

# Line Properties

---

```
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        |

# Lineseries Properties

---

## 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-to-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

# Lineseries Properties

---

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.

# Lineseries Properties

---

- 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.

# Lineseries Properties

---

- **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:

# Lineseries Properties

---

- '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 =  $\frac{1}{72}$  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    |

# Lineseries Properties

---

| 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` (the factory default for axes objects), 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, `hgggroup`, or `hgtransform`

*Parent of object.* Handle of the object's parent. The parent is normally the axes, `hgggroup`, 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.

# Lineseries Properties

---

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

# Lineseries Properties

---

*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

# Lineseries Properties

---

*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          |

# LineStyle (Line Specification)

---

| 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.

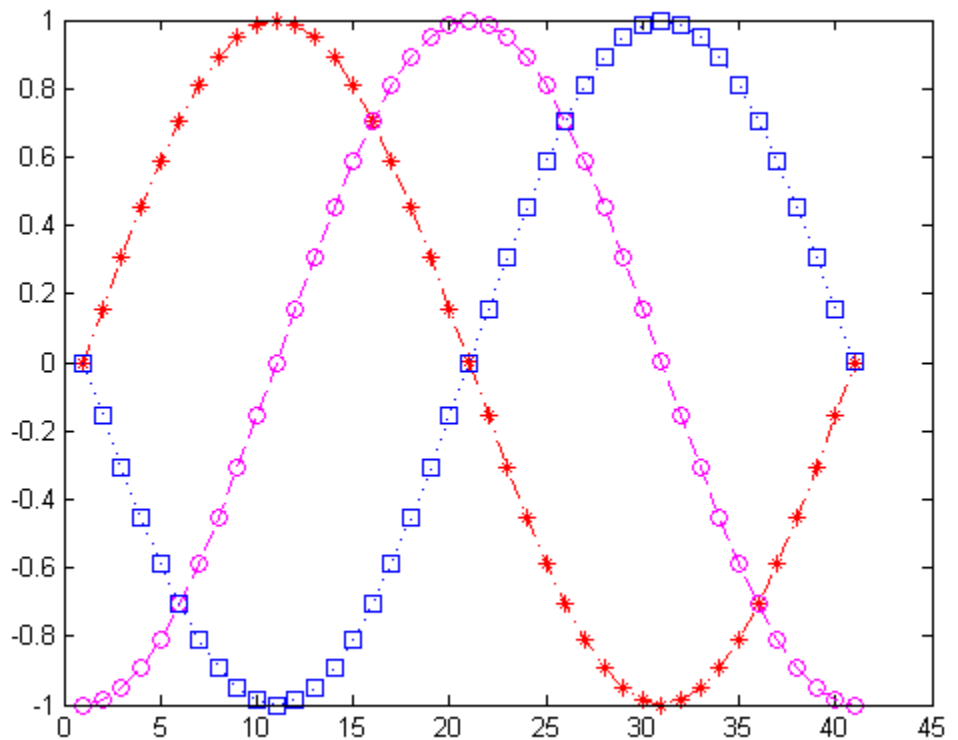
## Examples

Plot the sine function over three different ranges using different line styles, colors, and markers.

# LineSpec (Line Specification)

---

```
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
```

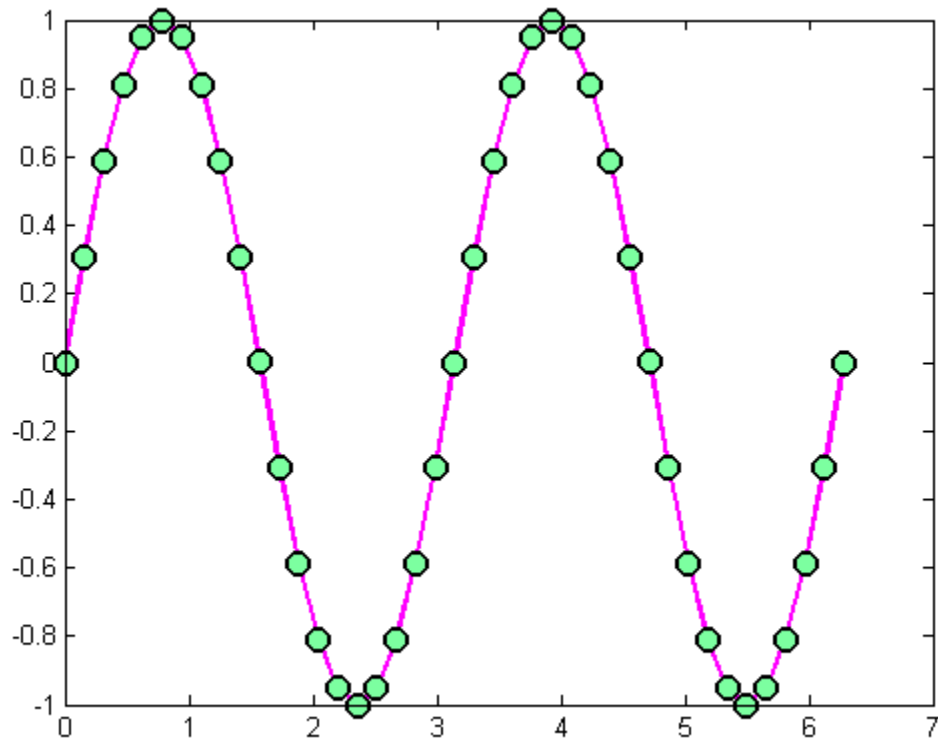


---

Create a plot illustrating how to set line properties:

# LineStyle (Line Specification)

```
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:

|                  |                               |
|------------------|-------------------------------|
| <code>x</code>   | Link $x$ -axis only.          |
| <code>y</code>   | Link $y$ -axis only.          |
| <code>xy</code>  | Link $x$ -axis and $y$ -axis. |
| <code>off</code> | 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.

---

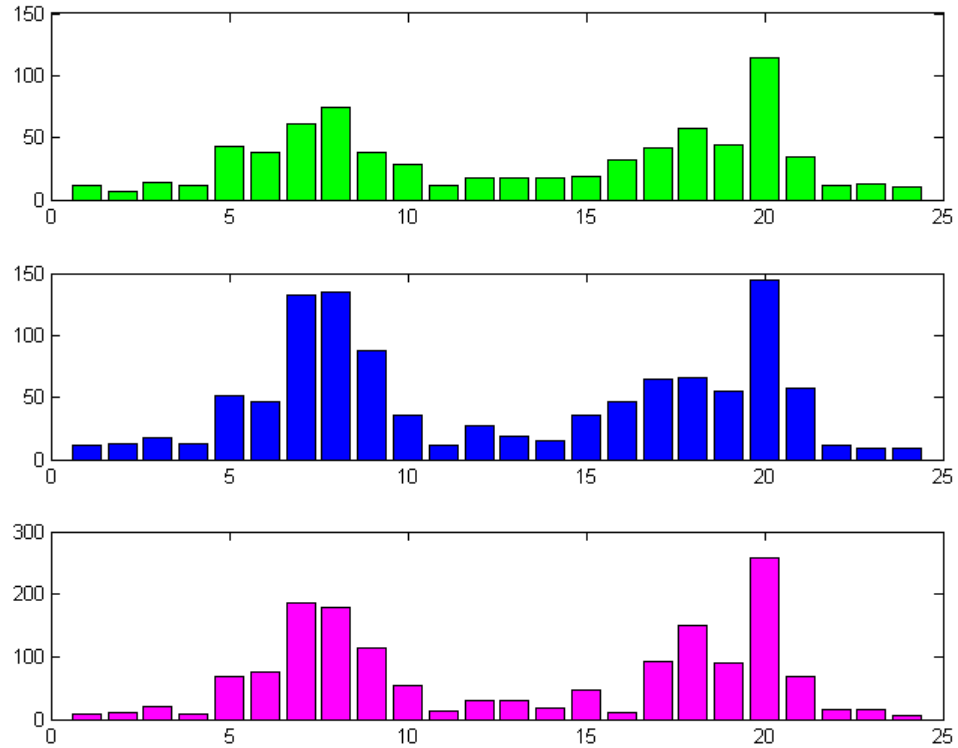
## Examples

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:

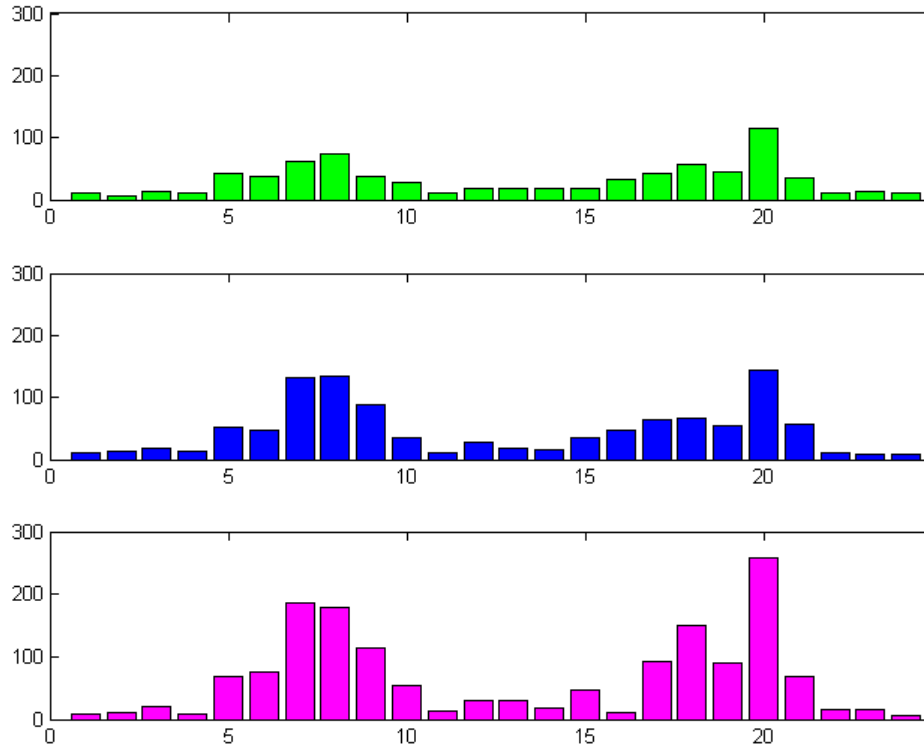
```
load count.dat % Contains a 3-column vector named count
figure
ax(1) = subplot(3,1,1);
bar(ax(1),count(:,1),'g');
ax(2) = subplot(3,1,2);
bar(ax(2),count(:,2),'b');
ax(3) = subplot(3,1,3);
bar(ax(3),count(:,3),'m');
```


# linkaxes



```
% Wait three seconds and then link the axes  
pause(3)  
linkaxes([ax(3) ax(2) ax(1)], 'xy'); % Base Y-limits on bottom subplot
```

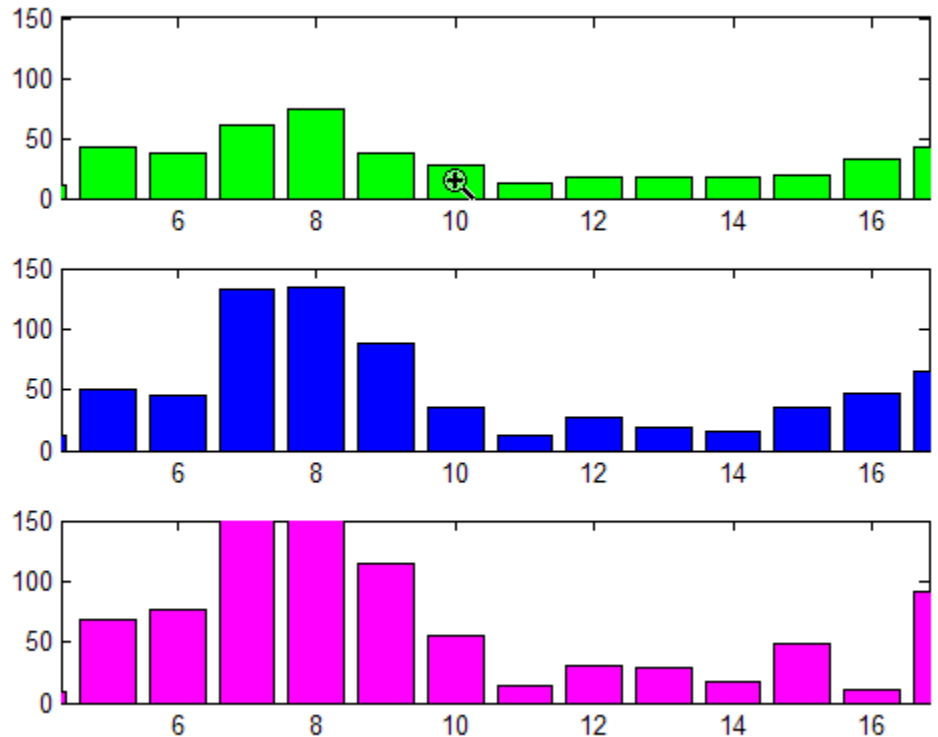




Click the **Zoom in** icon  to enter Zoom mode, and click the mouse near the middle of the top axes. You observe all three axes respond the same. If you pan in any of the axes, all of them also respond the same.

# linkaxes

---

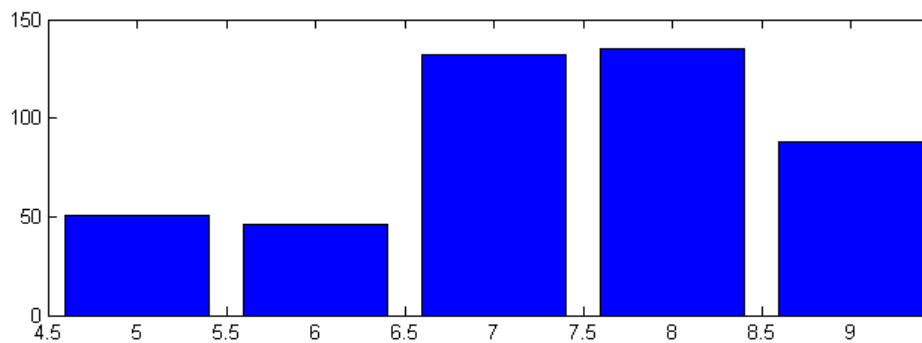
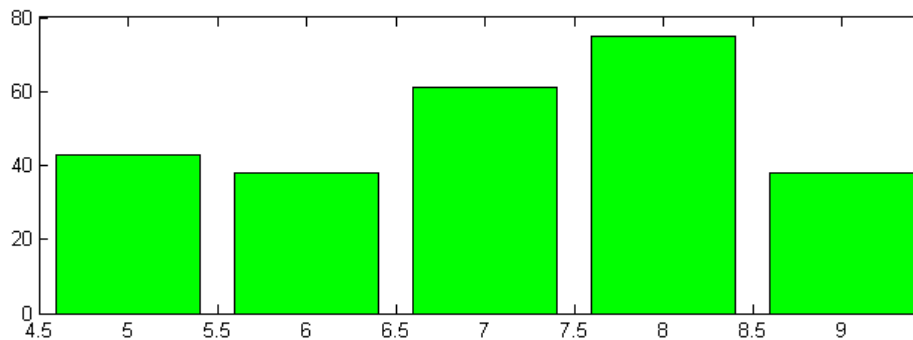


---

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
```



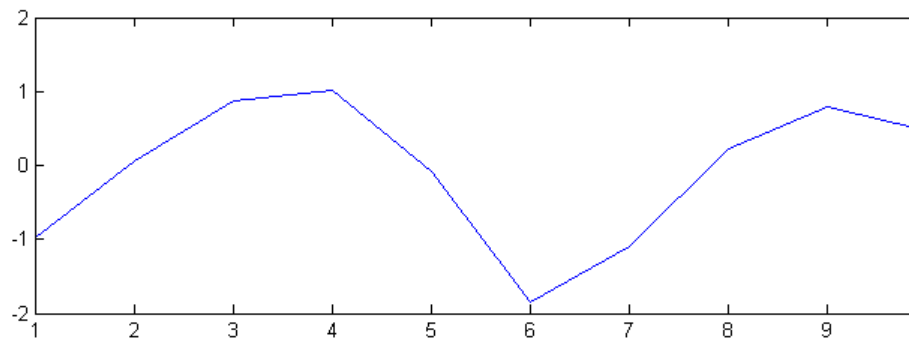
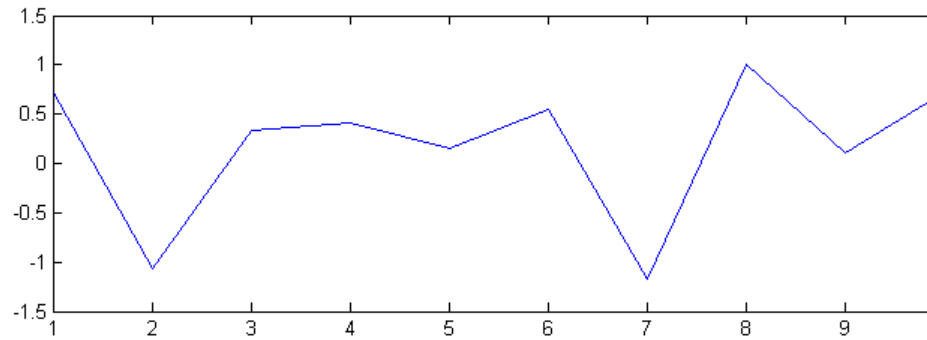
Choose the Pan tool (**Tools > Pan**) (or type `pan` on). Drag the top axes. Both axes pan uniformly in  $x$ , but only the one you pan moves in the  $y$  direction.

# linkaxes

---

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
```

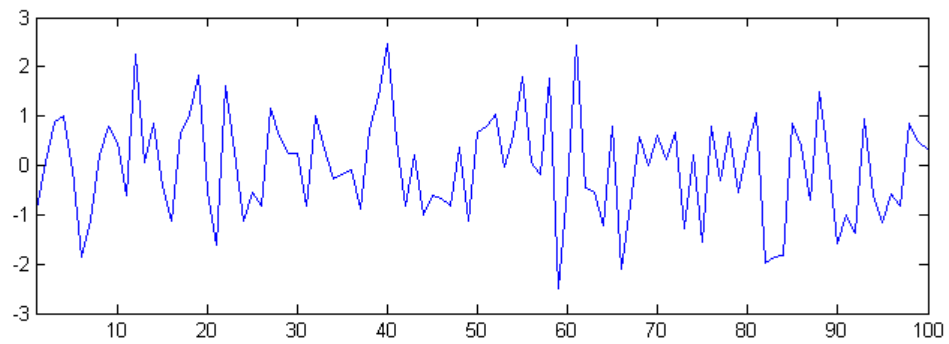
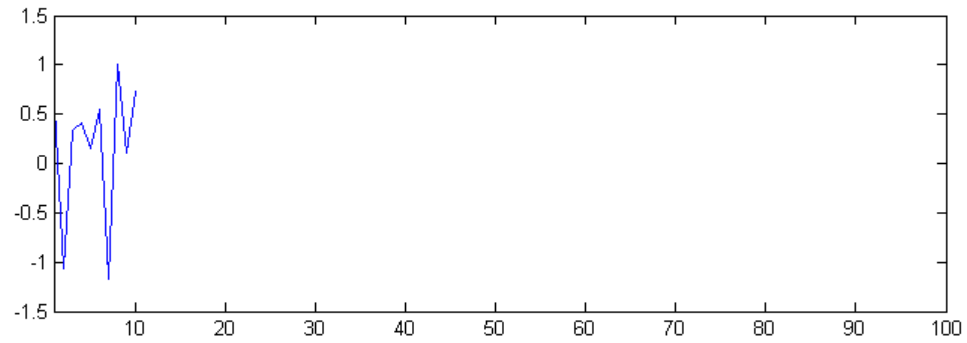


In order to display the full range of  $x$ -values, override the axes limits that `linkaxes` established.

```
set(a2,'xlimmode','auto'); % Now both axes run from 1-100 in x
                           % You could also use set(a2,'xlim',[1 100])
```

# linkaxes

---



## See Also

[linkdata](#) | [linkprop](#) | [pan](#) | [zoom](#)

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Automatically update graphs when variables change                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Syntax</b>      | <pre>linkdata on linkdata off linkdata linkdata(<i>figure_handle</i>,...) linkobj = linkdata(<i>figure_handle</i>)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b> | <p><code>linkdata on</code> turns on data linking for the current figure.</p> <p><code>linkdata off</code> turns data linking off.</p> <p><code>linkdata</code> by itself toggles the state of data linking.</p> <p><code>linkdata(<i>figure_handle</i>,...)</code> applies the function to the specified figure handle.</p> <p><code>linkobj = linkdata(<i>figure_handle</i>)</code> returns a <i>linkdata object</i> for the specified figure. The object has one read-only property, <code>Enable</code>, the string 'on' or 'off', depending on the linked state of the figure.</p> <p>Data linking connects graphs in figure windows to variables in the base or a function's workspace via their <code>XDataSource</code>, <code>YDataSource</code>, and <code>ZDataSource</code> properties. When you turn on data linking for a figure, MATLAB compares variables in the current (base or function caller) workspace with the <code>XData</code>, <code>YData</code>, and <code>ZData</code> properties of graphs in the affected figure to try to match them. When a match is found, the appropriate <code>XDataSource</code>, <code>YDataSource</code> and/or <code>ZDataSource</code> for the graph are set to strings that name the matching variables.</p> <p>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.</p> <p>When a figure containing graphs is linked and any variable identified as <code>XDataSource</code>, <code>YDataSource</code>, and/or <code>ZDataSource</code> changes its values in the workspace, all graphs displaying it in that and other linked figures</p> |

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 2-2648
- “Restoring Links that Break” on page 2-2648
- “Linking Rapidly Changing Data” on page 2-2649
- “Linking Brushed Graphs” on page 2-2649

## 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 2-2651 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 2-534 in the brush reference page.

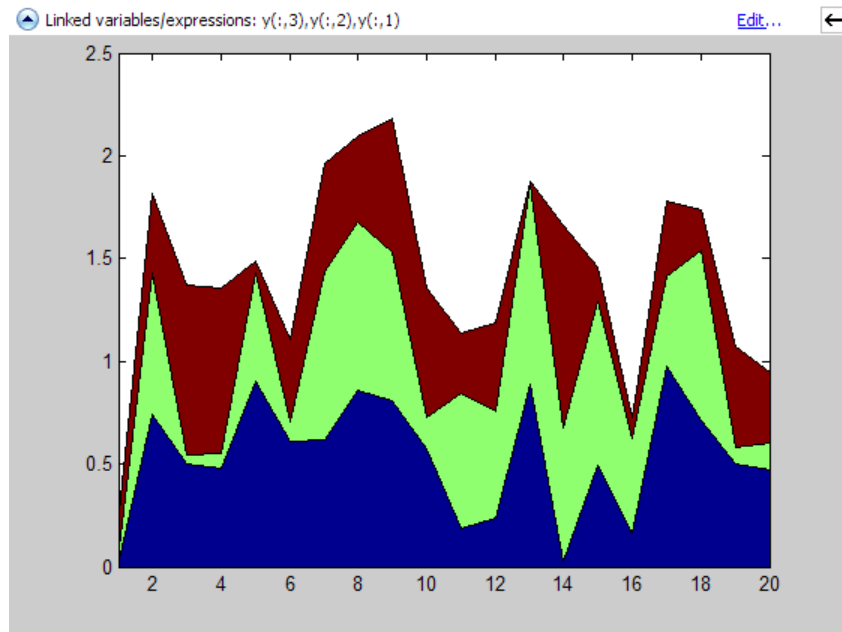
## Examples

### Example 1

Create two variables, graph them as areseries, and link the plot to them:

```
x = [1:20];  
y = rand(20,3);  
area(x,y)  
linkdata on
```

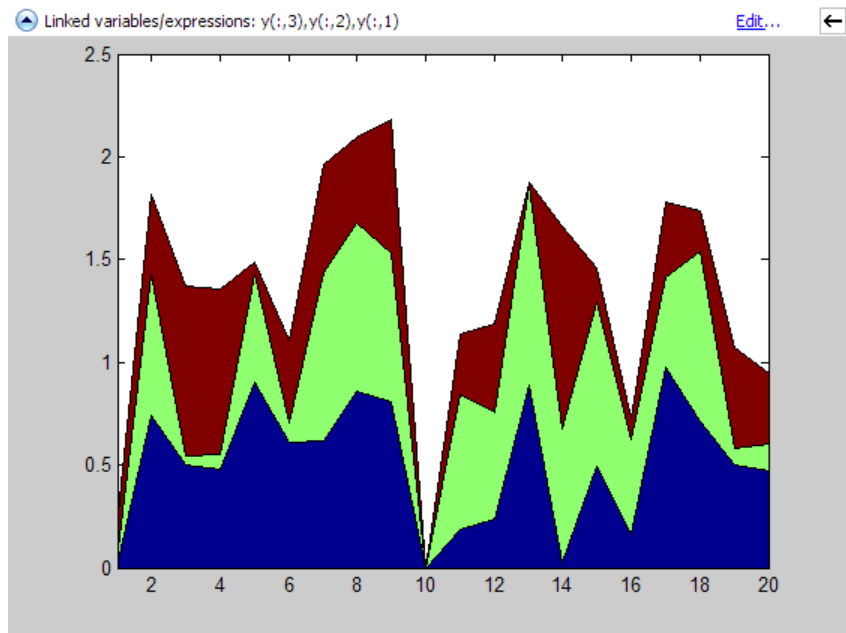
# linkdata



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,

# linkdata

---

```
x = eig(randn(20,20))
```

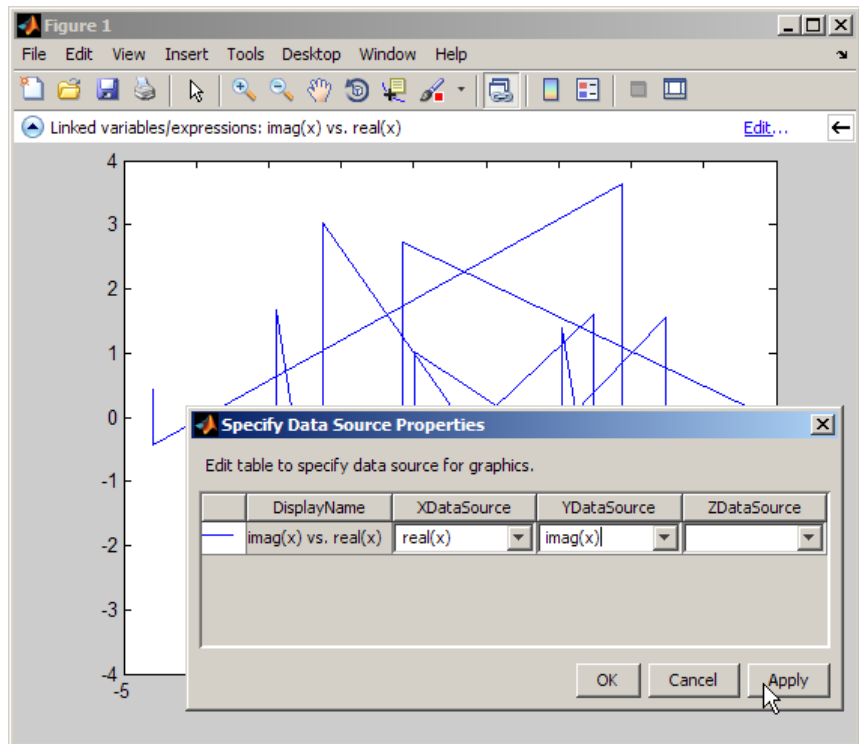
```
whos
```

| 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.

# linkdata

---

---

**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”

## 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 2-2656 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.

## About Link Objects

- To stop properties from linking, use the `removeprop` method.

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 2-2657 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                                                          |
|---------------------------|------------------------------------------------------------------|
| <code>addtarget</code>    | Add specified graphics object to the link object's targets.      |
| <code>removetarget</code> | Remove specified graphics object from the link object's targets. |
| <code>addprop</code>      | Add specified property to the linked properties.                 |
| <code>removeprop</code>   | 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';
```

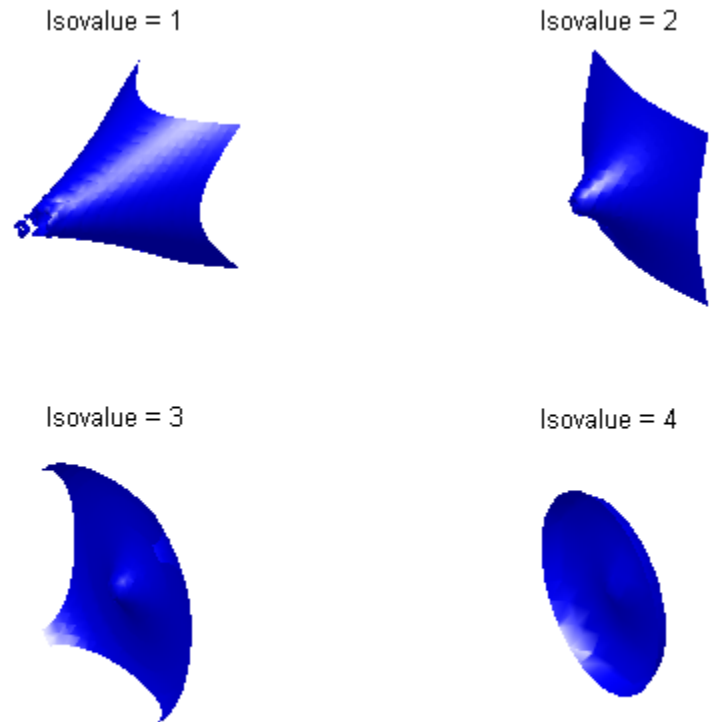
- 2 Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters. MATLAB functions used are `subplot`, `patch`, `isosurface`, `title`, and `num2str`.

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 is 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.

# linkprop

---

- 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:

# linsolve

- 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.

| <b>Keyword</b> | <b>Generator</b>                                            | <b>Multiple Stream and Substream Support</b> | <b>Approximate Period In Full Precision</b> |
|----------------|-------------------------------------------------------------|----------------------------------------------|---------------------------------------------|
| 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                                          | $2^{127}$                                   |

# RandStream.list

---

| <b>Keyword</b> | <b>Generator</b>                                                   | <b>Multiple Stream and Substream Support</b> | <b>Approximate Period In Full Precision</b> |
|----------------|--------------------------------------------------------------------|----------------------------------------------|---------------------------------------------|
| 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.                                                 |

# listdlg

---

| Parameter | Description                                       |
|-----------|---------------------------------------------------|
| 'fus'     | Frame/uicontrol spacing, in pixels. Default is 8. |
| 'ffs'     | Frame/figure spacing, in pixels. Default is 8.    |

---

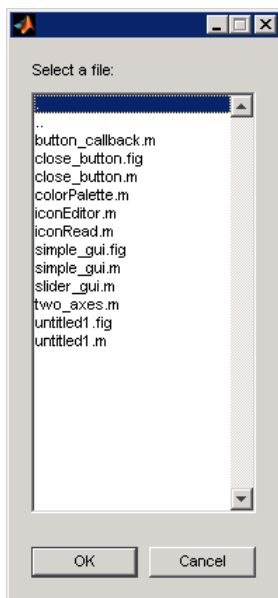
**Note** A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

---

## 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:

|                                    |                                                                                                                                                |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>var1, var2, ...</code>       | Load the listed variables. Use the '*' wildcard to match patterns. For example, <code>load('A*')</code> loads all variables that start with A. |
| <code>'-regex', expressions</code> | 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

# load

---

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;
```

# load

---

```
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”](#)
- [“Ways to Load MAT-Files”](#)
- [“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 =
```

# load (COM)

---

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 (serial)

---

```
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.

# loadlibrary

---

## 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”.

### **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.

### **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**.

### **Name-Value Pair Arguments**

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.

## **addheader - Additional header file**

string

Additional header file, specified as the comma-separated pair consisting of 'addheader' and a string. Specify the file name without a file extension. Loads the functions defined in the additional header file. Each file specified by addheader must have a corresponding #include statement in the base header file.

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.

# loadlibrary

---

## Output Arguments

### **notfound** - Names of functions

cell array

Names of functions found in header files but missing from the library, returned as cell array.

### **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.

# loadlibrary

---

## Definitions

### Prototype File

A file of MATLAB commands which you can modify and use in place of a header file.

### Thunk File

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](#)

## Related Examples

- “Create Alias Function Name Using Prototype File”

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Modify load process for object                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>          | <code>b = loadobj (a)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b>     | <p><code>b = loadobj (a)</code> is called by the <code>load</code> function if the class of <code>a</code> defines a <code>loadobj</code> method. <code>load</code> returns <code>b</code> as the value loaded from a MAT-file.</p> <p>Define a <code>loadobj</code> method when objects of the class require special processing when loaded from MAT-files. If you define a <code>saveobj</code> method, then define a <code>loadobj</code> method to restore the object to the desired state. Define <code>loadobj</code> as a static method so it can accept as an argument whatever object or structure that you saved in the MAT-file. See “Implement <code>loadobj</code> as a Static Method”.</p> <p>When loading a subclass object, <code>load</code> calls only the subclass <code>loadobj</code> method. If a superclass defines a <code>loadobj</code> 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 <code>loadobj</code> methods.</p> <p>If any superclass in a class hierarchy defines a <code>loadobj</code> method, then the subclass <code>loadobj</code> method must ensure that the subclass and superclass objects load properly. Ensure proper loading by calling the superclass <code>loadobj</code> (or other methods) from the subclass <code>loadobj</code> method. See “Saving and Loading Objects from Class Hierarchies”.</p> |
| <b>Input Arguments</b> | <p><b>a</b></p> <p>The input argument, <code>a</code>, can be:</p> <ul style="list-style-type: none"><li>• The object as loaded from the MAT-file.</li><li>• A structure created by <code>load</code> (if <code>load</code> cannot resolve the object).</li><li>• A struct returned by the <code>saveobj</code> method.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>See Also</b>        | <code>load</code>   <code>save</code>   <code>saveobj</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Tutorials</b>       | <ul style="list-style-type: none"><li>• “Control Save and Load”</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

# log

---

**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.

$$\log(z) = \log(\text{abs}(z)) + i*\text{atan2}(y,x)$$

**Examples** The statement `abs(log(-1))` is a clever way to generate  $\pi$ .

```
ans =
```

```
3.1416
```

**See Also** `exp` | `log10` | `log2` | `logm` | `reallog`



**Purpose** Common (base 10) logarithm

**Syntax**  $Y = \log_{10}(X)$

**Description** The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

$Y = \log_{10}(X)$  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`

**Purpose** Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

**Syntax**  $Y = \log_2(X)$   
 $[F, E] = \log_2(X)$

**Description**  $Y = \log_2(X)$  computes the base 2 logarithm of the elements of  $X$ .  
 $[F, E] = \log_2(X)$  returns arrays  $F$  and  $E$ . Argument  $F$  is an array of real values, usually in the range  $0.5 \leq \text{abs}(F) < 1$ . For real  $X$ ,  $F$  satisfies the equation:  $X = F \cdot 2.^E$ . Argument  $E$  is an array of integers that, for real  $X$ , satisfy the equation:  $X = F \cdot 2.^E$ .

**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  $[F, E] = \log_2(X)$  yields the values:

| <b>X</b> | <b>F</b>  | <b>E</b> |
|----------|-----------|----------|
| 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`

# logical

---

**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

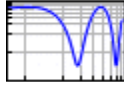
`true` | `false` | `islogical` | Logical Operators: Elementwise | Logical Operators: Short-circuit

# loglog

---

## Purpose

Log-log scale plot



## Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineStyle,...)
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  $X_n$  versus  $Y_n$  pairs. If only  $X_n$  or  $Y_n$  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,LineStyle,...)` plots all lines defined by the  $X_n, Y_n, LineSpec$  triples, where `LineStyle` determines line type, marker symbol, and color of the plotted lines. You can mix  $X_n, Y_n, LineSpec$  triples with  $X_n, Y_n$  pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineStyle,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

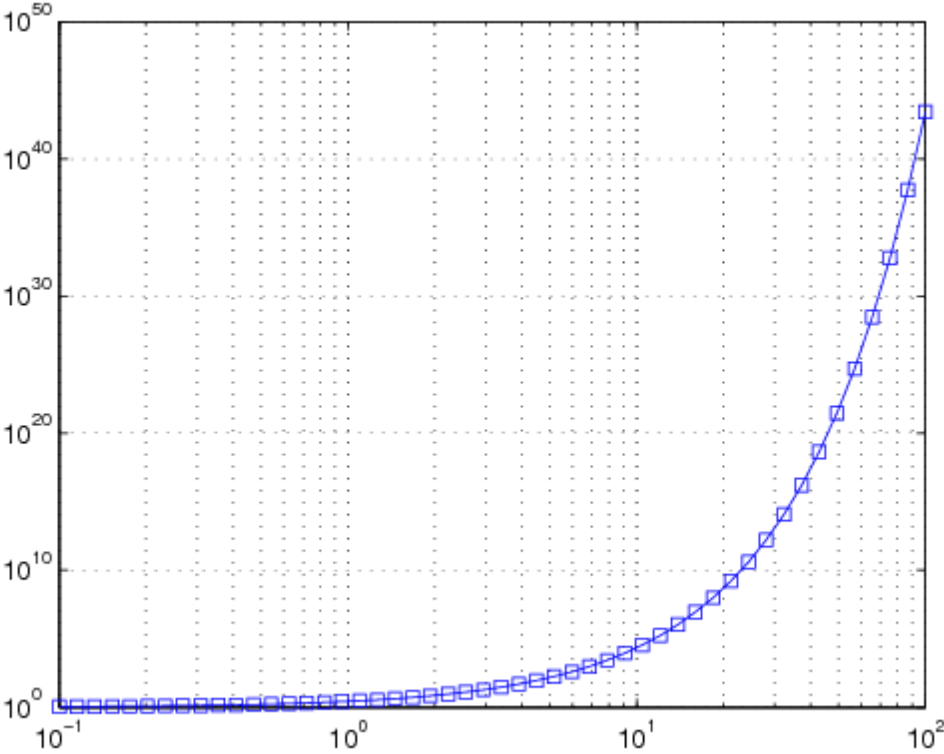
The OpenGL renderer does not support logarithmic-scale axes. MATLAB automatically selects a different renderer when using logarithmic scaling. If you set the figure `Renderer` property to `opengl`, axis scales become linear. See the figure `Renderer` property for more information on renderers.

## Examples

Create a simple `loglog` plot with square markers.

```
x = logspace(-1,2);  
loglog(x,exp(x),'-s')  
grid on
```

# loglog



**See Also**

LineSpec | plot | semilogx | semilogy



**Purpose** Matrix logarithm

**Syntax**  $L = \text{logm}(A)$   
 $[L, \text{exitflag}] = \text{logm}(A)$

**Description**  $L = \text{logm}(A)$  is the principal matrix logarithm of  $A$ , the inverse of  $\text{expm}(A)$ .  $L$  is the unique logarithm for which every eigenvalue has imaginary part lying strictly between  $-\pi$  and  $\pi$ . 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.

$[L, \text{exitflag}] = \text{logm}(A)$  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  $\text{logm}(A)$ .

Some matrices, like  $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ , do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

**Limitations** For most matrices:

$$\text{logm}(\text{expm}(A)) = A = \text{expm}(\text{logm}(A))$$

These identities may fail for some  $A$ . For example, if the computed eigenvalues of  $A$  include an exact zero, then  $\text{logm}(A)$  generates infinity. Or, if the elements of  $A$  are too large,  $\text{expm}(A)$  may overflow.

**Examples** Suppose  $A$  is the 3-by-3 matrix

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

```
      0      0      -1
```

and  $Y = \text{expm}(A)$  is

```
Y =
  2.7183    1.7183    1.0862
      0    1.0000    1.2642
      0      0    0.3679
```

Then  $A = \text{logm}(Y)$  produces the original matrix  $A$ .

```
Y =
  1.0000    1.0000    0.0000
      0      0    2.0000
      0      0   -1.0000
```

But  $\text{log}(A)$  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`

# logspace

---

**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

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | Search for keyword in all help entries                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Alternatives</b> | As an alternative to the <code>lookfor</code> function, use the Function Browser.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Syntax</b>       | <code>lookfor topic</code><br><code>lookfor topic -all</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b>  | <p><code>lookfor topic</code> searches for the string <code>topic</code> 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, <code>lookfor</code> displays the H1 line.</p> <p><code>lookfor topic -all</code> searches the entire first comment block of a MATLAB program file looking for <code>topic</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Examples</b>     | <p>For example:</p> <pre>lookfor inverse</pre> <p>finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with</p> <pre>which inverse</pre> <p>or</p> <pre>what inverse</pre> <p>These functions run more quickly, but probably fail to find anything because MATLAB does not have a function <code>inverse</code>.</p> <p>In summary, <code>what</code> lists the functions in a given folder, <code>which</code> finds the folder containing a given function or file, and <code>lookfor</code> finds all functions in all folders that might have something to do with a given keyword.</p> <p>Even more extensive than the <code>lookfor</code> 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</p> |

# lookfor

---

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 Using the Function Browser”
- “Search Syntax and Tips”

**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`

# ls

---

**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  $\text{rank}(A) < n$ , `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  $V$ .

$x = \text{lscov}(A, b, V, \text{alg})$  specifies the algorithm used to compute  $x$  when  $V$  is a matrix.  $\text{alg}$  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.

$[x, \text{stdx}] = \text{lscov}(\dots)$  returns the estimated standard errors of  $x$ . When  $A$  is rank deficient,  $\text{stdx}$  contains zeros in the elements corresponding to the necessarily zero elements of  $x$ .

$[x, \text{stdx}, \text{mse}] = \text{lscov}(\dots)$  returns the mean squared error. If  $b$  is assumed to have covariance matrix  $\sigma^2 V$  (or  $(\sigma^2) \times \text{diag}(1./W)$ ), then  $\text{mse}$  is an estimate of  $\sigma^2$ .

$[x, \text{stdx}, \text{mse}, S] = \text{lscov}(\dots)$  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  $x$ .  $\text{lscov}$  cannot return  $S$  if it is called with multiple right-hand sides, that is, if  $\text{size}(B, 2) > 1$ .

The standard formulas for these quantities, when  $A$  and  $V$  are full rank, are

- $x = \text{inv}(A' * \text{inv}(V) * A) * A' * \text{inv}(V) * B$
- $\text{mse} = B' * (\text{inv}(V) - \text{inv}(V) * A * \text{inv}(A' * \text{inv}(V) * A) * A' * \text{inv}(V)) * B ./ (m - n)$
- $S = \text{inv}(A' * \text{inv}(V) * A) * \text{mse}$
- $\text{stdx} = \text{sqrt}(\text{diag}(S))$

However,  $\text{lscov}$  uses methods that are faster and more stable, and are applicable to rank deficient cases.

$\text{lscov}$  assumes that the covariance matrix of  $B$  is known only up to a scale factor.  $\text{mse}$  is an estimate of that unknown scale factor, and  $\text{lscov}$  scales the outputs  $S$  and  $\text{stdx}$  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
mse =
    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 \

**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`:

# 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 \

# lsqr

---

## 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  $\text{norm}(b-A*x)$ . 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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $M \setminus x$  and `mfun(x, 'transp')` returns  $M' \setminus x$ .

`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    | <code>lsqr</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.           |
| 1    | <code>lsqr</code> iterated <code>maxit</code> times but did not converge.                                             |
| 2    | Preconditioner $M$ was ill-conditioned.                                                                               |
| 3    | <code>lsqr</code> stagnated. (Two consecutive iterates were the same.)                                                |
| 4    | One of the scalar quantities calculated during <code>lsqr</code> 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  $0 \leq iter \leq maxit$ .

`[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.

# lsqr

---

## 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 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.

`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  $Y = U+L - \text{eye}(\text{size}(A))$ . 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  $P*(R\backslash A)*Q = L*U$  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  $R(:,p)\backslash A(:,q) = L*U$ .

`[...]` = `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

$$A(i,j) \geq \text{thresh}(2) * \max(\text{abs}(A(j:m,j)))$$

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  $1/\min(\text{thresh})$  or less.

If `A` is not as described above, MATLAB uses an asymmetric strategy. In this case, the sparsest row `i` where

$$A(i,j) \geq \text{thresh}(1) * \max(\text{abs}(A(j:m,j)))$$

is selected. A value of 1.0 results in conventional partial pivoting. Entries in `L` have an absolute value of  $1/\text{thresh}(1)$  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  $P*A*Q \neq L*U$ . 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

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix};$$

To see the LU factorization, call `lu` with two output arguments.

$$[L1,U] = \text{lu}(A)$$

L1 =

$$\begin{bmatrix} 0.1429 & 1.0000 & 0 \\ 0.5714 & 0.5000 & 1.0000 \\ 1.0000 & 0 & 0 \end{bmatrix}$$

U =

$$\begin{bmatrix} 7.0000 & 8.0000 & 0 \\ 0 & 0.8571 & 3.0000 \\ 0 & 0 & 4.5000 \end{bmatrix}$$

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 1s 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,  $X = \text{inv}(A)$ , is actually computed from the inverses of the triangular factors

$$X = \text{inv}(U)*\text{inv}(L1)$$

Using three arguments on the left side to get the permutation matrix as well,

$$[L2,U,P] = \text{lu}(A)$$

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,  $\text{inv}(U) * \text{inv}(L)$  results in the permutation of  $\text{inv}(A)$  given by  $\text{inv}(P) * \text{inv}(A)$ .

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

$$x = A \backslash b$$

The solution is actually computed by solving two triangular systems

$$y = L \backslash b$$

$$x = U \backslash y$$

### **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.

$$B = \text{bucky};$$

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

$$[L,U,P,Q] = \text{lu}(B);$$

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 \,/

# luinc

---

**Purpose** Sparse incomplete LU factorization

---

**Note** `luinc` will be removed in a future version. 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                                                                                                                                                                                                                                        |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>droptol</code> | Drop tolerance of the incomplete factorization.                                                                                                                                                                                                    |
| <code>milu</code>    | If <code>milu</code> is 1, <code>luinc</code> 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.                |
| <code>udiag</code>   | If <code>udiag</code> is 1, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is 0.                                                                                                   |
| <code>thresh</code>  | 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. <code>thresh</code> is described in greater detail in the <code>lu</code> 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:

$$\text{nnz}(L) + \text{nnz}(U) = \text{nnz}(A) + n, \text{ where } A \text{ is } n\text{-by-}n.$$

The product  $L*U$  agrees with  $A$  over its sparsity pattern.  
 $(L*U) .* \text{spones}(A) - A$  has entries of the order of  $\text{eps}$ .

$[L,U] = \text{luinc}(A,\text{options})$  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  $A$ .  $\text{luinc}(A,\text{options})$  returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

$[L,U,P] = \text{luinc}(\dots)$  returns a unit lower triangular matrix in  $L$ , an upper triangular matrix in  $U$ , and a permutation matrix in  $P$ .

$[L,U,P] = \text{luinc}(A, '0')$  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.  
 $(L*U) .* \text{spones}(P*A) - P*A$  has entries of the order of  $\text{eps}$ .

$[L,U,P] = \text{luinc}(A,\text{options})$  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 1s 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.

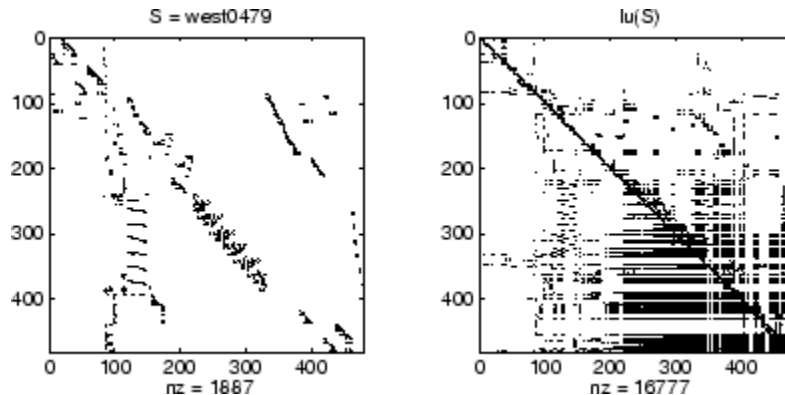
## Limitations

`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);
```



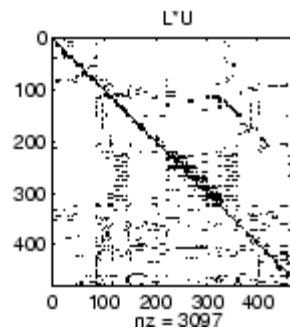
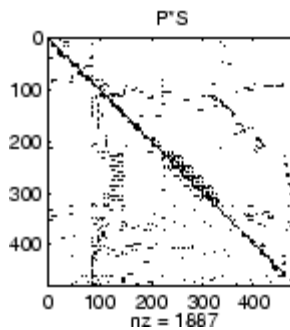
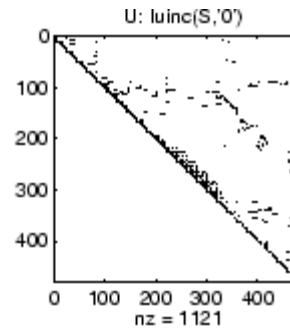
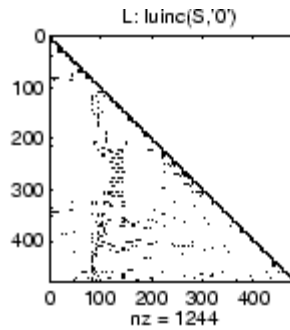
Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S, '0');
```

```
D = (L*U).*spones(P*S) - P*S;
```

`spones(U)` and `spones(triu(P*S))` are identical.

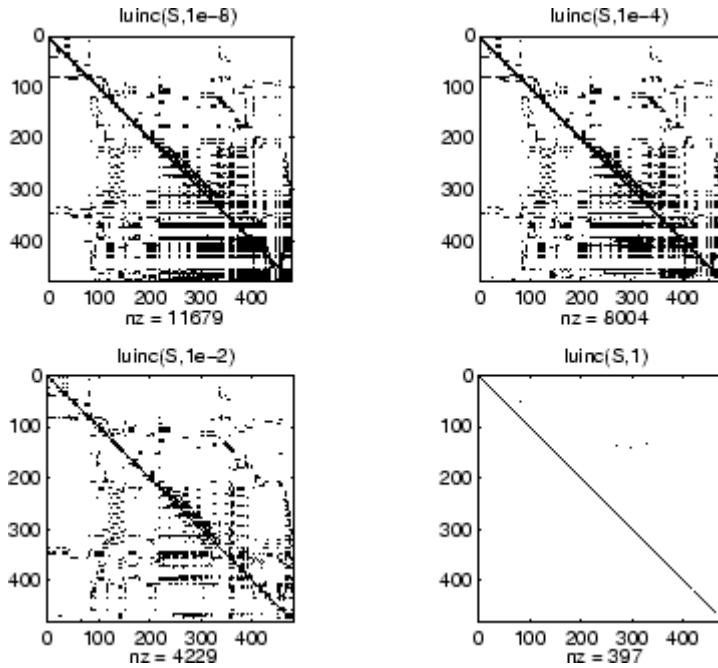
`spones(L)` and `spones(tril(P*S))` disagree at 73 places on the diagonal, where L is 1 and P\*S is 0, and also at position (206,113), where L is 0 due to cancellation, and P\*S is -1. D has entries of the order of eps.

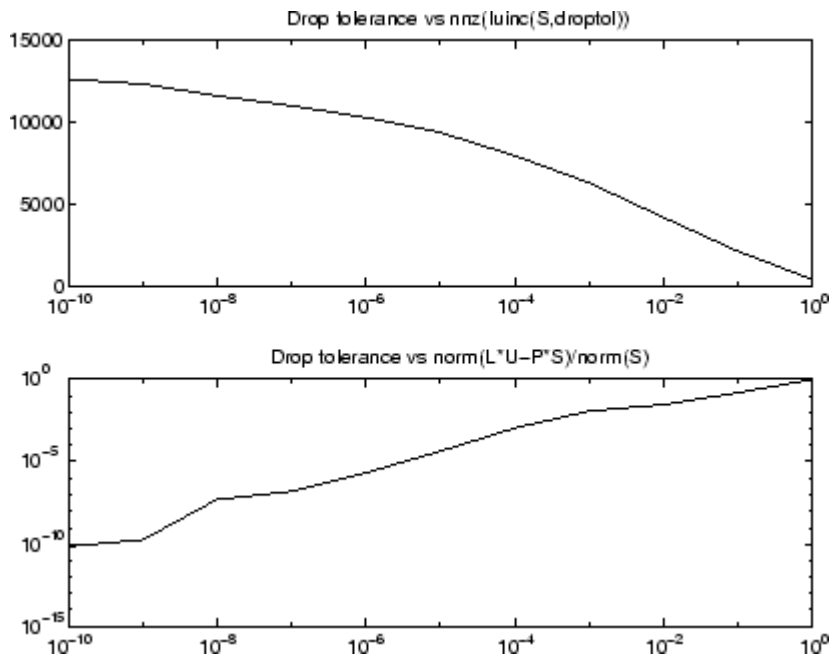


```
[ILO,IU0,IP0] = luinc(S,0);  
[IL1,IU1,IP1] = luinc(S,1e-10);
```

```
·  
·  
·
```

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  $\text{norm}(L*U-P*S, 1) / \text{norm}(S, 1)$  in the second figure below.





## Algorithms

`luinc(A, '0')` is based on the “KJJ” 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  $n/2 + 2$ .

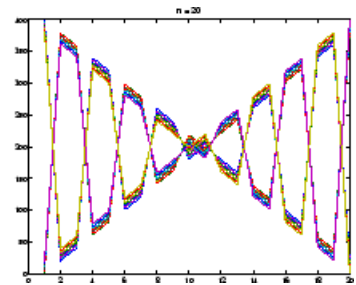
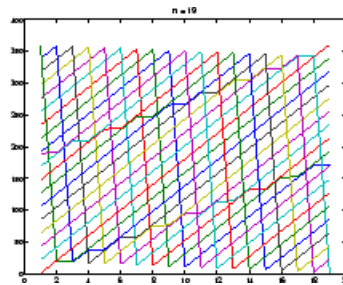
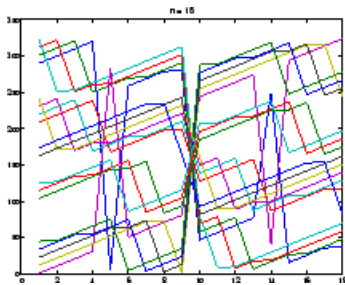
```
[(3:20)', r(3:20)']
```

```
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`

# makehgtform

---

**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”

# containers.Map

---

**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`.

# containers.Map

---

## 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

|                     |                                                                |
|---------------------|----------------------------------------------------------------|
| <code>isKey</code>  | Determine if <code>containers.Map</code> object contains key   |
| <code>keys</code>   | Identify keys of <code>containers.Map</code> object            |
| <code>length</code> | Length of <code>containers.Map</code> object                   |
| <code>remove</code> | Remove key-value pairs from <code>containers.Map</code> object |
| <code>size</code>   | Size of <code>containers.Map</code> object                     |
| <code>values</code> | Identify values in <code>containers.Map</code> object          |

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects in the MATLAB Programming Fundamentals 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

# containers.Map

---

```
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'    'May'
```

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
```

# containers.Map

---

## 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

# mat2cell

---

**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

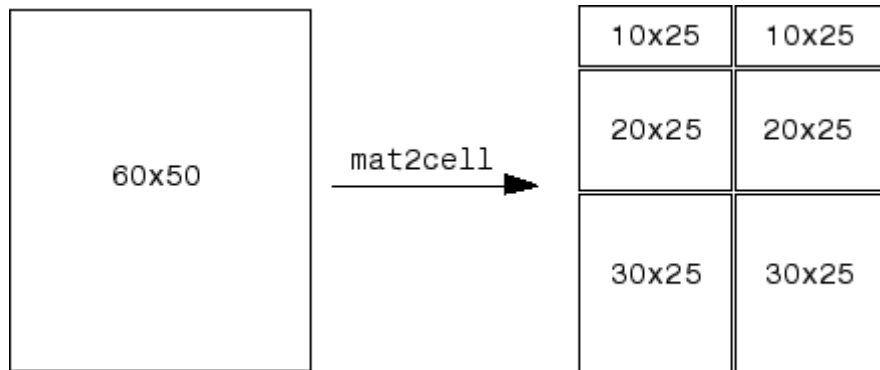
**A**  
Any type of array.

### **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  $n == \text{numel}(\text{rowDist})$ .

## 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]
```

# mat2cell

---

```
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](#)

**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](#)



**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.

# material

---

## 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 `material.m` file to see the actual values set (enter the command type `material`).

## See Also

`light` | `lighting` | `patch` | `surface`

## How To

- “Lighting Overview”

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Load and save parts of variables in MAT-files                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Syntax</b>          | <pre>matObj = matfile(filename) matObj = matfile(filename, 'Writable', isWritable)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b>     | <p><code>matObj = matfile(filename)</code> constructs a <code>matlab.io.MatFile</code> object that can load or save parts of variables in MAT-file <code>filename</code>. MATLAB does not load any data from the file into memory when creating the object.</p> <p><code>matObj = matfile(filename, 'Writable', isWritable)</code> enables or disables write access to the file for object <code>matObj</code>. Possible values for <code>isWritable</code> are logical true (1) or false (0).</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Input Arguments</b> | <p><b>filename</b></p> <p>String enclosed in single quotation marks that specifies the name of a MAT-file.</p> <p><code>filename</code> can include a full or partial path, otherwise <code>matfile</code> searches for the file along the MATLAB search path. If <code>filename</code> does not include an extension, <code>matfile</code> appends <code>.mat</code>.</p> <p>If the file does not exist, <code>matfile</code> creates a Version 7.3 MAT-file on the first assignment to a variable.</p> <p><code>matfile</code> only supports partial loading and saving for MAT-files in Version 7.3 format (described in <i>MAT-File Versions</i>). 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.</p> <p><b>'Writable'</b></p> <p>Parameter to use with the <code>isWritable</code> argument.</p> <p><b>isWritable</b></p> <p>Logical value that specifies whether to allow saving to the file. Possible values:</p> |

- `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;
```

---

# matfile

---

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**

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:

|                           |                                                                                                       |
|---------------------------|-------------------------------------------------------------------------------------------------------|
| <code>true</code><br>(1)  | Enable saving. If the file is read only, change the system permissions with <code>fileattrib</code> . |
| <code>false</code><br>(0) | Disable saving with <code>matfile</code> . 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:

|                           |                                                                                                       |
|---------------------------|-------------------------------------------------------------------------------------------------------|
| <code>true</code><br>(1)  | Enable saving. If the file is read only, change the system permissions with <code>fileattrib</code> . |
| <code>false</code><br>(0) | Disable saving with <code>matfile</code> . MATLAB does not change the system permissions.             |

**Default:** `true` for new files, `false` for existing files

## **Methods**

|                   |                                                  |
|-------------------|--------------------------------------------------|
| <code>size</code> | Array dimensions                                 |
| <code>who</code>  | Names of variables in MAT-file                   |
| <code>whos</code> | 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 Programming Fundamentals 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

# matlabroot

---

**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”

# matlab (UNIX)

---

**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 -nosplash
matlab -nouserjavapath
matlab -r matlab_command
matlab -singleCompThread
```

---

**Note** You can enter more than one of these options in the same `matlab` command. If you use `-Ddebugger` to start MATLAB in debug mode, the first option in the command must be `-Ddebugger`.

---

**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

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                                |
|--------------------|--------------------------------------------|
| <code>-help</code> | Display <code>matlab</code> command usage. |
| <code>-h</code>    | The same as <code>-help</code> .           |

`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                                                                                                                                                             |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-n</code> | 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. |
| <code>-e</code> | Display <i>all</i> environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The   |

## Values for envDispOption (Continued)

| Option | Description                                                                                                                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | 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                                                                                                                                                                                                                                             |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-arch</code>          | Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term <code>arch</code> with a string representing a recognized system architecture.                                                 |
| <code>v=variant</code>      | Execute the version of MATLAB found in the folder <code>bin/\$ARCH/variant</code> instead of <code>bin/\$ARCH</code> . Replace the term <code>variant</code> with a string representing a MATLAB version.                                               |
| <code>v=arch/variant</code> | Execute the version of MATLAB found in the folder <code>bin/arch/variant</code> instead of <code>bin/\$ARCH</code> . Replace the terms <code>arch</code> and <code>variant</code> 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                                                                                                                                     |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-display</b> xDisp | Send X commands to X Window Server display xDisp. This supersedes the value of the DISPLAY environment variable.                                |
| <b>-nodisplay</b>     | Start the Oracle JVM software, but do not start the MATLAB desktop. Do not display any X commands, and ignore the DISPLAY environment variable, |

matlab modeOption starts MATLAB without its usual desktop component. Enter only one of the options shown below.

## Values for modeOption

| Option            | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-desktop</b>   | 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.                                                                                                                                                                                                                                                                                                                                                    |
| <b>-nodesktop</b> | Start MATLAB without bringing up the MATLAB desktop. The JVM software <i>is</i> started. Use the current window in the operating system to enter commands. Use this option to run without an X-window, for example, in VT100 mode, or 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 |

## Values for modeOption (Continued)

| Option        | Description                                                                                                                                                                                                                                                                                                   |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | preferences to open the Preferences dialog box and doc to open the Help browser. Do not use nodedesktop to provide a Command Window-only interface; instead, select <b>Desktop &gt; Desktop Layout &gt; Command Window Only</b> .                                                                             |
| <b>-nojvm</b> | 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 (e.g., `dbx`, `gdb`, `xdb`, `cvd`). A full path can be specified for debugger.

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 `-Ddebugger` argument, they will be handled as if they were part of the options after the `-Ddebugger` 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.

---

**Note** 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 -nosplash` starts MATLAB but does not display the splash screen during startup.

`matlab -nouserjavapath` disables `javaclasspath.txt` and `javalibrarypath.txt` files.

# matlab (UNIX)

---

`matlab -r matlab_command` starts MATLAB and executes the specified MATLAB command.

`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 | Definition and Standard Assignment Behavior                                                                                                                                                                                                                                                                               |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARCH     | The machine architecture.<br><br>The value ARCH passed with the <code>-arch</code> or <code>-arch/ext</code> argument to the script is tried first, then the value of the environment variable <code>MATLAB_ARCH</code> is tried next, and finally it is computed. The first one that gives a valid architecture is used. |
| DISPLAY  | The hostname of the X Window display MATLAB uses for output.                                                                                                                                                                                                                                                              |

| Variable        | Definition and Standard Assignment Behavior                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                 | <p>The value of <code>Xdisplay</code> passed with the <code>-display</code> argument to the script is used; otherwise, the value in the environment is used. <code>DISPLAY</code> is ignored by MATLAB if the <code>-nodisplay</code> argument is passed.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| LD_LIBRARY_PATH | <p>Final Load library path. The name <code>LD_LIBRARY_PATH</code> is platform dependent.</p> <p>The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in <code>.matlab7rc.sh</code> as <code>LDPATH_PREFIX</code>. 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 <code>LD_LIBRARY_PATH</code> from the environment possibly augmented in <code>.matlab7rc.sh</code>. The final sublist is defined in <code>.matlab7rc.sh</code> as <code>LDPATH_SUFFIX</code>.</p>                                                               |
| LM_LICENSE_FILE | <p>The FLEX lm license variable.</p> <p>The license file value passed with the <code>-c</code> argument to the script is used; otherwise it is the value set in <code>.matlab7rc.sh</code>. In general, the final value is a colon-separated list of license files and/or <code>port@host</code> entries. The shipping <code>.matlab7rc.sh</code> file starts out the value by prepending <code>LM_LICENSE_FILE</code> in the environment to a default <code>license.file</code>.</p> <p>Later in the <code>matlab</code> script, if the <code>-c</code> option is not used, the <code>matlabroot/etc</code> folder is searched for the files that start with <code>license.dat.DEMO</code>. These files are assumed to contain demo</p> |

# matlab (UNIX)

---

| Variable     | Definition and Standard Assignment Behavior                                                                                                                                                                                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | licenses and are added automatically to the end of the current list.                                                                                                                                                                                                                                                                               |
| MATLAB       | <p>The MATLAB root folder.</p> <p>The default computed by the script is used unless MATLABdefault is reset in .matlab7rc.sh.</p> <p>Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.</p>                                                                                                                                        |
| MATLAB_DEBUG | <p>Normally set to the name of the debugger.</p> <p>The -Ddebugger argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.</p>                                                                                                                                                                   |
| MATLAB_JAVA  | <p>The path to the root of the Java Runtime Environment.</p> <p>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.</p> |
| MATLABPATH   | <p>The MATLAB search path.</p> <p>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 the MATLAB search path upon startup. See userpath for details.</p>                                                                            |



| Variable | Definition and Standard Assignment Behavior                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SHELL    | <p>The shell to use when the “!” or unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in <code>.matlab7rc.sh</code>.</p> <p>Note that an additional environment variable called <code>MATLAB_SHELL</code> takes precedence over <code>SHELL</code>. MATLAB checks internally for <code>MATLAB_SHELL</code> first and, if empty or not defined, then checks <code>SHELL</code>. If <code>SHELL</code> is also empty or not defined, MATLAB uses <code>/bin/sh</code>. The value of <code>MATLAB_SHELL</code> should be an absolute path, i.e. <code>/bin/sh</code>, not simply <code>sh</code>.</p> <p>Currently, the shipping <code>.matlab7rc.sh</code> file does not reset <code>SHELL</code> and also does not reference or set <code>MATLAB_SHELL</code>.</p> |
| TOOLBOX  | <p>Path of the toolbox folder.</p> <p>A nonempty value in the environment is used first. Otherwise, <code>matlabroot/toolbox</code>, computed by the script, is used unless <code>TOOLBOX</code> is reset in <code>.matlab7rc.sh</code>. Currently <code>TOOLBOX</code> is not reset in the shipping <code>.matlab7rc.sh</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

# matlab (UNIX)

---

| Variable    | Definition and Standard Assignment Behavior                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| XAPPLRESDIR | The X application resource folder (Linux systems only).<br><br>A nonempty value in the environment is used first unless XAPPLRESDIR is reset in <code>.matlab7rc.sh</code> . |
| XKEYSYMDB   | The X keysym database file (Linux systems only).<br><br>A nonempty value in the environment is used first unless XKEYSYMDB is reset in <code>.matlab7rc.sh</code> .          |

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

- “Starting the MATLAB Program on Linux Platforms”
- “Starting the MATLAB Program on Macintosh Platforms”
- “Startup Options”

**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 directory
- 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:

# matlab (Windows)

---

- 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                   |
|--------------|-------------------------------|
| <b>-help</b> | Display matlab command usage. |
| <b>-h</b>    | The same as <b>-help</b> .    |
| <b>-?</b>    | The same as <b>-help</b> .    |

`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. Any required file must be on the MATLAB search path or in the startup directory.

`matlab -regserver` registers MATLAB as a Component Object Model (COM) server.

`matlab -sd "startdir"` specifies the startup directory for MATLAB (the current directory 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,

# matlab (Windows)

---

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                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-shield minimum</b> | <p>This is the default setting. It protects the range 0x50000000 to 0x70000000 during MATLAB startup until just before startup processes <code>matlabrc</code>. It ensures there is at least approximately 500 MB of contiguous memory up to this point.</p> <p>Start with this, the default value. To use the default, do not specify a shield option upon startup.</p> <p>If MATLAB fails to start successfully using the default option, <b>-shield minimum</b>, instead use <b>-shield none</b>.</p> <p>If MATLAB starts successfully with the default value for <i>shieldOption</i> and you want to try to ensure an even larger contiguous block after startup, try using the <b>-shield medium</b> option.</p> |
| <b>-shield medium</b>  | <p>This protects the same range as for minimum, 0x50000000 to 0x70000000, but protects the range until just after startup processes <code>matlabrc</code>. It ensures there is at least approximately 500 MB of contiguous memory up to this point.</p> <p>If MATLAB fails to start successfully with the <b>-shield medium</b> option, instead use the default option (<b>-shield minimum</b>).</p> <p>If MATLAB starts successfully with the <b>-shield medium</b> option and you want to try</p>                                                                                                                                                                                                                   |

| Option                 | Description                                                                                                                                                                                                                                                                      |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        | to ensure an even larger contiguous block after startup, try using the <b>-shield maximum</b> option.                                                                                                                                                                            |
| <b>-shield maximum</b> | This protects the maximum possible range, which can be up to approximately 1.5 GB, until just after startup processes <code>matlabrc</code> .<br><br>If MATLAB fails to start successfully with the <b>-shield maximum</b> option, instead use the <b>-shield medium</b> option. |
| <b>-shield none</b>    | This completely disables address shielding. Use this if MATLAB fails to start successfully with the default ( <b>-shield minimum</b> ) 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.

# matlab (Windows)

---

| Variable Name   | Description                                                                                                                                                                                                                                           |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LM_LICENSE_FILE | This variable specifies the License File to use. If you use the <b>-c</b> 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

- “Starting the MATLAB Program 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

---

```
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`

**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
```

# MaximizeCommandWindow

---

```
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”

## 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.

---

# mean

---

**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](#)

**Purpose** Median value of array

**Syntax** `M = median(A)`  
`M = median(A,dim)`

**Description** `M = median(A)` returns the median values of the elements along different dimensions of an array. `A` should be of type `single` or `double`.  
If `A` is a vector, `median(A)` returns the median value of `A`.  
If `A` is a matrix, `median(A)` treats the columns of `A` as vectors, returning a row vector of median values.  
If `A` is a multidimensional array, `median(A)` treats the values along the first nonsingleton dimension as vectors, returning an array of median values.  
`M = median(A,dim)` returns the median values for elements along the dimension of `A` specified by scalar `dim`.

**Examples**

```
A = [1 2 4 4; 3 4 6 6; 5 6 8 8; 5 6 8 8];
median(A)

ans =

     4     5     7     7

median(A,2)

ans =

     3
     5
     7
     7
```

**See Also** `corrcoef` | `cov` | `max` | `mean` | `min` | `mode` | `std` | `var`

# memmapfile

---

**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.

# memmapfile

---

| 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
```

# memmapfile

---

```
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)`

# memory

---

**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 2-2799
- “Memory Available for All Arrays” on page 2-2800
- “Memory Used By MATLAB” on page 2-2801
- “Total Physical Memory (RAM)” on page 2-2801

`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 2-2799
- “Memory Available for All Arrays” on page 2-2800
- “Memory Used By MATLAB” on page 2-2801

`[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 2-2802
- “System Memory” on page 2-2802
- “Physical Memory” on page 2-2803

## 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                                             |
|----------------------------|---------------------------------------------------------|
| <code>memory</code>        | String labelled Maximum possible array:                 |
| <code>user = memory</code> | Structure field <code>user.MaxPossibleArrayBytes</code> |

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                                             |
|----------------------------|---------------------------------------------------------|
| <code>memory</code>        | String labelled Memory available for all arrays:        |
| <code>user = memory</code> | Structure field <code>user.MemAvailableAllArrays</code> |

## 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                                     |
|----------------------------|-------------------------------------------------|
| <code>memory</code>        | String labelled Memory used by MATLAB:          |
| <code>user = memory</code> | Structure field <code>user.MemUsedMATLAB</code> |

### 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                            |
|---------------------|----------------------------------------|
| <code>memory</code> | 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] =<br>memory | Available<br>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] =<br>memory | Available<br>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] =<br>memory | Available<br>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:

$$\text{Mem Usage} < \text{MemUsedMATLAB} < \text{Mem Usage} + \text{VM Size (in bytes)}$$

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 MemUsedMATLAB. 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 than 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;
```

# 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”

# menu

---

**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`.

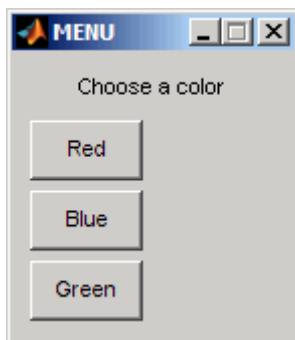
## **Examples** **Example 1**

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

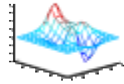
[guide](#) | [input](#) | [uicontrol](#) | [uimenu](#)

# 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.

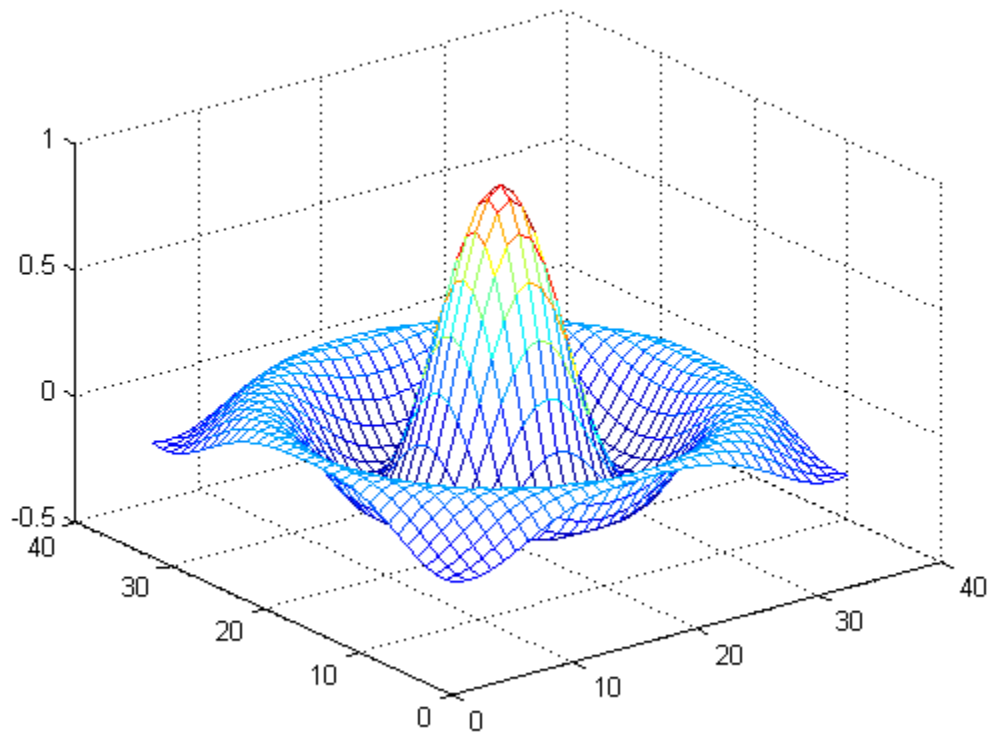
**Examples**

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);
```

# mesh

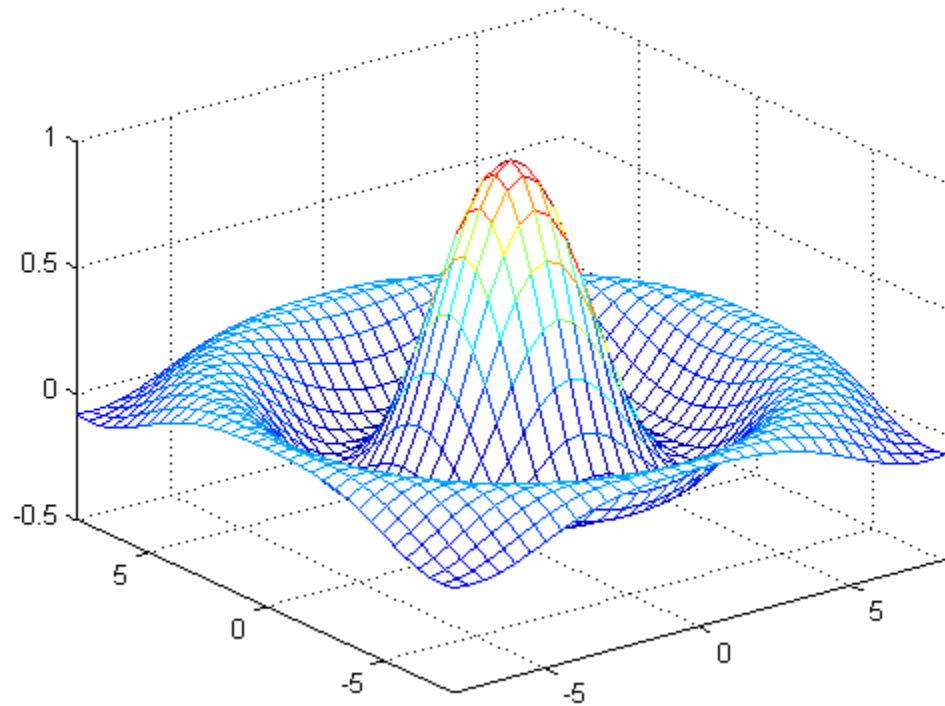
---



---

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]);
```

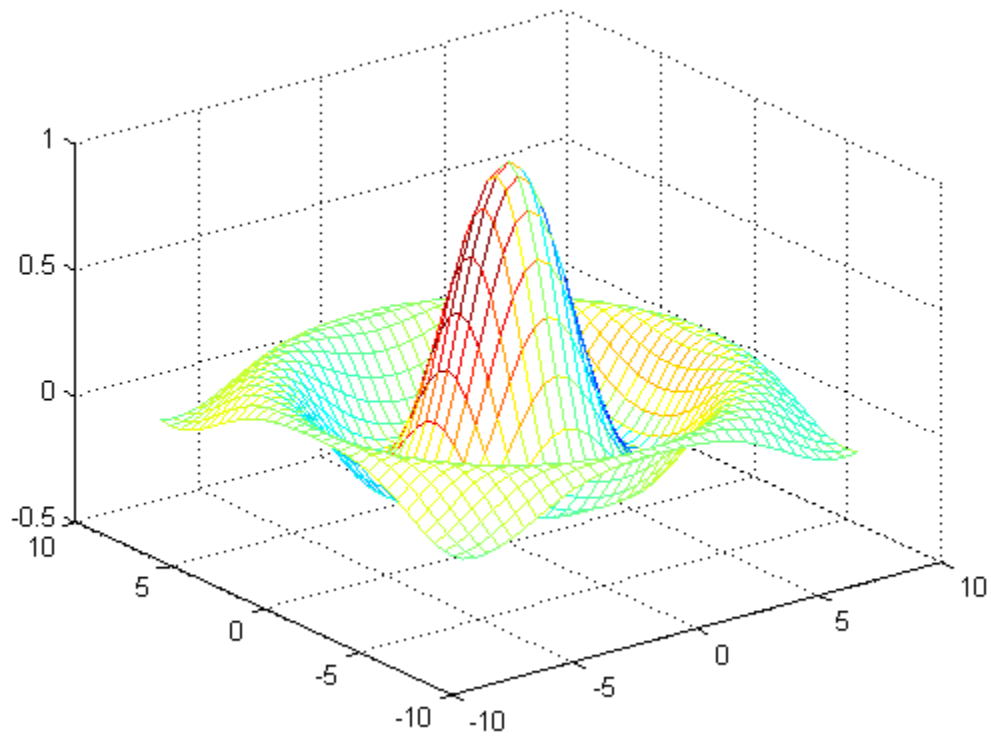


Specifying color matrix for the data in the mesh function

```
C = gradient(Z);  
figure;  
mesh(X,Y,Z,C);
```

# mesh

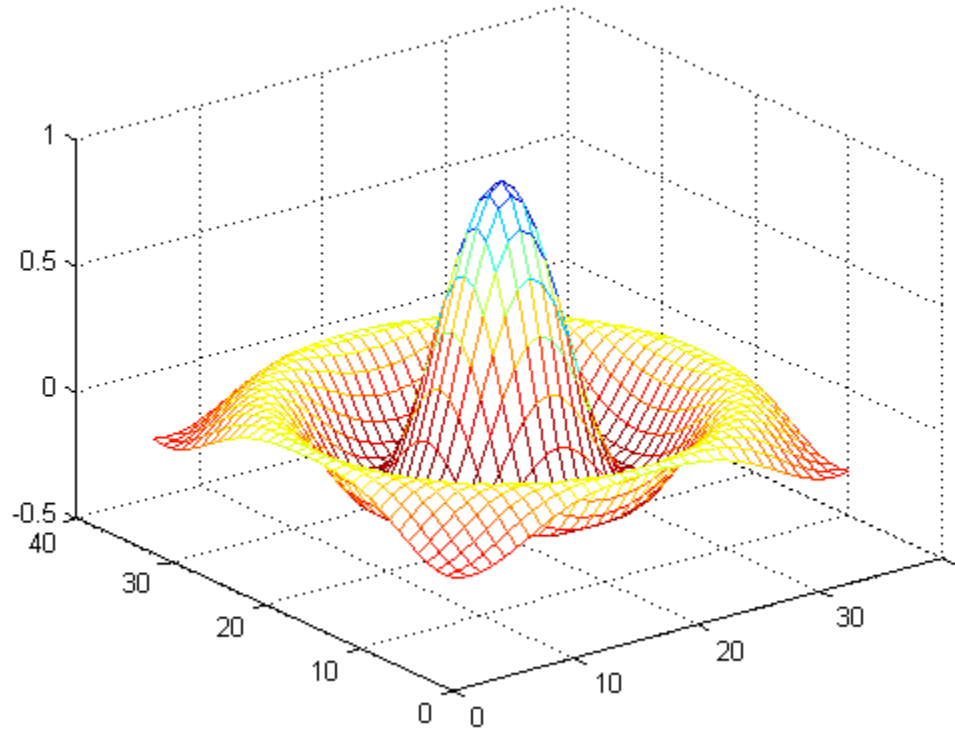
---



---

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` | `TriScatteredInterp` | `meshgrid` | `surface` | `surf` | `waterfall` | `axis` | `colormap` | `hold` | `shading` | `view`

**How To**

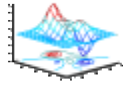
- “Representing Data as a Surface”

# 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

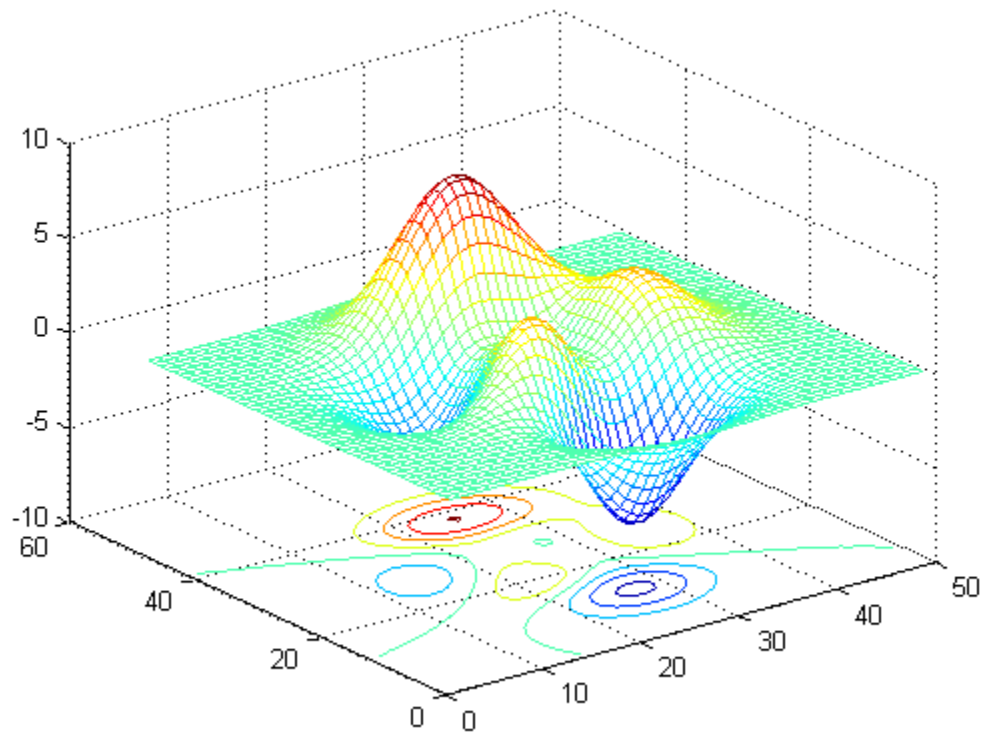


color (to simulate a wireframe with hidden-surface elimination), or none when drawing a standard see-through wireframe. The current colormap determines the edge color. The hidden command controls the simulation of hidden-surface elimination in the mesh, and the shading command controls the shading model.

## Examples

Produce a combination mesh and contour plot of the peaks surface with meshc:

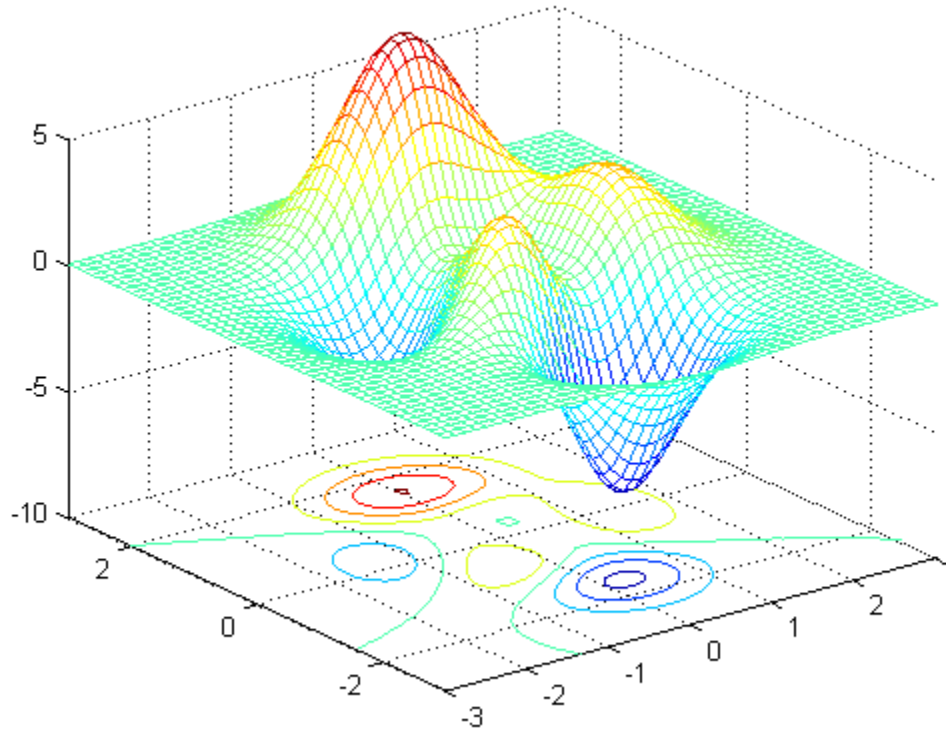
```
figure
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(Z);
```



---

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. 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

# meshc

---

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.

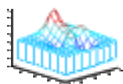
`meshc` calls `mesh`, turns `hold` on, and then calls `contour` and positions the contour on the  $x$ - $y$  plane. For additional control over the appearance of the contours, issue these commands directly. You can combine other types of graphs in this manner, for example `surf` and `pcolor` plots.

`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` | `surf1` | `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.

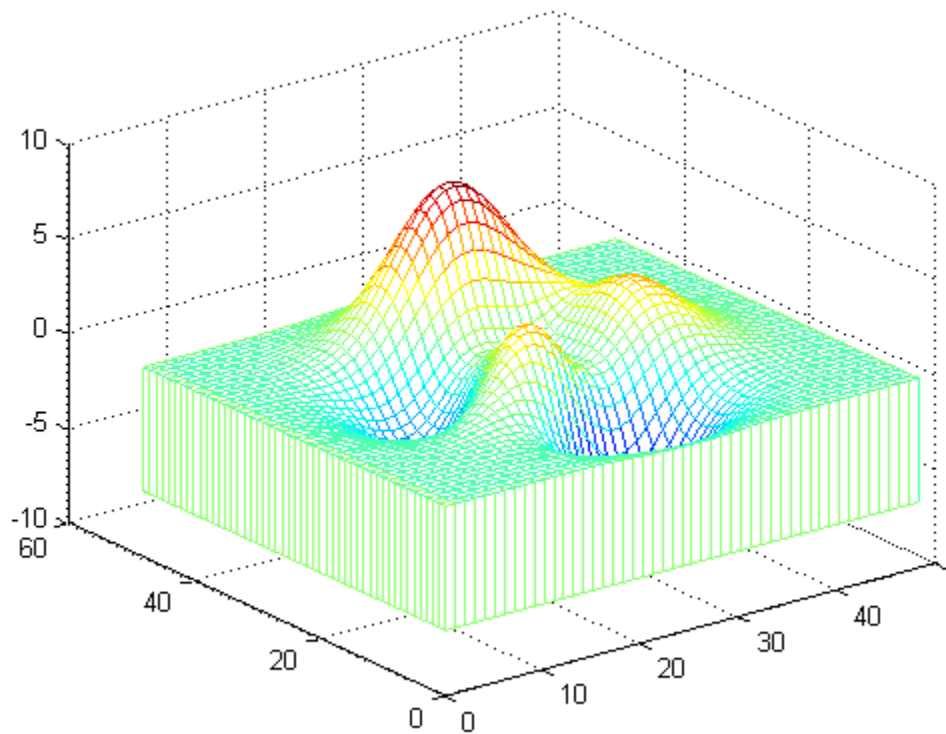
A mesh is drawn as a `Surfaceplot` graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background

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

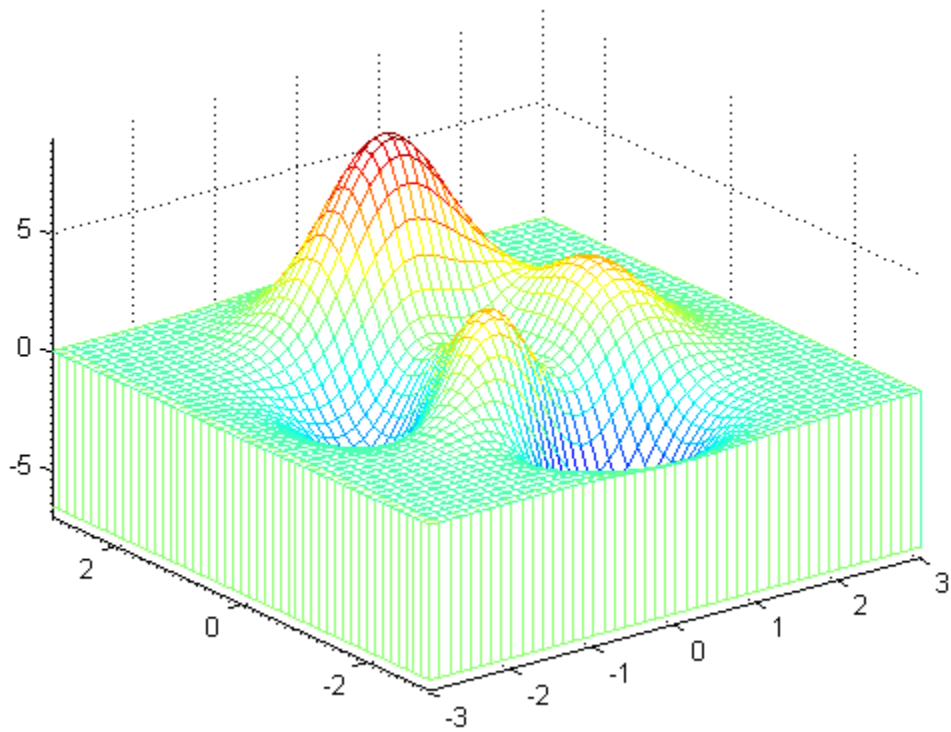
Generate a curtain plot for the peaks function using meshz

```
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]);
```

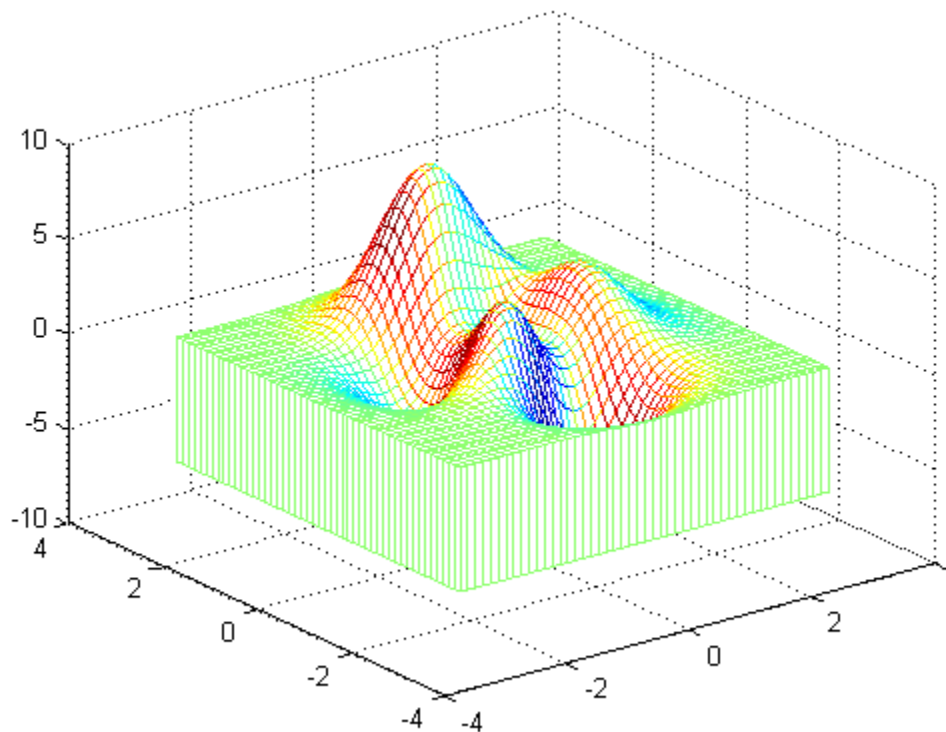


---

Specifying a color matrix

```
C = gradient(Z);  
figure;  
meshz(X,Y,Z,C);
```



**See Also**

`mesh` | `meshc` | `contour` | `hidden` | `meshgrid` | `surface` | `surf` |  
`surfc` | `surf1` | `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 =
```

```

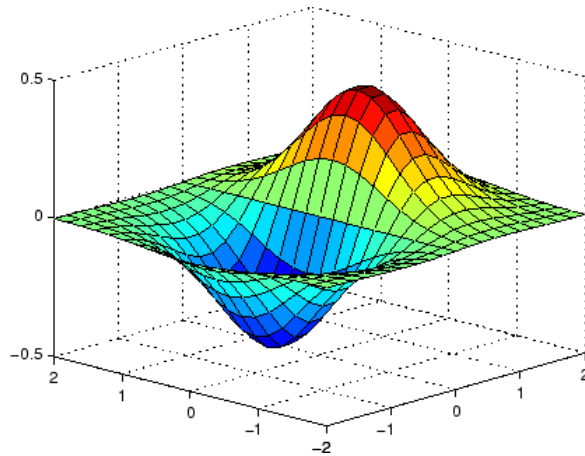
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

Use `meshgrid` to create a 3-D surface plot of a function:

# meshgrid

---

```
[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');
```

# meta.class

---

## Properties

| Property                                                                     | Purpose                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Abstract attribute, default = false                                          | If true, this class is an abstract class (cannot be instantiated).<br><br>See “Defining Abstract Classes” for more information.                                                                                                                                                                                                                          |
| AllowedSubclasses attribute, write only from <code>classdef</code> statement | List classes that can subclass this class. Specify subclasses as <code>meta.class</code> objects in the form: <ul style="list-style-type: none"><li>• A single <code>meta.class</code> object</li><li>• A cell array of <code>meta.class</code> objects</li></ul> Specify <code>meta.class</code> objects using the <code>?ClassName</code> 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.<br><br>See “Tips for Saving and Loading”                                                                                               |
| ContainingPackage read only                                                  | A <code>meta.package</code> object describing the package within which this class is contained, or an empty object if this class is not in a package.<br><br>See “Packages Create Namespaces”.                                                                                                                                                           |
| Description read only                                                        | Currently not used                                                                                                                                                                                                                                                                                                                                       |
| DetailedDescription read only                                                | Currently not used                                                                                                                                                                                                                                                                                                                                       |

| <b>Property</b>                                             | <b>Purpose</b>                                                                                                                                                                                |
|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EventList read only                                         | An array of <code>meta.event</code> objects describing each event defined by this class, including all inherited events.<br><br>See “Events”.                                                 |
| Events read only Use EventList instead                      | A cell array of <code>meta.event</code> objects describing each event defined by this class, including all inherited events.                                                                  |
| EnumerationMemberList                                       | An array of <code>meta.EnumeratedValue</code> objects describing the member names defined by an enumeration class.<br><br>See “Enumerations” for more information on enumeration classes.     |
| EnumeratedValues read only Use EnumeratedMemberList instead | A cell array of <code>meta.EnumeratedValue</code> objects describing the member names defined by an enumeration class.<br><br>See “Enumerations” for more information on enumeration classes. |
| Hidden attribute, default = false                           | If set to <code>true</code> , the class does not appear in the output of MATLAB commands or tools that display class names.                                                                   |
| InferiorClasses attribute, default = {}                     | A cell array of <code>meta.class</code> objects defining the precedence of classes represented by the list as inferior to this class.<br><br>See “Class Precedence”                           |
| MethodList read only                                        | An array of <code>meta.method</code> objects describing each method defined by this class, including all inherited public and protected methods.<br><br>See “How to Use Methods”.             |

# meta.class

| Property                                                                | Purpose                                                                                                                                                                          |
|-------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Methods read only<br>Use <code>MethodList</code><br>instead             | A cell array of <code>meta.method</code> 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 <code>meta.class</code> object (char array)                                                                                               |
| PropertyList read only                                                  | An array of <code>meta.property</code> objects describing each property defined by this class, including all inherited public and protected properties.<br>See “Properties”.     |
| Properties read only<br>Use <code>PropertyList</code><br>instead        | A cell array of <code>meta.property</code> objects describing each property defined by this class, including all inherited public and protected properties.<br>See “Properties”. |
| Sealed attribute,<br>default = false                                    | If true, the class cannot be subclassed.                                                                                                                                         |
| SuperClassList<br>read only                                             | An array of <code>meta.class</code> objects describing each direct superclass from which this class is derived.<br>See “Creating Subclasses — Syntax and Techniques”.            |
| SuperClasses<br>read only Use<br><code>SuperClassList</code><br>instead | A cell array of <code>meta.class</code> 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(cname)
    % Do error checking
    mcls = meta.class.fromName(cname);
    ...
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    |

# meta.DynamicProperty

---

| 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 | <p>If true, the property has no implementation, but a concrete subclass must redefine this property without <code>Abstract</code> being set to true.</p> <ul style="list-style-type: none"><li>• Abstract properties cannot define set or get access methods. See “Property Access Methods”</li><li>• Abstract properties cannot define initial values. “Assigning a Default Value”</li><li>• All subclasses must specify the same values as the superclass for the property <code>SetAccess</code> and <code>GetAccess</code> attributes.</li><li>• <code>Abstract=true</code> should be used with the class attribute <code>Sealed=false</code> (the default).</li></ul> |

| Property                            | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Access                              | <p><code>public</code> – unrestricted access</p> <p><code>protected</code> – access from class or subclasses</p> <p><code>private</code> – access by class members only (not subclasses)</p> <p>List of classes that have get and set access to this property. Specify classes as <code>meta.class</code> objects in the form:</p> <ul style="list-style-type: none"><li>• A single <code>meta.class</code> object</li><li>• A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access.</li></ul> <p>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>).</p> |
| Constant attribute, default = false | <p>Set to <code>true</code> if you want only one value for this property in all instances of the class.</p> <ul style="list-style-type: none"><li>• Subclasses inherit constant properties, but cannot change them.</li><li>• Constant properties cannot be <code>Dependent</code></li><li>• <code>SetAccess</code> is ignored.</li></ul> <p>See “Properties with Constant Values”</p>                                                                                                                                                                                                                                                                                                                                                                                                              |
| DefaultValue                        | <p>Querying this property returns an error because dynamic properties cannot define default values.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

# meta.DynamicProperty

---

| Property                                 | Purpose                                                                                                                                                                                                                                   |
|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DefiningClass                            | The meta.class object representing the class that defines this property.                                                                                                                                                                  |
| GetAccess attribute, default = public    | public – unrestricted access<br>protected – access from class or subclasses<br>private – access by class members only                                                                                                                     |
| SetAccess attribute, default = public    | public – unrestricted access<br>protected – access from class or subclasses<br>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.<br>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<br>attribute, default<br>= 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 to the 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”

# meta.event

---

## 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 |

| Property     | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Hidden       | If true, the event does not appear in the list of events returned by the events function (or other event listing functions or viewers)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| ListenAccess | <p>Determines where you can create listeners for the event.</p> <ul style="list-style-type: none"><li>• <code>public</code> — unrestricted access</li><li>• <code>protected</code> — access from methods in class or subclasses</li><li>• <code>private</code> — access by class methods only (not from subclasses)</li><li>• List classes that have listen access to this event. Specify classes as <code>meta.class</code> objects in the form:<ul style="list-style-type: none"><li>▪ A single <code>meta.class</code> object</li><li>▪ A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access.</li></ul></li></ul> <p>See “Controlling Access to Class Members”</p> |

# meta.event

---

| Property      | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NotifyAccess  | <p>Determines where code can trigger the event.</p> <ul style="list-style-type: none"><li>• <code>public</code> — any code can trigger event</li><li>• <code>protected</code> — can trigger event from methods in class or subclasses</li><li>• <code>private</code> — can trigger event by class methods only (not from subclasses)</li><li>• List classes that have notify access to this event. Specify classes as <code>meta.class</code> objects in the form:<ul style="list-style-type: none"><li>▪ A single <code>meta.class</code> object</li><li>▪ A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access.</li></ul></li></ul> <p>See “Controlling Access to Class Members”</p> |
| DefiningClass | <p>The <code>meta.class</code> object representing the class that defines this event.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## See Also

`meta.class` | `meta.property` | `meta.method` | `metaclass`

## How To

- “Events”
- “Getting Information About Classes and Objects”

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Superclasses</b>   | Heterogeneous                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Purpose</b>        | Superclass for MATLAB object metadata                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b>    | <p>The <code>meta.MetaData</code> 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.</p> <p>The <code>meta.MetaData</code> class forms the root of the metadata class hierarchy, which enables the formation of arrays of metadata objects belonging to different specific classes.</p> <p>MATLAB uses instances of the <code>meta.MetaData</code> class as the default object to fill in missing array elements.</p> <p><code>findobj</code> and <code>findprop</code>, 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.</p> <p>See the <code>matlab.mixin.Heterogeneous</code> class for more information on heterogeneous hierarchies.</p> |
| <b>Construction</b>   | You cannot create an instance of the <code>meta.MetaData</code> class directly. MATLAB constructs instances of this class as required.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Copy Semantics</b> | Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Examples</b>       | <p>This example shows how the <code>meta.MetaData</code> class facilitates working with metaclasses.</p> <p>Create a <code>meta.class</code> instance representing the MATLAB <code>timeseries</code> class:</p> <pre>&gt;&gt; mc = ?timeseries;</pre> <p>MATLAB uses <code>meta.MetaData</code> objects to fill empty array elements:</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

# meta.MetaData

---

```
>> 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 | <p>If true, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method.</p> <ul style="list-style-type: none"><li>Subclasses are not required to define the same number of input and output arguments.</li></ul> |



| Property                                        | Purpose                                                                                                                                                                                                                                                      |
|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                 | <ul style="list-style-type: none"> <li>• The method can have comments after the function line</li> <li>• Does not contain <code>function</code> or <code>end</code> keywords, only the function syntax (e.g., <code>[a,b] = myMethod(x,y)</code>)</li> </ul> |
| Access attribute, default = <code>public</code> | Determines what code can call this method. <ul style="list-style-type: none"> <li>• <code>public</code> — unrestricted access</li> <li>• <code>protected</code> — access from methods in class</li> </ul>                                                    |
| DefiningClass                                   | The <code>meta.class</code> object representing the class that defines this method.                                                                                                                                                                          |
| Description read only                           | Currently not used                                                                                                                                                                                                                                           |
| DetailedDescription read only                   | Currently not used                                                                                                                                                                                                                                           |
| Hidden attribute, default = <code>false</code>  | When <code>false</code> , the method name shows in the list of methods displayed using the <code>methods</code> or <code>methodsview</code> commands. If set to <code>true</code> , the method name is not included in these listings.                       |
| Name read only                                  | Name of the method.                                                                                                                                                                                                                                          |

# meta.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:<br><br><code>classname.methodname()</code><br><br>Or with an instance of the class, like any method:<br><br><code>o.methodname()</code><br><br>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. |

# meta.package

---

## Methods

| Method         | Purpose                                                                                                      |
|----------------|--------------------------------------------------------------------------------------------------------------|
| fromName       | Static method returns a <code>meta.package</code> object for a specified package name.                       |
| getAllPackages | Static method returns a cell array of <code>meta.package</code> 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)

# meta.abstractDetails

---

## 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
```

# meta.abstractDetails

---

```
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.forName`

## Concepts

- “Defining Abstract Classes”
- “Getting Information About Classes and Objects”



**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 PropertyList 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    |

# meta.property

| Property                            | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AbortSet attribute, default = false | <p>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.</p> <p>See “Listen for Changes to Property Values”</p>                                                                                                                                                                                                                                                                                                                        |
| Abstract attribute, default = false | <p>If true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true.</p> <ul style="list-style-type: none"><li>• Abstract properties cannot define set or get access methods. See “Property Access Methods”</li><li>• Abstract properties cannot define initial values. “Assigning a Default Value”</li><li>• All subclasses must specify the same values as the superclass for the property SetAccess and GetAccess attributes.</li><li>• Abstract=true should be used with the class attribute Sealed=false (the default).</li></ul> |
| Access                              | <p>public – unrestricted access</p> <p>protected – access from class or subclasses</p> <p>private – access by class members only (not subclasses)</p> <p>List of classes that have get and set access to this property. Specify classes as meta.class objects in the form:</p> <ul style="list-style-type: none"><li>• A single meta.class object</li></ul>                                                                                                                                                                                                                                               |

| Property                                   | Purpose                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                            | <ul style="list-style-type: none"> <li>• A cell array of <code>meta.class</code> objects. An empty cell array, {}, is the same as <code>private</code> access.</li> </ul> <p>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>).</p> |
| <p>Constant attribute, default = false</p> | <p>Set to true if you want only one value for this property in all instances of the class.</p> <ul style="list-style-type: none"> <li>• Subclasses inherit constant properties, but cannot change them.</li> <li>• Constant properties cannot be <code>Dependent</code></li> <li>• <code>SetAccess</code> is ignored.</li> </ul> <p>See “Properties with Constant Values”</p>                       |
| <p>DefaultValue</p>                        | <p>Property default value (if specified in class definition). See also <code>HasDefault</code> property. Abstract, dependent and dynamic properties cannot specify default values.</p>                                                                                                                                                                                                              |
| <p>DefiningClass</p>                       | <p>The <code>meta.class</code> object representing the class that defines this property.</p>                                                                                                                                                                                                                                                                                                        |

# meta.property

---

| Property                              | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GetAccess attribute, default = public | <p>public – unrestricted access</p> <p>protected – access from class or subclasses</p> <p>private – access by class members only</p> <p>List classes that have get access to this property. Specify classes as <code>meta.class</code> objects in the form:</p> <ul style="list-style-type: none"><li>• A single <code>meta.class</code> object</li><li>• A cell array of <code>meta.class</code> objects. An empty cell array, {}, is the same as private access.</li></ul> <p>See “Controlling Access to Class Members”</p> |
| Dependent attribute, default = false  | <p>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.</p> <p>See “Property Get Methods”</p>                                                                                                                                                                                                                                                                          |
| Transient attribute, default = false  | <p>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.</p>                                                                                                                                                                                                                                                                                                                                                                      |
| GetMethod read only                   | <p>Function handle of the get method associated with this property. Empty if there is no get method specified. See “Get Method Syntax”</p>                                                                                                                                                                                                                                                                                                                                                                                    |

| 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    | <p>public – unrestricted access</p> <p>protected – access from class or subclasses</p> <p>private – access by class members only</p> <p>immutable — property can be set only in the constructor.</p> <p>See “Mutable and Immutable Properties”</p> <p>List classes that have set access to this property. Specify classes as meta.class objects in the form:</p> <ul style="list-style-type: none"> <li>• A single meta.class object</li> <li>• A cell array of meta.class objects. An empty cell array, {}, is the same as private access.</li> </ul> <p>See “Controlling Access to Class Members”</p> |

# meta.property

---

| 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 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 2-2870 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 2-2875.

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 2-2874. 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 2-2874.

For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.

## MEX Script Switches

| Switch                 | Function                                                                                                                                                                                                                                                                             |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>@rsp_file</code> | (Windows systems only) Include the contents of the text file <code>rsp_file</code> as command-line arguments to <code>mex</code> .                                                                                                                                                   |
| <code>-arch</code>     | Build an output file for architecture <code>arch</code> . To determine the value for <code>arch</code> , type <code>computer('arch')</code> at the MATLAB Command Prompt on the target machine. Valid values for <code>arch</code> depend on the architecture of the build platform. |
| <code>-c</code>        | Compile only. Creates an object file, but not a binary MEX-file.                                                                                                                                                                                                                     |

(Continued)

| Switch                            | Function                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-compatibleArrayDims</code> | <p>Build a binary MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to <math>2^{31}-1</math> elements. Default option. (See the <code>-largeArrayDims</code> option.)</p> <p>In verbose mode, if you do not specify either the <code>-largeArrayDims</code> or the <code>-compatibleArrayDims</code> switches, MATLAB displays a message showing the default switch.</p> |
| <code>-cxx</code>                 | <p>(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.</p>                                                                                                                                |
| <code>-Dname</code>               | <p>Define a symbol name to the C preprocessor. Equivalent to a <code>#define name</code> directive in the source.</p> <p>Do not add a space after this switch.</p>                                                                                                                                                                                                                                   |
| <code>-Dname=value</code>         | <p>Define a symbol name and value to the C preprocessor. Equivalent to a <code>#define name value</code> directive in the source.</p> <p>Do not add a space after this switch.</p>                                                                                                                                                                                                                   |
| <code>-f optionsfile</code>       | <p>Specify location and name of options file to use. Overrides the mex default-options-file search mechanism.</p>                                                                                                                                                                                                                                                                                    |

**(Continued)**

| <b>Switch</b>      | <b>Function</b>                                                                                                                                                                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -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 <i>pathname</i> | Add <i>pathname</i> to the list of folders to search for #include files.<br><br>Do not add a space after this switch.                                                                                                                                      |
| -inline            | This option has been removed.                                                                                                                                                                                                                              |
| -lname             | Link with object library. On Windows systems, <i>name</i> expands to <i>name.lib</i> or <i>libname.lib</i> and on UNIX systems, to <i>libname.so</i> or <i>libname.dylib</i> .<br><br>Do not add a space after this switch.                                |
| -L <i>folder</i>   | Add <i>folder</i> 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”.<br><br>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.)<br><br>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 <i>dirname</i>    | Place all output files in folder <i>dirname</i> .                                                                                                                                                                                                                                                                                             |
| -output <i>resultname</i> | Create binary MEX-file named <i>resultname</i> . 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)**

| <b>Switch</b>     | <b>Function</b>                                                                                                                                                                                                  |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>-Uname</i>     | Remove any initial definition of the C preprocessor symbol <i>name</i> . (Inverse of the <i>-D</i> option.)<br><br>Do not add a space after this switch.                                                         |
| <i>-v</i>         | 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. |
| <i>name=value</i> | Override an options file variable for variable <i>name</i> . For examples, see <a href="#">Override Option Details</a> on <a href="#">mex</a> reference page.                                                    |

## Tips

### 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" yprime.c
```

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**

The following command compiles 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 2-2874.

## **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 2-2878.

`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 compiler 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.

# mex.getCompilerConfigurations

---

## 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):

# mex.getCompilerConfigurations

---

```
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: ''
```

# mex.getCompilerConfigurations

---

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<sub>1</sub>*, *v<sub>2</sub>*, ... 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<sub>1</sub>*, *v<sub>2</sub>*, ..., *v<sub>N</sub>* provide values that correspond to and replace the conversion specifiers.

For example, if *msgString* is “Error on line %d, command %s”, then *v<sub>1</sub>* is the line number at which the error was detected, and *v<sub>2</sub>* 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.

# MException

---

## 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                                                                                                                     |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>identifier</code> | Identifies the error.                                                                                                           |
| <code>message</code>    | Formatted error message that is displayed.                                                                                      |
| <code>stack</code>      | Structure containing stack trace information such as file name, function name, and line number where the MException was thrown. |
| <code>cause</code>      | 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                                                                                            |
|------------------------|--------------------------------------------------------------------------------------------------------|
| <code>addCause</code>  | Appends an MException to the <code>cause</code> field of another MException.                           |
| <code>eq</code>        | Tests scalar MException objects for equality.                                                          |
| <code>getReport</code> | Returns a formatted message string that uses the same format as errors thrown by internal MATLAB code. |
| <code>isequal</code>   | Tests scalar MException objects for equality.                                                          |
| <code>last</code>      | Returns an MException object for the most recently thrown exception.                                   |
| <code>ne</code>        | 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 <code>stack</code> field of the <code>MException</code> . |

## Tips

Valid escape sequences for the `msgString` argument are `\b`, `\f`, `\n`, `\r`, `\t`, and `\x` or `\o` 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'' 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 modified 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
end
```

# MException

---

```
        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”



---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Binary MEX-file name extension                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>      | <pre>ext = mexext extlist = mexext('all')</pre>                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b> | <p><code>ext = mexext</code> returns the file name extension for the current platform.</p> <p><code>extlist = mexext('all')</code> returns a struct with fields <code>arch</code> and <code>ext</code> describing MEX-file name extensions for all platforms.</p>                                                                                                                                                                                        |
| <b>Tips</b>        | For a table of file extensions, see “Using Binary MEX-Files”.                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Examples</b>    | <p>Find the MEX-file extension for the system you are currently working on:</p> <pre>ext = mexext  ext =     mexw32</pre> <p>Find the MEX-file extension for an Apple Macintosh system:</p> <pre>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</pre> |
| <b>See Also</b>    | <code>mex</code>                                                                                                                                                                                                                                                                                                                                                                                                                                         |

# 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

|                        |                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Download files from FTP server                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>          | <code>mget(ftpobj, contents)</code><br><code>mget(ftpobj, contents, target)</code>                                                                                                                                                                                                                                                                                                                      |
| <b>Description</b>     | <p><code>mget(ftpobj, contents)</code> retrieves the file or folder specified by <code>contents</code> from an FTP server into the MATLAB current folder.</p> <p><code>mget(ftpobj, contents, target)</code> retrieves the file or folder into the local folder specified by <code>target</code>, which includes an absolute or relative path.</p>                                                      |
| <b>Input Arguments</b> | <p><b>ftpobj</b><br/>FTP object created by <code>ftp</code>.</p> <p><b>contents</b><br/>String enclosed in single quotation marks that specifies either a file name or a folder name. Can include a wildcard character (*).</p> <p><b>target</b><br/>String enclosed in single quotation marks that specifies the absolute or relative path of the local folder to contain the downloaded contents.</p> |
| <b>Examples</b>        | <p>Connect to an FTP server and retrieve the file README into the current MATLAB folder:</p> <pre>mw = ftp('ftp.mathworks.com');<br/>mget(mw, 'README');<br/>close(mw);</pre>                                                                                                                                                                                                                           |
| <b>See Also</b>        | <code>cd</code>   <code>ftp</code>   <code>mput</code>                                                                                                                                                                                                                                                                                                                                                  |

# 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 NaNs.

### 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”

# minres

---

## 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  $\|b-A*x\|/\|b\|$  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  $\text{inv}(\text{sqrt}(M))*A*\text{inv}(\text{sqrt}(M))*y =$



$\text{inv}(\text{sqrt}(M)) * b$  for  $y$  and then return  $x = \text{inv}(\text{sqrt}(M)) * y$ . If  $M$  is  $[]$  then `minres` applies no preconditioner.  $M$  can be a function handle `mfun`, such that `mfun(x)` returns  $M \backslash x$ .

`minres(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If  $x_0$  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 <code>tol</code> within <code>maxit</code> iterations.                        |
| 1    | minres iterated <code>maxit</code> 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 <code>minres</code> 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  $\text{norm}(b - A * x) / \text{norm}(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = minres(A,b,...)` also returns the iteration number at which  $x$  was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = minres(A,b,...)` also returns a vector of estimates of the `minres` residual norms at each iteration, including  $\text{norm}(b - A * x_0)$ .

`[x,flag,relres,iter,resvec,resvecg] = 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.

# minres

---

## **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`

**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

$$c_1x^{k-1} + c_2x^{k-2} + \dots + c_{k-1}x + c_k$$

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

$$\left(\frac{x}{2} - 1\right)^2 - 1 = \frac{x^2}{4} - x$$

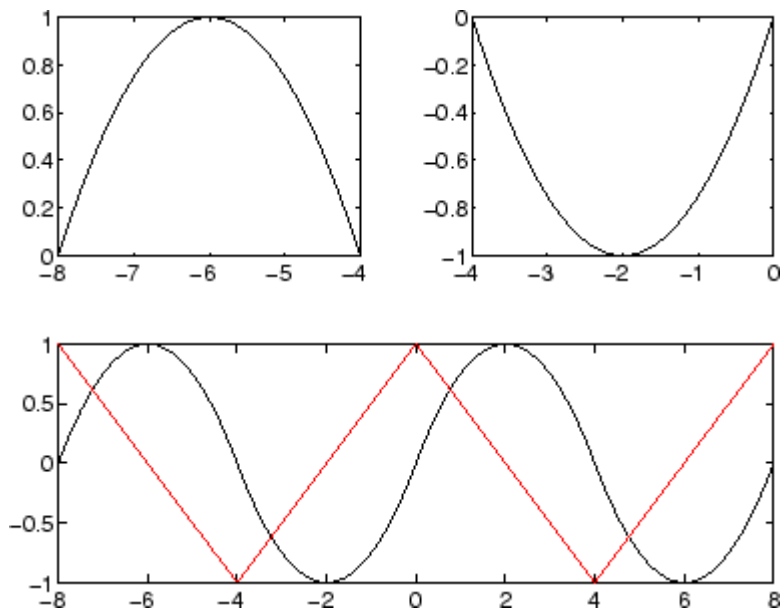
but shifted to the interval [-4,0].

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*1,1).*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  $m \neq n$ , 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  $A^*x = B$ .

$x = \text{mldivide}(A,B)$  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.

# mldivide

---

## 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 inaccurate.  
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

```

# mldivide

---

(5,1)      5.0000

## Tips

- If  $A$  is a square matrix,  $A \setminus B$  is roughly equal to  $\text{inv}(A) * B$ , but MATLAB processes  $A \setminus B$  differently and more robustly.
- If the rank of  $A$  is less than the number of columns in  $A$ , then  $x = A \setminus B$  is not necessarily the minimum norm solution. The more computationally expensive  $x = \text{pinv}(A) * B$  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**  
 $x = B/A$   
 $x = \text{mrdivide}(B,A)$

**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  $m \neq n$ , 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  $x*A = B$ .

$x = \text{mrdivide}(B,A)$  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  $B/A = (A' \setminus B')'$ .
- If  $A$  is a square matrix,  $B/A$  is roughly equal to  $B * \text{inv}(A)$ , 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                                                                                                                                                                                                                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>message</code> | Message describing the suspicious construct that code analysis caught.                                                                                                                                                                                                                                                                          |
| <code>line</code>    | Vector of file line numbers to which the message refers.                                                                                                                                                                                                                                                                                        |
| <code>column</code>  | 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,filepath]=mlint('filename')` additionally returns `filepath`, 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 STRCMP(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 'nohandle' 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 'nohandle' 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 'nohandle' 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 STRCMP(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 2-2926 for more information about this data structure. |
| Video    | Structure containing information about the video data in the file. See “Video Data” on page 2-2926 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),<br>MPEG-1 (.mpg),<br>Windows Media Video (.wmv, .asf, .asx),<br>and any format supported by Microsoft DirectShow.                                                                                                                                                          |
| Macintosh     | AVI (.avi),<br>MPEG-1 (.mpg),<br>MPEG-4 (.mp4, .m4v),<br>Apple QuickTime Movie (.mov),<br>and any format supported by QuickTime as listed on<br><a href="http://support.apple.com/kb/HT3775">http://support.apple.com/kb/HT3775</a> .                                                  |
| Linux         | Any format supported by your installed plug-ins<br>for GStreamer 0.10 or above, as listed on<br><a href="http://gstreamer.freedesktop.org/documentation/plugins.html">http://gstreamer.freedesktop.org/documentation/plugins.html</a> ,<br>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;
```

# mmreader

---

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])
```

# mmreader

---

```
% 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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Modulus after division                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>      | <code>M = mod(X,Y)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b> | <p><code>M = mod(X,Y)</code> if <code>Y ~= 0</code>, returns <code>X - n.*Y</code> where <code>n = floor(X./Y)</code>. If <code>Y</code> is not an integer and the quotient <code>X./Y</code> is within roundoff error of an integer, then <code>n</code> is that integer. The inputs <code>X</code> and <code>Y</code> must be real arrays of the same size, or real scalars.</p> <p>The following are true by convention:</p> <ul style="list-style-type: none"> <li>• <code>mod(X,0)</code> is <code>X</code></li> <li>• <code>mod(X,X)</code> is <code>0</code></li> <li>• <code>mod(X,Y)</code> for <code>X~=Y</code> and <code>Y~=0</code> has the same sign as <code>Y</code>.</li> </ul> |
| <b>Tips</b>        | <p><code>rem(X,Y)</code> for <code>X~=Y</code> and <code>Y~=0</code> has the same sign as <code>X</code>.</p> <p><code>mod(X,Y)</code> and <code>rem(X,Y)</code> are equal if <code>X</code> and <code>Y</code> have the same sign, but differ by <code>Y</code> if <code>X</code> and <code>Y</code> have different signs.</p> <p>The <code>mod</code> function is useful for congruence relationships: <i>x</i> and <i>y</i> are congruent (mod <i>m</i>) if and only if <code>mod(x,m) == mod(y,m)</code>.</p>                                                                                                                                                                                |
| <b>Examples</b>    | <pre>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</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

# mod

---

## See Also

rem

**Purpose** Most frequent values in array

**Syntax**

```
M = mode(X)
M = mode(X, dim)
[M,F] = mode(X, ...)
[M,F,C] = mode(X, ...)
```

**Description** `M = mode(X)` for vector `X` computes the sample mode `M`, (i.e., the most frequently occurring value in `X`). If `X` is a matrix, then `M` is a row vector containing the mode of each column of that matrix. If `X` is an `N`-dimensional array, then `M` is the mode of the elements along the first nonsingleton dimension of that 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.

`M = mode(X, dim)` computes the mode along the dimension `dim` of `X`.

`[M,F] = mode(X, ...)` also returns array `F`, each element of which represents the number of occurrences of the corresponding element of `M`. The `M` and `F` output arrays are of equal size.

`[M,F,C] = mode(X, ...)` also returns cell array `C`, each element of which is a sorted vector of all values that have the same frequency as the corresponding element of `M`. All three output arrays `M`, `F`, and `C` are of equal size.

**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** **Example 1**

Find the mode of the 3-by-4 matrix shown here:

# mode

---

```
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
```

```
mode(X)  
ans =  
     0     0     1     4
```

Find the mode along the second (row) dimension:

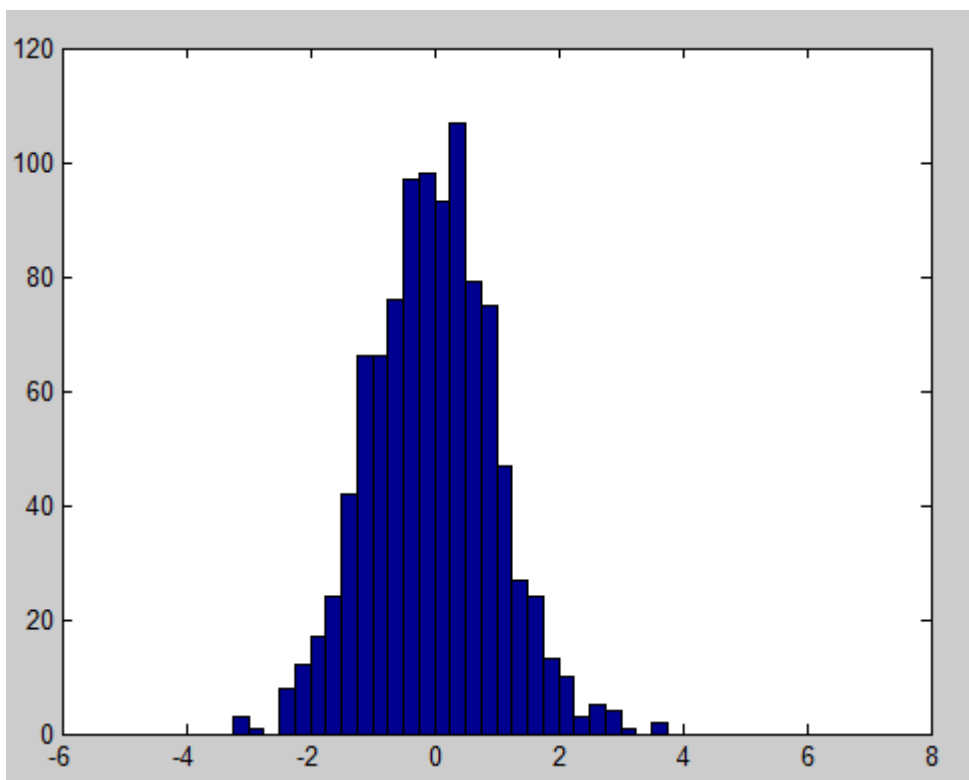
```
mode(X, 2)  
ans =  
     3  
     0  
     0
```

## Example 2

Find the mode of a continuous variable grouped into bins:

```
rng(0,'twister');           % Reset the random number generator  
y = randn(1000,1);  
edges = -6:.25:6;  
[n,bin] = histc(y,edges);  
m = mode(bin)  
m =  
     26  
  
edges([m, m+1])  
ans =  
     0.2500     0.5000  
  
hist(y,edges+.125)
```



**See Also**

[mean](#) | [median](#) | [hist](#) | [histc](#)

# 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.

| <b>Press the...</b> | <b>To...</b>                                                                                                                                 |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| 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 <b>Ctrl+C</b> to terminate <b>more</b> 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 `resizectl.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 $h \geq 0$ ) | Offset of left side from left edge of screen   |
| $h$ (for $h < 0$ )    | Offset of right side from right edge of screen |
| $v$ (for $v \geq 0$ ) | 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;
```

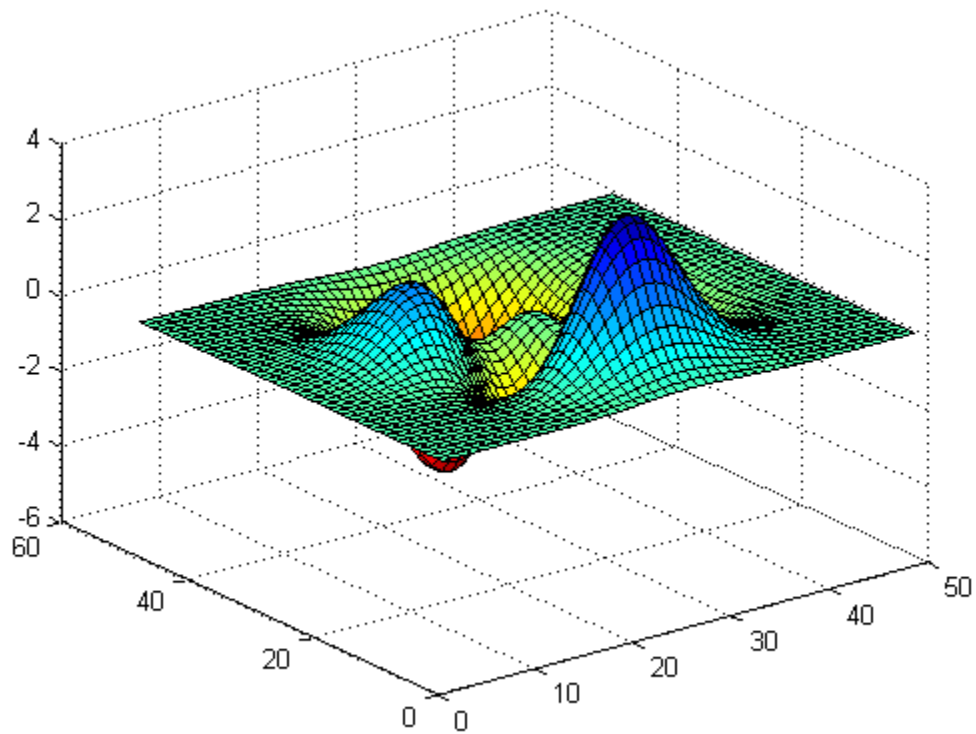
# movie

---

```
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](#)

# movie2avi

---

**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).<br><br>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'.<br><br>On UNIX operating systems, the only valid value is 'None'.<br><br>On Windows systems, valid values include: <ul style="list-style-type: none"><li>• 'MSVC'</li><li>• 'RLE'</li><li>• 'Cinepak' on 32-bit systems.</li></ul> | 'Indeo5' on Windows systems.<br><br>'None' on UNIX systems. |



| Parameter Name | Value                                                                                                                                                                                                                                                                                                                                                     | Default                      |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|
|                | <ul style="list-style-type: none"> <li>'Indeo3' or 'Indeo5' on 32-bit Windows XP systems.</li> </ul> <p>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.</p> |                              |
| '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.<br><br>Valid only for compressed movies.                                                                                                                            | 75                           |
| 'videoname'    | A descriptive name for the video stream, no more than 64 characters.                                                                                                                                                                                                                                                                                      | <i>filename</i>              |

**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 `WindowState` in the Figure Properties.

---

If `CreateMode` is a structure, it can have fields `WindowState` and `Interpreter`. The `WindowState` 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

```

$$A*B$$

$$\text{ans} =$$

```

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  $0 \leq \mu \leq 255$ , to linear signal amplitude in the range  $-s < Y < s$  where  $s = 32124/32768 \approx .9803$ . 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`

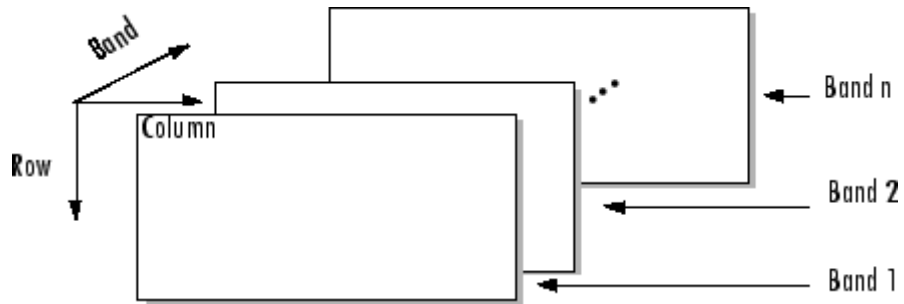
# multibandread

**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 2-2965 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 2-2966 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                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>filename</code>  | String containing the name of the file to be read.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>size</code>      | Three-element vector of integers consisting of [height, width, N], where <ul style="list-style-type: none"> <li>• height is the total number of rows</li> <li>• width is the total number of elements in each row</li> <li>• N is the total number of bands.</li> </ul> <p>This will be the dimensions of the data if it is read in its entirety.</p>                                                                                                                                                                                                                                                                                                                                  |
| <code>precision</code> | String specifying the format of the data to be read, such as <code>'uint8'</code> , <code>'double'</code> , <code>'integer*4'</code> , or any of the other precisions supported by the <code>fread</code> function. <p>Note: You can also use the <code>precision</code> parameter to specify the format of the output data. For example, to read <code>uint8</code> data and output a <code>uint8</code> array, specify a precision of <code>'uint8=&gt;uint8'</code> (or <code>'*uint8'</code>). To read <code>uint8</code> data and output it in the MATLAB software in single precision, specify <code>'uint8=&gt;single'</code>. See <code>fread</code> for more information.</p> |

# multibandread

---

| 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 <ul style="list-style-type: none"><li>• 'bsq' — Band-Sequential</li><li>• 'bil' — Band-Interleaved-by-Line</li><li>• 'bip' — Band-Interleaved-by-Pixel</li></ul> <p>For more information about these interleave methods, see the <code>multibandwrite</code> reference page.</p> |
| byteorder  | String specifying the byte ordering (machine format) in which the data is stored, such as <ul style="list-style-type: none"><li>• 'ieee-le' — Little-endian</li><li>• 'ieee-be' — Big-endian</li></ul> <p>See <code>fopen</code> for a complete list of supported formats.</p>                                                            |

## 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                                                                                                                                                       |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>dim</i> | Text string specifying the dimension to subset along. It can have any of these values: <ul style="list-style-type: none"><li>• 'Column'</li><li>• 'Row'</li></ul> |

| Parameter     | Description                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>method</i> | <ul style="list-style-type: none"><li>• 'Band'</li></ul> Text string specifying the subsetting method. It can have either of these values: <ul style="list-style-type: none"><li>• 'Direct'</li><li>• 'Range'</li></ul> If you leave out this element of the subset cell array, <code>multibandread</code> uses 'Direct' as the default.                                                                                             |
| <i>index</i>  | If <i>method</i> is 'Direct', <i>index</i> is a vector specifying the indices to read along the Band dimension.<br><br>If <i>method</i> is 'Range', <i>index</i> is a three-element vector of [start, increment, stop] specifying the range and step size to read along the dimension specified in <i>dim</i> . If <i>index</i> is a two-element vector, <code>multibandread</code> assumes that the value of <i>increment</i> 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.

# multibandread

---

```
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 2-2971 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.

# multibandwrite

---

---

**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.

---

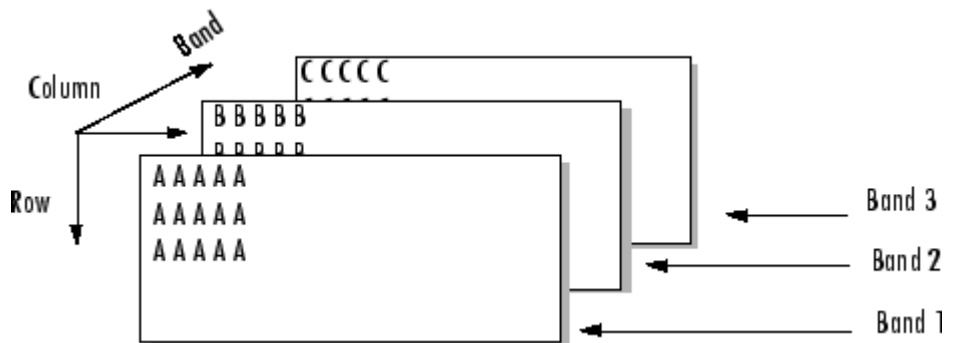
`multibandwrite(..., param, value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

| Parameter   | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'precision' | String specifying the form and size of each element written to the file. See the help for <code>fwrite</code> 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, <code>multibandwrite</code> writes ASCII null values to fill the space. To specify a different fill value, use the parameter <code>'fillvalue'</code> .<br><br>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 <code>fopen</code> using <code>'r+'</code> 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 <code>fopen</code> 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.

# multibandwrite

---

| Method                    | String | Description                        | Example                                                                       |
|---------------------------|--------|------------------------------------|-------------------------------------------------------------------------------|
| Band-Interleaved-by-Line  | 'bil'  | Write an entire row from each band | AAAAABBBBBCCCC<br>AAAAABBBBBCCCC<br>AAAAABBBBBCCCC                            |
| Band-Interleaved-by-Pixel | 'bip'  | Write a pixel from each band       | ABCABCABCABC...                                                               |
| Band-Sequential           | 'bsq'  | Write each band in its entirety    | AAAAA<br>AAAAA<br>AAAAA<br>BBBBB<br>BBBBB<br>BBBBB<br>CCCCC<br>CCCCC<br>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. MATLAB identifiers are

- Variable names
- Function and local function names
- Structure fieldnames
- Object names
- File names with a `.m`, `.mex`, or `.mdl` extension

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,...])
```

**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 NaNs.

NaN(m,n) or NaN([m,n]) is an m-by-n matrix of NaNs.

NaN(m,n,p,...) or NaN([m,n,p,...]) is an m-by-n-by-p-by-... array of NaNs.

---

**Note** The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

---

NaN(...,classname) is an array of NaNs of class specified by classname. classname must be either 'single' or 'double'.

**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)`

# NaN

---

- Multiplication, such as  $0 \cdot \text{Inf}$
- Division, such as  $0/0$  and  $\text{Inf}/\text{Inf}$
- 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

# nargchk

---

```
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.

# nargin

---

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`

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Validate number of input arguments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <code>narginchk(minargs, maxargs)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Description</b> | <p><code>narginchk(minargs, maxargs)</code> throws an error if the number of inputs specified in the call to the currently executing function is less than <code>minargs</code> or greater than <code>maxargs</code>. If the number of inputs is between <code>minargs</code> and <code>maxargs</code> (inclusive), <code>narginchk</code> does nothing.</p> <p>When too few inputs are supplied, the message identifier and message are:</p> <pre>identifier: 'MATLAB:narginchk:notEnoughInputs'<br/>message: 'Not enough input arguments.'</pre> <p>When too many inputs are supplied, the message identifier and message are:</p> <pre>identifier: 'MATLAB:narginchk:tooManyInputs'<br/>message: 'Too many input arguments.'</pre> |
| <b>Examples</b>    | <p>This function uses <code>narginchk</code> to verify that a minimum of 2 and maximum of 5 input arguments are received from the calling function:</p> <pre>function check_inputs(A, B, varargin)<br/>minargs=2; maxargs=5;<br/><br/>% Number of inputs must be &gt;=minargs and &lt;=maxargs.<br/>narginchk(minargs, maxargs)<br/><br/>fprintf('Received 2 required, %d optional inputs.\n\n', ...<br/>    size(varargin, 2))</pre> <p>Call the example function, passing 1 input argument:</p> <pre>check_inputs(23)<br/>Error using check_inputs<br/>Not enough input arguments.</pre>                                                                                                                                            |

# narginchk

---

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](#)



|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Number of function output arguments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Syntax</b>          | nargout<br>nargout(fx)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b>     | <p>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.</p> <p>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.</p>                                                                                                  |
| <b>Input Arguments</b> | <p><b>fx</b></p> <p>Either a function handle or a string in single quotes that specifies the name of a function.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Examples</b>        | <p><b>Outputs for Current Function</b></p> <p>Create a function in a file named <code>subtract.m</code> that calculates a second return value only when requested.</p> <pre>function [dif,absdif] = subtract(y,x) dif = y - x; if nargout &gt; 1     disp('Calculating absolute value')     absdif = abs(dif); end</pre> <p><b>Outputs Defined for a Function</b></p> <p>Determine how many outputs a function can return.</p> <p>The function named <code>subtract</code> created in the previous example has two outputs in its declaration statement (<code>dif</code> and <code>absdif</code>).</p> |

```
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**  $C = \text{nchoosek}(n, k)$   
 $C = \text{nchoosek}(v, k)$

**Description**  $C = \text{nchoosek}(n, k)$  where  $n$  and  $k$  are nonnegative integers, returns  $n!/((n-k)! k!)$ . This is the number of combinations of  $n$  things taken  $k$  at a time.

$C = \text{nchoosek}(v, k)$ , where  $v$  is a row vector of length  $n$ , creates a matrix whose rows consist of all possible combinations of the  $n$  elements of  $v$  taken  $k$  at a time. Matrix  $C$  contains  $n!/((n-k)! k!)$  rows and  $k$  columns.

Inputs  $n$ ,  $k$ , and  $v$  support classes of `float double` and `float single`.

**Examples** The command `nchoosek(2:2:10,4)` returns the even numbers from two to ten, taken four at a time:

|   |   |   |    |
|---|---|---|----|
| 2 | 4 | 6 | 8  |
| 2 | 4 | 6 | 10 |
| 2 | 4 | 8 | 10 |
| 2 | 6 | 8 | 10 |
| 4 | 6 | 8 | 10 |

**Limitations** When  $C = \text{nchoosek}(n, k)$  has a large coefficient, a warning will be produced indicating possible inexact results. In such cases, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.

$C = \text{nchoosek}(v, k)$  is only practical for situations where  $n$  is less than about 15.

**See Also** `perms`



|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Rectangular grid in N-D space                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>           | $[X1, X2, X3, \dots, Xn] = \text{ndgrid}(x1gv, x2gv, x3gv, \dots, xngv)$<br>$[X1, X2, \dots, Xn] = \text{ndgrid}(xgv)$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>      | <p><math>[X1, X2, X3, \dots, Xn] = \text{ndgrid}(x1gv, x2gv, x3gv, \dots, xngv)</math> replicates the grid vectors <math>x1gv, x2gv, x3gv, \dots, xngv</math> to produce a full grid. This grid is represented by the output coordinate arrays <math>X1, X2, X3, \dots, Xn</math>. The <math>i</math>th dimension of any output array <math>Xi</math> contains copies of the grid vector <math>xigv</math>.</p> <p><math>[X1, X2, \dots, Xn] = \text{ndgrid}(xgv)</math> is the same as <math>[X1, X2, \dots, Xn] = \text{ndgrid}(xgv, xgv, \dots, xgv)</math>. 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.</p> <p>The coordinate arrays <math>[X1, X2, X3, \dots, Xn]</math> are typically used to evaluate functions of several variables and to create surface and volumetric plots.</p> |
| <b>Input Arguments</b>  | <p><b>xigv</b><br/>Grid vector specifying a series of grid point coordinates in the <math>i</math>th dimension.</p> <p><b>xgv</b><br/>Generic grid vector specifying a series of point coordinates.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Output Arguments</b> | <p><b><math>Xi</math></b><br/>The <math>i</math>th dimension of the output array <math>Xi</math> are copies of elements of the grid vector <math>xigv</math>. The output arrays specify the full grid.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Tips</b>             | <p>The <code>ndgrid</code> function is similar to <code>meshgrid</code>, however <code>ndgrid</code> supports 1-D to N-D while <code>meshgrid</code> 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 <math>x1gv</math>,</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

# ndgrid

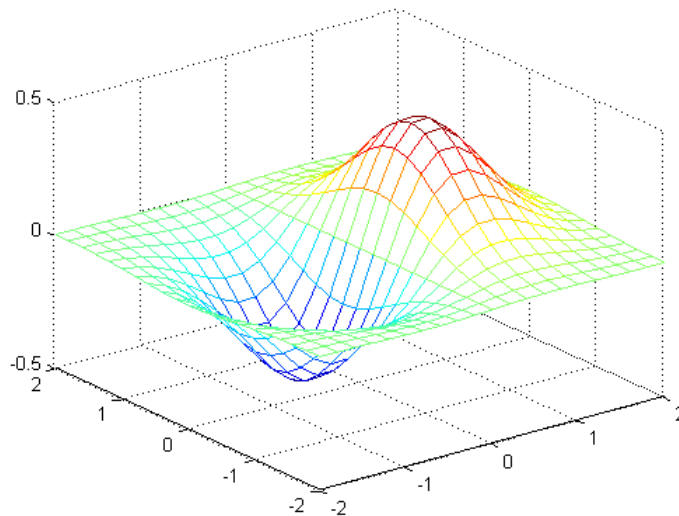
---

`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”

**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

# DelaunayTri.nearestNeighbor

---

**Purpose** Point closest to specified location

**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

DelaunayTri.pointLocation

## ne (MException)

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Compare scalar MException objects for inequality                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Syntax</b>      | <code>eObj1 ~= eObj2</code>                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b> | <code>eObj1 ~= eObj2</code> tests MException objects <code>eObj1</code> and <code>eObj2</code> for inequality, returning logical 1 (true) if the two objects are not identical, otherwise returning logical 0 (false).                                                                                                                                                                                                            |
| <b>See Also</b>    | <code>try</code>   <code>catch</code>   <code>error</code>   <code>assert</code>   <code>MException</code>   <code>isequal(MException)</code>   <code>eq(MException)</code>   <code>getReport(MException)</code>   <code>disp(MException)</code>   <code>throw(MException)</code>   <code>rethrow(MException)</code>   <code>throwAsCaller(MException)</code>   <code>addCause(MException)</code>   <code>last(MException)</code> |



|                         |                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Simplex neighbor information                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>           | <code>SN = neighbors(TR, SI)</code>                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>      | <code>SN = neighbors(TR, SI)</code> returns the simplex neighbor information for the specified simplices <code>SI</code> .                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Input Arguments</b>  | <code>TR</code>                                                                                                                                  | Triangulation representation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|                         | <code>SI</code>                                                                                                                                  | <code>SI</code> is a column vector of simplex indices that index into the triangulation matrix <code>TR.Triangulation</code> . If <code>SI</code> is not specified the neighbor information for the entire triangulation is returned, where the neighbors associated with simplex <code>i</code> are defined by the <code>i</code> 'th row of <code>SN</code> .                                                                                                                                                                          |
| <b>Output Arguments</b> | <code>SN</code>                                                                                                                                  | <code>SN</code> is an <code>m</code> -by- <code>n</code> matrix, where <code>m = length(SI)</code> , the number of specified simplices, and <code>n</code> is the number of neighbors per simplex. Each row <code>SN(i, :)</code> represents the neighbors of the simplex <code>SI(i)</code> .<br><br>By convention, the simplex opposite vertex <code>(j)</code> of simplex <code>SI(i)</code> is <code>SN(i, j)</code> . If a simplex has one or more boundary facets, the nonexistent neighbors are represented by <code>NaN</code> . |
| <b>Definitions</b>      | <i>A simplex</i> is a triangle/tetrahedron or higher-dimensional equivalent. A <i>facet</i> is an edge of a triangle or a face of a tetrahedron. |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Examples</b>         | <b>Example 1</b>                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|                         | Load a 3-D triangulation and use <code>TriRep</code> to compute the neighbors of all tetrahedra.<br><br><code>load tetmesh</code>                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

# TriRep.neighbors

---

```
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

`DelaunayTri`

**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.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”

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Make .NET assembly visible to MATLAB                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Syntax</b>           | <pre>asmInfo = NET.addAssembly(globalName) asmInfo = NET.addAssembly(privateName)</pre>                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Description</b>      | <p><code>asmInfo = NET.addAssembly(globalName)</code> loads a global .NET assembly into MATLAB.</p> <p><code>asmInfo = NET.addAssembly(privateName)</code> loads a private .NET assembly.</p>                                                                                                                                                                                                                                                               |
| <b>Tips</b>             | <ul style="list-style-type: none"><li>• MATLAB dynamically loads the <code>mscorlib.dll</code> and <code>system.dll</code> assemblies from the .NET Framework class library the first time you type "NET." or "System.". You do not need to call <code>NET.addAssembly</code> to access classes in these assemblies.</li><li>• Refer to your .NET product documentation for the name of the assembly and its deployment type (global or private).</li></ul> |
| <b>Limitations</b>      | <ul style="list-style-type: none"><li>• <code>NET.addAssembly</code> does not support assemblies generated by the MATLAB Builder NE product.</li></ul>                                                                                                                                                                                                                                                                                                      |
| <b>Input Arguments</b>  | <p><b>globalName</b></p> <p>One of the following:</p> <ul style="list-style-type: none"><li>• String representing the name of a global assembly.</li><li>• Instance of <code>System.Reflection.AssemblyName</code> class.</li></ul> <p><b>privateName</b></p> <p>String representing the full path of a private assembly.</p>                                                                                                                               |
| <b>Output Arguments</b> | <p><b>asmInfo</b></p> <p><code>NET.Assembly</code> object containing names of the members of the assembly.</p>                                                                                                                                                                                                                                                                                                                                              |

# NET.addAssembly

---

## 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](#)

|                       |                                                                                                                                    |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>        | Members of .NET assembly                                                                                                           |
| <b>Description</b>    | NET.Assembly object returns names of the members of an assembly.                                                                   |
| <b>Construction</b>   | The NET.addAssembly function creates an instance of this class.                                                                    |
| <b>Properties</b>     | <b>AssemblyHandle</b><br>Instance of System.Reflection.Assembly class of the added assembly.                                       |
|                       | <b>Classes</b><br>nClassx1 cell array of class names of the assembly, where nClass is the number of classes                        |
|                       | <b>Enums</b><br>nEnumx1 cell array of enums of the assembly, where nEnum is the number of enums                                    |
|                       | <b>Structures</b><br>nStructx1 cell array of structures of the assembly, where nStruct is the number of structures                 |
|                       | <b>GenericTypes</b><br>nGenTypex1 cell array of generic types of the assembly, where nGenType is the number of generic types       |
|                       | <b>Interfaces</b><br>nInterfacex1 cell array of interface names of the assembly, where nInterface is the number of interfaces      |
|                       | <b>Delegates</b><br>nDelegatex1 cell array of delegates of the assembly, where nDelegate is the number of delegates                |
| <b>Copy Semantics</b> | Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation. |

# NET.Assembly

---

## See Also

`NET.addAssembly`

## How To

- “What Classes Are in a .NET Assembly?”



**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`

# 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.

# NET.createArray

---

```
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](#)

# NET.createGeneric

---

|                         |                                                                                                                                                                  |                                                                                                                                                                                                     |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Create instance of specialized .NET generic type                                                                                                                 |                                                                                                                                                                                                     |
| <b>Syntax</b>           | <code>genObj = createGeneric(className,paramTypes,<br/>varargin ctorArgs)</code>                                                                                 |                                                                                                                                                                                                     |
| <b>Description</b>      | <code>genObj = createGeneric(className,paramTypes,varargin<br/>ctorArgs)</code> creates an instance <code>genObj</code> of generic type <code>className</code> . |                                                                                                                                                                                                     |
| <b>Input Arguments</b>  | <code>className</code>                                                                                                                                           | Fully qualified string with the generic type name.                                                                                                                                                  |
|                         | <code>paramTypes</code>                                                                                                                                          | Allowed cell types are: strings with fully qualified parameter type names and instances of the <code>NET.GenericClass</code> class when parameterization with another parameterized type is needed. |
|                         | <code>ctorArgs</code>                                                                                                                                            | Optional, variable length (0 to N) list of constructor arguments matching the arguments of the .NET generic class constructor intended to be invoked.                                               |
| <b>Output Arguments</b> | <code>genObj</code>                                                                                                                                              | 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`

# NET.disableAutoRelease

---

**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.StringRetVal readData(ComNamespace.ComClass this, System.String s)
```

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”



- 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”

# NET.GenericClass

---

**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 Programming Fundamentals 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”

# NET.invokeGenericMethod

---

## 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”

# NET.NetException

---

<b>Purpose</b>	Capture error information for .NET exception
<b>Description</b>	Process information from a NET.NetException object to handle .NET errors. This class is derived from MException.
<b>Construction</b>	<code>e = NET.NetException(msgID, errMsg, netObj)</code> constructs instance <code>e</code> of NET.NetException class.
<b>Input Arguments</b>	<b>msgID</b> message identifier <b>errMsg</b> error message string <b>netObj</b> System.Exception object that caused the exception
<b>Properties</b>	<b>ExceptionObject</b> System.Exception class causing the error.
<b>Methods</b>	<b>Inherited Methods</b> See the methods of the base class MException.
<b>Copy Semantics</b>	Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.
<b>Examples</b>	Display error information after trying to load an unknown assembly: <pre>try NET.addAssembly('C:\Work\invalidfile.dll') catch e e.message; if(isa(e, 'NET.NetException')) eObj = e.ExceptionObject</pre>

```
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);
```



---

<b>Purpose</b>	Create variable in NetCDF file
<b>Syntax</b>	<code>nccreate(filename,varname)</code> <code>nccreate(filename,varname,Name,Value)</code>
<b>Description</b>	<p><code>nccreate(filename,varname)</code> creates a scalar double variable named <code>varname</code> in the NetCDF file <code>filename</code>. If <code>filename</code> does not exist, <code>nccreate</code> creates the file using the <code>netcdf4_classic</code> format. To create a nonscalar variable, use the <code>Dimensions</code> argument.</p> <p><code>nccreate(filename,varname,Name,Value)</code> creates a variable named <code>varname</code> with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>filename</b></p> <p>Text string specifying a NetCDF file, or the name you want to assign to a new NetCDF file.</p> <p><b>varname</b></p> <p>Text string specifying the name you want to assign to a variable in a NetCDF file.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>Dimensions</b></p> <p>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: <code>{dname1,dlength1,dname2,dlength2,...}</code>. If a dimension exists, specifying its length is optional. Use <code>Inf</code> to specify an unlimited dimension.</p>

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.

<b>MATLAB Data Type</b>	<b>Corresponding NetCDF Variable Type</b>
<code>double</code>	<code>NC_DOUBLE</code>
<code>single</code>	<code>NC_FLOAT</code>
<code>int64</code>	<code>NC_INT64*</code>
<code>uint64</code>	<code>NC_UINT64*</code>
<code>int32</code>	<code>NC_INT</code>
<code>uint32</code>	<code>NC_UINT*</code>
<code>int16</code>	<code>NC_SHORT</code>
<code>uint16</code>	<code>NC_USHORT*</code>
<code>int8</code>	<code>NC_BYTE</code>
<code>uint8</code>	<code>NC_UBYTE*</code>
<code>char</code>	<code>NC_CHAR</code>

\* 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` | `ncwritescema` | `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

# ncdisp

---

## 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.

```
vinfos = ncinfo('example.nc','peaks');
varSize = vinfos.Size;
disp(vinfos);
hasUnLimDim = any([vinfos.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` | `ncwritschema` | `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 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]);
```

# ncread

---

```
surf(double(subsetdata));
```

## See Also

`ncdisp` | `ncinfo` | `netcdf` | `ncwrite` | `ncreadatt`

## Tutorials

- “Importing Network Common Data Form (NetCDF) Files and OPeNDAP Data”

---

<b>Purpose</b>	Read attribute value from NetCDF data source
<b>Syntax</b>	<code>attvalue = ncreadatt(source,location,attname)</code>
<b>Description</b>	<code>attvalue = ncreadatt(source,location,attname)</code> reads the attribute <code>attname</code> from the group or variable specified by <code>location</code> in <code>source</code> , where <code>source</code> is the name of a NetCDF file or the URL of a NetCDF data source.
<b>Input Arguments</b>	<p><b>source</b></p> <p>Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.</p> <p><b>location</b></p> <p>Text string specifying a group or variable in the NetCDF data source. To read global attributes, set <code>location</code> to <code>'/'</code> (forward slash).</p> <p><b>attname</b></p> <p>Text string specifying the name of an attribute that you want to read in the NetCDF data source.</p>
<b>Output Arguments</b>	<p><b>attvalue</b></p> <p>Data associated with the attribute.</p>
<b>Examples</b>	<p>Read a global attribute.</p> <pre>creation_date = ncreadatt('example.nc','/','creation_date'); disp(creation_date);</pre> <hr/> <p>Read an attribute associated with a variable.</p> <pre>scale_factor = ncreadatt('example.nc','temperature','scale_factor'); disp(scale_factor);</pre>

# ncreadatt

---

---

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”



---

<b>Purpose</b>	Write data to NetCDF file
<b>Syntax</b>	<code>ncwrite(filename,varname,vardata)</code> <code>ncwrite(filename,varname,vardata,start, stride)</code>
<b>Description</b>	<p><code>ncwrite(filename,varname,vardata)</code> writes the numerical or char data in <code>vardata</code> to an existing variable <code>varname</code> in the NetCDF file <code>filename</code>. <code>ncwrite</code> writes the data in <code>vardata</code> starting at the beginning of the variable and extends unlimited dimensions automatically, if needed.</p> <p>If the NetCDF file or the variable do not exist, use <code>nccreate</code> to create them first.</p> <p><code>ncwrite(filename,varname,vardata,start, stride)</code> writes <code>vardata</code> to an existing variable <code>varname</code> in file <code>filename</code> beginning at the location given by <code>start</code>. <code>stride</code> 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.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• If the variable <code>varname</code> already exists, <code>ncwrite</code> expects the datatype of <code>vardata</code> to match the NetCDF variable data type.</li><li>• If the variable <code>varname</code> has a <code>_FillValue</code>, <code>scale_factor</code> or <code>add_offset</code> attribute, <code>ncwrite</code> expects data in double format and casts <code>vardata</code> to the NetCDF data type, after applying the following attribute conventions in sequence:<ol style="list-style-type: none"><li>1 Subtract the value of the <code>add_offset</code> attribute from <code>vardata</code>.</li><li>2 Divide <code>vardata</code> by the value of the <code>scale_factor</code> attribute.</li><li>3 Replace NaNs in <code>vardata</code> by the value of the <code>_FillValue</code> attribute. If this attribute does not exist, <code>ncwrite</code> tries to use the fill value for this variable as reported by the NetCDF library.</li></ol></li></ul>
<b>Input Arguments</b>	<p><b>filename</b></p> <p>Text string specifying the name of a NetCDF file. If the file does not exist, use <code>nccreate</code> to create it first.</p>

**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](#) | [ncreate](#)

**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](#) | [ncreate](#) | [netcdf](#)

**Tutorials**

- “Exporting to Network Common Data Form (NetCDF) Files”

# ncwritescema

---

**Purpose** Add NetCDF schema definitions to NetCDF file

**Syntax** `ncwritescema(filename,schema)`

**Description** `ncwritescema(filename,schema)` creates or adds attributes, dimensions, variable definitions and group structure defined in `schema` to the file `filename`.

Use `ncwritescema` in combination with `ncinfo` to create a new NetCDF file based on the schema of an existing file. You can also use `ncwritescema` to add variable definitions, attributes, dimensions, or group structure to an existing file.

---

**Note** `ncwritescema` 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** `ncwritescema` 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, `ncwritescema` 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;
ncwritschema('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 `ncwritschema` 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');
ncwritschema('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”

**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.

<code>nccreate</code>	Create variable in NetCDF file
<code>ncdisp</code>	Display contents of NetCDF data source in Command Window
<code>ncinfo</code>	Return information about NetCDF data source
<code>ncread</code>	Read data from variable in NetCDF data source
<code>ncreadatt</code>	Read attribute value from NetCDF data source
<code>ncwrite</code>	Write data to NetCDF file
<code>ncwriteatt</code>	Write attribute to NetCDF file
<code>ncwritschema</code>	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

<code>netcdf</code>	Summary of MATLAB Network Common Data Form (NetCDF) capabilities
<code>netcdf.getChunkCache</code>	Retrieve chunk cache settings for NetCDF library
<code>netcdf.inqLibVers</code>	Return NetCDF library version information
<code>netcdf.setChunkCache</code>	Set default chunk cache settings for NetCDF library
<code>netcdf.setDefaultFormat</code>	Change default netCDF file format

## File Operations

<code>netcdf.abort</code>	Revert recent netCDF file definitions
<code>netcdf.close</code>	Close netCDF file

<code>netcdf.create</code>	Create new NetCDF dataset
<code>netcdf.endDef</code>	End netCDF file define mode
<code>netcdf.inq</code>	Return information about netCDF file
<code>netcdf.inqFormat</code>	Determine format of NetCDF file
<code>netcdf.inqGrps</code>	Retrieve array of child group IDs
<code>netcdf.inqUnlimDims</code>	Return list of unlimited dimensions in group
<code>netcdf.open</code>	Open NetCDF data source
<code>netcdf.reDef</code>	Put open netCDF file into define mode
<code>netcdf.setFill</code>	Set netCDF fill mode
<code>netcdf.sync</code>	Synchronize netCDF file to disk
<b>Dimensions</b>	
<code>netcdf.defDim</code>	Create netCDF dimension
<code>netcdf.inqDim</code>	Return netCDF dimension name and length
<code>netcdf.inqDimID</code>	Return dimension ID
<code>netcdf.renameDim</code>	Change name of netCDF dimension
<b>Variables</b>	
<code>netcdf.defVar</code>	Create NetCDF variable
<code>netcdf.defVarChunking</code>	Define chunking behavior for NetCDF variable
<code>netcdf.defVarDeflate</code>	Define compression parameters for NetCDF variable
<code>netcdf.defVarFill</code>	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

<b>Purpose</b>	Revert recent netCDF file definitions
<b>Syntax</b>	<code>netcdf.abort(ncid)</code>
<b>Description</b>	<p><code>netcdf.abort(ncid)</code> 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 <code>netcdf.create</code>) or when you explicitly enter define mode (using <code>netcdf.redef</code>). Once you leave define mode (using <code>netcdf.endDef</code>), you cannot revert the definitions you made while in define mode. <code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>. A call to <code>netcdf.abort</code> closes the file.</p> <p>This function corresponds to the <code>nc_abort</code> function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>
<b>Examples</b>	<p>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.</p> <pre>% Create a netCDF file ncid = netcdf.create('foo.nc','NC_NOCLlobber');  % 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 ==&gt; netcdflib NetCDF: Not a valid ID  Error in ==&gt; defDim at 22 dimid = netcdflib('def_dim', ncid,dimname,dimlen);</pre>

# netcdf.abort

---

## See Also

[netcdf.create](#) | [netcdf.endDef](#) | [netcdf.reDef](#)



<b>Purpose</b>	Close netCDF file
<b>Syntax</b>	<code>netcdf.close(ncid)</code>
<b>Description</b>	<p><code>netcdf.close(ncid)</code> terminates access to the netCDF file identified by <code>ncid</code>.</p> <p><code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p>This function corresponds to the <code>nc_close</code> function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>
<b>Examples</b>	<p>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.</p> <pre>ncid = netcdf.open('foo.nc', 'NC_WRITE')  netcdf.close(ncid)</pre>
<b>See Also</b>	<code>netcdf.create</code>   <code>netCDF.open</code>

# netcdf.copyAtt

---

<b>Purpose</b>	Copy attribute to new location
<b>Syntax</b>	<code>netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)</code>
<b>Description</b>	<p><code>netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)</code> copies an attribute from one variable to another, possibly across files. <code>ncid_in</code> and <code>ncid_out</code> are netCDF file identifiers returned by <code>netcdf.create</code> or <code>netcdf.open</code>. <code>varid_in</code> identifies the variable with an attribute that you want to copy. <code>varid_out</code> identifies the variable to which you want to associate a copy of the attribute.</p> <p>This function corresponds to the <code>nc_copy_att</code> function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>

## 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_NOCLlobber');

% 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
NOGLOBBER	Prevent overwriting of existing file with the
GLOBBER	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_NOCLlobber')

% Define a dimension.
lat_dimID = netcdf.defDim(ncid,'latitude',360);

% Define an unlimited dimension.
long_dimID = netcdf.defDim(ncid,'longitude',netcdf.getConstant('NC_UNLIMITED'))
```

**See Also** `netcdf.getConstant`

<b>Purpose</b>	Create group in NetCDF file
<b>Syntax</b>	<code>childGrpID = netcdf.defGrp(parentGroupId,childGroupName)</code>
<b>Description</b>	<code>childGrpID = netcdf.defGrp(parentGroupId,childGroupName)</code> creates a child group with the name specified by <code>childGroupName</code> , that is the child of the parent group specified by <code>parentGroupId</code>
<b>Input Arguments</b>	<p><b>parentGroupId</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>childGroupName</b> Text string specifying the name that you want to assign to the group.</p>
<b>Output Arguments</b>	<p><b>childGrpID</b> Identifier of a NetCDF group.</p>
<b>Examples</b>	<p>This example creates a NetCDF dataset and then defines a group.</p> <pre>ncid = netcdf.create('myfile.nc','netcdf4'); childGroupId = netcdf.defGrp(ncid,'mygroup'); netcdf.close(ncid);</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_def_grp</code> function in the NetCDF library C API.</p> <p>For copyright information, read the files <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code>.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqGrps</code>

# 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_NOCLlobber');
%
% 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`



<b>Purpose</b>	Define chunking behavior for NetCDF variable
<b>Syntax</b>	<code>netcdf.defVarChunking(ncid,varid,storage,chunkDims)</code>
<b>Description</b>	<p><code>netcdf.defVarChunking(ncid,varid,storage,chunkDims)</code>. sets the chunk settings for the variable specified by <code>varid</code>. Chunking is a technique to improve performance. <code>storage</code> specifies the type of chunking to use and <code>chunkDims</code> 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.</p> <p>You cannot specify the chunk size for variables in a NetCDF file created with the netCDF-3 mode (<code>CLASSIC_MODEL</code>).</p>
<b>Input Arguments</b>	<p><b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>varid</b> Identifier of a NetCDF variable, returned by <code>netcdf.defVar</code>.</p> <p><b>storage</b> Text string specifying whether NetCDF should break the variable into chunks when writing to a file. If set to <code>CHUNKED</code>, NetCDF breaks the variable into chunks; if set to <code>CONTIGUOUS</code>, NetCDF does not break the data into chunks.</p> <p><b>chunkDims</b> Array specifying the dimensions of the chunk. Because MATLAB uses FORTRAN-style ordering, the order of dimensions in <code>chunkdims</code> is reversed relative to what would be in the C API. If <code>storage</code> is <code>CONTIGUOUS</code>, you can omit <code>chunkDims</code>.</p>

# netcdf.defVarChunking

---

**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`

<b>Purpose</b>	Define compression parameters for NetCDF variable
<b>Syntax</b>	<code>netcdf.defVarDeflate(ncid, varid, shuffle, deflate, deflateLevel)</code>
<b>Description</b>	<code>netcdf.defVarDeflate(ncid, varid, shuffle, deflate, deflateLevel)</code> sets the compression parameters for the NetCDF variable specified by <code>varid</code> in the location specified by <code>ncid</code> .
<b>Input Arguments</b>	<p><b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>varid</b> Identifier of a NetCDF variable, returned by <code>netcdf.defVar</code>.</p> <p><b>shuffle</b> Boolean value. To turn on the shuffle filter, set this argument to <code>true</code>. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.</p> <p><b>deflate</b> Boolean value. To turn on compression, set this argument to <code>true</code> and set the <code>deflateLevel</code> argument to the desired compression level.</p> <p><b>deflateLevel</b> Numeric value between 0 and 9 specifying the amount of compression, where 0 is no compression and 9 is the most compression.</p>
<b>Examples</b>	<p>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.</p> <pre>ncid = netcdf.create('myfile.nc', 'NETCDF4'); latdimid = netcdf.defDim(ncid, 'lat', 1800);</pre>

# netcdf.defVarDeflate

---

```
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`

<b>Purpose</b>	Define fill parameters for NetCDF variable
<b>Syntax</b>	<code>netcdf.defVarFill(ncid, varid, noFillMode, fillValue)</code>
<b>Description</b>	<p><code>netcdf.defVarFill(ncid, varid, noFillMode, fillValue)</code> sets the fill parameters for the NetCDF variable identified by <code>varid</code>. <code>ncid</code> specifies the location.</p> <p>For netCDF-4 files, you can only specify fill values when the NetCDF is in definition mode (before calling <code>netcdf.endDef</code>). For NetCDF files in classic and 64-bit offset modes, you can turn no-fill mode on and off at any time.</p>
<b>Input Arguments</b>	<p><b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>varid</b> Identifier of a NetCDF variable, returned by <code>netcdf.defVar</code>.</p> <p><b>noFillMode</b> Boolean value. When set to <code>true</code>, turns off use of fill values for the variable, which can be helpful in high performance applications. When <code>true</code>, <code>netcdf.defVarFill</code> ignores the value of the <code>fillValue</code> argument. To use the fill value, set this to <code>false</code>.</p> <p><b>fillValue</b> 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.</p>
<b>Examples</b>	<p>This example creates a NetCDF file and defines a fill value for a variable.</p> <pre>ncid = netcdf.create('myfile.nc', 'NETCDF4'); dimid = netcdf.defDim(ncid, 'latitude', 180);</pre>

# netcdf.defVarFill

---

```
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`

<b>Purpose</b>	Define checksum parameters for NetCDF variable
<b>Syntax</b>	<code>netcdf.defVarFletcher32(ncid,varid,setting)</code>
<b>Description</b>	<code>netcdf.defVarFletcher32(ncid,varid,setting)</code> defines the checksum settings for the NetCDF variable specified by <code>varid</code> in the file specified by <code>ncid</code> .
<b>Input Arguments</b>	<p><b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>varid</b> Identifier of a NetCDF variable, returned by <code>netcdf.defVar</code>.</p> <p><b>setting</b> Text string specifying whether Fletcher32 checksum error detection is used with the variable. To turn on Fletcher32 checksum, specify the value <code>FLETCHER32</code>. To turn off the use of checksum error detection, specify the value <code>NOCHECKSUM</code>.</p>
<b>Examples</b>	<p>This example creates a NetCDF dataset and turns on the Fletcher32 checksum for a variable.</p> <pre>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);</pre>
<b>References</b>	This function corresponds to the <code>nc_def_var_fletcher32</code> function in the NetCDF library C API.

# netcdf.defVarFletcher32

---

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

## **See Also**

`netcdf` | `netcdf.inqVarFletcher32`



**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

---

```
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](#)

<b>Purpose</b>	End netCDF file define mode
<b>Syntax</b>	<pre>netcdf.endDef(ncid) netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)</pre>
<b>Description</b>	<p><code>netcdf.endDef(ncid)</code> takes a netCDF file out of define mode and into data mode. <code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p><code>netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)</code> 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 <code>h_minfree</code> parameter:</p> <pre>ncid = netcdf.endDef(ncid,20000,4,0,4);</pre> <p>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.</p> <p>This function corresponds to the <code>nc_enddef</code> and <code>nc__enddef</code> functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>
<b>Examples</b>	<p>When you create a file using <code>netcdf.create</code>, the functions opens the file in define mode. This example uses <code>netcdf.endDef</code> to take the file out of define mode.</p> <pre>% Create a netCDF file. ncid = netcdf.create('foo.c','NC_NOCLlobber');  % Define a dimension. dimid = netcdf.defDim(ncid, 'lat', 50);  % Leave define mode. netcdf.endDef(ncid)</pre>

## netcdf.endDef

---

```
% Test if still in define mode.
dimid = netcdf.defDim(ncid, 'lon', 50); % should fail
??? Error using ==> netcdf.lib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdf.lib('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.
```

## netcdf.getAtt

---

```
[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](#)

<b>Purpose</b>	Retrieve chunk cache settings for NetCDF library
<b>Syntax</b>	<code>[csize, nelems, premp] = netcdf.getChunkCache()</code>
<b>Description</b>	<code>[csize, nelems, premp] = netcdf.getChunkCache()</code> returns the default chunk cache settings.
<b>Output Arguments</b>	<p><b>csize</b></p> <p>Scalar double specifying the total size of the raw data chunk cache in bytes.</p> <p><b>nelems</b></p> <p>Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.</p> <p><b>premp</b></p> <p>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.</p>
<b>Examples</b>	<p>Determine information about the chunk cache size used by the NetCDF library.</p> <pre>[csize, nelems, premp] = netcdf.getChunkCache();</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_get_chunk_cache</code> function in the NetCDF library C API.</p> <p>For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.setChunkCache</code>

# 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_NOGLOBBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOGLOBBBER`.

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
```

# netcdf.getVar

---

## **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 =
```

# netcdf.inq

---

1

unlimdimID =

3

## **See Also**

`netcdf.create` | `netcdf.open`

<b>Purpose</b>	Retrieve list of dimension identifiers in group
<b>Syntax</b>	<pre>dimIDs = netcdf.inqDimIDs(ncid) dimIDs = netcdf.inqDimIDs(ncid,includeParents)</pre>
<b>Description</b>	<p><code>dimIDs = netcdf.inqDimIDs(ncid)</code> returns a list of dimension identifiers in the group specified by <code>ncid</code>.</p> <p><code>dimIDs = netcdf.inqDimIDs(ncid,includeParents)</code> includes all dimensions in all parent groups if <code>includeParents</code> is true.</p>
<b>Input Arguments</b>	<p><b>ncid</b></p> <p>Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code>, or of a NetCDF group, returned by <code>netcdf.defGrp</code>.</p> <p><b>includeParents</b></p> <p>Boolean value. If set to true, <code>netcdf.inqDimIDs</code> includes the dimensions of all parent groups.</p> <p style="text-align: center;"><b>Default:</b> false</p>
<b>Output Arguments</b>	<p><b>dimIDs</b></p> <p>Array of dimension IDs</p>
<b>Examples</b>	<p>This example opens the NetCDF sample file and gets the IDs of all the dimensions.</p> <pre>ncid = netcdf.open('example.nc','NOWRITE'); gid = netcdf.inqNcid(ncid,'grid1'); dimids = netcdf.inqDimIDs(gid); dimids_all = netcdf.inqDimIDs(gid, true); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_dimids</code> function in the NetCDF library C API.

# netcdf.inqDimIDs

---

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)
```

# netcdf.inqFormat

---

```
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`

<b>Purpose</b>	Retrieve name of group
<b>Syntax</b>	<code>groupName = netcdf.inqGrpName(ncid)</code>
<b>Description</b>	<code>groupName = netcdf.inqGrpName(ncid)</code> returns the name of a group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>groupName</b> Text string specifying name of a group. The root group has the name <code>'/'</code> .
<b>Examples</b>	<p>This example opens the NetCDF sample file and gets the names of groups in the dataset.</p> <pre>ncid = netcdf.open('example.nc', 'nowrite'); name = netcdf.inqGrpName(ncid); netcdf.close(ncid);</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_inq_grpname</code> function in the NetCDF library C API.</p> <p>For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqGrpNameFull</code>

# netcdf.inqGrpNameFull

---

<b>Purpose</b>	Complete pathname of group
<b>Syntax</b>	<code>groupName = netcdf.inqGrpNameFull(ncid)</code>
<b>Description</b>	<code>groupName = netcdf.inqGrpNameFull(ncid)</code> returns the complete pathname of the group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>groupName</b> Text string specifying complete path of group.  The root group has the name <code>'/'</code> . The names of parent groups and child groups use the forward slash <code>'/'</code> separator, as in UNIX folder names, for example, <code>/group1/subgrp2/subsubgrp3</code> .
<b>Examples</b>	Open the NetCDF sample dataset and retrieve the names of all groups.  <pre>ncid = netcdf.open('example.nc', 'NOWRITE'); gid = netcdf.inqNcid(ncid, 'grid2'); fullName = netcdf.inqGrpNameFull(gid); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_grpname_full</code> function in the netCDF library C API.  For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqGrpName</code>

<b>Purpose</b>	Retrieve ID of parent group.
<b>Syntax</b>	<code>parentGroupID = netcdf.inqGrpParent(ncid)</code>
<b>Description</b>	<code>parentGroupID = netcdf.inqGrpParent(ncid)</code> returns the ID of the parent group given the location of the child group, specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>parentGroupID</b> Identifier of the NetCDF group or file that is the parent of the specified file or group.
<b>Examples</b>	<p>This example opens the NetCDF sample file and gets the full path of the parent of the specified group.</p> <pre>ncid = netcdf.open('example.nc', 'NOWRITE'); gid = netcdf.inqNcid(ncid, 'grid2'); parentId = netcdf.inqGrpParent(gid); fullName = netcdf.inqGrpNameFull(parentId); netcdf.close(ncid);</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_inq_grp_parent</code> function in the NetCDF library C API.</p> <p>For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqGrps</code>

# netcdf.inqGrps

---

<b>Purpose</b>	Retrieve array of child group IDs
<b>Syntax</b>	<code>childGrps = netcdf.inqGrps(ncid)</code>
<b>Description</b>	<code>childGrps = netcdf.inqGrps(ncid)</code> returns all the child group IDs in the parent group, specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>childGrps</b> Array containing identifiers of child groups in the specified NetCDF file or group.
<b>Examples</b>	This example opens the sample NetCDF file and then gets information about the groups it contains.  <pre>ncid = netcdf.open('example.nc','nowrite'); childGroups = netcdf.inqGrps(ncid); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_grps</code> function in the netCDF library C API.  For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqNcid</code>

<b>Purpose</b>	Return ID of named group
<b>Syntax</b>	<code>childGroupId = netcdf.inqNcid(ncid,childGroupName)</code>
<b>Description</b>	<code>childGroupId = netcdf.inqNcid(ncid,childGroupName)</code> returns the ID of the child group, specified by the name <code>childGroupName</code> , in the file or group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> . <b>childGroupName</b> Text string specifying the name of a NetCDF group.
<b>Output Arguments</b>	<b>childGroupID</b> Identifier of a NetCDF group.
<b>Examples</b>	This example opens the sample NetCDF dataset and then gets the ID of a group in the dataset. <pre>ncid = netcdf.open('example.nc','nowrite'); gid = netcdf.inqNcid(ncid,'grid1'); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_ncid</code> function in the netCDF library C API. Read the files <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> for more information.
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqGrpName</code>   <code>netcdf.inqGrpNameFull</code>

# netcdf.inqUnlimDims

---

<b>Purpose</b>	Return list of unlimited dimensions in group
<b>Syntax</b>	<code>unlimdimIDs = netcdf.inqUnlimDims(ncid)</code>
<b>Description</b>	<code>unlimdimIDs = netcdf.inqUnlimDims(ncid)</code> returns the IDs of all unlimited dimensions in the group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>unlimDimIDs</b> An array containing the identifiers of each unlimited dimension. <code>unlimDimIDs</code> is empty if there are no unlimited dimensions.
<b>Examples</b>	<p>This example opens the NetCDF sample dataset and gets the IDs of all the unlimited dimensions.</p> <pre>ncid = netcdf.open('example.nc', 'NOWRITE'); dimids = netcdf.inqUnlimDims(ncid)  dimids =       []  netcdf.close(ncid);</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_inq_unlim_dims</code> function in the NetCDF library C API.</p> <p>For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexncccopyright.txt</code> files.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.defDim</code>   <code>netcdf.inqDim</code>   <code>netcdf.inqDimID</code>   <code>netcdf.renameDim</code>   <code>netcdf.inqDimIDs</code>



<b>Purpose</b>	IDs of all variables in group
<b>Syntax</b>	<code>varids = netcdf.inqVarIDs(ncid)</code>
<b>Description</b>	<code>varids = netcdf.inqVarIDs(ncid)</code> returns IDs of the all the variables in the group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .
<b>Output Arguments</b>	<b>varids</b> Array containing identifiers of variables in a NetCDF file or group.
<b>Examples</b>	<p>This example opens the NetCDF sample file and gets the IDs of all the variables in a group.</p> <pre>ncid = netcdf.open('example.nc', 'NOWRITE'); gid = netcdf.inqNcid(ncid, 'grid1'); varids = netcdf.inqVarIDs(gid); netcdf.close(ncid);</pre>
<b>References</b>	<p>This function corresponds to the <code>nc_inq_varids</code> function in the NetCDF library C API.</p> <p>For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> files.</p>
<b>See Also</b>	<code>netcdf</code>   <code>netcdf.inqDimIDs</code>   <code>netcdf.inqVarID</code>

# netcdf.inqVarChunking

---

<b>Purpose</b>	Determine chunking settings for NetCDF variable
<b>Syntax</b>	<code>[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)</code>
<b>Description</b>	<code>[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)</code> returns the type of chunking and the dimensions of a chunk for the NetCDF variable specified by <code>varid</code> , in the file or group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> .  <b>varid</b> Identifier of NetCDF variable, returned by <code>netcdf.defVar</code> .
<b>Output Arguments</b>	<b>storage</b> Text string specifying if NetCDF breaks the data into chunks when writing to a file. <code>CHUNKED</code> indicates the data is chunked; <code>CONTIGUOUS</code> indicates that the data is not chunked.  <b>chunkSizes</b> Array specifying the dimensions of the chunk.  Because MATLAB uses FORTRAN-style ordering, the order of dimensions in <code>chunkdims</code> is reversed relative to what would be in the NetCDF C API.  If the storage type specified is <code>CONTIGUOUS</code> , <code>netcdf.inqVarChunking</code> returns an empty array, <code>[]</code> .
<b>Examples</b>	This example opens the NetCDF sample dataset and gets the values of chunking parameters associated with a variable.  <pre>ncid = netcdf.open('example.nc','NOWRITE');</pre>

```
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 `mexncopyright.txt` files.

## See Also

`netcdf` | `netcdf.defVar` | `netcdf.defVarChunking`

# netcdf.inqVarDeflate

---

<b>Purpose</b>	Determine compression settings for NetCDF variable
<b>Syntax</b>	<code>[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid, varid)</code>
<b>Description</b>	<code>[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)</code> returns the compression parameters for the NetCDF variable specified by <code>varid</code> in the location specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or of a NetCDF group, returned by <code>netcdf.defGrp</code> . <b>varid</b> Identifier of NetCDF variable, returned by <code>netcdf.defVar</code> .
<b>Output Arguments</b>	<b>shuffle</b> Boolean value. <code>true</code> 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. <b>deflate</b> Boolean value. <code>true</code> indicates that compression is enabled for this variable. The <code>deflateLevel</code> argument specifies the level of compression. <b>deflateLevel</b> Scalar value between 0 and 9 specifying the amount of compression, where 0 is no compression and 9 is the most compression
<b>Examples</b>	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

---

<b>Purpose</b>	Determine values of fill parameters for NetCDF variable
<b>Syntax</b>	<code>[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)</code>
<b>Description</b>	<code>[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)</code> returns the fill mode and the fill value for the variable <code>varid</code> in the file or group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier of a NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or a NetCDF group, returned by <code>netcdf.defGrp</code> .  <b>varid</b> Identifier of NetCDF variable.
<b>Output Arguments</b>	<b>noFillMode</b> Boolean value. <code>true</code> indicates that use of the fill values for the variable has been disabled.  <b>fillValue</b> Specifies the value to use in the variable when no other value is specified and use of fill values has been enabled.
<b>Examples</b>	This example opens the NetCDF sample dataset and gets the fill mode and fill value used with a variable.  <pre>ncid = netcdf.open('example.nc','NOWRITE'); varid = netcdf.inqVarID(ncid,'temperature'); [noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_var_fill</code> 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

---

<b>Purpose</b>	Fletcher32 checksum setting for NetCDF variable
<b>Syntax</b>	<code>setting = netcdf.inqVarFletcher32(ncid,varid)</code>
<b>Description</b>	<code>setting = netcdf.inqVarFletcher32(ncid,varid)</code> returns the Fletcher32 checksum setting for the NetCDF variable specified by <code>varid</code> in the file or group specified by <code>ncid</code> .
<b>Input Arguments</b>	<b>ncid</b> Identifier for NetCDF file, returned by <code>netcdf.create</code> or <code>netcdf.open</code> , or group, returned by <code>netcdf.defGrp</code> .  <b>varid</b> Identifier of NetCDF variable.
<b>Output Arguments</b>	<b>setting</b> Text string specifying whether the Fletcher32 checksum is turned on for the specified variable. <code>netcdf.inqVarFletcher32</code> returns the text string <code>FLETCHER32</code> if the checksum is turned on for the variable; otherwise, <code>NOCHECKSUM</code> .
<b>Examples</b>	This example opens the sample NetCDF file and gets information about the checksum setting for a variable.  <pre>ncid = netcdf.open('example.nc', 'NOWRITE'); varid = netcdf.inqVarID(ncid, 'temperature'); setting = netcdf.inqVarFletcher32(ncid, varid); netcdf.close(ncid);</pre>
<b>References</b>	This function corresponds to the <code>nc_inq_var_fletcher32</code> function in the netCDF library C API.  For copyright information, read the <code>netcdfcopyright.txt</code> and <code>mexnccopyright.txt</code> 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)
```

# netcdf.inqAttName

---

gattname =

creation\_date

## See Also

netcdf.inqAtt | netcdf.inqAttID

<b>Purpose</b>	Return netCDF dimension name and length
<b>Syntax</b>	<code>[dimname, dimlen] = netcdf.inqDim(ncid,dimid)</code>
<b>Description</b>	<p><code>[dimname, dimlen] = netcdf.inqDim(ncid,dimid)</code> returns the name, <code>dimname</code>, and length, <code>dimlen</code>, of the dimension specified by <code>dimid</code>. If <code>ndims</code> is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and <code>ndims-1</code>. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.</p> <p><code>ncid</code> is a netCDF file identifier returned by <code>netcdf.create</code> or <code>netcdf.open</code>.</p> <p>This function corresponds to the <code>nc_inq_dim</code> function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>
<b>Examples</b>	<p>The example opens the example netCDF file include with MATLAB, <code>example.nc</code>.</p> <pre>ncid = netcdf.open('example.nc','NC_NOWRITE');  % Get name and length of first dimension [dimname, dimlen] = netcdf.inqDim(ncid,0)  dimname =  x  dimlen =      50</pre>
<b>See Also</b>	<code>netcdf.inqDimID</code>

# 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`



**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);
```

# netcdf.open

---

## 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);
```

## netcdf.putAtt

---

```
% 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** This example creates a new netCDF file and writes a variable to the file.

```
% Create a 50 element vector for a variable.
my_vardata = linspace(0,50,50);

% Open netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE')

% Define the dimensions of the variable.
```

## netcdf.putVar

---

```
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
```

### See Also

`netcdf.getVar`

<b>Purpose</b>	Put open netCDF file into define mode
<b>Syntax</b>	<code>netcdf.reDef(ncid)</code>
<b>Description</b>	<p><code>netcdf.reDef(ncid)</code> 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. <code>ncid</code> is a valid NetCDF file ID, returned from a previous call to <code>netcdf.open</code> or <code>netcdf.create</code>.</p> <p>This function corresponds to the <code>nc_redef</code> function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See <code>netcdf</code> for more information.</p>
<b>Examples</b>	<p>This example opens a local copy of the example netCDF file included with MATLAB, <code>example.nc</code>.</p> <pre>% 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 ==&gt; netcdflib NetCDF: Operation not allowed in data mode  Error in ==&gt; 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);</pre>
<b>See Also</b>	<code>netcdf.create</code>   <code>netcdf.endDef</code>   <code>netcdf.open</code>

# netcdf.renameAtt

---

**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 identifier 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`

**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
```

# netcdf.renameVar

---

varname =

Avagadros\_number

## **See Also**

`netCDF.defVar` | `netCDF.inqVar` | `netCDF.putVar`



<b>Purpose</b>	Set default chunk cache settings for NetCDF library
<b>Syntax</b>	<code>netcdf.setChunkCache(csize, nelems, premp)</code>
<b>Description</b>	<p><code>netcdf.setChunkCache(csize, nelems, premp)</code> sets the default chunk cache settings used by the NetCDF library.</p> <p>Settings apply for subsequent file open or create operations, for the remainder of the MATLAB session or until you issue a <code>clear mex</code> call. This function does not change the chunk cache settings of files already open.</p>
<b>Input Arguments</b>	<p><b>csize</b></p> <p>Scalar double specifying the total size of the raw data chunk cache in bytes.</p> <p><b>nelems</b></p> <p>Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.</p> <p><b>premp</b></p> <p>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.</p>
<b>Examples</b>	<p>This example sets the cache chunk size used by the NetCDF library.</p> <pre>netcdf.setChunkCache(32000000, 2003, .75)</pre>
<b>References</b>	This function corresponds to the <code>nc_set_chunk_cache</code> function in the NetCDF library C API.

# netcdf.setChunkCache

---

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

## **See Also**

`netcdf` | `netcdf.getChunkCache`

**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`

# netcdf.setFill

---

**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`

**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`

# newplot

---

**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.

<b>NextPlot</b>	<b>What Happens</b>
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 <code>HandleVisibility</code> property is set to <code>on</code> and reset figure <code>NextPlot</code> property to <code>add</code> .  This clears the current figure and is equivalent to issuing the <code>clf</code> command.
replace	Remove all child objects (regardless of the setting of the <code>HandleVisibility</code> property) and reset figure properties to their defaults, except <code>NextPlot</code> is reset to <code>add</code> regardless of user-defined defaults.  <ul style="list-style-type: none"> <li>• <code>Position</code>, <code>Units</code>, <code>PaperPosition</code>, and <code>PaperUnits</code> are not reset.</li> </ul> This clears and resets the current figure and is equivalent to issuing the <code>clf reset</code> command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

<b>NextPlot</b>	<b>Description</b>
<code>add</code>	Draw into the current axes, retaining all graphics objects already present.
<code>replacechildren</code>	Remove all child objects whose <code>HandleVisibility</code> property is set to <code>on</code> , but do not reset axes properties. This clears the current axes like the <code>cla</code> command.
<code>replace</code>	Remove all child objects (regardless of the setting of the <code>HandleVisibility</code> property) and reset axes properties to their defaults, except <code>Position</code> and <code>Units</code> .  This clears and resets the current axes like the <code>cla reset</code> command.

## See Also

`axes` | `cla` | `clf` | `figure` | `hold` | `ishold` | `reset` | `figure` | `axes`

## How To

- Controlling Graphics Output



<b>Purpose</b>	Make next IFD current IFD
<b>Syntax</b>	<code>tiffobj.nextDirectory()</code>
<b>Description</b>	<code>tiffobj.nextDirectory()</code> makes the next image file directory (IFD) in the file the current IFD. <code>Tiff</code> object methods operate on the current IFD. Use this method to navigate among IFDs in a TIFF file containing multiple images.
<b>Examples</b>	<p>Open a <code>Tiff</code> object and change the current IFD to the next IFD in the file. Replace <code>myfile.tif</code> with the name of a TIFF file on your MATLAB path. The TIFF file should contain multiple images.</p> <pre>t = Tiff('myfile.tif', 'r'); t.nextDirectory();</pre>
<b>References</b>	This method corresponds to the <code>TIFFReadDirectory</code> 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 <a href="#">LibTIFF - TIFF Library and Utilities</a> .
<b>See Also</b>	<code>Tiff.setDirectory</code>
<b>Tutorials</b>	<ul style="list-style-type: none"><li>• “Exporting Image Data and Metadata to TIFF Files”</li><li>• “Reading Image Data and Metadata from TIFF Files”</li></ul>

# nextpow2

---

**Purpose** Next higher power of 2

**Syntax** `p = nextpow2(A)`

**Description** `p = nextpow2(A)` returns the smallest power of two that is greater than or equal to the absolute value of A. (That is, p that satisfies  $2^p \geq \text{abs}(A)$ ).

This function is useful for optimizing FFT operations, which are most efficient when sequence length is an exact power of two.

**Examples** For any integer n in the range from 513 to 1024, `nextpow2(n)` is 10.

For vector input, `nextpow2(n)` returns an element-by-element result:

```
A = [1 2 3 4 5 9 519]
nextpow2(A)
ans =
```

```
0    1    2    2    3    4   10
```

**See Also** `fft` | `log2` | `pow2`

---

<b>Purpose</b>	Number of nonzero matrix elements
<b>Syntax</b>	<code>n = nnz(X)</code>
<b>Description</b>	<code>n = nnz(X)</code> returns the number of nonzero elements in matrix <code>X</code> . The density of a sparse matrix is <code>nnz(X)/prod(size(X))</code> .
<b>Examples</b>	The matrix  <code>w = sparse(wilkinson(21));</code>  is a tridiagonal matrix with 20 nonzeros on each of three diagonals, so <code>nnz(w) = 60</code> .
<b>See Also</b>	<code>find</code>   <code>isa</code>   <code>nonzeros</code>   <code>nzmax</code>   <code>size</code>   <code>whos</code>

# 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`

# norm

---

## 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  $\text{sum}(\text{abs}(v).^p)^{1/p}$ .

`n = norm(v,Inf)` returns the largest element of  $\text{abs}(v)$ .

`n = norm(v,-Inf)` returns the smallest element of  $\text{abs}(v)$ .

## 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`

---

<b>Purpose</b>	2-norm estimate
<b>Syntax</b>	<pre>nrm = normest(S) nrm = normest(S,tol) [nrm,count] = normest(...)</pre>
<b>Description</b>	<p>This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.</p> <p><code>nrm = normest(S)</code> returns an estimate of the 2-norm of the matrix <code>S</code>.</p> <p><code>nrm = normest(S,tol)</code> uses relative error <code>tol</code> instead of the default tolerance <code>1.e-6</code>. The value of <code>tol</code> determines when the estimate is considered acceptable.</p> <p><code>[nrm,count] = normest(...)</code> returns an estimate of the 2-norm and also gives the number of power iterations used.</p>
<b>Algorithms</b>	<p>The power iteration involves repeated multiplication by the matrix <code>S</code> and its transpose, <code>S'</code>. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.</p>
<b>See Also</b>	<code>cond</code>   <code>condest</code>   <code>norm</code>   <code>rcond</code>   <code>svd</code>

# not

---

**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”



- 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”
  - “Publish MATLAB Code”

# notify (handle)

---

**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`

**Purpose** Current date and time

**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`

# nthroot

---

**Purpose** Real nth root of real numbers

**Syntax** `y = nthroot(X, n)`

**Description** `y = nthroot(X, n)` returns the real nth 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**

$Z = \text{null}(A)$  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  $Z' * Z = I$ .

$Z = \text{null}(A, 'r')$  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 =
```

# null

---

```
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](#)

**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)`.

# num2cell

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

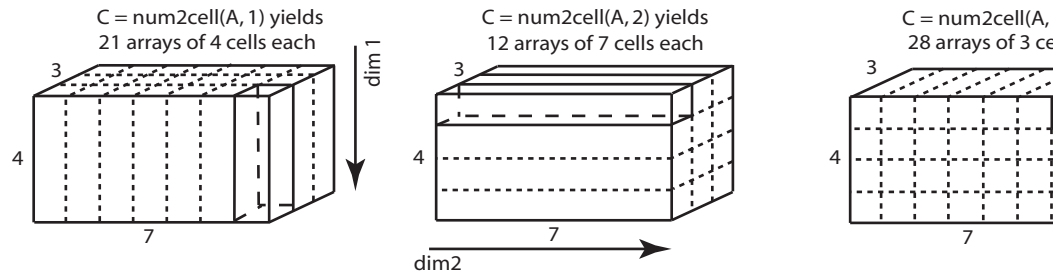
### C - Resulting array

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.

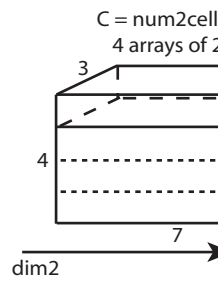
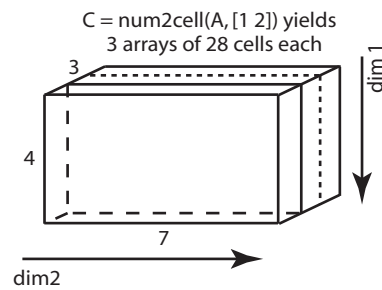
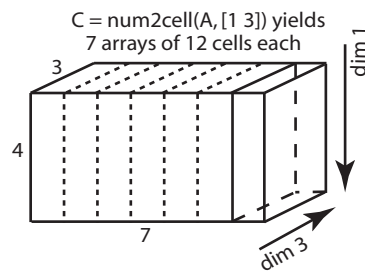
For example, given a 4-by-7-by-3 array, `A`, this figure shows how `num2cell` creates cells corresponding to `dim` values of 1, 2, and 3.



- If `dim` is a vector containing `N` values, then `C` has `numel(A)/prod([size(A,dim(1)), ..., size(A, dim(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 a 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
```

# num2cell

---

```
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:

## num2cell

---

```
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
```

# num2cell

---

```
30  
40
```

```
c = num2cell(A,[2 3])
```

```
c =
```

```
[1x3x2 double]  
[1x3x2 double]  
[1x3x2 double]  
[1x3x2 double]
```

## See Also

```
cat | mat2cell | cell2mat
```

**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
3fb9999999999999a
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

# num2str

---

**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`. When `A` contains floating-point numbers, the output format depends upon the magnitude of the original values and sometimes includes exponents.

`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

**Complex Number Support:** Yes

### **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

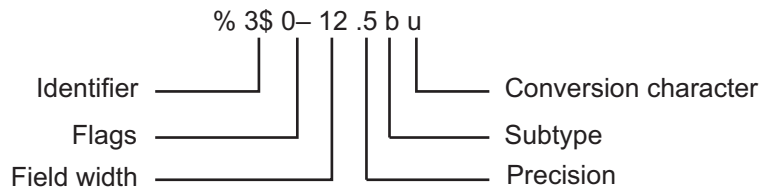
Format of the output fields, specified as a string.

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<br>%bo<br>%bu | Double-precision hexadecimal, octal, or decimal value<br>Example: %bx prints pi as 400921fb54442d18 |

| Value Type | Conversion               | Details                                                                                     |
|------------|--------------------------|---------------------------------------------------------------------------------------------|
|            | %tx or %tX<br>%to<br>%tu | Single-precision hexadecimal, octal, or decimal value<br>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.
- Escape characters, including:

' ' Single quotation mark

%% Percent character

|                  |                              |
|------------------|------------------------------|
| <code>\\</code>  | Backslash                    |
| <code>\a</code>  | Alarm                        |
| <code>\b</code>  | Backspace                    |
| <code>\f</code>  | Form feed                    |
| <code>\n</code>  | New line                     |
| <code>\r</code>  | Carriage return              |
| <code>\t</code>  | Horizontal tab               |
| <code>\v</code>  | Vertical tab                 |
| <code>\xN</code> | Hexadecimal number, <i>N</i> |
| <code>\N</code>  | Octal number, <i>N</i>       |

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.

## 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'.

## Tips

- num2str is useful for labeling and titling plots with numeric values.

## 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
```

## num2str

---

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`

|                    |                                                                                                                                                                                                                                                                                                           |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Total number of strips in image                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>      | <code>numStrips = tiffobj.numberOfStrips()</code>                                                                                                                                                                                                                                                         |
| <b>Description</b> | <code>numStrips = tiffobj.numberOfStrips()</code> returns the total number of strips in the image.                                                                                                                                                                                                        |
| <b>Examples</b>    | <p>Open a Tiff object and determine the number of strips in the image. Replace <code>myfile.tif</code> with the name of a TIFF file on your MATLAB path:</p> <pre>t = Tiff('myfile.tif', 'r'); % % Check if image has a stripped organization if ~t.isTiled()     nStrips = t.numberOfStrips(); end</pre> |
| <b>References</b>  | This method corresponds to the <code>TIFFNumberOfStrips</code> 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 <a href="#">LibTIFF - TIFF Library and Utilities</a> .                                 |
| <b>See Also</b>    | <code>Tiff.numberOfTiles</code>   <code>Tiff.isTiled</code>                                                                                                                                                                                                                                               |
| <b>Tutorials</b>   | <ul style="list-style-type: none"><li>• “Exporting Image Data and Metadata to TIFF Files”</li><li>• “Reading Image Data and Metadata from TIFF Files”</li></ul>                                                                                                                                           |

# Tiff.numberOfTiles

---

**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”



**Purpose** Number of elements in array or subscripted array expression

**Syntax**  
`n = numel(A)`  
`n = numel(A, index1, index2, ... indexn)`

**Description** `n = numel(A)` returns the number of elements, `n`, in array `A`.  
`n = numel(A, index1, index2, ... indexn)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`. To handle the variable number of arguments, `numel` is typically written with the header function `n = numel(A, varargin)`, where `varargin` is a cell array with elements `index1, index2, ... indexn`.

The MATLAB software implicitly calls the `numel` built-in function whenever an expression generates a comma-separated list. This includes brace indexing (i.e., `A{index1, index2, ..., indexN}`), and dot indexing (i.e., `A.fieldname`).

**Tips** It is important to note the significance of `numel` with regards to the overloaded `subsref` and `subsasgn` functions. In the case of the overloaded `subsref` function for brace and dot indexing (as described in the last paragraph), `numel` is used to compute the number of expected outputs (`nargout`) returned from `subsref`. For the overloaded `subsasgn` function, `numel` is used to compute the number of expected inputs (`nargin`) to be assigned using `subsasgn`. The `nargin` value for the overloaded `subsasgn` function is the value returned by `numel` plus 2 (one for the variable being assigned to, and one for the structure array of subscripts).

As a class designer, you must ensure that the value of `n` returned by the built-in `numel` function is consistent with the class design for that object. If `n` is different from either the `nargout` for the overloaded `subsref` function or the `nargin` for the overloaded `subsasgn` function, then you need to overload `numel` to return a value of `n` that is consistent with the class' `subsref` and `subsasgn` functions. Otherwise, MATLAB produces errors when calling these functions.

# numel

---

## Examples

Create a 4-by-4-by-2 matrix. `numel` counts 32 elements in the 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(a)
ans =
    32
```

## See Also

[margin](#) | [nargout](#) | [prod](#) | [size](#) | [subsasgn](#) | [subsref](#)

**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`

# ode15i

---

**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`.

|                      |                                                                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code>  | A function handle that evaluates the left side of the differential equations, which are of the form $f(t,y,y') = 0$ .                                                               |
| <code>tspan</code>   | A vector specifying the interval of integration, $[t_0, t_f]$ . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> . |
| <code>y0, yp0</code> | Vectors of initial conditions for $y$ and $y'$ respectively.                                                                                                                        |
| <code>options</code> | Optional integration argument created using the <code>odeset</code> function. See <code>odeset</code> for details.                                                                  |

The following table lists the output arguments for `ode15i`.

|                |                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------|
| <code>T</code> | Column vector of time points                                                                                     |
| <code>Y</code> | 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  $t_0$  to  $t_f$  with initial conditions  $y_0$  and  $yp_0$ . `odefun` is a function handle. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that  $f(t_0,y_0,yp_0) = 0$ . 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  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use  $tspan = [t_0, t_1, \dots, t_f]$ .

“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:

|                    |                                                                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.x</code> | Steps chosen by the solver. If you specify the <code>Events</code> option and a terminal event is detected, <code>sol.x(end)</code> contains the end of the step at which the event occurred. |
| <code>sol.y</code> | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> .                                                                                                          |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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   | <code>RelTol</code> , <code>AbsTol</code> , <code>NormControl</code>                       |
| Solver output   | <code>OutputFcn</code> , <code>OutputSel</code> , <code>Refine</code> , <code>Stats</code> |
| Event location  | <code>Events</code>                                                                        |
| Step size       | <code>MaxStep</code> , <code>InitialStep</code>                                            |
| Jacobian matrix | <code>Jacobian</code> , <code>JPattern</code> , <code>Vectorized</code>                    |

## 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`, and `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

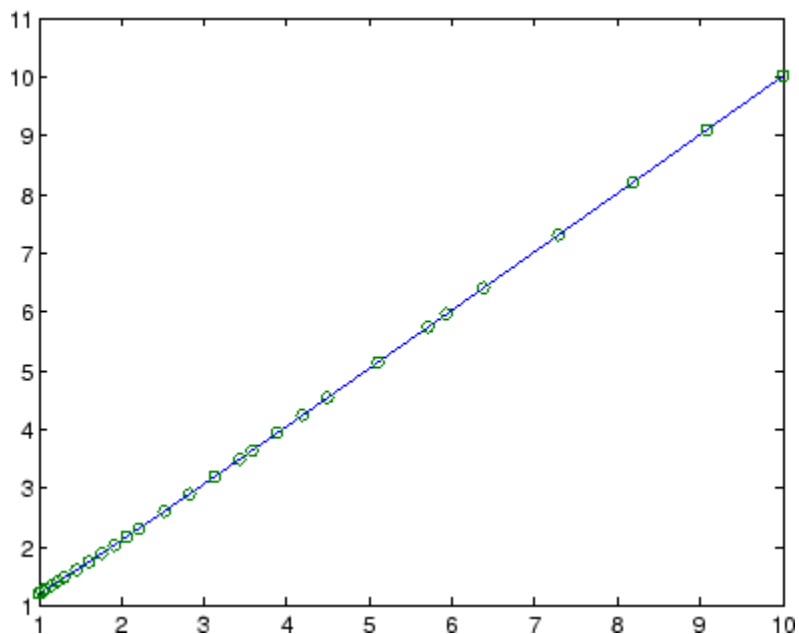
This example uses a helper function `decic` to hold fixed the initial value for  $y(t_0)$  and compute a consistent initial value for  $y'(t_0)$  for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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.

|                 |                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>T</code>  | Column vector of time points.                                                                                                          |
| <code>Y</code>  | Solution array. Each row in <code>Y</code> corresponds to the solution at a time returned in the corresponding row of <code>T</code> . |
| <code>TE</code> | The time at which an event occurs.                                                                                                     |
| <code>YE</code> | The solution at the time of the event.                                                                                                 |

|     |                                                         |
|-----|---------------------------------------------------------|
| IE  | The index <i>i</i> of the event function that vanishes. |
| sol | Structure to evaluate the solution.                     |

## Description

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ tf}]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $\text{tf}$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, \text{tf}$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, \text{tf}]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $yp_0$  as the value of the InitialSlope property. The default is the zero vector. If a problem is a DAE, and  $y_0$  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. 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3190 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,<br>NormControl            | √     | √     | √      | √      | √      | √      | √       |
| OutputFcn,<br>OutputSel,<br>Refine, Stats | √     | √     | √      | √      | √      | √      | √       |
| NonNegative                               | √     | √     | √      | √ *    | —      | √ *    | √ *     |
| Events                                    | √     | √     | √      | √      | √      | √      | √       |
| MaxStep,<br>InitialStep                   | √     | √     | √      | √      | √      | √      | √       |
| Jacobian,<br>JPattern,<br>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

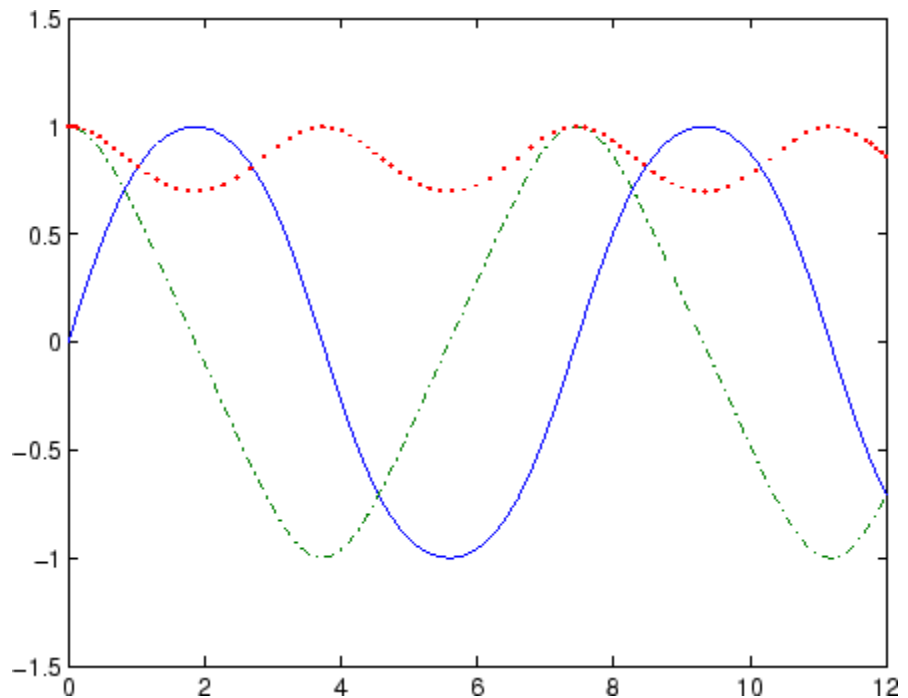
```
function dy = rigid(t,y)
dy = zeros(3,1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```

# ode15s

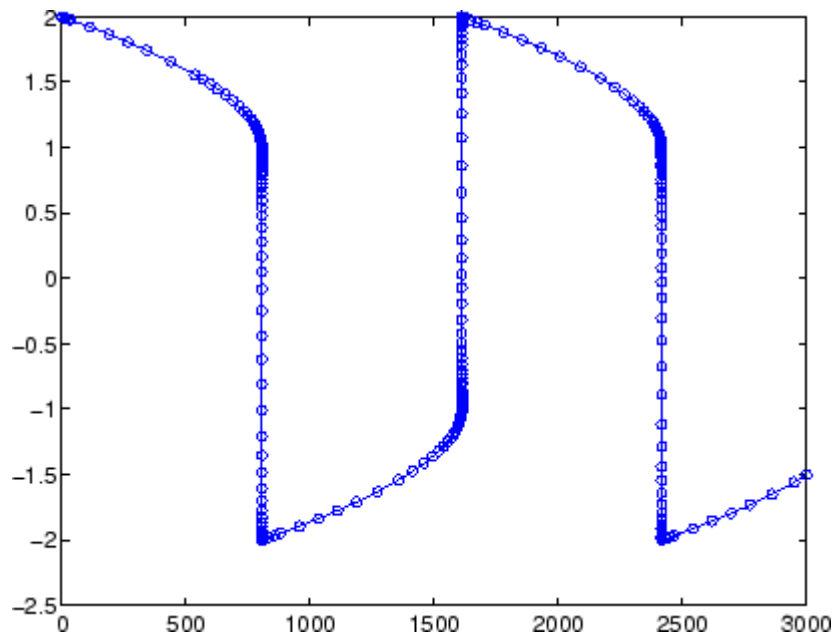
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate 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
```

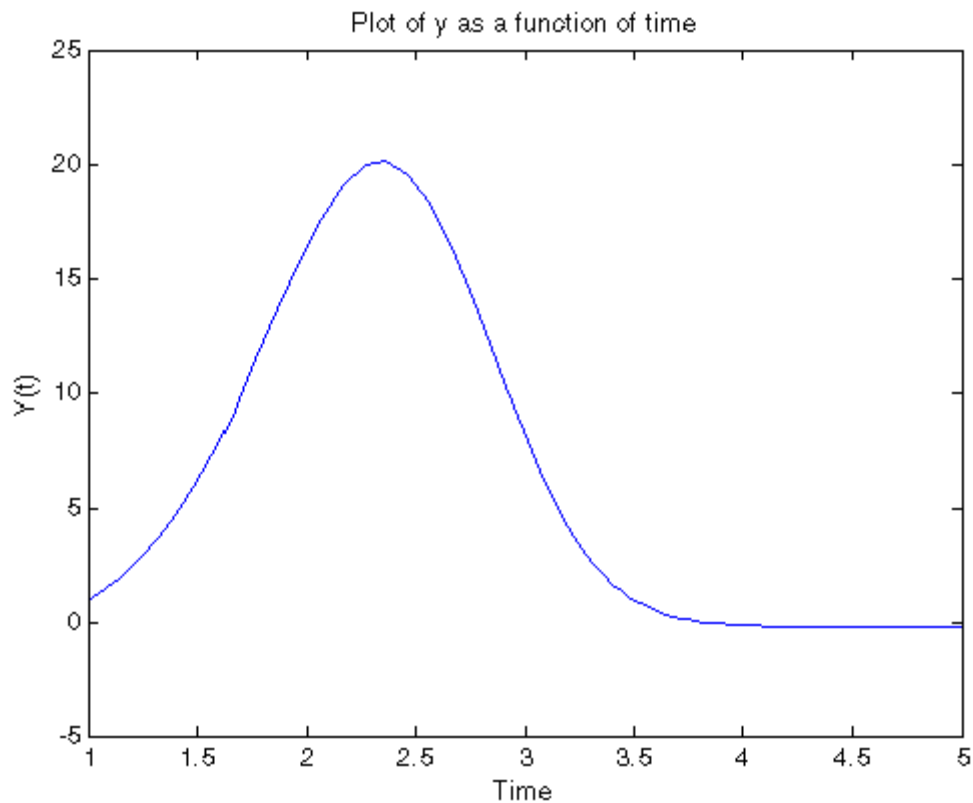
Plot the solution  $y(t)$  as a function of time:

```
plot(T, Y);
```

# ode15s

---

```
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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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>i</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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred.<br><code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                        |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3205 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.

# ode23

| 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

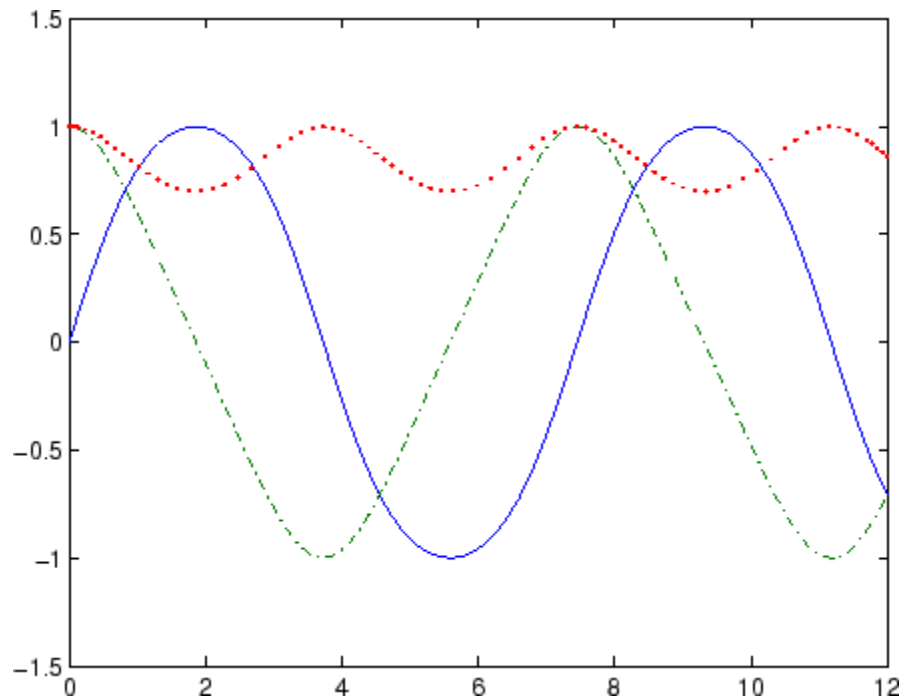
```
function dy = rigid(t,y)
dy = zeros(3,1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```

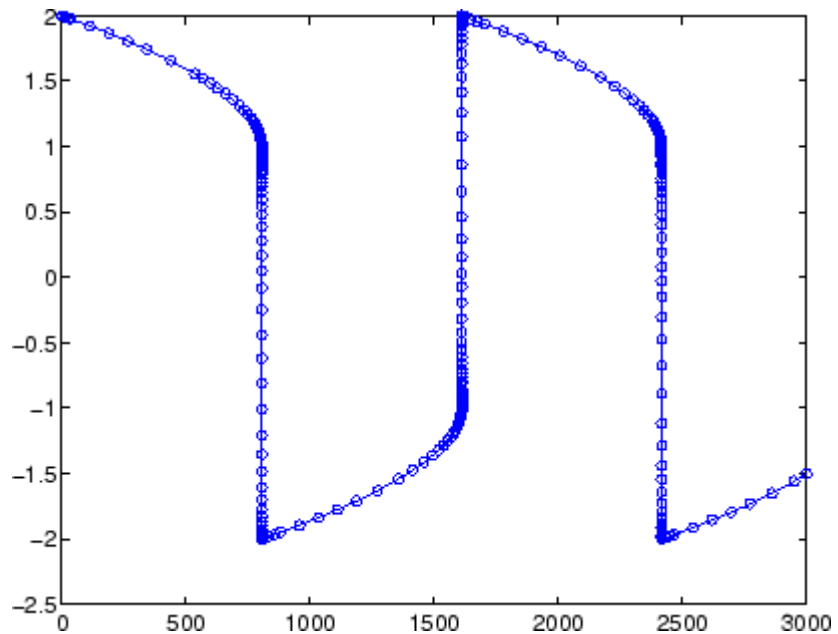
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

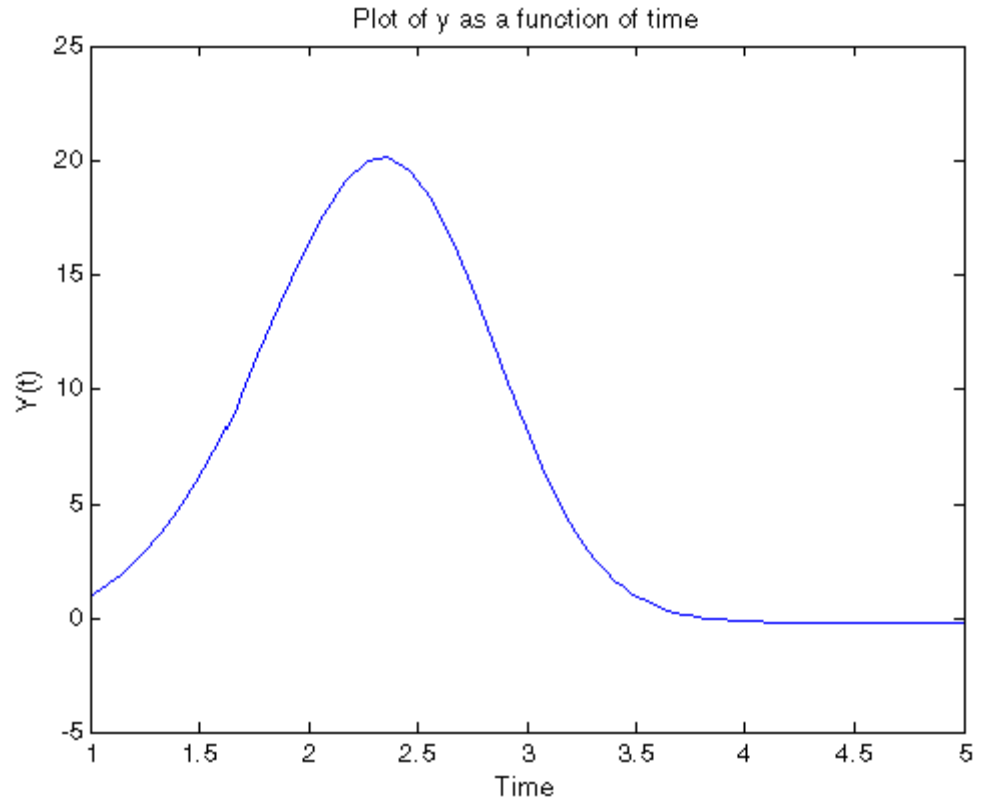
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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.

|                 |                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>T</code>  | Column vector of time points.                                                                                                          |
| <code>Y</code>  | Solution array. Each row in <code>Y</code> corresponds to the solution at a time returned in the corresponding row of <code>T</code> . |
| <code>TE</code> | The time at which an event occurs.                                                                                                     |
| <code>YE</code> | The solution at the time of the event.                                                                                                 |

|     |                                                         |
|-----|---------------------------------------------------------|
| IE  | The index <i>i</i> of the event function that vanishes. |
| sol | Structure to evaluate the solution.                     |

## Description

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ tf}]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $\text{tf}$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, \text{tf}$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, \text{tf}]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. When solving DAEs, it is very advantageous to formulate the problem so that  $M$  is a diagonal matrix (a semi-explicit DAE).

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                 |
|---------------|---------------------|--------------------------|------------------------------------------------------------------------------------|
| 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3220 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,<br>NormControl            | √     | √     | √      | √      | √      | √      | √       |
| OutputFcn,<br>OutputSel,<br>Refine, Stats | √     | √     | √      | √      | √      | √      | √       |
| NonNegative                               | √     | √     | √      | √ *    | —      | √ *    | √ *     |
| Events                                    | √     | √     | √      | √      | √      | √      | √       |
| MaxStep,<br>InitialStep                   | √     | √     | √      | √      | √      | √      | √       |
| Jacobian,<br>JPattern,<br>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);
```

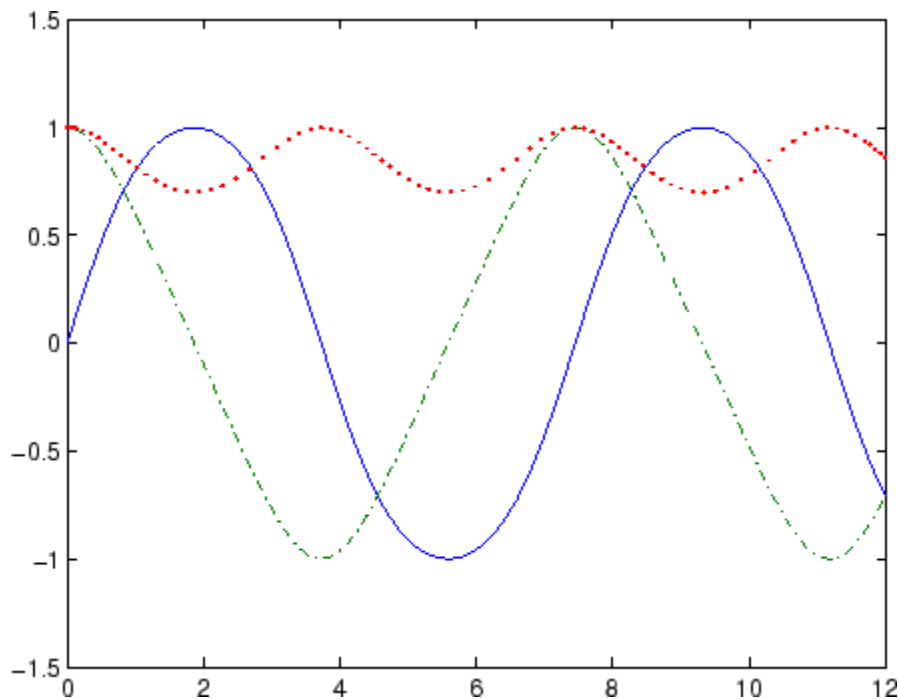
In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time 0.

```
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)
```

# ode23s

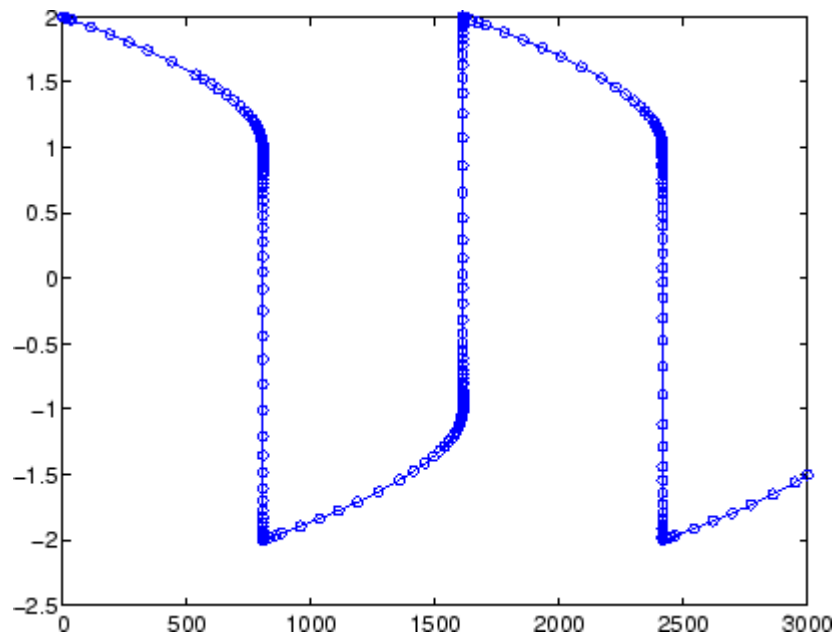
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

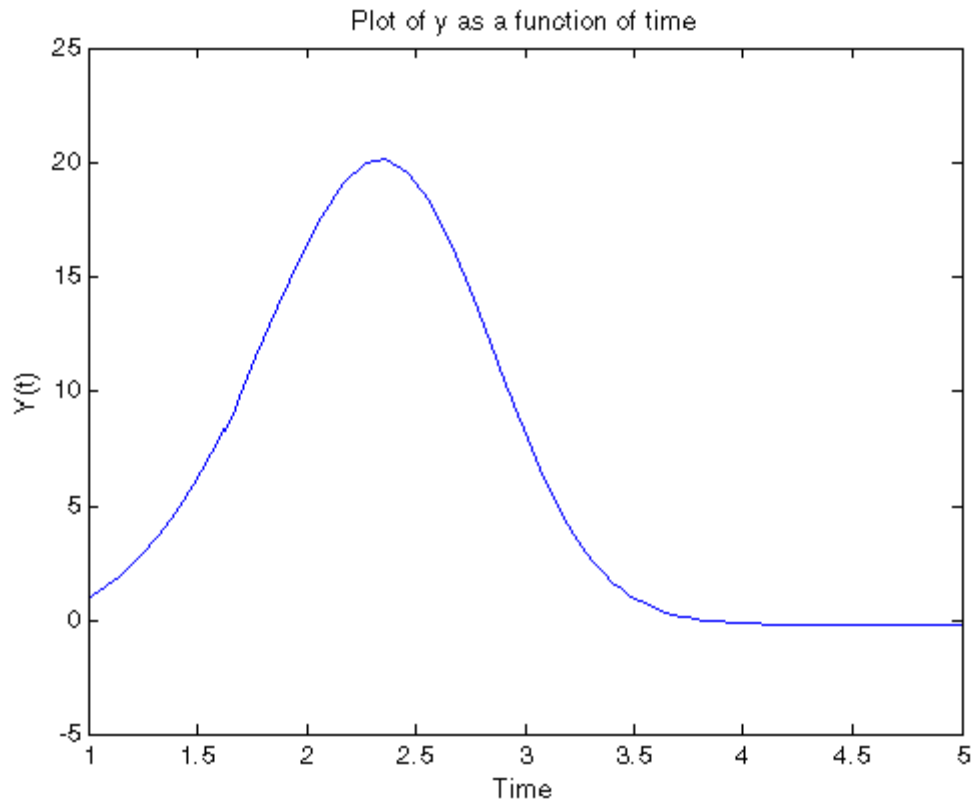
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                        |
| <code>tspan</code>  | A vector specifying the interval of integration, $[t_0, t_f]$ . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements $[t_0 \ t_f]$ , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ } t_f]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $t_f$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, t_f]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3235 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

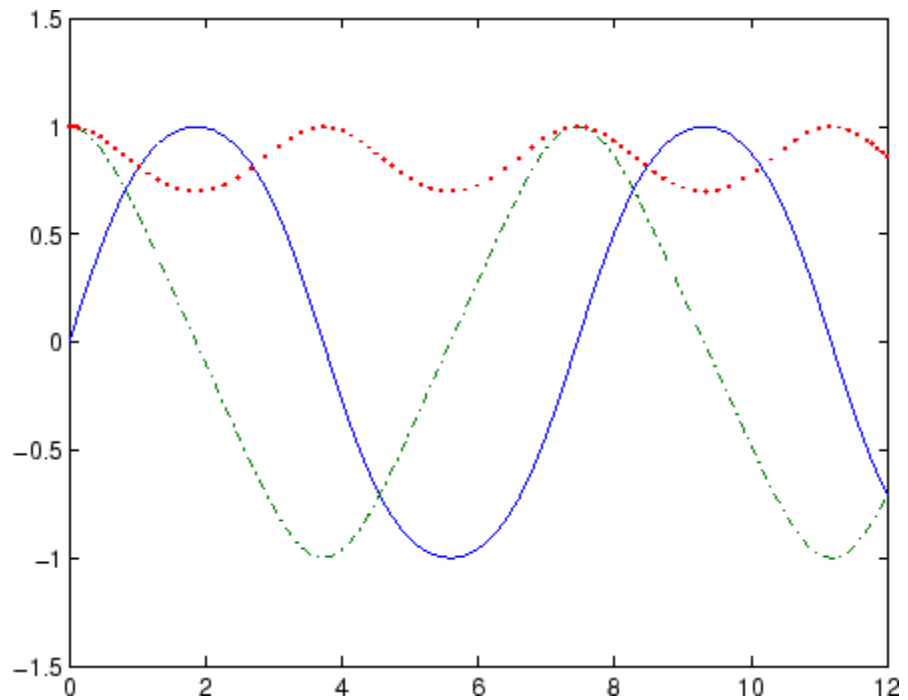
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```



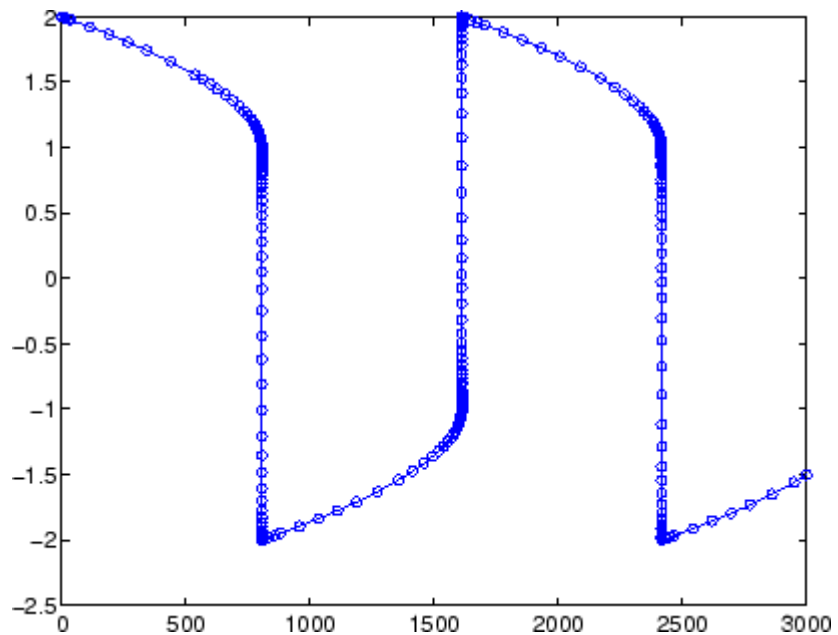
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

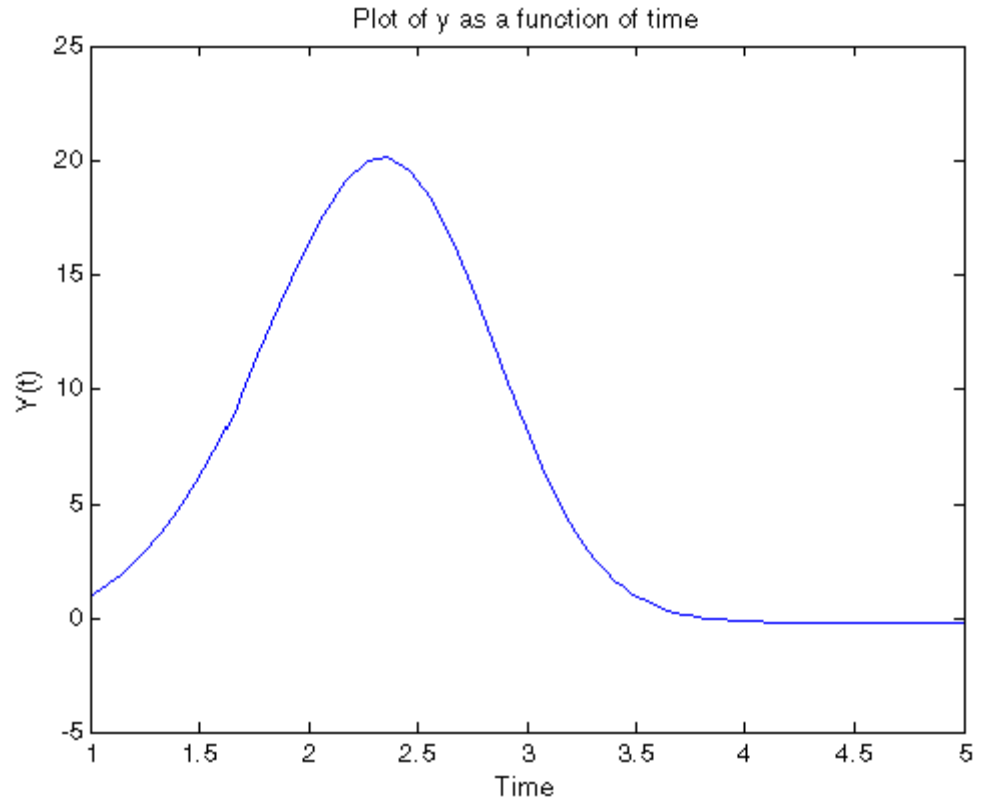
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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.

|                 |                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>T</code>  | Column vector of time points.                                                                                                          |
| <code>Y</code>  | Solution array. Each row in <code>Y</code> corresponds to the solution at a time returned in the corresponding row of <code>T</code> . |
| <code>TE</code> | The time at which an event occurs.                                                                                                     |
| <code>YE</code> | The solution at the time of the event.                                                                                                 |

|     |                                                         |
|-----|---------------------------------------------------------|
| IE  | The index <i>i</i> of the event function that vanishes. |
| sol | Structure to evaluate the solution.                     |

## Description

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ } t_f]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $t_f$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, t_f]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. When solving DAEs, it is very advantageous to formulate the problem so that  $M$  is a diagonal matrix (a semi-explicit DAE).

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                 |
|---------------|---------------------|--------------------------|------------------------------------------------------------------------------------|
| 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3250 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,<br>NormControl            | √     | √     | √      | √      | √      | √      | √       |
| OutputFcn,<br>OutputSel,<br>Refine, Stats | √     | √     | √      | √      | √      | √      | √       |
| NonNegative                               | √     | √     | √      | √ *    | —      | √ *    | √ *     |
| Events                                    | √     | √     | √      | √      | √      | √      | √       |
| MaxStep,<br>InitialStep                   | √     | √     | √      | √      | √      | √      | √       |
| Jacobian,<br>JPattern,<br>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

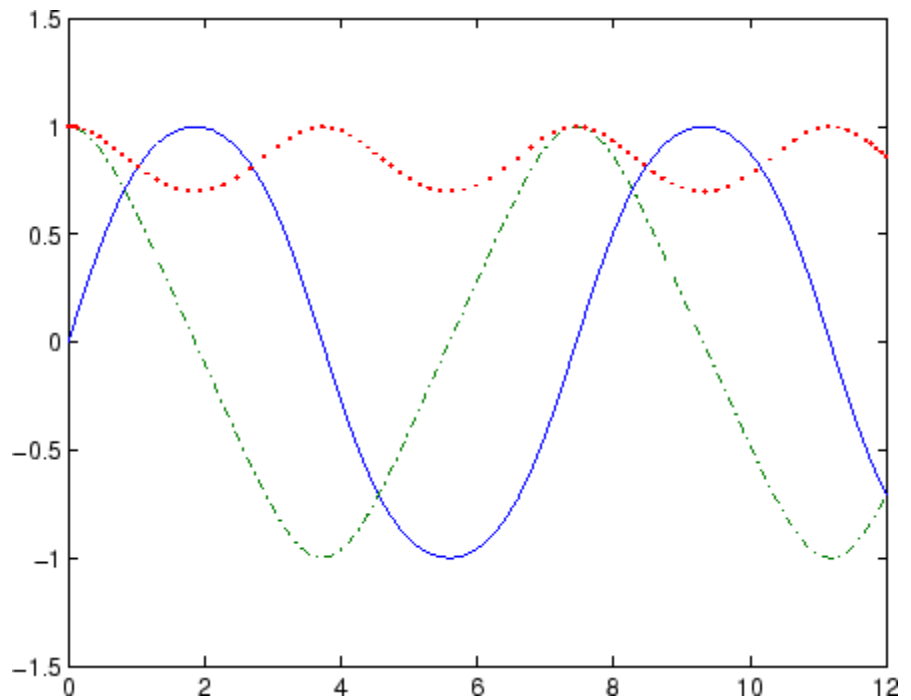
```
function dy = rigid(t,y)
dy = zeros(3,1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```

## ode23tb

---

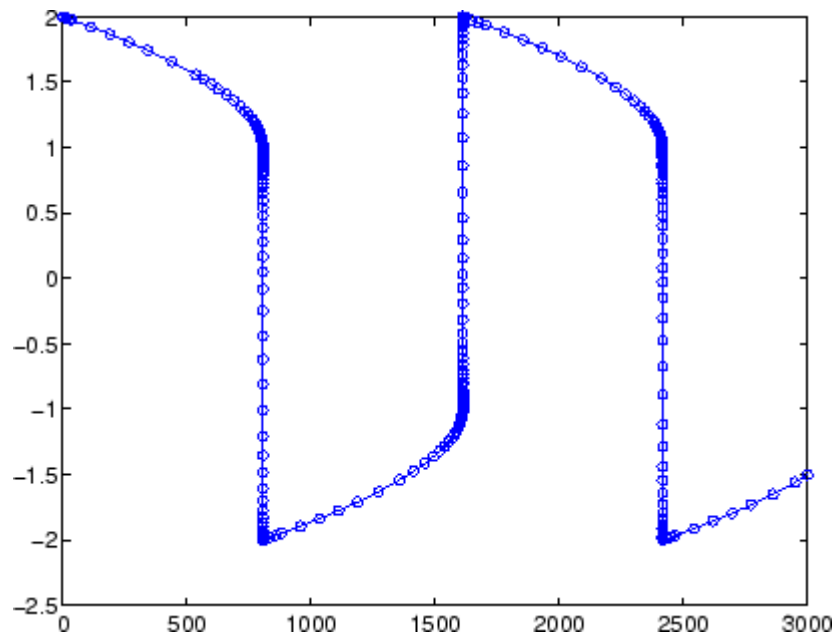
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

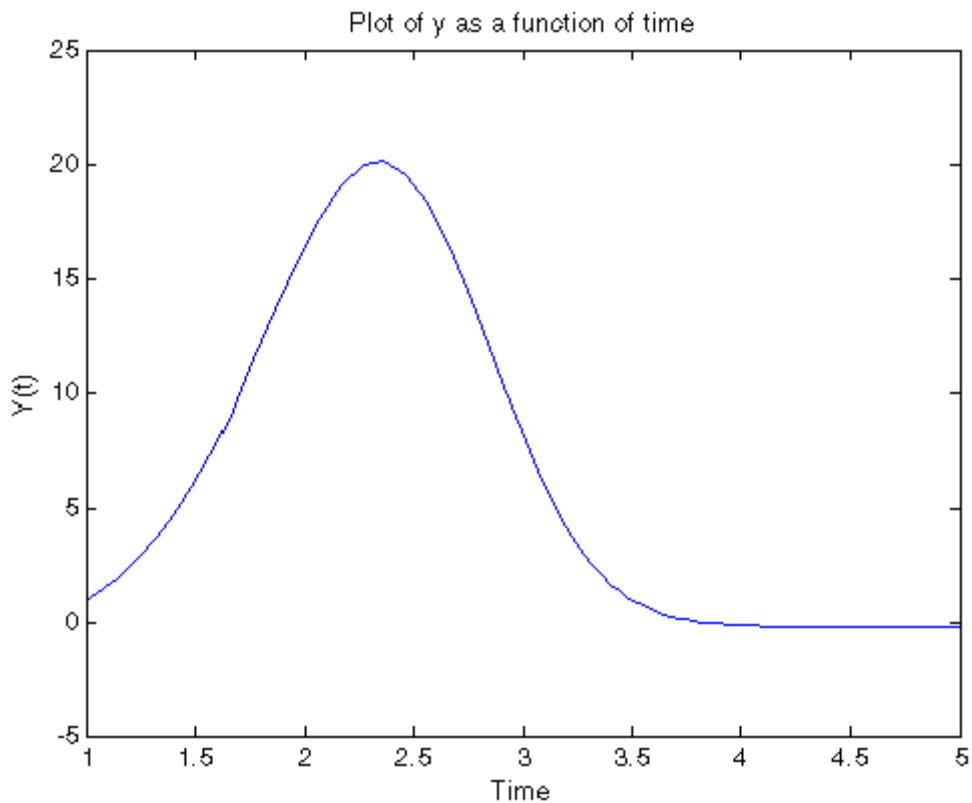
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
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; 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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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.

|                      |                                                                                                                     |
|----------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>y0</code>      | A vector of initial conditions.                                                                                     |
| <code>options</code> | 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.

|                 |                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>T</code>  | Column vector of time points.                                                                                                          |
| <code>Y</code>  | Solution array. Each row in <code>Y</code> corresponds to the solution at a time returned in the corresponding row of <code>T</code> . |
| <code>TE</code> | The time at which an event occurs.                                                                                                     |
| <code>YE</code> | 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

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ } t_f]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $t_f$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, t_f]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3265 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.

# ode45

| 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

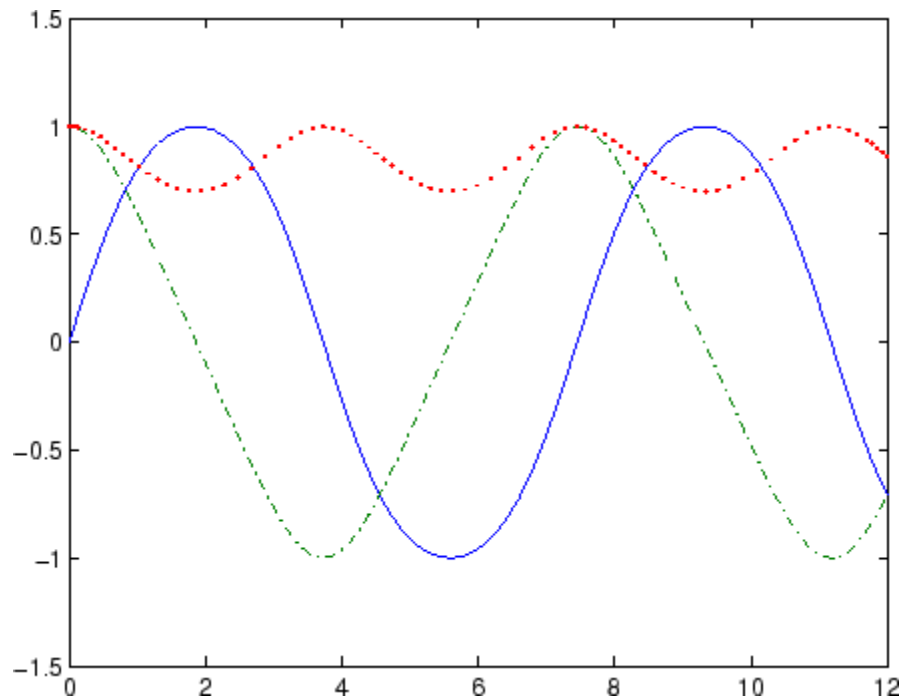
```
function dy = rigid(t,y)
dy = zeros(3,1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```

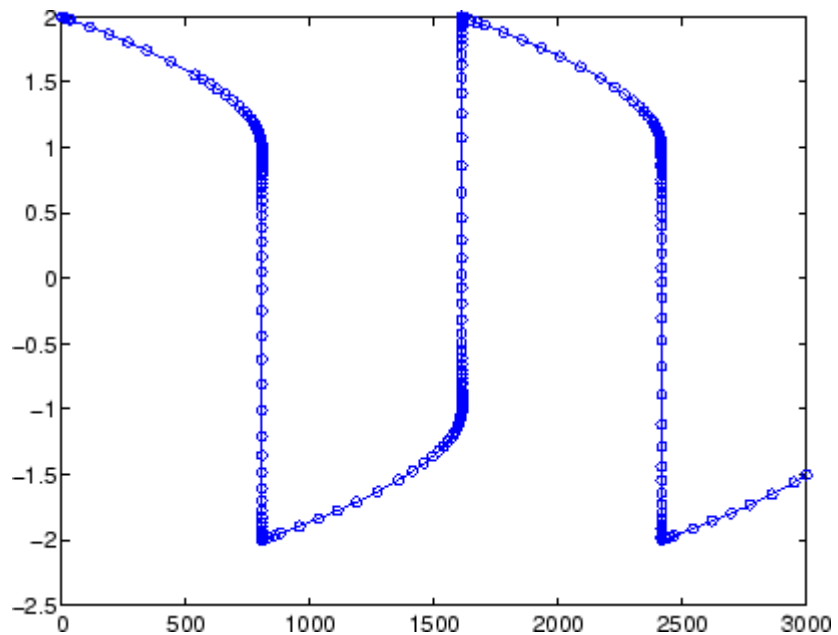
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

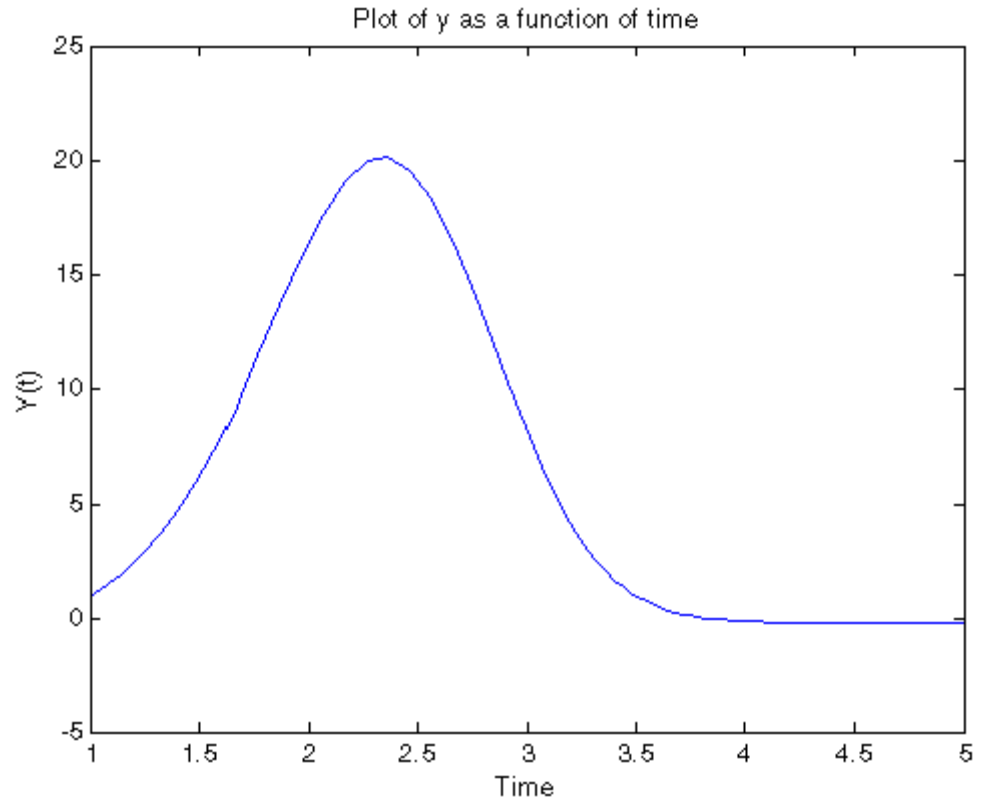
```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

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.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>odefun</code> | 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 <code>ode23s</code> solver can solve only equations with constant mass matrices. <code>ode15s</code> and <code>ode23t</code> can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).                                                                                                                                                                                                                                       |
| <code>tspan</code>  | A vector specifying the interval of integration, <code>[t0,tf]</code> . The solver imposes the initial conditions at <code>tspan(1)</code> , and integrates from <code>tspan(1)</code> to <code>tspan(end)</code> . To obtain solutions at specific times (all increasing or all decreasing), use <code>tspan = [t0,t1,...,tf]</code> .<br><br>For <code>tspan</code> vectors with two elements <code>[t0 tf]</code> , the solver returns the solution evaluated at every integration step. For <code>tspan</code> 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.

|                 |                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>T</code>  | Column vector of time points.                                                                                                          |
| <code>Y</code>  | Solution array. Each row in <code>Y</code> corresponds to the solution at a time returned in the corresponding row of <code>T</code> . |
| <code>TE</code> | The time at which an event occurs.                                                                                                     |
| <code>YE</code> | 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

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0)$  with  $\text{tspan} = [t_0 \text{ } t_f]$  integrates the system of differential equations  $y' = f(t, y)$  from time  $t_0$  to  $t_f$  with initial conditions  $y_0$ . The first input argument, `odefun`, is a function handle. The function,  $f = \text{odefun}(t, y)$ , 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  $t_0, t_1, \dots, t_f$  (all increasing or all decreasing), use  $\text{tspan} = [t_0, t_1, \dots, t_f]$ .

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

$[T, Y] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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.

$[T, Y, TE, YE, IE] = \text{solver}(\text{odefun}, \text{tspan}, y_0, \text{options})$  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  $[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$ . 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:

|                         |                                                                                      |
|-------------------------|--------------------------------------------------------------------------------------|
| <code>sol.x</code>      | Steps chosen by the solver.                                                          |
| <code>sol.y</code>      | Each column <code>sol.y(:,i)</code> contains the solution at <code>sol.x(i)</code> . |
| <code>sol.solver</code> | Solver name.                                                                         |

If you specify the `Events` option and events are detected, `sol` also includes these fields:

|                     |                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sol.xe</code> | Points at which events, if any, occurred. <code>sol.xe(end)</code> contains the exact point of a terminal event, if any.                           |
| <code>sol.ye</code> | Solutions that correspond to events in <code>sol.xe</code> .                                                                                       |
| <code>sol.ie</code> | Indices into the vector returned by the function specified in the <code>Events</code> 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  $y_0$  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  $y_0$  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. When solving DAEs, it is very advantageous to formulate the problem so that  $M$  is a diagonal matrix (a semi-explicit DAE).

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                 |
|---------------|---------------------|--------------------------|------------------------------------------------------------------------------------|
| 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. |

| <b>Solver</b> | <b>Problem Type</b> | <b>Order of Accuracy</b> | <b>When to Use</b>                                                                              |
|---------------|---------------------|--------------------------|-------------------------------------------------------------------------------------------------|
| 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 2-3280 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,<br>NormControl            | √     | √     | √      | √      | √      | √      | √       |
| OutputFcn,<br>OutputSel,<br>Refine, Stats | √     | √     | √      | √      | √      | √      | √       |
| NonNegative                               | √     | √     | √      | √ *    | —      | √ *    | √ *     |
| Events                                    | √     | √     | √      | √      | √      | √      | √       |
| MaxStep,<br>InitialStep                   | √     | √     | √      | √      | √      | √      | √       |
| Jacobian,<br>JPattern,<br>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

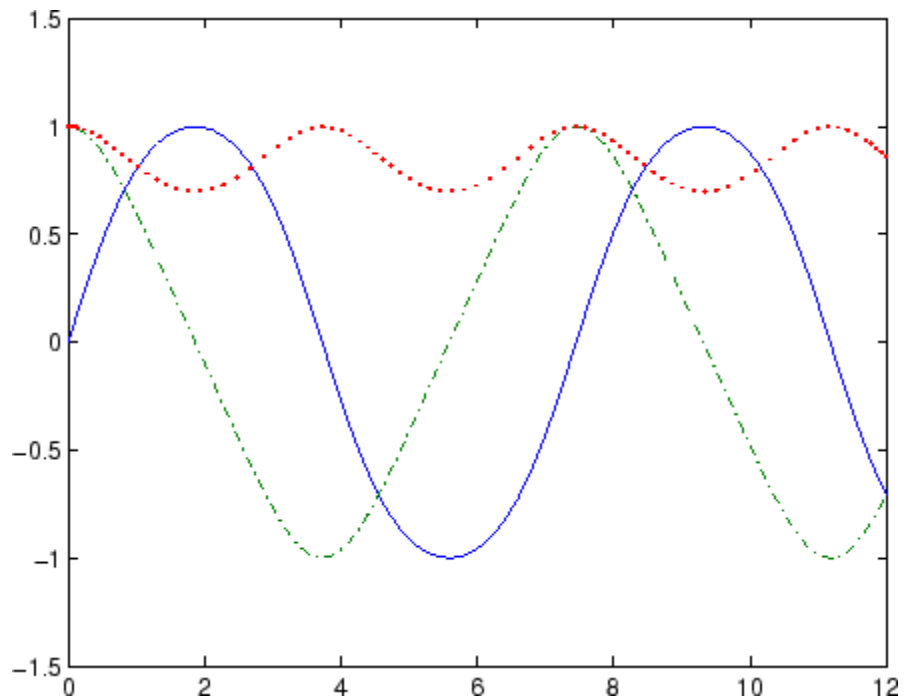
```
function dy = rigid(t,y)
dy = zeros(3,1); % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

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)
```

# ode113

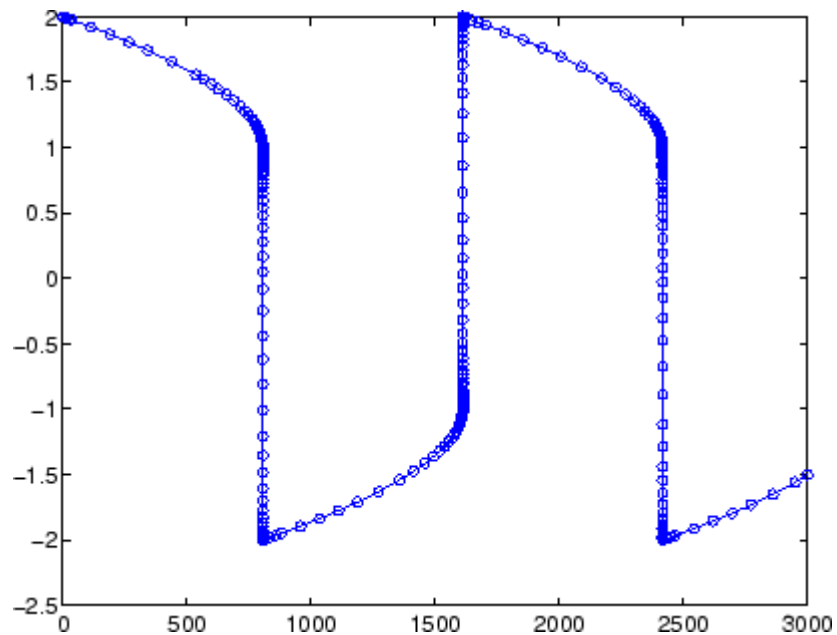
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

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; % Evaluate ODE at time t
```

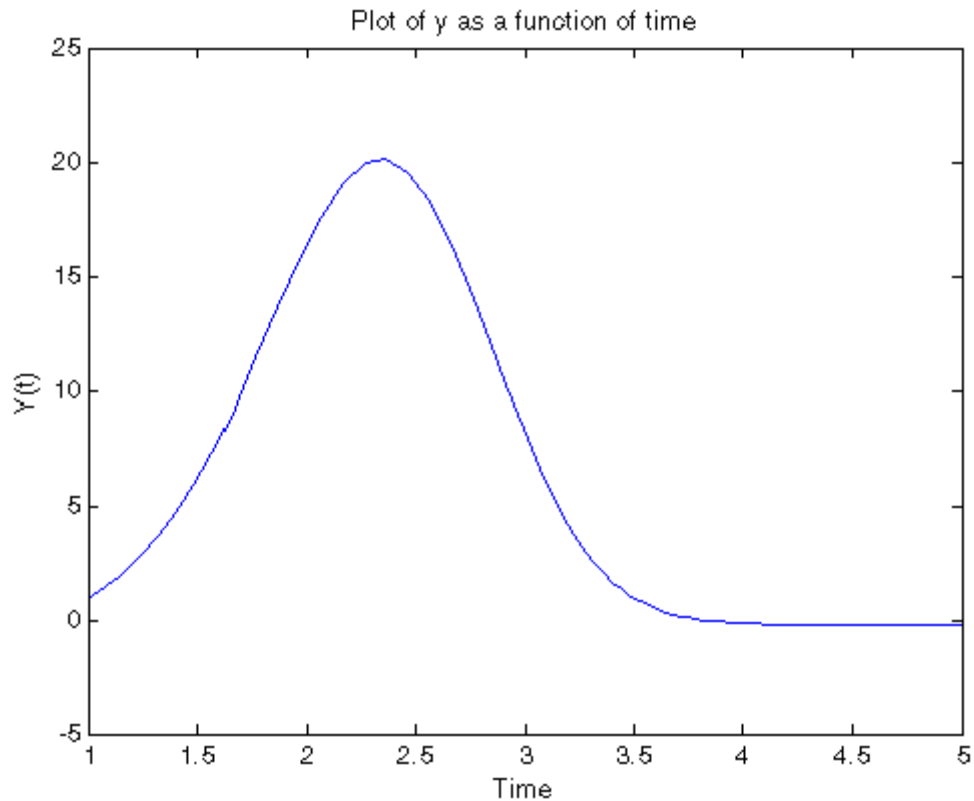
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
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**

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`

# odeset

---

**Purpose** Create or alter options structure for ordinary differential equation solvers

**Syntax**

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(olddopts,'name1',value1,...)
options = odeset(olddopts,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(olddopts,'name1',value1,...)` alters an existing options structure `olddopts`. This sets `options` equal to the existing structure `olddopts`, overwrites any values in `olddopts` 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(olddopts,newopts)` alters an existing options structure `olddopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `olddopts`.

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 2-3285
- “Solver Output Properties” on page 2-3287
- “Step-Size Properties” on page 2-3290
- “Event Location Property” on page 2-3292
- “Jacobian Matrix Properties” on page 2-3293
- “Mass Matrix and DAE Properties” on page 2-3297
- “ode15s and ode15i-Specific Properties” on page 2-3299

---

**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`.

$$|e(i)| \leq \max(\text{RelTol} * \text{abs}(y(i)), \text{AbsTol}(i))$$

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                                                                               |
|--------------------------|-------------------------------------|-------------------------------------------------------------------------------------------|
| <code>RelTol</code>      | Positive scalar<br>{1e-3}           | Relative error tolerance that applies to all components of the solution vector $y$ .      |
| <code>AbsTol</code>      | Positive scalar<br>or vector {1e-6} | Absolute error tolerances that apply to the individual components of the solution vector. |
| <code>NormControl</code> | 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  $i$ th 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  $\text{norm}(e) \leq \max(\text{RelTol} \cdot \text{norm}(y), \text{AbsTol})$ . 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 <code>[]</code> .          |
| 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 <code>Refine</code> .                                             |
| Stats       | on   {off}         | Determines whether the solver should display statistics about its computations. By default, <code>Stats</code> 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 <code>myfun(tspan,y0,'init')</code> before beginning the integration to allow the output function to initialize. <code>tspan</code> and <code>y0</code> are the input arguments to the ODE solver.                                                                                                                                                                                                                                                                                                                                                                                                             |
| {[]} | The solver calls <code>status = myfun(t,y,[])</code> after each integration step on which output is requested. <code>t</code> contains points where output was generated during the step, and <code>y</code> is the numerical solution at the points in <code>t</code> . If <code>t</code> is a vector, the <i>i</i> th column of <code>y</code> corresponds to the <i>i</i> th element of <code>t</code> .<br><br>When <code>length(tspan) &gt; 2</code> the output is produced at every point in <code>tspan</code> . When <code>length(tspan) = 2</code> the output is produced according to the <code>Refine</code> 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 <b>Stop</b> 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<br>{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 2-3285 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

$$[\text{value}, \text{isterminal}, \text{direction}] = \text{events}(t, y)$$

`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.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \dots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

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 2-3296 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 1s 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 (`.`<sup>`^`</sup>) and array multiplication (`.`<sup>`*`</sup>) 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

$$[dFd_y, dFd_p] = Fjac(t, y, y_p)$$

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 `{dFd_yPattern, dFd_pPattern}`, 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

$$M(t,y)y' = f(t,y) \quad (2-3)$$

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 2-3) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse  $M(t,y)$ , solving Equation 2-3 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<br>(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`

**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

# odextend

---

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  $y' = \text{vdp1}(t,y)$ , 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](#)

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Cleanup tasks upon function completion                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>      | <code>cleanupObj = onCleanup(cleanupFun)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p><code>cleanupObj = onCleanup(cleanupFun)</code> creates an object that, when destroyed, executes the function <code>cleanupFun</code>. 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 <b>Ctrl+C</b>.</p> <p>If you reference or pass <code>cleanupObj</code> outside your function, then <code>cleanupFun</code> does not run when that function terminates. Instead, it runs whenever the MATLAB destroys the object.</p> |

## 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**

```
Y = ones(n)
Y = ones(m,n)
Y = ones([m n])
Y = ones(m,n,p,...)
Y = ones([m n p ...])
Y = ones(size(A))
Y = ones
ones(m, n,...,classname)
ones([m,n,...],classname)
```

**Description** `Y = ones(n)` returns an  $n$ -by- $n$  matrix of 1s. An error message appears if  $n$  is not a scalar.

`Y = ones(m,n)` or `Y = ones([m n])` returns an  $m$ -by- $n$  matrix of ones.

`Y = ones(m,n,p,...)` or `Y = ones([m n p ...])` returns an  $m$ -by- $n$ -by- $p$ -by-... array of 1s.

---

**Note** The size inputs  $m$ ,  $n$ ,  $p$ , ... should be nonnegative integers. Negative integers are treated as 0.

---

`Y = ones(size(A))` returns an array of 1s that is the same size as  $A$ .

`Y = ones` with no arguments is the scalar 1.

`ones(m, n,...,classname)` or `ones([m,n,...],classname)` is an  $m$ -by- $n$ -by-... array of ones of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

**Examples** `x = ones(2,3,'int8');`

**See Also** `eye` | `zeros` | `complex` | `true`

**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 <i>st</i> when called with the syntax:<br><br><code>st = open(name)</code> |
| .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.

# openfig

---

- 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 whether your system is currently using hardware or 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 2-3315 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)
```

## **OpenGLWobbleTessellatorBug**

Symptom: Rendering complex patch object causes segmentation violation and returns a tessellator error message in the stack trace.

Command to enable:

```
opengl('OpenGLWobbleTessellatorBug',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 is 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.

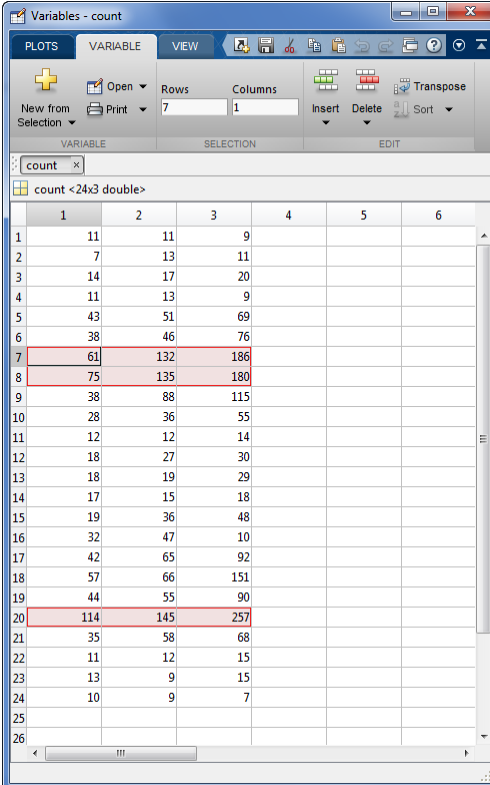
Make a scatter plot of data in the sample MAT-file `count.dat`, and open the variable `count` in the Variables editor.

```
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.

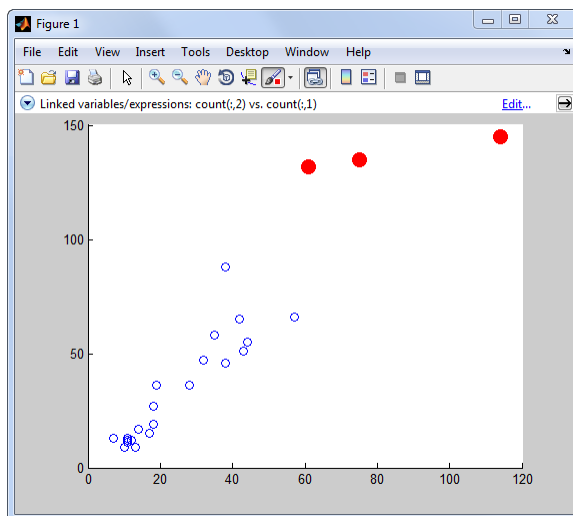
Select the rows 7, 8, and 20. (Select noncontiguous rows by holding down the **Ctrl** key and clicking in each row.)



|    | 1   | 2   | 3   | 4 | 5 | 6 |
|----|-----|-----|-----|---|---|---|
| 1  | 11  | 11  | 9   |   |   |   |
| 2  | 7   | 13  | 11  |   |   |   |
| 3  | 14  | 17  | 20  |   |   |   |
| 4  | 11  | 13  | 9   |   |   |   |
| 5  | 43  | 51  | 69  |   |   |   |
| 6  | 38  | 46  | 76  |   |   |   |
| 7  | 61  | 132 | 186 |   |   |   |
| 8  | 75  | 135 | 180 |   |   |   |
| 9  | 38  | 88  | 115 |   |   |   |
| 10 | 28  | 36  | 55  |   |   |   |
| 11 | 12  | 12  | 14  |   |   |   |
| 12 | 18  | 27  | 30  |   |   |   |
| 13 | 18  | 19  | 29  |   |   |   |
| 14 | 17  | 15  | 18  |   |   |   |
| 15 | 19  | 36  | 48  |   |   |   |
| 16 | 32  | 47  | 10  |   |   |   |
| 17 | 42  | 65  | 92  |   |   |   |
| 18 | 57  | 66  | 151 |   |   |   |
| 19 | 44  | 55  | 90  |   |   |   |
| 20 | 114 | 145 | 257 |   |   |   |
| 21 | 35  | 58  | 68  |   |   |   |
| 22 | 11  | 12  | 15  |   |   |   |
| 23 | 13  | 9   | 15  |   |   |   |
| 24 | 10  | 9   | 7   |   |   |   |
| 25 |     |     |     |   |   |   |
| 26 |     |     |     |   |   |   |

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(olddopts,'param1',value1,...)
options = optimset(olddopts,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(olddopts, 'param1', value1, ...)` creates a copy of `olddopts`, modifying the specified parameters with the specified values.

`options = optimset(olddopts, newopts)` combines an existing options structure `olddopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `olddopts`.

## Options

The following table lists the available options for the MATLAB optimization functions.

| Option  | Value                                          | Description                                                                                                                                                                                                | Solvers                                                                                            |
|---------|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Display | 'off'  <br>'iter'  <br>{'final'}  <br>'notify' | Level of display. 'off' displays no output; 'iter' displays output at each iteration (not available for <code>lsqnonneg</code> ); 'final' displays just the final output; 'notify' displays output only if | <code>fminbnd</code> ,<br><code>fminsearch</code> , <code>fzero</code> ,<br><code>lsqnonneg</code> |

# optimset

| 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.<br>Built-in functions: <ul style="list-style-type: none"> <li>• @optimplotx plots the current point</li> <li>• @optimplotfval plots the function value</li> <li>• @optimplotfunccount plots the function</li> </ul> | 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 -  $\lambda$ *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
```

# ordeig

---

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 =
```

# orderfields

---

```
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

**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  $E(\text{select})$  where E is the vector of eigenvalues as they appear along the diagonal of  $AA^{-1}BB$ .

---

**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  $AA^{-1}BB$ .

---

[...] = 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 ( $\text{real}(E) < 0$ )      |
| 'rhp'   | Right-half plane ( $\text{real}(E) > 0$ )     |
| 'udi'   | Interior of unit disk ( $\text{abs}(E) < 1$ ) |
| 'udo'   | Exterior of unit disk ( $\text{abs}(E) > 1$ ) |

[...] = ordqz(AA,BB,Q,Z,clusters) reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with  $E = \text{ordeig}(AA,BB)$ , such that all eigenvalues with the same `clusters`

value from 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  $X = U*U^*U'$  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  $E(\text{select})$  where  $E$  is the vector of eigenvalues as they appear along  $T$ 's diagonal.

---

**Note** To extract  $E$  from  $T$ , use  $E = \text{ordeig}(T)$ , instead of  $\text{eig}$ . 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 ( $\text{real}(E) < 0$ )      |
| 'rhp'   | Right-half plane ( $\text{real}(E) > 0$ )     |
| 'udi'   | Interior of unit disk ( $\text{abs}(E) < 1$ ) |
| 'udo'   | Exterior of unit disk ( $\text{abs}(E) > 1$ ) |

[US,TS] = ordschur(U,T,clusters) reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with  $E = \text{ordeig}(T)$ , 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  $B' * B = \text{eye}(\text{rank}(A))$ . 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.

# otherwise

---

## 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.');
```

## 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 Windows platforms, 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`



|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>       | Padé approximation of time delays                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Syntax</b>        | <code>[num,den] = padecoeff(T,N)</code>                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b>   | <p><code>[num,den] = padecoeff(T,N)</code> returns the <math>N</math>th-order Padé approximation of the continuous-time delay <math>T</math> in transfer function form. The row vectors <code>num</code> and <code>den</code> contain the numerator and denominator coefficients in descending powers of <math>s</math>. Both are <math>N</math>th-order polynomials.</p> <p>Class support for input <math>T</math>:</p> <p>float: double, single</p> |
| <b>Class Support</b> | Input $T$ support floating-point values of type <code>single</code> or <code>double</code> .                                                                                                                                                                                                                                                                                                                                                          |
| <b>References</b>    | [1] Golub, G. H. and C. F. Van Loan <i>Matrix Computations</i> , 3rd ed. Johns Hopkins University Press, Baltimore: 1996, pp. 572–574.                                                                                                                                                                                                                                                                                                                |
| <b>See Also</b>      | <code>pade</code>                                                                                                                                                                                                                                                                                                                                                                                                                                     |

# pagesetupdlg

**Purpose** Page setup dialog box

**Syntax** `dlg = pagesetupdlg(fig)`

---

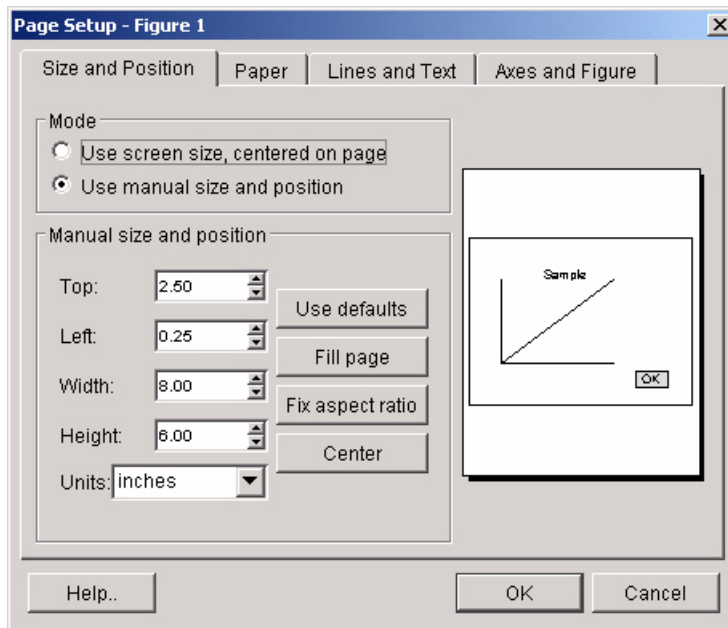
**Note** `pagesetupdlg` is not recommended. Use `printpreview` instead.

---

**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(gcf, '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

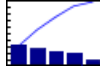
Use the **Pan** tool  on the figure toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure’s **Tools** menu. For details, see “Panning — Shifting Your View of the Graph”.

## 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 NaNs. 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.

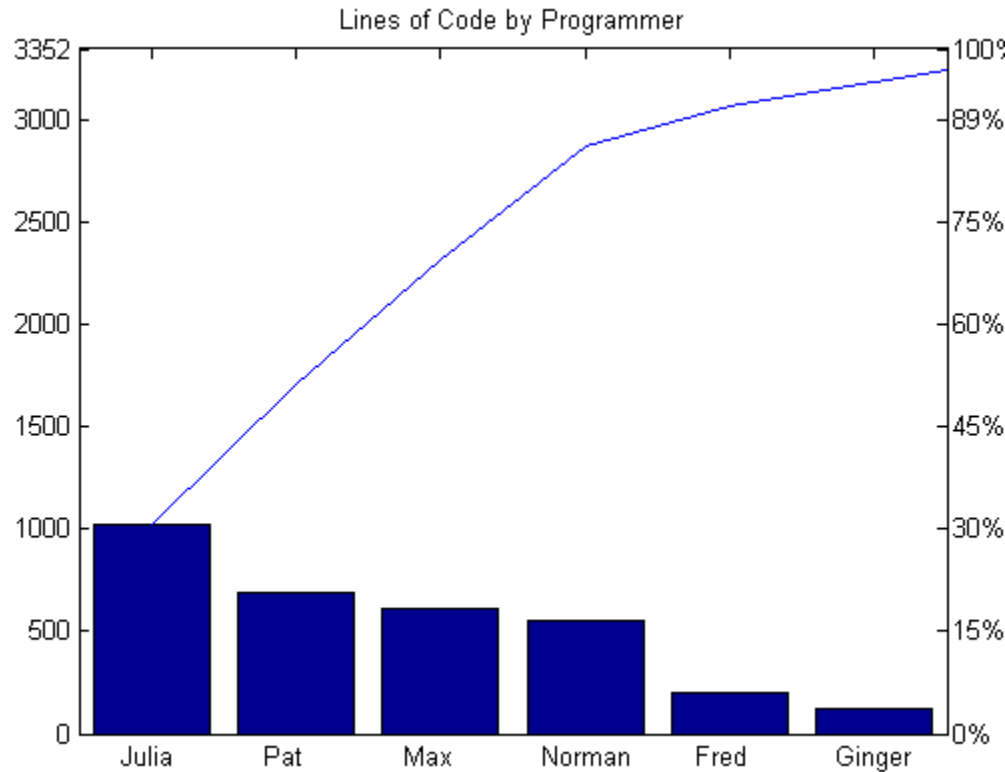
**Examples**

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')
```

# pareto

---



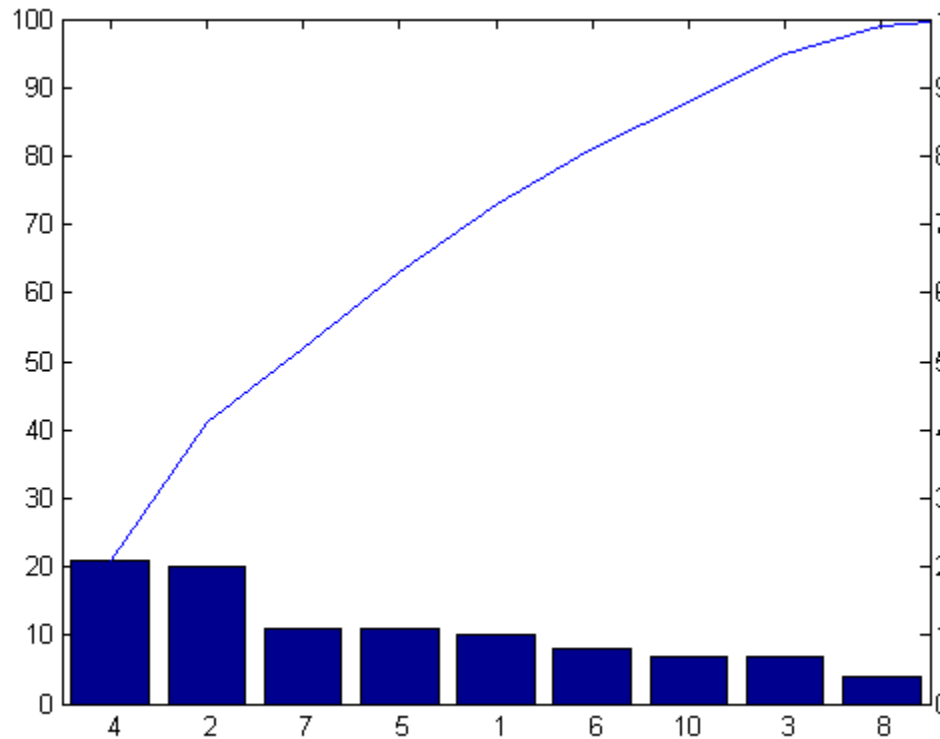
---

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`.

# pareto

---

## See Also

hist | bar

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Parallel for loop                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>      | <pre>parfor loopvar = initval:endval; <i>statements</i>; end parfor (loopvar = initval:endval, M); <i>statements</i>; end</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b> | <p>parfor loopvar = initval:endval; <i>statements</i>; 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.</p> <p>parfor (loopvar = initval:endval, M); <i>statements</i>; end executes statements in a loop using a maximum of M workers or threads, where M is a nonnegative integer.</p> |
| <b>Tips</b>        | <ul style="list-style-type: none"><li>• If you have Parallel Computing Toolbox software, see the function reference pages for parfor and matlabpool for additional information.</li><li>• If you have MATLAB Coder software, see the parfor function reference page for additional information.</li></ul>                                                                                                                                                                                                                                                                                                                                 |
| <b>Examples</b>    | <p>Perform three large eigenvalue computations using three workers or cores with Parallel Computing Toolbox software:</p> <pre>matlabpool(3) parfor i=1:3, c(:,i) = eig(rand(1000)); end</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>See Also</b>    | for                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

# inputParser.parse

---

|                        |                                                                                                                                                                                                             |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Parse function inputs                                                                                                                                                                                       |
| <b>Syntax</b>          | parse(p, argList)                                                                                                                                                                                           |
| <b>Description</b>     | parse(p, argList) parses and validates the inputs in arglist.                                                                                                                                               |
| <b>Input Arguments</b> | <b>p</b><br>Object of class inputParser.<br><b>argList</b><br>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](#) | [createClassFromWsd1](#) | [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 2-3369.

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 2-3366.

`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 2-3366 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:

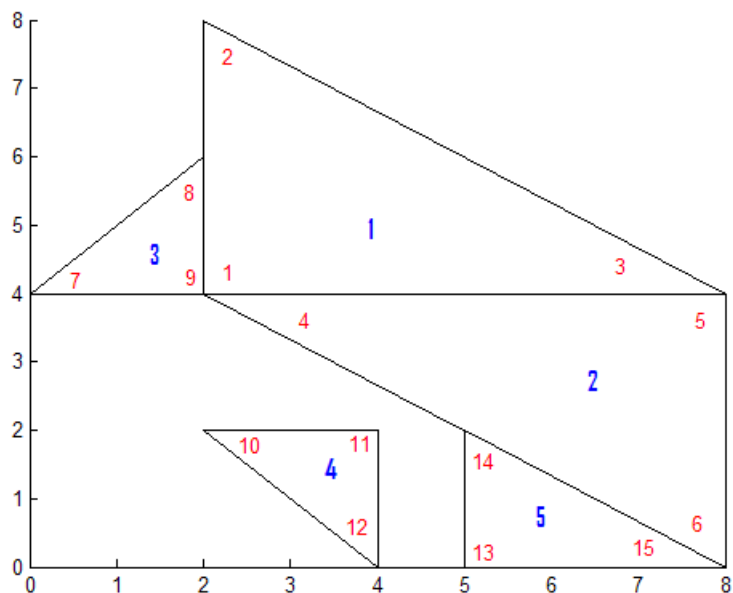
```
xdata = [2 2 0 2 5;  
         2 8 2 4 5;
```

```

        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

# patch

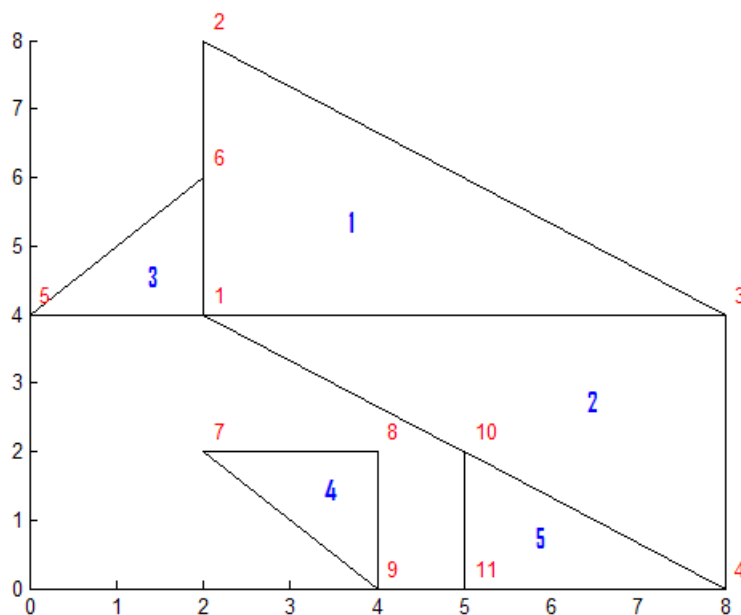
---

```
% 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.

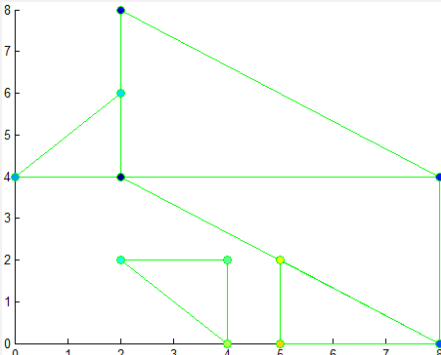
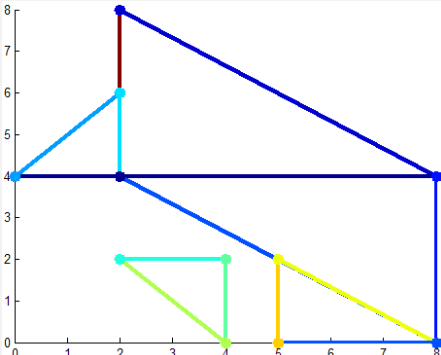
## Specifying Edge Colors

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:

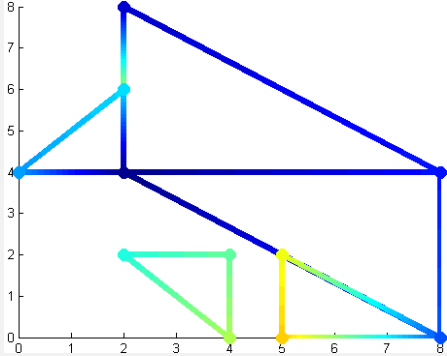
```
xdata = [2     2     0     2     5;  
         2     8     2     4     5;  
         8     8     2     4     8];  
ydata = [4     4     4     2     0;  
         8     4     6     2     2;  
         4     0     4     0     0];  
cdata = [15    0     4     6    10;  
         1     2     5     7     9;  
         2     3     0     8     3];  
p = patch(xdata,ydata,cdata,'Marker','o','MarkerFaceColor','flat','FaceColor','g');
```

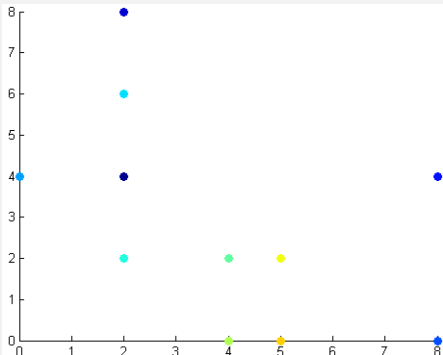
For more detailed information on how the `EdgeColor` property works, see the [Patch Properties](#) page.

| Desired Look                                                                                                    | EdgeColor Value        | Sample Code                           |
|-----------------------------------------------------------------------------------------------------------------|------------------------|---------------------------------------|
| All edges have the same color, around all faces. This option does not rely on the <code>FaceColor</code> value. | <code>ColorSpec</code> | <code>set(p, 'EdgeColor', 'g')</code> |

| Desired Look                                                                                                                                                                                                                                                                                                                                                        | EdgeColor Value | Sample Code                                                      |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                    |                 |                                                                  |
| <p>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.</p>  | <p>'flat'</p>   | <pre>set(p, 'EdgeColor', 'flat', ...       'LineWidth', 3)</pre> |

# patch

| Desired Look                                                                                                                                                                                                                                                                                                                                                                                                             | EdgeColor Value              | Sample Code                                                                                  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|----------------------------------------------------------------------------------------------|
| <p>Each edge corresponds to the vertex colors, interpolated between vertices. This option requires that the <code>FaceColor</code> property be <code>flat</code> or <code>interp</code>. By default, if you specify <code>CData</code> when creating the patch object, its <code>FaceColor</code> property is <code>interp</code>.</p>  | <p><code>'interp'</code></p> | <pre>set(gcf, 'Renderer', 'zbuffer') set(p, 'EdgeColor', 'interp', ... 'LineWidth', 5)</pre> |
| <p>Edges have no color. This option does not rely on the <code>FaceColor</code> value. If set, markers retain vertex colors.</p>                                                                                                                                                                                                                                                                                         | <p><code>'none'</code></p>   | <pre>set(p, 'EdgeColor', 'none')</pre>                                                       |

| Desired Look                                                                      | EdgeColor Value | Sample Code |
|-----------------------------------------------------------------------------------|-----------------|-------------|
|  |                 |             |

### Specifying Face Colors Using Face/Vertex Input Matrices

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:

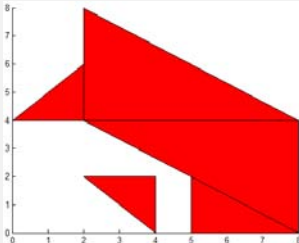
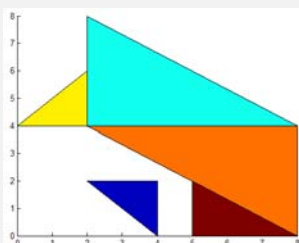
% For this example, there are five triangles ( $m = 5$ ) sharing  
 % eleven unique vertices ( $k = 11$ ).

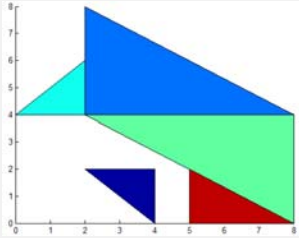
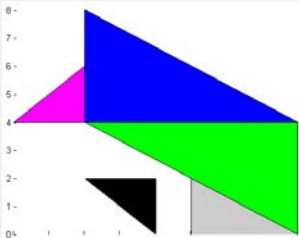
```
verts = [2 4; ...
         2 8; ...
         8 4; ...
         8 0; ...
         0 4; ...
         2 6; ...
         2 2; ...
         4 2; ...
         4 0; ...
         5 2; ...
         5 0 ];
faces = [1 2 3; ...
         1 3 4; ...
         5 6 1; ...
         7 8 9; ...
         11 10 4];
```

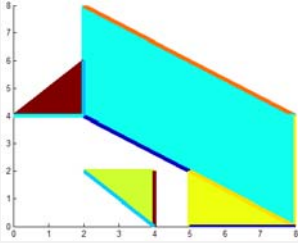
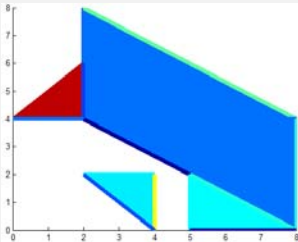
# patch

```
p = patch('Faces',faces,'Vertices',verts,'FaceColor','b');
```

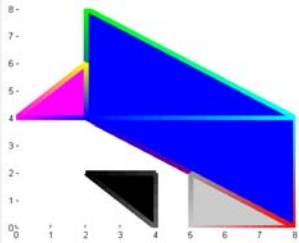
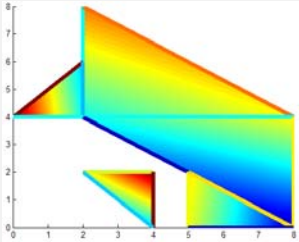
For more information on the relevant properties, see [FaceColor](#), [FaceVertexCData](#), and [CDataMapping](#).

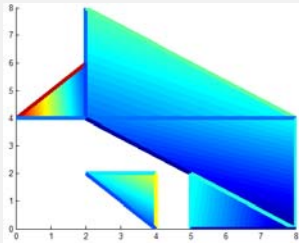
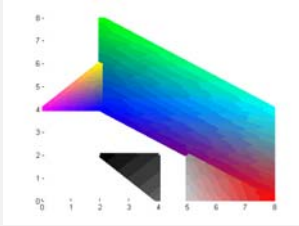
| Desired Look                                                                                                                                                                      | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                                         | Sample Code                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>All faces have the same color.</p>                                                            | <ul style="list-style-type: none"> <li>• <b>FaceColor:</b> ColorSpec</li> <li>• <b>FaceVertexCData:</b> [] (no input)<br/>An empty array is the default value, and patch ignores any input until you set FaceColor to 'flat' or 'interp'.</li> <li>• <b>Color source:</b> truecolor</li> <li>• <b>CDataMapping:</b> 'direct' or 'scaled'.<br/>'scaled' is the default value, but neither affects the outcome.</li> </ul> | <pre>set(p,'FaceColor','r')<br/><br/>or<br/><br/>set(p,'FaceColor',[1 0 0])</pre>                                                                                                    |
| <p>Each face has a single, unique color, indexed from a selected section of the colormap.</p>  | <ul style="list-style-type: none"> <li>• <b>FaceColor:</b> 'flat'</li> <li>• <b>FaceVertexCData:</b> m-by-1 matrix of index values</li> <li>• <b>Color source:</b> A selected portion of the colormap</li> <li>• <b>CDataMapping:</b> 'scaled'</li> </ul>                                                                                                                                                                | <pre>clear cdata<br/>set(gca,'CLim',[0 40])<br/>cdata = [15 30 25 2 60];<br/>set(p,'FaceColor','flat',...<br/>    'FaceVertexCData',cdata,...<br/>    'CDataMapping','scaled')</pre> |

| Desired Look                                                                                                                                                         | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                           | Sample Code                                                                                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Each face has a single, unique color, indexed from the whole colormap.</p>       | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• FaceVertexCData: m-by-1 matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <pre>clear cdata set(gca,'CLim',[0 40]) cdata = [15 30 25 2 60]'; set(p,'FaceColor','flat',... 'FaceVertexCData',cdata,... 'CDataMapping','direct')</pre> |
| <p>Each face has a single, unique color, determined by truecolor value input.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• FaceVertexCData: m-by-3 matrix of truecolor values, from 0 to 1</li> <li>• Color source: truecolor</li> <li>• CDataMapping: 'direct' or 'scaled'.</li> </ul> <p>'scaled' is the default value, but neither affects the outcome.</p>                                                                                                  | <pre>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)</pre>          |

| Desired Look                                                                                                                                                                                                                                                                  | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                          | Sample Code                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>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.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• FaceVertexCData: k-by-1 matrix of index values</li> <li>• Color source: A selected portion of the colormap</li> <li>• CDataMapping: 'scaled'</li> </ul>                                                                                                                                                                             | <pre>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')</pre> |
| <p>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.</p>                | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• FaceVertexCData: k-by-1 matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <pre>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)</pre> |



| Desired Look                                                                                                                                                                                                                                                      | Parameter Values                                                                                                                                                                                                                                                                                          | Sample Code                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>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.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• FaceVertexCData: k-by-3 matrix of truecolor values, from 0 to 1</li> <li>• Color source: truecolor</li> <li>• CDataMapping: 'direct' or 'scaled'.</li> </ul> <p>'scaled' is the default value, but neither affects the outcome.</p> | <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','flat',...     'FaceVertexCData',cdata,...     'EdgeColor','interp',...     'LineWidth',5)</pre> |
| <p>Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have 'flat' or 'interp' color.</p>                                | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• FaceVertexCData: k-by-1 matrix of index values</li> <li>• Color source: A selected portion of the colormap</li> <li>• CDataMapping: 'scaled'</li> </ul>                                                                           | <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,...     'EdgeColor','flat',...     'LineWidth',5,...     'CDataMapping','scaled')</pre>                                                                                           |

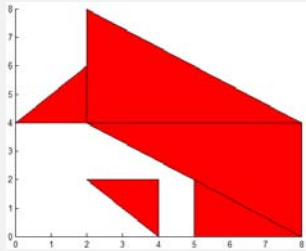
| Desired Look                                                                                                                                                                                                            | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                             | Sample Code                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Each unique vertex has a single, unique color, indexed from the whole colormap. Edges may have 'flat' or 'interp' color.</p>        | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• FaceVertexCData: k-by-1 matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <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>                                                                                    |
| <p>Each unique vertex has a single, unique color, determined by truecolor value input. Edges may have 'flat' or 'interp' color.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• FaceVertexCData: k-by-3 matrix of truecolor values, from 0 to 1</li> <li>• Color source: truecolor</li> <li>• CDataMapping: 'direct' or 'scaled'</li> </ul> <p>'scaled' is the default value, but neither affects the outcome.</p>                                                                                                   | <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> |

### Specifying Face Colors Using x-, y-, and z-Coordinate Input

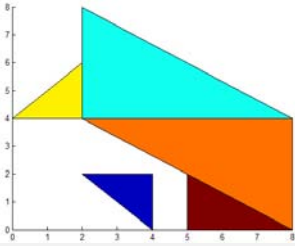
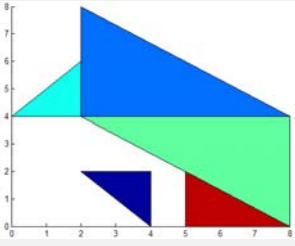
The following options apply to the face colors of your patch object when you specify the faces using x-, y-, and z-coordinates. To explore the options, start with a base patch object:

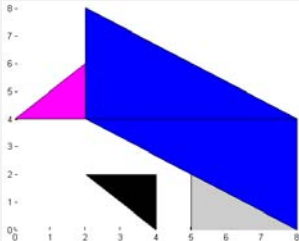
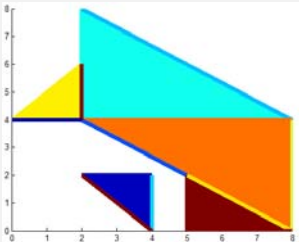
```
% For this example, there are five (m=5) triangles (n=3).
% The total number of vertices is mxn, or k = 15.
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);
p = patch(xdata,ydata,zdata,'b')
```

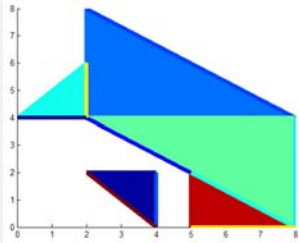
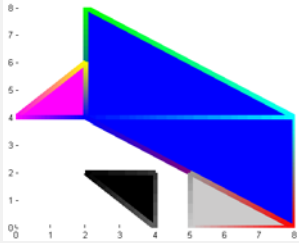
For more information on the relevant properties, see `FaceColor`, `CData`, and `CDataMapping`.

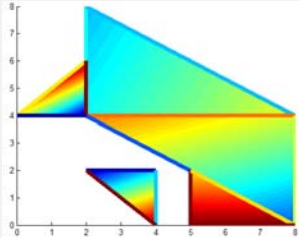
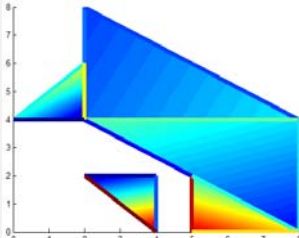
| Desired Look                                                                                                             | Parameter Values                                                                                                                                                                                                                                                                                                                                                                       | Sample Code                                                            |
|--------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| <p>All faces have the same color.</p>  | <ul style="list-style-type: none"> <li>• <code>FaceColor</code>: <code>ColorSpec</code></li> <li>• <code>CData</code>: <code>[]</code> (no input)</li> <li>• Color source: <code>truecolor</code></li> <li>• <code>CDataMapping</code>: <code>'direct'</code> or <code>'scaled'</code>.</li> </ul> <p><code>'scaled'</code> is the default value, but neither affects the outcome.</p> | <pre>set(p,'FaceColor','r')  or  set(p,'FaceColor',[1 0 0])</pre>      |
| <p>Each face has a single, unique color, indexed from a selected section of the colormap.</p>                            | <ul style="list-style-type: none"> <li>• <code>FaceColor</code>: <code>'flat'</code></li> <li>• <code>CData</code>: m-by-1 matrix of index values</li> </ul>                                                                                                                                                                                                                           | <pre>clear cdata set(gca,'CLim',[0 40]) cdata = [15 30 25 2 60];</pre> |

# patch

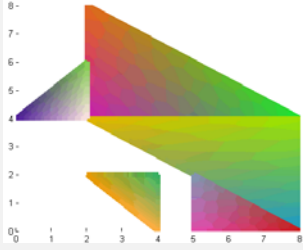
| Desired Look                                                                                                                                                     | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                | Sample Code                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                 | <ul style="list-style-type: none"> <li>• Color source: A selected portion of the colormap</li> <li>• CDataMapping: 'scaled'</li> </ul>                                                                                                                                                                                                                                                          | <pre>set(p,'FaceColor','flat',... 'CData',cdata,... 'CDataMapping','scaled')</pre>                                                             |
| <p>Each face has a single, unique color, indexed from the whole colormap.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• CData: m-by-1 matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <pre>clear cdata set(gca,'CLim',[0 40]) cdata = [15 30 25 2 60]; set(p,'FaceColor','flat',... 'CData',cdata,... 'CDataMapping','direct')</pre> |

| Desired Look                                                                                                                                                                                                                                                                    | Parameter Values                                                                                                                                                                                                                                                                                       | Sample Code                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Each face has a single, unique color, determined by truecolor value input.</p>                                                                                                              | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• CData: m-by-1-by-3 matrix of truecolor values, from 0 to 1</li> <li>• Color source: truecolor</li> <li>• CDataMapping: 'direct' or 'scaled'.<br/><br/>'scaled' is the default value, but neither affects the outcome.</li> </ul> | <pre>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)</pre>                                                                          |
| <p>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.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• CData: m-by-n matrix of index values</li> <li>• Color source: A selected portion of the colormap</li> <li>• CDataMapping: 'scaled'</li> </ul>                                                                                    | <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,... 'EdgeColor','flat',... 'LineWidth',5,... 'CDatamapping','scaled')</pre> |

| Desired Look                                                                                                                                                                                                                                                  | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                 | Sample Code                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>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.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• CData: m-by-n matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <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>                                                                                                                                                   |
| <p>Each 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.</p>   | <ul style="list-style-type: none"> <li>• FaceColor: 'flat'</li> <li>• CData: m-by-n-by-3 matrix of truecolor values, from 0 to 1</li> <li>• Color source: truecolor</li> <li>• CDataMapping: 'direct' or 'scaled'</li> </ul> <p>'scaled' is the default value, but neither affects the outcome.</p>                                                                                              | <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> |

| Desired Look                                                                                                                                                                                                               | Parameter Values                                                                                                                                                                                                                                                                                                                                                                                  | Sample Code                                                                                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Each vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have 'flat' or 'interp' color.</p>  | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• CData: m-by-n matrix of index values</li> <li>• Color source: A selected portion of the colormap</li> <li>• CDataMapping: 'scaled'</li> </ul>                                                                                                                                                                             | <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','interp',...      'CData',cdata,...      'EdgeColor','flat',...      'LineWidth',5,...      'CDataMapping','scaled')</pre> |
| <p>Each vertex has a single, unique color, indexed from the whole colormap. Edges may have 'flat' or 'interp' color.</p>                | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• CData: m-by-n matrix of index values</li> <li>• Color source: colormap</li> <li>• CDataMapping: 'direct'</li> </ul> <p>'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'.</p> | <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','interp',...      'CData',cdata,...      'CDataMapping','direct',...      'EdgeColor','flat',...      'LineWidth',5)</pre> |
| <p>Each vertex has a single, unique color, determined by truecolor value input. Edges</p>                                                                                                                                  | <ul style="list-style-type: none"> <li>• FaceColor: 'interp'</li> <li>• CData: m-by-n-by-3 matrix of truecolor values, from 0 to 1</li> </ul>                                                                                                                                                                                                                                                     | <pre>clear cdata cdata(:,:,1) = [0.8 0.1 0.2                 0.9 0.3 1;                 ...                 ...                 ...]</pre>                                                                                                                    |

# patch

| Desired Look                                                                                                                | Parameter Values                                                                                                                                                                               | Sample Code                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>may have 'flat' or 'interp' color.</p>  | <ul style="list-style-type: none"><li>• Color source: truecolor</li><li>• CDataMapping: 'direct' or 'scaled'.</li></ul> <p>'scaled' is the default value, but neither affects the outcome.</p> | <pre>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)</pre> |

## 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

# Patch Properties

---

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  $\geq 0$  and  $\leq 1$

*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'})
```

# Patch Properties

---

```
% 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.

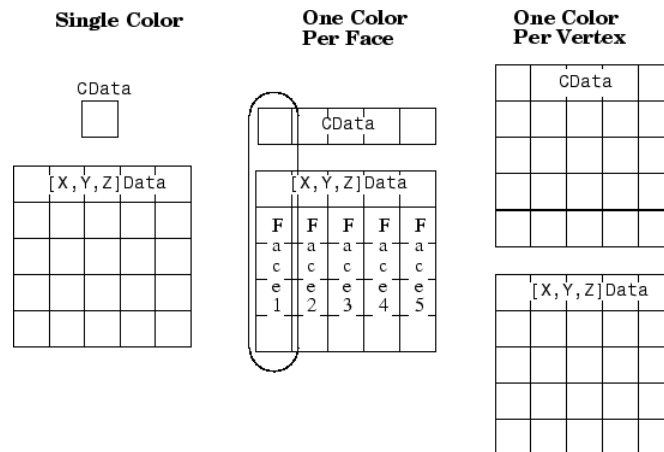
## `CData`

scalar | vector | matrix

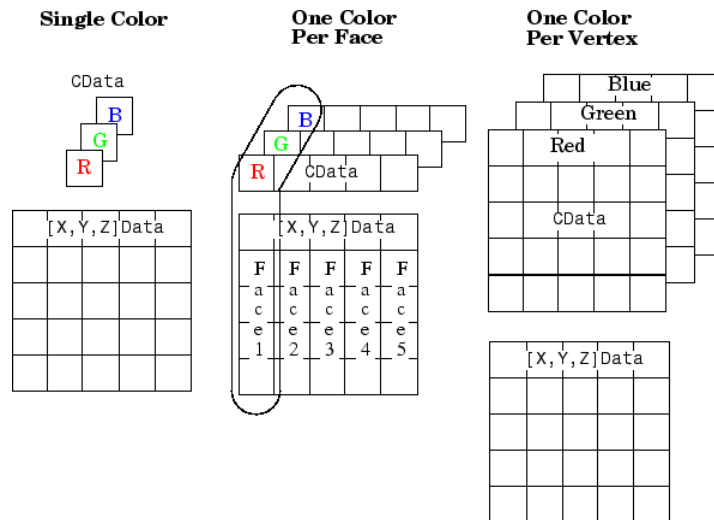
# Patch Properties

*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

# Patch Properties

---

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  $\geq 0$  and  $\leq 1$

*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.

# Patch Properties

---

- 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.

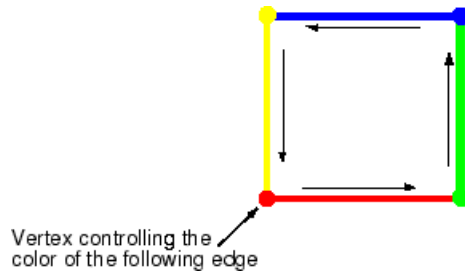
Note that you cannot specify `flat` or `interp` `EdgeAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

## 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.

# Patch Properties

---

- `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`).

# Patch Properties

---

## 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

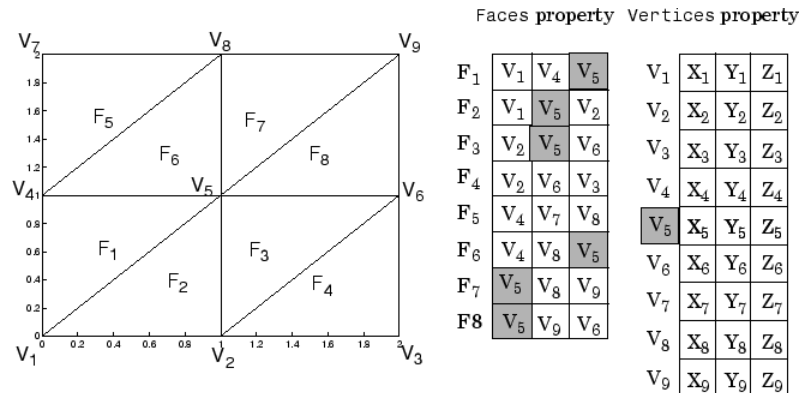
view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

## 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.



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.

# Patch Properties

---

## 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.

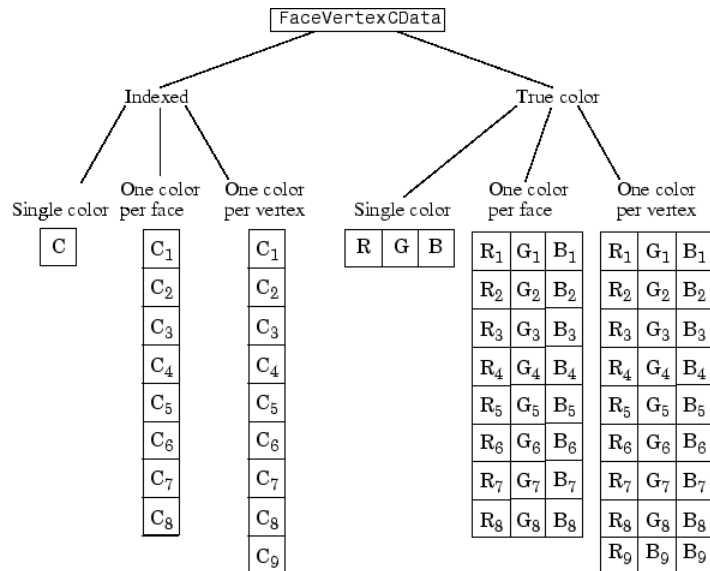


- An  $n$ -by-1 matrix, where  $n$  is the number of rows in the Vertices property, which specifies one color per vertex.

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.



# Patch Properties

---

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.

# Patch Properties

---

- 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 =  $\frac{1}{72}$  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     |

# Patch Properties

---

| 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.

# Patch Properties

---

## 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 of the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

## SpecularExponent

scalar  $\geq 1$

*Harshness of specular reflection.* Controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

## SpecularStrength

scalar  $\geq 0$  and  $\leq 1$



*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'`.

# Patch Properties

---

## 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

---

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | View or change search path                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Alternatives</b> | As an alternative to the <code>path</code> function, use the Set Path dialog box.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>       | <pre>path path('newpath') path(<b>path</b>, 'newpath') path('newpath', <b>path</b>) p = path</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b>  | <p><code>path</code> displays the MATLAB search path, which is stored in <code>pathdef.m</code>.</p> <p><code>path('newpath')</code> changes the search path to <code>newpath</code>, where <code>newpath</code> is a string array of folders.</p> <p><code>path(<b>path</b>, 'newpath')</code> adds the <code>newpath</code> folder to the end of the search path. If <code>newpath</code> is already on the search path, then <code>path(<b>path</b>, 'newpath')</code> moves <code>newpath</code> to the end of the search path.</p> <p><code>path('newpath', <b>path</b>)</code> adds the <code>newpath</code> folder to the top of the search path. If <code>newpath</code> is already on the search path, then <code>path('newpath', <b>path</b>)</code> moves <code>newpath</code> to the top of the search path. To add multiple folders in one statement, instead use <code>addpath</code>.</p> <p><code>p = path</code> returns the search path to string variable <code>p</code>.</p> |
| <b>Examples</b>     | <p>Display the search path:</p> <pre>path</pre> <p>MATLAB returns, for example</p> <pre>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</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

```
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')
```

# path

---

Add a new folder to the search path on UNIX<sup>8</sup> 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

- “Using the MATLAB Search Path”
- “Making Files and Folders Accessible to MATLAB”

8. 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<sup>9</sup> platforms was a colon (:). Use `pathsep` to work programmatically with the content of the search path file.

**See Also** `fileparts` | `filesep` | `fullfile` | `path`

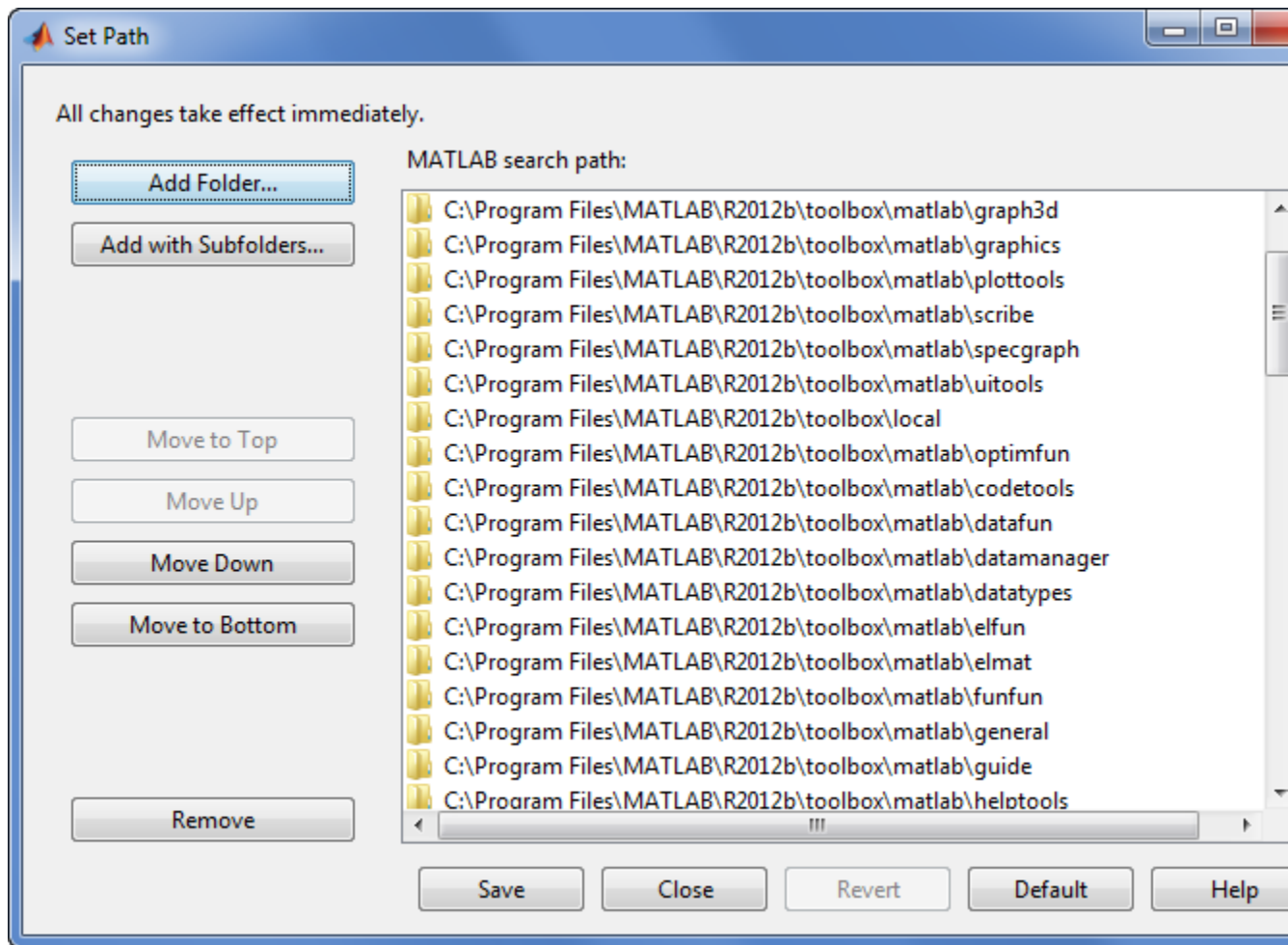
**How To**

- “Using the MATLAB Search Path”

9. UNIX is a registered trademark of The Open Group in the United States and other countries.



|                    |                                                                                                                                    |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Open Set Path dialog box to view and change search path                                                                            |
| <b>Syntax</b>      | <code>pathtool</code>                                                                                                              |
| <b>Description</b> | <code>pathtool</code> 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

- “Using 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`.

# pause

---

## 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,

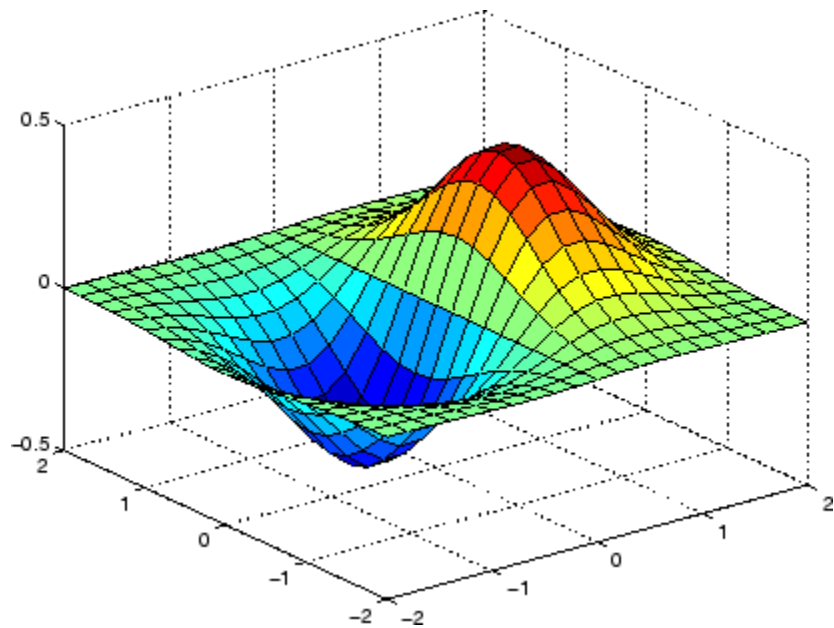
```
pbaspect(pbaspect)
```

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  $-2 \leq x \leq 2$ ,  $-2 \leq y \leq 2$ ,

```
[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.

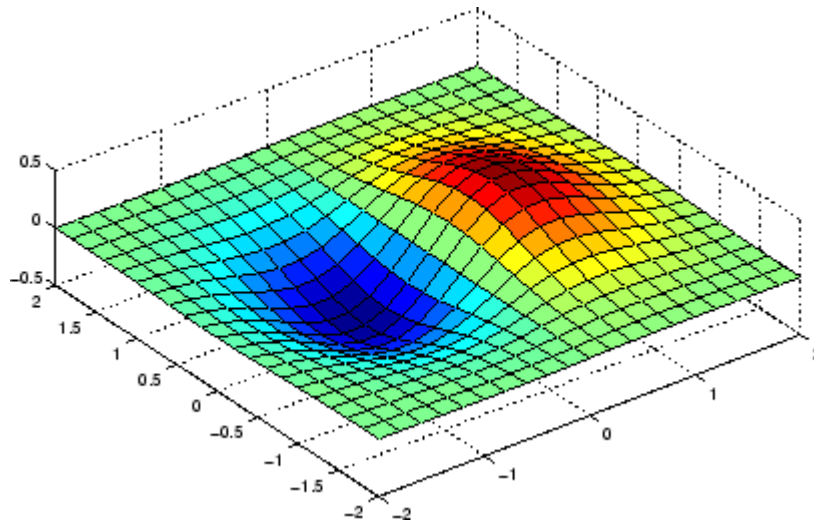
```
daspect
ans =
    4    4    1
```

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

---

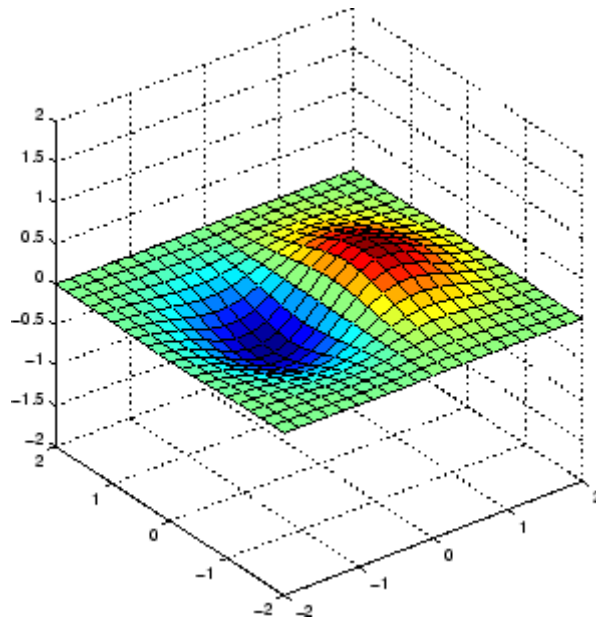


```
pbaspect
ans =
     4     4     1
```

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])
```

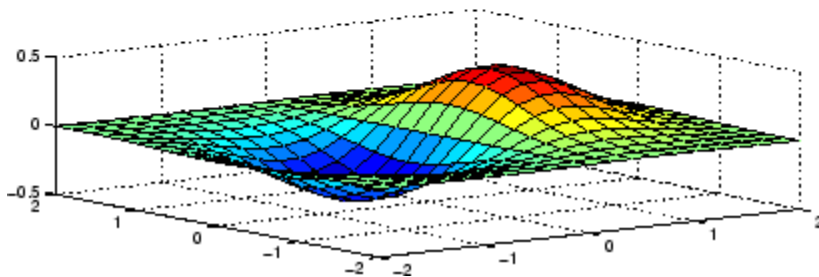
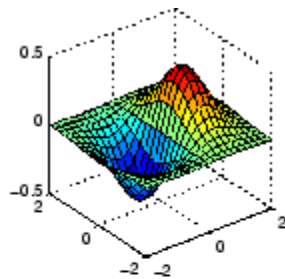




Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $\text{inv}(M)*A*x = \text{inv}(M)*b$  for  $x$ . If  $M$  is `[]`

then `pcg` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns  $M \backslash x$ .

`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    | <code>pcg</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.           |
| 1    | <code>pcg</code> iterated <code>maxit</code> times but did not converge.                                             |
| 2    | Preconditioner <code>M</code> was ill-conditioned.                                                                   |
| 3    | <code>pcg</code> stagnated. (Two consecutive iterates were the same.)                                                |
| 4    | One of the scalar quantities calculated during <code>pcg</code> 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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = pcg(A,b,...)` also returns the iteration number at which `x` was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = pcg(A,b,...)` also returns a vector of the residual norms at each iteration including  $\text{norm}(b-A*x0)$ .

## 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,f10,rr0,it0,rv0] = pcg(A,b,1e-8,100);

```

`f10` 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,f11,rr1,it1,rv1] = pcg(A,b,1e-8,100,L,L');
```

`f11` 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  $100 \times 100$  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,f12,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.');
```

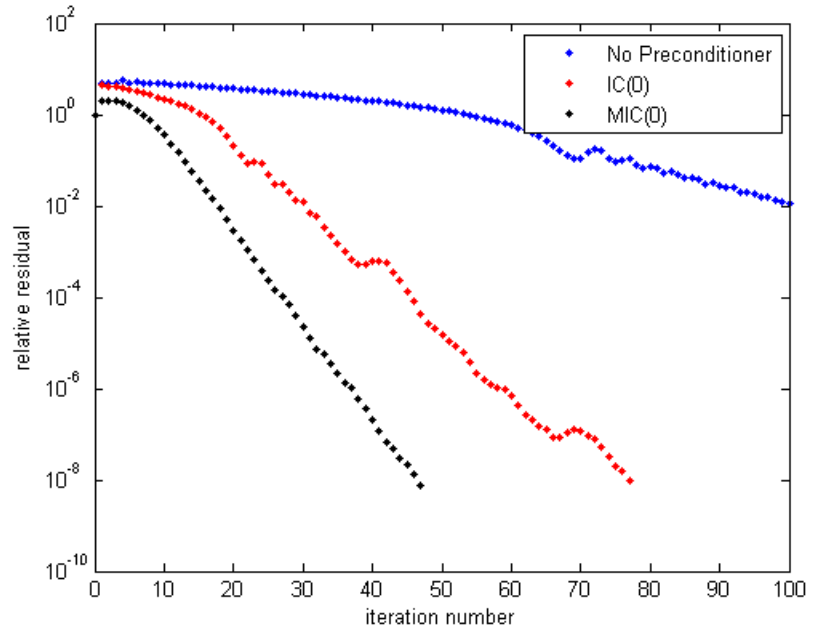
```
hold on;  
semilogy(0:it1,rv1/norm(b),'r.');
```

```
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  $x_k \leq x \leq x_{k+1}$ ,  $P(x)$  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  $y$ .

---



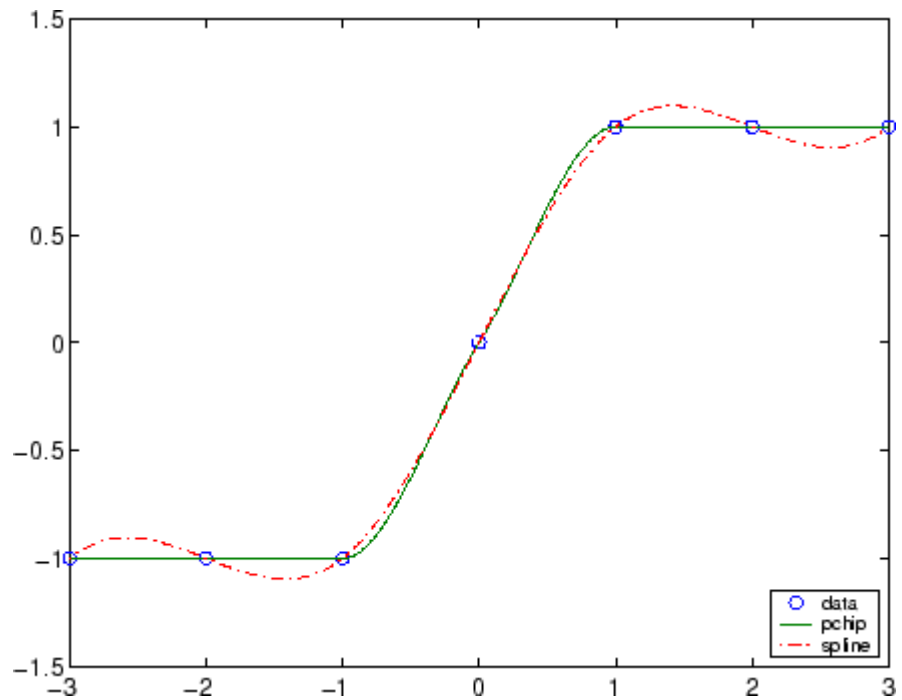
**Tips**

`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`

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Create protected function file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Syntax</b>          | <pre>pcode(fun) pcode(fun1,...,funN) pcode(fun, '-inplace')</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b>     | <p><code>pcode(fun)</code> encrypts the code in <code>fun.m</code> and produces a file called <code>fun.p</code>, known as a P-file. If <code>fun</code> 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 <code>.m</code> file or folder can be anywhere on the search path.</p> <p><code>pcode(fun1,...,funN)</code> creates N P-files from the listed files. If any inputs are folders, then MATLAB creates a P-file for every <code>.m</code> file the folders contain.</p> <p><code>pcode(fun, '-inplace')</code> creates P-files in the same folder as the script or function files.</p>                                         |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• The <code>pcode</code> 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.</li><li>• When encrypting all files in a folder, <code>pcode</code> does not encrypt any files within subfolders.</li><li>• A P-file takes precedence over the corresponding <code>.m</code> file for execution, even after modifications to the <code>.m</code> file.</li><li>• MATLAB does not display any of the help comments that might be in the original <code>.m</code> file.</li></ul> |
| <b>Input Arguments</b> | <p><b>fun</b></p> <p>MATLAB file or directory containing MATLAB files. If <code>fun</code> resides within a package and/or class folder, then <code>pcode</code> creates the same package and/or class structure to house the resulting P-files.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

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)
```

# pcode

---

```
.          ..          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.

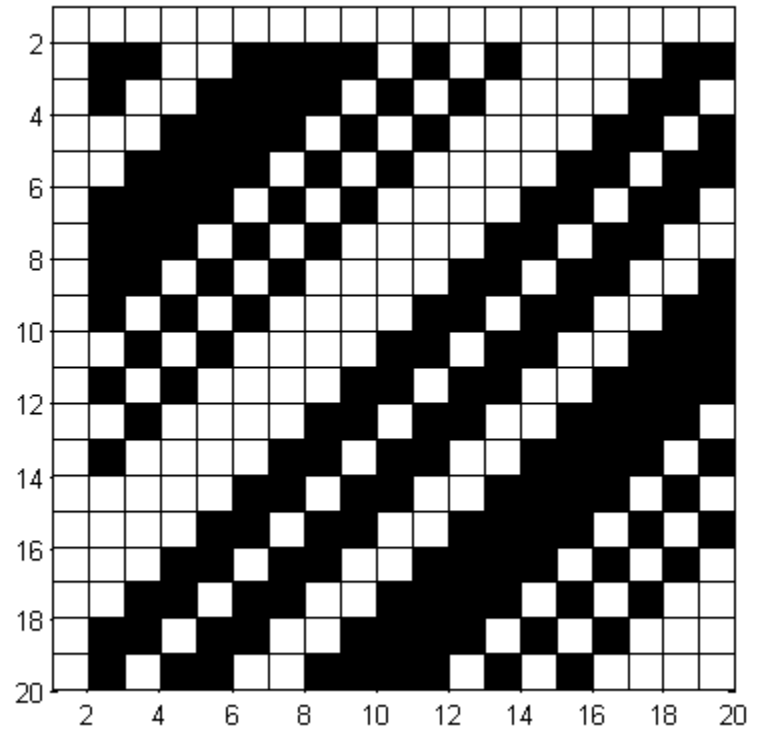
When you use `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices, and all elements of `C` are used.

## 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
```

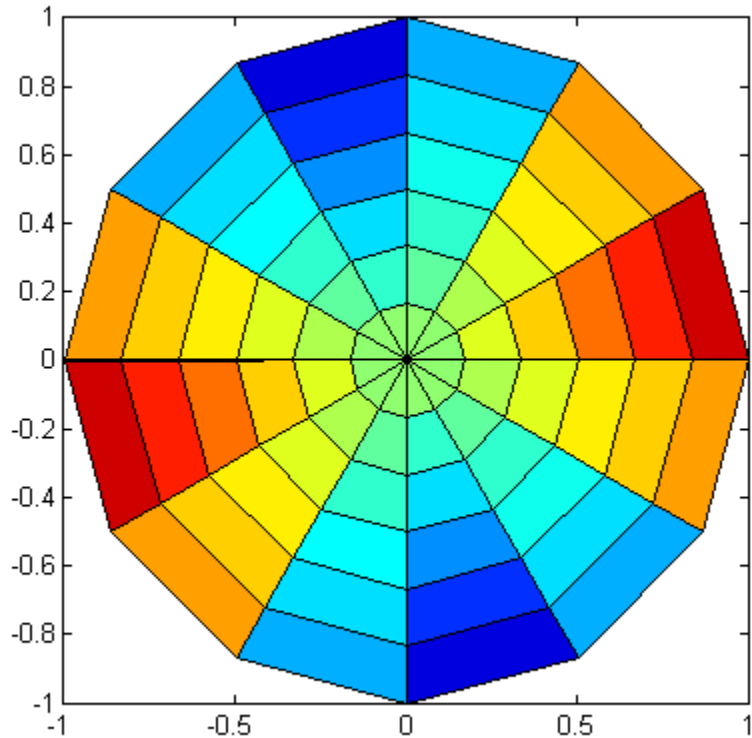




A simple color wheel illustrates a polar coordinate system.

```
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 $x_0 < x_1 < \dots < x_n$ . The length of xmesh must be $\geq 3$ .              |
| 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 $t_0 < t_1 < \dots < t_f$ . The length of tspan must be $\geq 3$ .                        |
| 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:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right) \quad (2-4)$$

The PDEs hold for  $t_0 \leq t \leq t_f$  and  $a \leq x \leq b$ . 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  $\geq 0$ .

In Equation 2-4,  $f(x,t,u,\partial u/\partial x)$  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

$$u(x,t_0) = u_0(x) \quad (2-5)$$

For all  $t$  and either  $x = a$  or  $x = b$ , the solution components satisfy a boundary condition of the form

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 \quad (2-6)$$

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 2-4). 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 2-4).

- `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 2-6). It has the form

$$[p1,q1,pr,qr] = bcfun(xl,u1,xr,ur,t)$$

$u1$  is the approximate solution at the left boundary  $x1 = a$  and  $ur$  is the approximate solution at the right boundary  $xr = b$ .  $p1$  and  $q1$  are column vectors corresponding to  $p$  and  $q$  evaluated at  $x1$ , 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`.

$u_i = \text{ui} = \text{sol}(:, :, i)$  is an approximation to the  $i$ th component of the solution vector  $u$ . The element  $\text{ui}(j, k) = \text{sol}(j, k, i)$  approximates  $u_i$  at  $(t, x) = (\text{tspan}(j), \text{xmesh}(k))$ .

$\text{ui} = \text{sol}(j, :, i)$  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 2-4 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  $0 \leq x \leq 1$  for times  $t \geq 0$ .

The PDE satisfies the initial condition

$$u(x, 0) = \sin \pi x$$

and boundary conditions

$$u(0, t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

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.
```

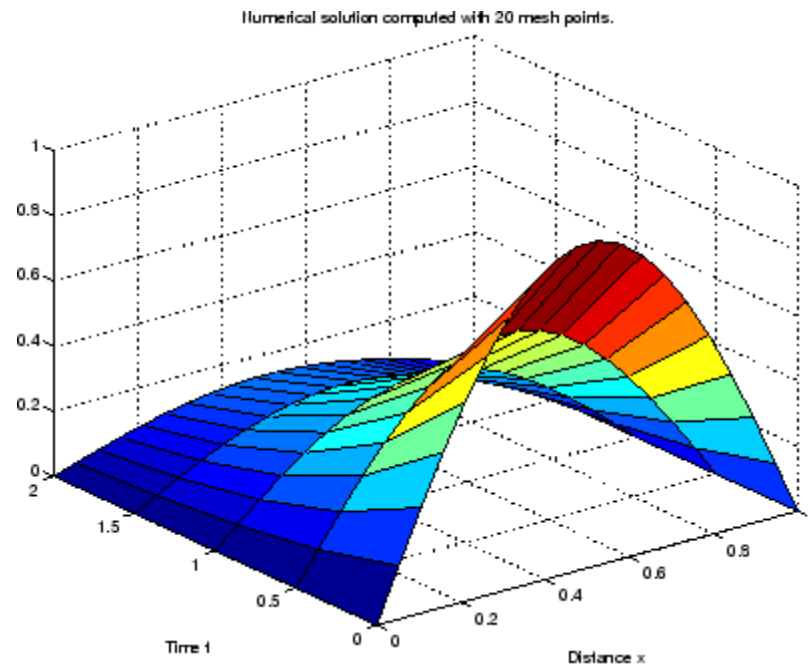
```
u = sol(:,:,1);

% 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')

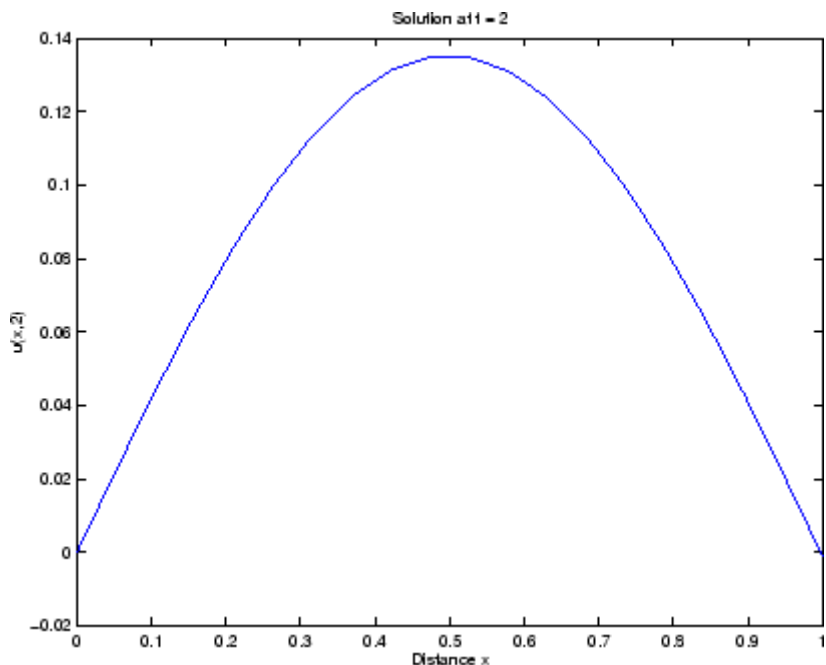
% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
% -----
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
% -----
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% -----
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
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  $0 \leq x \leq 1$  for times  $t \geq 0$ .

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

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot * \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}$$

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

$$\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot * \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot * \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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);

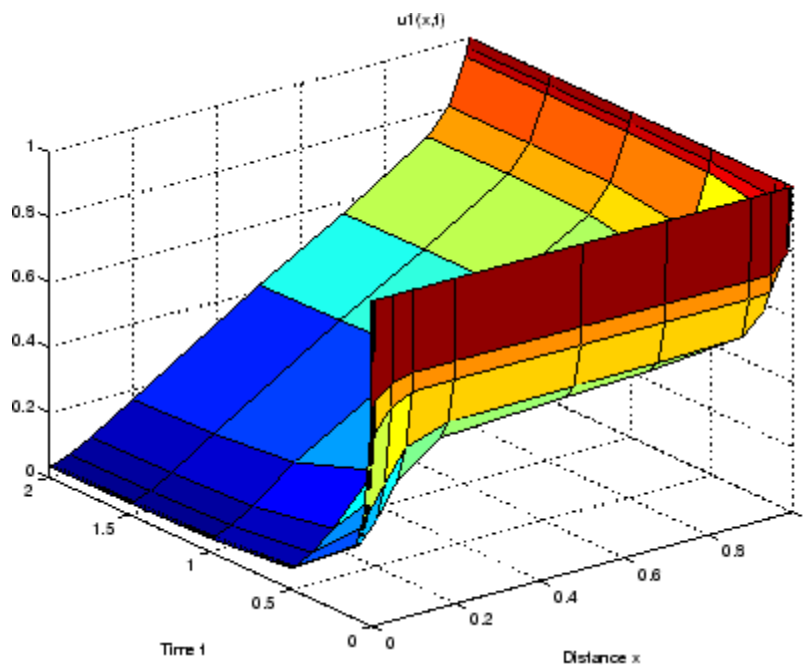
figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

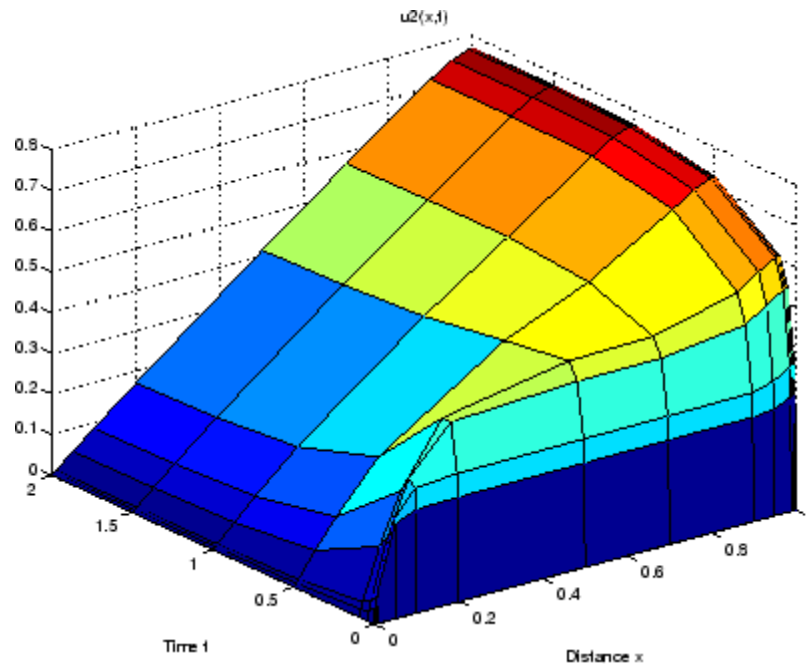
figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% -----
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y) - exp(-11.47*y);
s = [-F; F];
% -----
function u0 = pdex4ic(x);
u0 = [1; 0];
% -----
function [pl,ql,pr,qr] = 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 <code>[x0, x1, ..., xn]</code> specifying the points at which the elements of <code>ui</code> were computed. This is the same vector with which pdepe was called.                               |
| ui    | A vector <code>sol(j,:,i)</code> that approximates component <code>i</code> of the solution at time $t_j$ and mesh points <code>xmesh</code> , where <code>sol</code> is the solution returned by pdepe. |
| xout  | A vector of points from the interval <code>[x0,xn]</code> 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

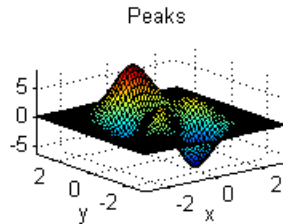
pdepe

# 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,de12(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`.

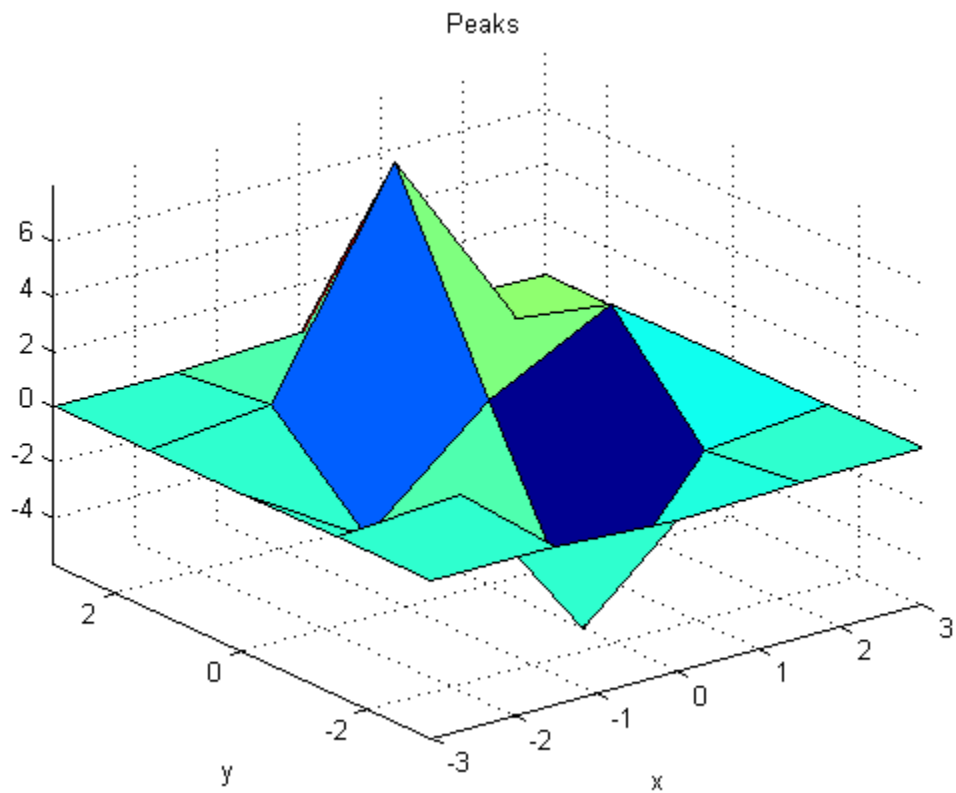
## Examples

This example creates a 5-by-5 matrix of peaks and displays the image:

```
peaks(5);
```

# peaks

---



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.";
```

# perl

---

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**  $P = \text{perms}(v)$

**Description**  $P = \text{perms}(v)$ , where  $v$  is a row vector of length  $n$ , creates a matrix whose rows consist of all possible permutations of the  $n$  elements of  $v$ . Matrix  $P$  contains  $n!$  rows and  $n$  columns.

**Examples** The command `perms([2 4 6])` returns *all* the permutations of the numbers 2, 4, and 6:

|   |   |   |
|---|---|---|
| 6 | 4 | 2 |
| 6 | 2 | 4 |
| 4 | 6 | 2 |
| 4 | 2 | 6 |
| 2 | 4 | 6 |
| 2 | 6 | 4 |

**Limitations** This function is only practical for situations where  $n$  is less than about 11.

**See Also** `nchoosek` | `permute` | `randperm`

# permute

---

**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.

**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`



**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](#)

# pi

---

**Purpose** Ratio of circle's circumference to its diameter

**Syntax** `pi`

**Description** `pi` returns the floating-point number nearest the value of  $\pi$ . The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

**Examples** Find the sine of  $\pi$ :

```
sin(pi)
```

```
returns
```

```
ans =
```

```
1.2246e-16
```

The expression `sin(pi)` is not exactly zero because `pi` is not exactly  $\pi$ .

**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  $X/\text{sum}(X)$  to determine the area of each slice of the pie. If  $\text{sum}(X) = 1$ , the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if  $\text{sum}(X) < 1$ .

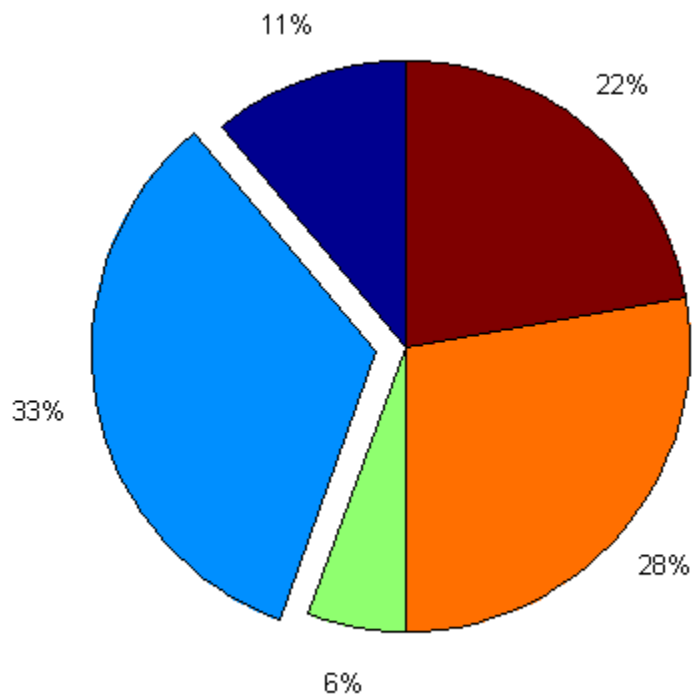
**Examples**

Emphasize the second slice in the chart by setting its corresponding `explode` element to 1.

# pie

---

```
x = [1 3 0.5 2.5 2];  
explode = [0 1 0 0 0];  
pie(x,explode)  
colormap jet
```



## See Also

`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  $X/\text{sum}(X)$  to determine the area of each slice of the pie. If  $\text{sum}(X) = 1$ , the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if  $\text{sum}(X) < 1$ .

**Examples**

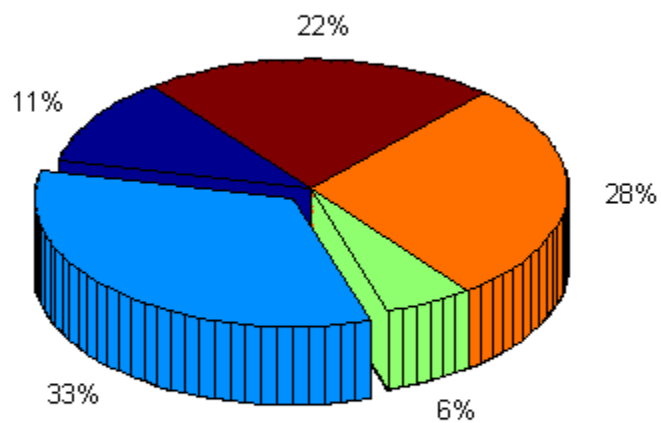
Offset a slice in the pie chart by setting the corresponding `explode` element to 1:

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
```

# pie3

---

```
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  $\text{norm}(x)$  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  $\text{rank}(A) = 3$ .

```
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  $b = 260 \cdot \text{ones}(8, 1)$ ,

```
b =  
    260  
    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  $A \cdot x = b$  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  $\text{norm}(A*x-b)$  and  $\text{norm}(A*y-b)$  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`

# planerot

---

**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`

**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);
```

# audioplayer.play

---

## 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”

**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

# audioplayer.playblocking

---

**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”

# plot

---

**Purpose** 2-D line plot

**Syntax**

```
plot(Y)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineStyle,...,Xn,Yn,LineStyle)
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,LineStyle,...,Xn,Yn,LineStyle)` plots lines defined by the `Xn,Yn,LineStyle` triplets, where `LineStyle` specifies the line type, marker symbol, and color. You can mix `Xn,Yn,LineStyle` triplets with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineStyle,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.

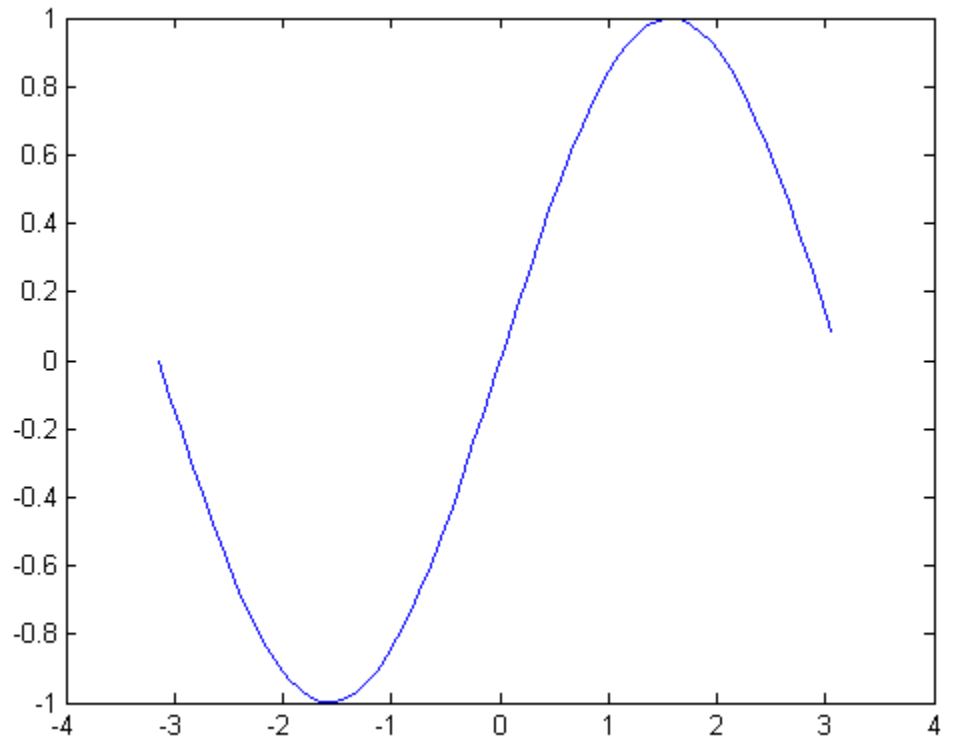
**Examples**

Plot a sine curve.

```
x = -pi:.1:pi;  
y = sin(x);  
plot(x,y)
```

# plot

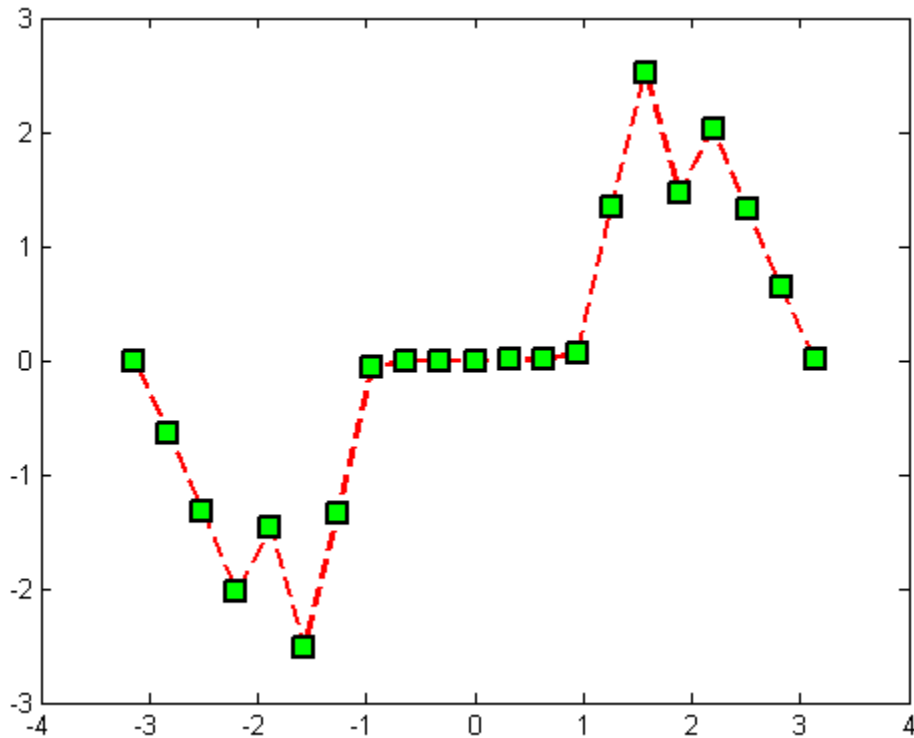
---



---

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)
```



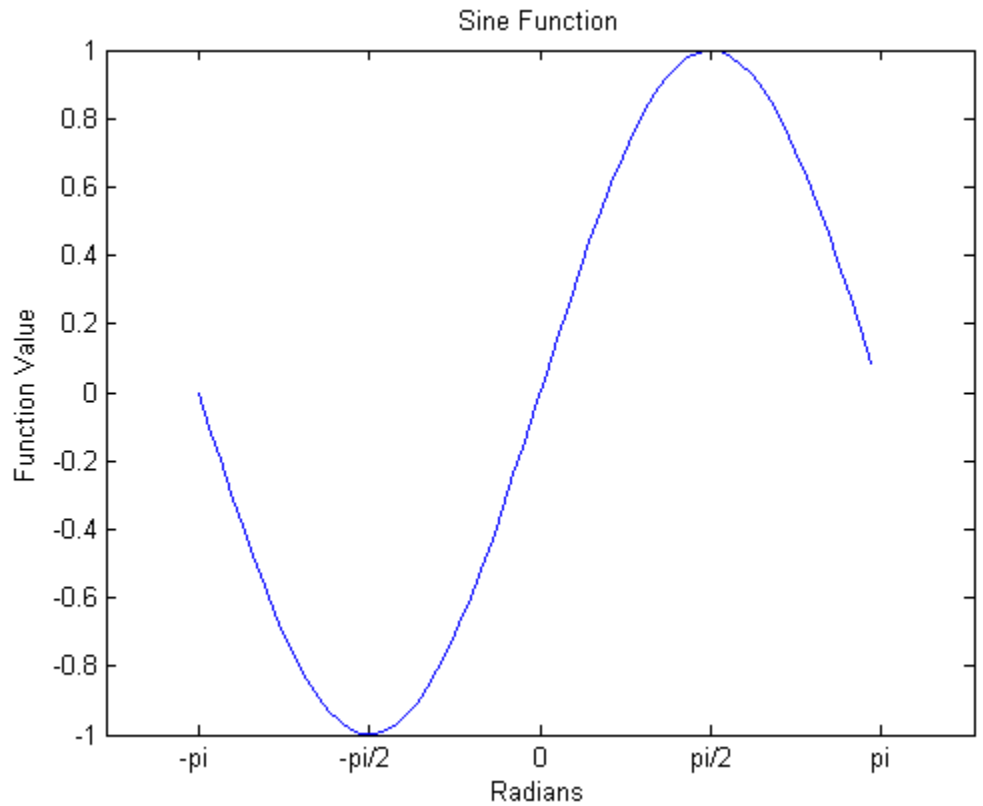
Modify axis tick marks and tick labels and annotate the graph.

```
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'})
```

# plot

---

```
title('Sine Function');  
xlabel('Radians');  
ylabel('Function Value');
```



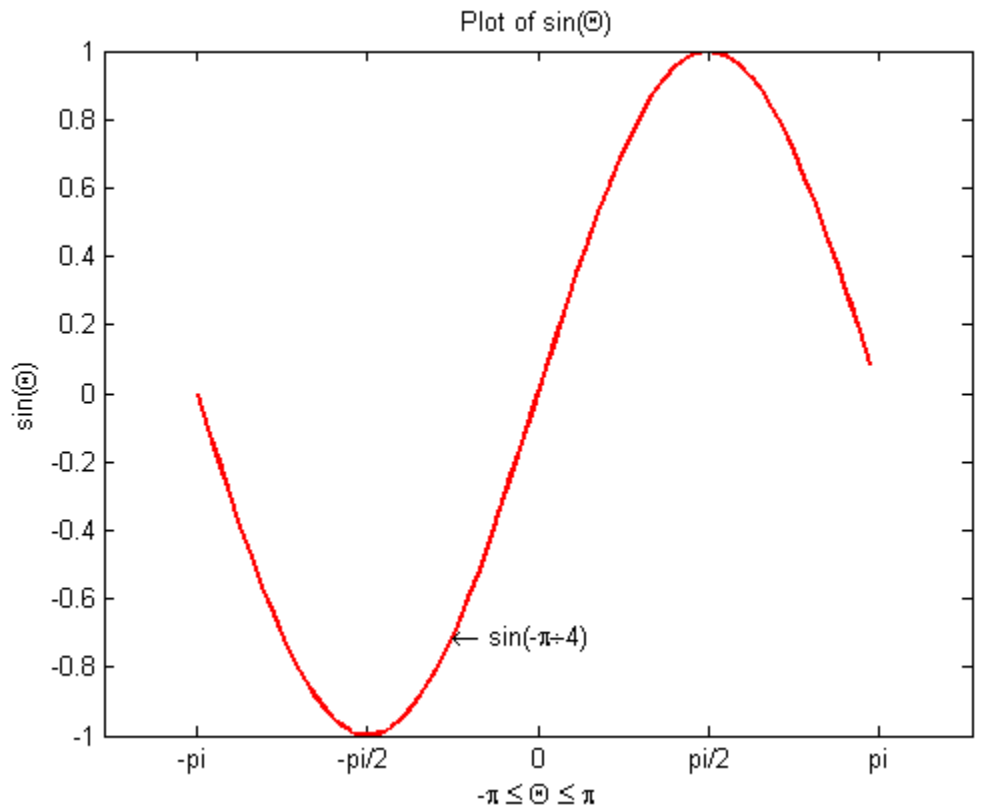
Add a plot title, axis labels, and annotations.

```
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

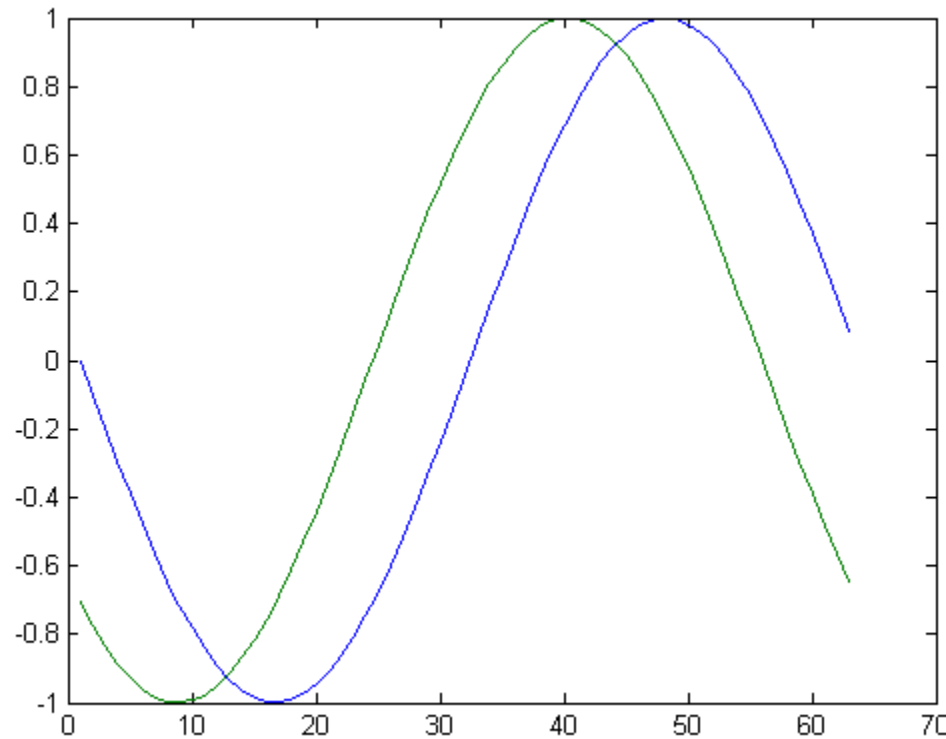
---



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)));
```





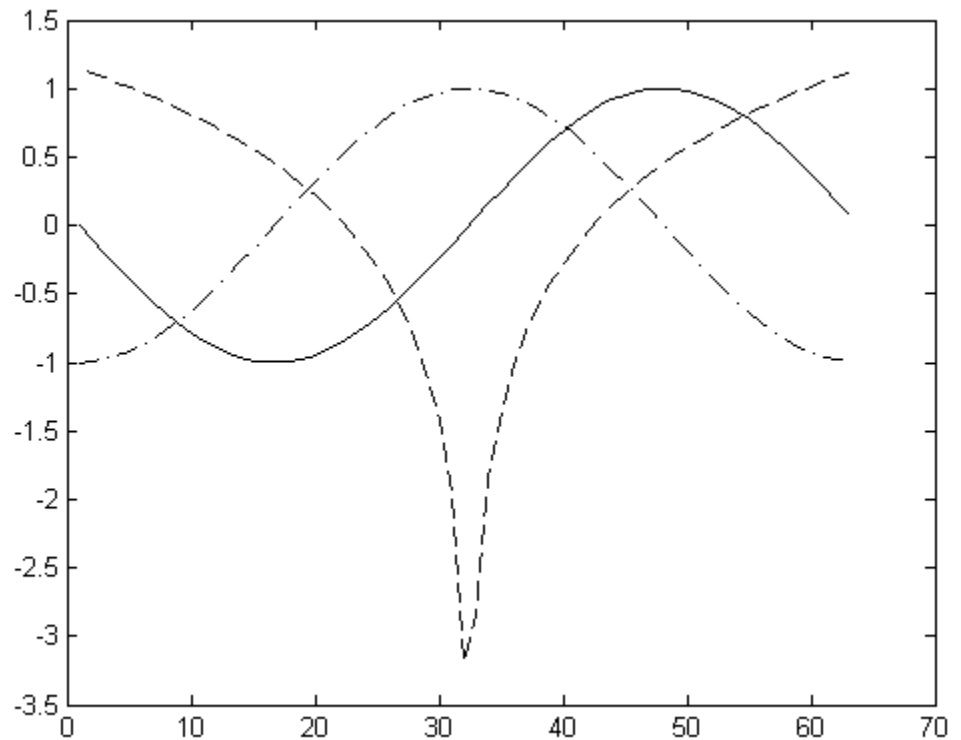
Set line color to be always black and line style order to cycle through solid, dash-dot, dash-dash, and dotted line styles.

```
set(0, 'DefaultAxesColorOrder', [0 0 0], ...  
      'DefaultAxesLineStyleOrder', '-|-|.|-|:|')  
plot(sin(x))  
hold all  
plot(cos(x))  
hold all
```

# plot

---

```
plot(log(abs(x)))
```



## See Also

[axis](#) | [axes](#) | [bar](#) | [gca](#) | [grid](#) | [hold](#) | [legend](#) | [line](#) | [lineseries](#) | [properties](#) | [LineStyle](#) | [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”

# plot3

---

**Purpose** 3-D line plot



**Syntax**

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineStyle,...)
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,LineStyle,...)` 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,

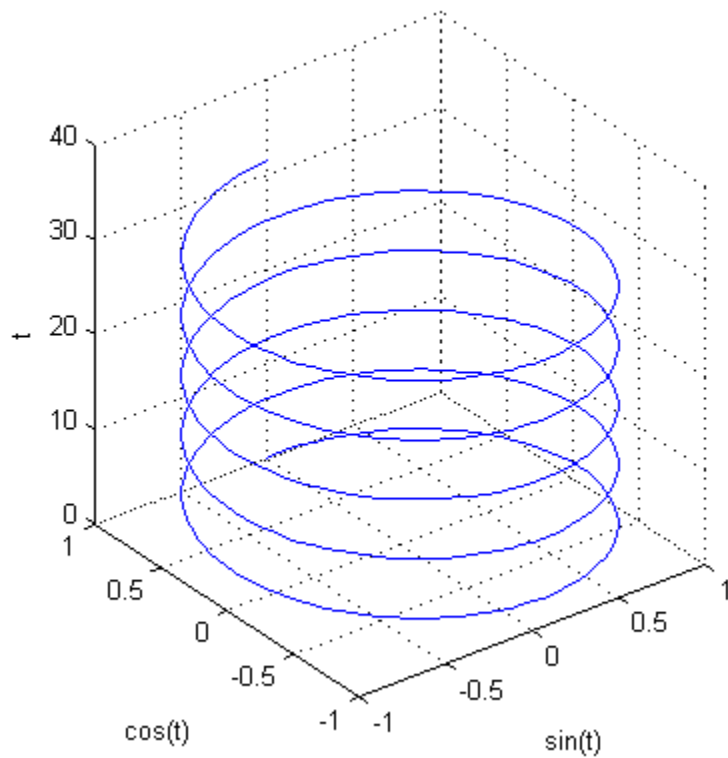
```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineStyle,X3,Y3,Z3)
```

See `LineStyle` and `plot` for information on line types and markers.

**Examples**

Plot a three-dimensional helix.

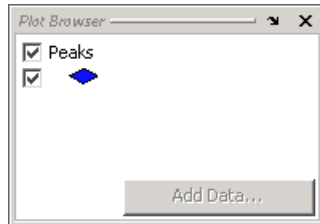
```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
xlabel('sin(t)')
ylabel('cos(t)')
zlabel('t')
grid on
axis square
```



## See Also

[axis](#) | [bar3](#) | [grid](#) | [line](#) | [LineStyle](#) | [lineseries](#) properties | [loglog](#) | [plot](#) | [scatter3](#) | [semilogx](#) | [semilogy](#) | [subplot](#)

**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`

# plottedit

---

**Purpose** Interactively edit and annotate plots

**Syntax**

```
plottedit on
plottedit off
plottedit
plottedit(h)
plottedit('state')
plottedit(h, 'state')
```

**Description** `plottedit 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.

`plottedit off` ends plot mode for the current figure.

`plottedit` toggles the plot edit mode for the current figure.

`plottedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plottedit('state')` specifies the `plottedit` 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 <b>Tools</b> menu in the menu bar  |
| hidetoolsmenu   | Removes the <b>Tools</b> 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.

---



`plottedit(h, 'state')` specifies the plottedit state for figure handle `h`.

## Tips

### Plot Editing Mode Graphical Interface Components

Use these toolbar buttons to add a legend, text, and arrows.

To start plot edit mode, click this button.

Use the Edit, Insert, and Tools menus to add objects or edit existing objects in a graph.

Double-click on an object to select it.

Position labels, legends, and other objects by clicking and dragging.

Access object-specific plot edit functions through context-sensitive pop-up menus.

## Examples

Start plot edit mode for figure 2.

```
plottedit(2)
```

End plot edit mode for figure 2.

```
plottedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

# plottedit

---

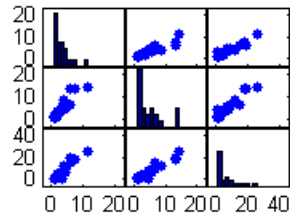
```
plottedit('hidetoolsmenu')
```

## See Also

[axes](#) | [line](#) | [open](#) | [plot](#) | [print](#) | [saveas](#) | [text](#) | [propedit](#)

**Purpose**

Scatter plot matrix

**Syntax**

```
plotmatrix(X,Y)
plotmatrix(X)
plotmatrix(...,'LineStyle')
[H,AX,BigAx,P] = plotmatrix(...)
```

**Description**

`plotmatrix(X,Y)` scatter plots the columns of `X` against the columns of `Y`. If `X` is  $p$ -by- $m$  and `Y` is  $p$ -by- $n$ , `plotmatrix` produces an  $n$ -by- $m$  matrix of axes.

`plotmatrix(X)` is the same as `plotmatrix(X,X)`, except that the diagonal is replaced by `hist(X(:,i))`.

`plotmatrix(...,'LineStyle')` uses a `LineStyle` to create the scatter plot. The default is `'.'`.

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in `H`, a matrix of handles to the individual subaxes in `AX`, a handle to a big (invisible) axes that frames the subaxes in `BigAx`, and a matrix of handles for the histogram plots in `P`. `BigAx` is left as the current axes so that a subsequent `title`, `xlabel`, or `ylabel` command is centered with respect to the matrix of axes.

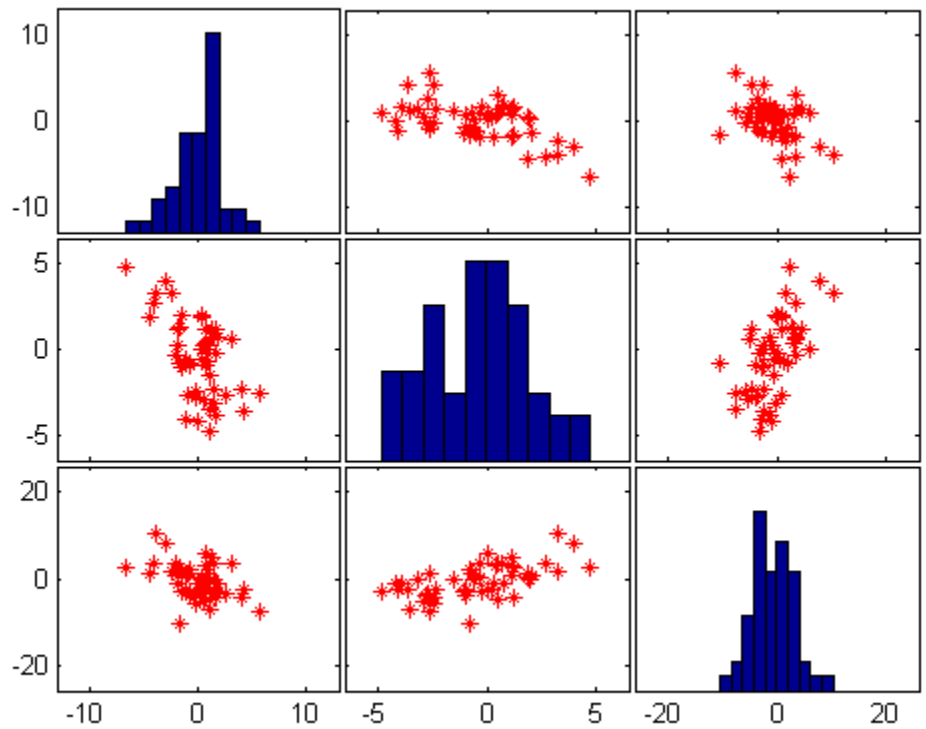
**Examples**

Generate plots of random data.

```
x = randn(50,3);
y = x*[-1 2 1;2 0 1;1 -2 3]';
plotmatrix(y,'*r')
```

# plotmatrix

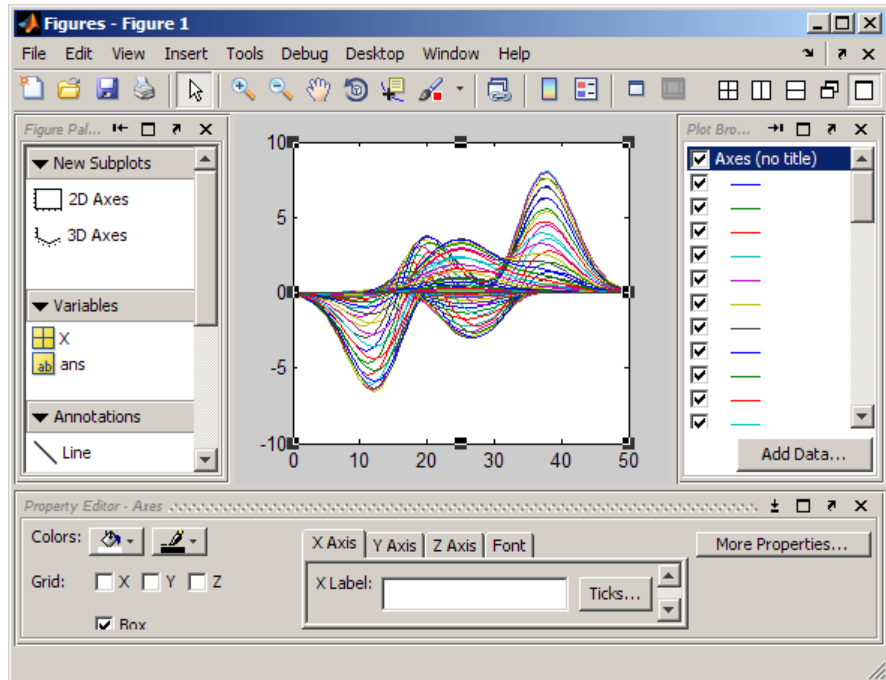
---



## 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

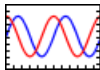
Click the larger **Plotting Tools** icon  on the figure toolbar to collectively enable plotting tools, and the smaller icon  to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see “Plotting 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')
```

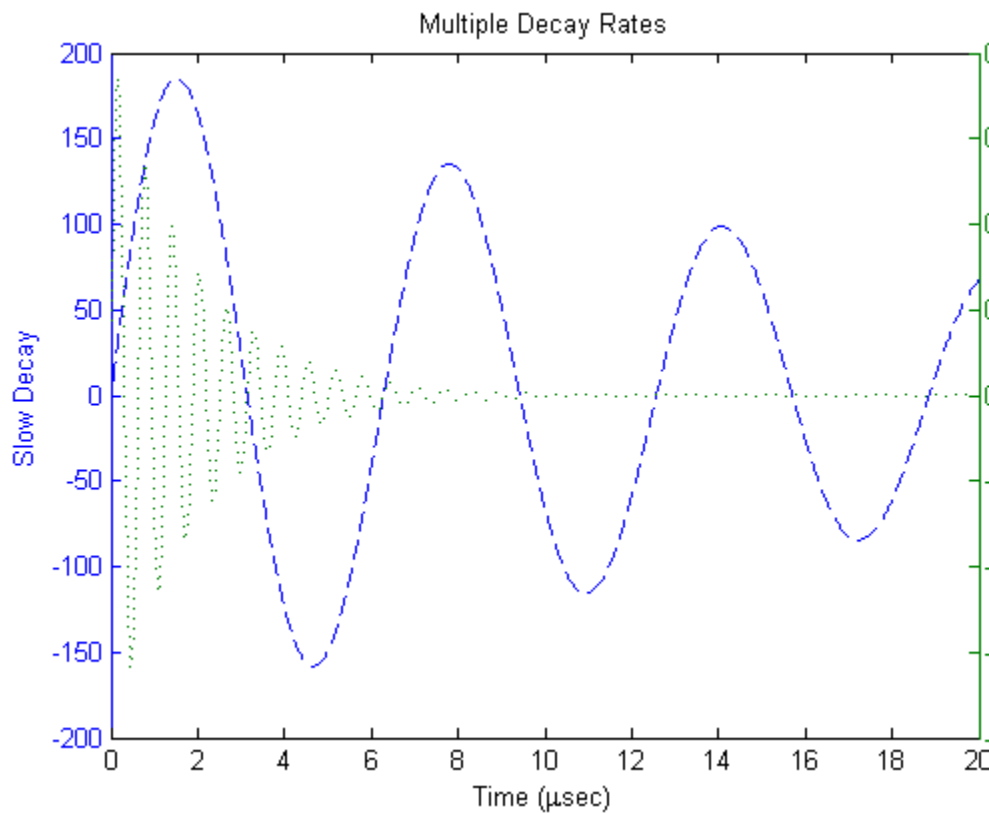
Use the `xlabel` and `title` commands to label the  $x$ -axis and add a title:

```
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

Simplex containing specified location

## 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). |

# DelaunayTri.pointLocation

---

## 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](#)

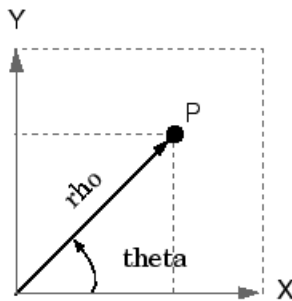
**Purpose** Transform polar or cylindrical coordinates to Cartesian

**Syntax**  
 $[X, Y] = \text{pol2cart}(\text{THETA}, \text{RHO})$   
 $[X, Y, Z] = \text{pol2cart}(\text{THETA}, \text{RHO}, Z)$

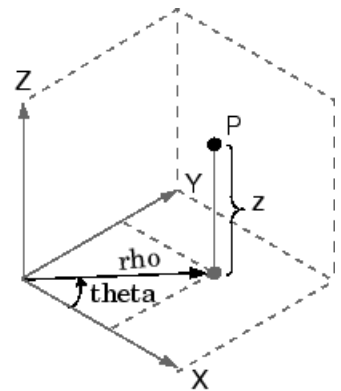
**Description**  $[X, Y] = \text{pol2cart}(\text{THETA}, \text{RHO})$  transforms the polar coordinate data stored in corresponding elements of THETA and RHO to two-dimensional Cartesian, or *xy*, coordinates. The arrays THETA and RHO must be the same size (or either can be scalar). The values in THETA must be in radians.

$[X, Y, Z] = \text{pol2cart}(\text{THETA}, \text{RHO}, Z)$  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**  
 $x = \text{rho} * \cos(\text{theta})$   
 $y = \text{rho} * \sin(\text{theta})$



**Polar to Cartesian Mapping**  
 $x = \text{rho} * \cos(\text{theta})$   
 $y = \text{rho} * \sin(\text{theta})$   
 $z = z$

# pol2cart

---

## See Also

[cart2pol](#) | [cart2sph](#) | [sph2cart](#)

**Purpose**

Polar coordinate plot

**Syntax**

```
polar(theta,rho)
polar(theta,rho,LineStyle)
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,LineStyle)` `LineStyle` 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`.

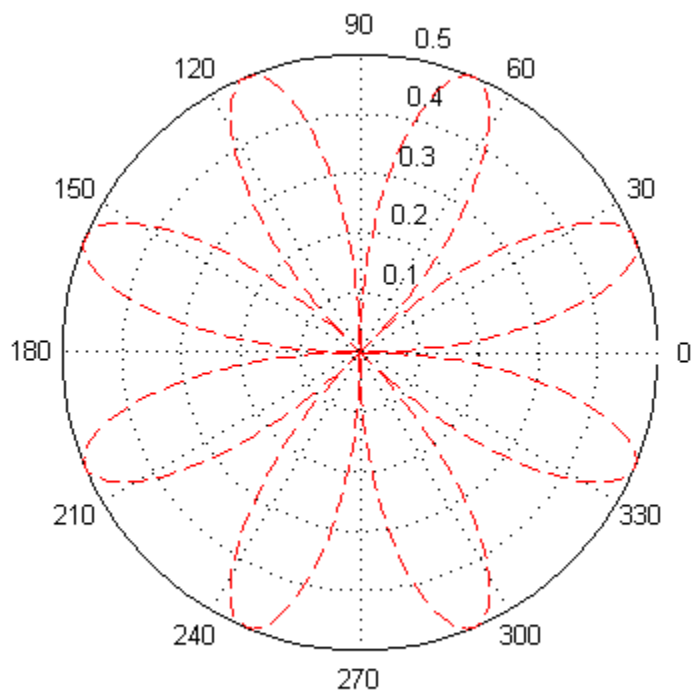
**Tips**

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  $\text{abs}(r)$ .

**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](#) | [LineStyle](#) | [plot](#) | [pol2cart](#) | [rose](#)



**Purpose**

Polynomial with specified roots

**Syntax**

```
p = poly(A)
p = poly(r)
```

**Description**

$p = \text{poly}(A)$  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  $c_1 s^n + \dots + c_n s + c_n + 1$

$p = \text{poly}(r)$  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

$$\det(\lambda I - A) = c_1 \lambda^n + \dots + c_n \lambda + c_{n+1}$$

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.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\dots(\lambda - \lambda_n)$$

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`.

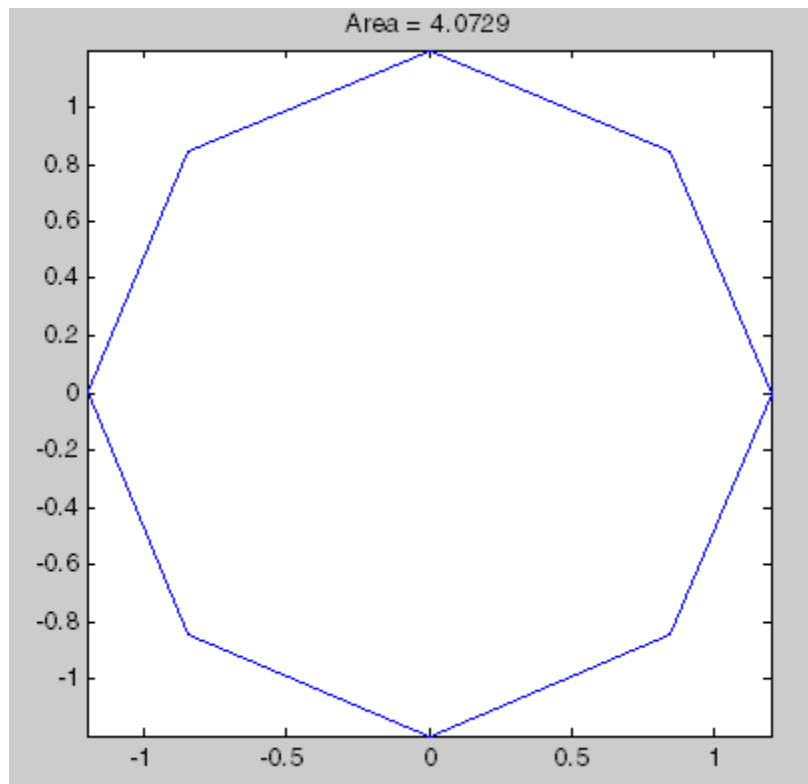
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

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

$$(3x^2 + 6x + 9)(x^2 + 2x)$$

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  $p$

$$(A_0 + \lambda A_1 + \dots + \lambda^p A_p)x = 0$$

where polynomial degree  $p$  is a non-negative integer, and  $A_0, A_1, \dots, A_p$  are input matrices of order  $n$ . The output consists of a matrix  $X$  of size  $n$ -by- $n \times p$  whose columns are the eigenvectors, and a vector  $e$  of length  $n \times p$  containing the eigenvalues.

If  $\lambda$  is the  $j$ th eigenvalue in  $e$ , and  $x$  is the  $j$ th column of eigenvectors in  $X$ , then  $(A_0 + \lambda A_1 + \dots + \lambda^p A_p)x$  is approximately 0.

$e = \text{polyeig}(A_0, A_1, \dots, A_p)$  is a vector of length  $n \times p$  whose elements are the eigenvalues of the polynomial eigenvalue problem.

`[X, e, s] = polyeig(A0,A1,...,AP)` also returns a vector  $s$  of length  $p \times n$  containing condition numbers for the eigenvalues. At least one of  $A_0$  and  $A_p$  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  $a_0, a_1, \dots, a_p$  is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both  $A_0$  and  $A_p$  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  $A_0$  and  $A_p$  is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling  $A_0, A_1, \dots, A_p$  to have  $\text{norm}(A_i)$  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:

$$p(x) = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}.$$

`[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  $(R_{\text{inv}} * R_{\text{inv}}') * \text{normr}^2 / \text{df}$ , 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  $\mu_1 = \text{mean}(x)$  and  $\mu_2 = \text{std}(x)$ . `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,  $\text{erf}(x)$ , by a polynomial in  $x$ . This is a risky project because  $\text{erf}(x)$  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  $\text{erf}(x)$  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

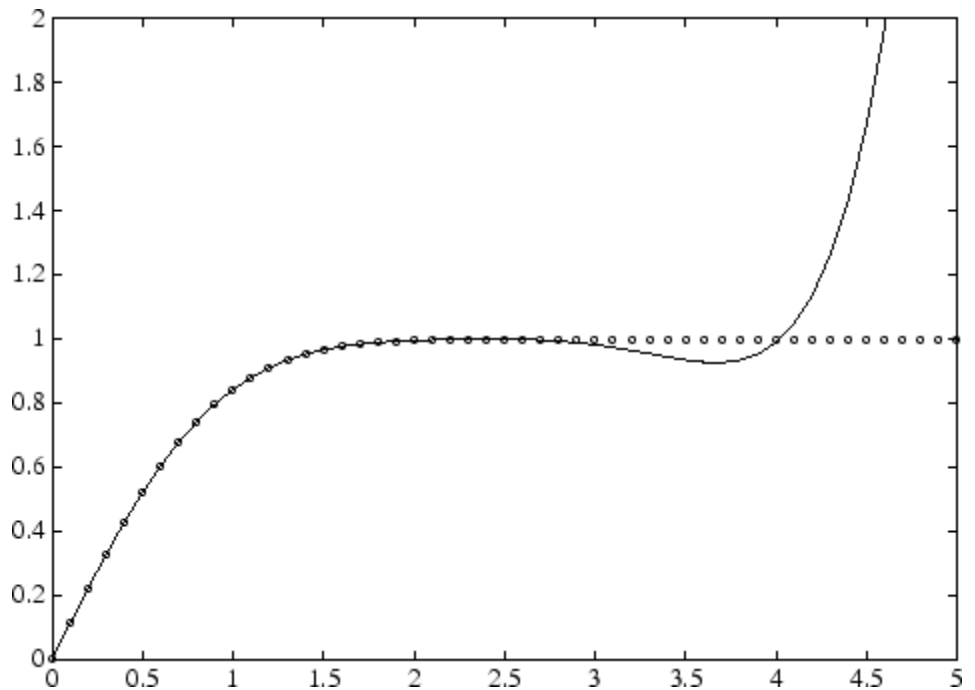
```
table = [x y f y-f]
```

```
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  $x$ .  $v_{i,j} = x_i^{n-j}$

# polyfit

---

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.

$$y = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$$

$x$  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  $y \pm \delta$  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  $\text{mu} = [\mu_1, \mu_2]$  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.

$C = \text{power}(A,B)$  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  $\text{abs}(Y)$  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  $[D1, \dots, Dr]$ -valued, and `xx` is a vector of length  $N$ , then `V` has size  $[D1, \dots, Dr, N]$ , with `V(:, ..., :, J)` the value of  $f$  at `xx(J)`.

If  $f$  is  $[D1, \dots, Dr]$ -valued, and `xx` has size  $[N1, \dots, Ns]$ , then `V` has size  $[D1, \dots, Dr, N1, \dots, Ns]$ , with `V(:, ..., :, J1, ..., Js)` the value of  $f$  at `xx(J1, ..., Js)`.

**Examples** Compare the results of integrating the function `cos`

```
a = 0; b = 10;  
int1 = quad(@cos,a,b)
```

```
int1 =  
    -0.5440
```

with the results of integrating the piecewise polynomial `pp` that approximates the cosine function by interpolating the computed values `x` and `y`.

```
x = a:b;  
y = cos(x);  
pp = spline(x,y);  
int2 = quad(@(x)ppval(pp,x),a,b)
```

```
int2 =
```

-0.5485

`int1` provides the integral of the cosine function over the interval  $[a,b]$ , while `int2` provides the integral over the same interval of the piecewise polynomial `pp`.

**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”

# preferences

---

|                    |                                                                                                                              |
|--------------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Open Preferences dialog box                                                                                                  |
| <b>Syntax</b>      | preferences                                                                                                                  |
| <b>Description</b> | preferences displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products. |
| <b>See Also</b>    | prefdir                                                                                                                      |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Preferences”</li></ul>                                                              |



**Purpose** Generate list of prime numbers

**Syntax** `p = primes(n)`

**Description** `p = primes(n)` returns a row vector of the prime numbers less than or equal to `n`. A prime number is one that has no factors other than 1 and itself.

**Examples** `p = primes(37)`

```
p = 2 3 5 7 11 13 17 19 23 29 31 37
```

**See Also** `factor`

## Purpose

Print figure or save to file

## Contents

“Syntax”

“Description” on page 2-3538

“Printer Drivers” on page 2-3540

“Graphics Format Files” on page 2-3544

“Printing Options” on page 2-3548

“Paper Sizes” on page 2-3551

“Printing Tips” on page 2-3552

“Examples” on page 2-3555

“Alternatives” on page 2-3557

## 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 2-3557 for an example. Also see “Specifying the Figure to Print” on page 2-3555 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:

| <b>Argument</b>                | <b>Description</b>                                                                                                                                                                                                                                                                               |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>handle</code>            | Print the specified object.                                                                                                                                                                                                                                                                      |
| <code>filename</code>          | Direct the output to the PostScript file designated by <code>filename</code> . If <code>filename</code> does not include an extension, <code>print</code> appends an appropriate extension.                                                                                                      |
| <code>-ddriver</code>          | Print the figure using the specified printer <code>driver</code> , (such as color PostScript). If you omit <code>-ddriver</code> , <code>print</code> uses the default value stored in <code>printopt.m</code> . The table in “Printer Drivers” on page 2-3540 lists all supported device types. |
| <code>-dformat</code>          | Copy the figure to the system Clipboard (Microsoft Windows platforms only). To be valid, the <code>format</code> for this operation must be either <code>-dmeta</code> (Windows Enhanced Metafile) or <code>-dbitmap</code> (Windows Bitmap).                                                    |
| <code>-dformat filename</code> | Export the figure to the specified file using the specified graphics <code>format</code> (such as TIFF). The table of “Graphics Format Files” on page 2-3544 lists all supported graphics file formats.                                                                                          |
| <code>-smodelName</code>       | Print the current Simulink model <code>modelName</code> .                                                                                                                                                                                                                                        |
| <code>-options</code>          | Specify print options that modify the action of the <code>print</code> command. (For example, the <code>-noui</code> option suppresses printing of user interface controls.) “Printing Options” on page 2-3548 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 2-3538 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** MathWorks is planning to leverage existing operating system (OS) support for printer drivers and devices. As a result, the ability to specify certain print devices using the `print -d` command, and certain graphics formats using the `print -d` command and/or the `saveas` command, will be removed in a future release. In the following table, the affected formats have an asterisk (\*) next to the `print` command option string. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

| <b>Printer Driver</b>                                              | <b>Print Command Option String</b> | <b>Ghostscript</b> |
|--------------------------------------------------------------------|------------------------------------|--------------------|
| <b>Canon® BubbleJet BJ10e</b>                                      | -dbj10e *                          | Yes                |
| <b>Canon BubbleJet BJ200 color</b>                                 | -dbj200 *                          | Yes                |
| <b>Canon Color BubbleJet BJC-70/BJC-600/BJC-4000</b>               | -dbjc600 *                         | Yes                |
| <b>Canon Color BubbleJet BJC-800</b>                               | -dbjc800 *                         | Yes                |
| <b>Epson® and compatible 9- or 24-pin dot matrix print drivers</b> | -depson *                          | Yes                |

| <b>Printer Driver</b>                                                                                                         | <b>Print Command Option String</b> | <b>Ghostscript</b> |
|-------------------------------------------------------------------------------------------------------------------------------|------------------------------------|--------------------|
| <b>Epson</b> and compatible 9-pin with interleaved lines (triple resolution)                                                  | -deps9high *                       | Yes                |
| <b>Epson LQ-2550</b> and compatible; color (not supported on HP-700)                                                          | -depsonc *                         | Yes                |
| <b>Fujitsu® 3400/2400/1200</b>                                                                                                | -depsonc *                         | Yes                |
| <b>HP® DesignJet 650C</b> color (not supported on Windows )                                                                   | -ddnj650c *                        | Yes                |
| <b>HP DeskJet 500</b>                                                                                                         | -ddjet500 *                        | Yes                |
| <b>HP DeskJet 500C</b> (creates black and white output)                                                                       | -dcdjmono *                        | Yes                |
| <b>HP DeskJet 500C</b> (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows ) | -dcdjcolor *                       | Yes                |
| <b>HP DeskJet 500C/540C</b> color (not supported on Windows )                                                                 | -dcdj500 *                         | Yes                |
| <b>HP Deskjet 550C</b> color (not supported on Windows )                                                                      | -dcdj550 *                         | Yes                |
| <b>HP DeskJet</b> and <b>DeskJet Plus</b>                                                                                     | -ddeskjet *                        | Yes                |
| <b>HP LaserJet</b>                                                                                                            | -dlaserjet *                       | Yes                |

| <b>Printer Driver</b>                                                                         | <b>Print Command Option String</b> | <b>Ghostscript</b> |
|-----------------------------------------------------------------------------------------------|------------------------------------|--------------------|
| <b>HP LaserJet+</b>                                                                           | -dljetplus *                       | Yes                |
| <b>HP LaserJet IIP</b>                                                                        | -dljet2p *                         | Yes                |
| <b>HP LaserJet III</b>                                                                        | -dljet3 *                          | Yes                |
| <b>HP LaserJet 4, 5L and 5P</b>                                                               | -dljet4 *                          | Yes                |
| <b>HP LaserJet 5 and 6</b>                                                                    | -dpxlmono *                        | Yes                |
| <b>HP PaintJet color</b>                                                                      | -dpaintjet *                       | Yes                |
| <b>HP PaintJet XL color</b>                                                                   | -dpjx1 *                           | Yes                |
| <b>HP PaintJet XL color</b>                                                                   | -dpjetx1 *                         | Yes                |
| <b>HP PaintJet XL300 color</b> (not supported on Windows )                                    | -dpjx1300 *                        | Yes                |
| <b>HPGL for HP 7475A and other compatible plotters.</b> (Renderer cannot be set to Z-buffer.) | -dhpgl *                           | No                 |
| <b>IBM 9-pin Proprinter</b>                                                                   | -dibmpro *                         | Yes                |
| <b>PostScript black and white</b>                                                             | -dps                               | No                 |
| <b>PostScript color</b>                                                                       | -dpsc                              | No                 |
| <b>PostScript Level 2 black and white</b>                                                     | -dps2                              | No                 |
| <b>PostScript Level 2 color</b>                                                               | -dpsc2                             | No                 |
| <b>Windows color</b> (Windows only)                                                           | -dwinc                             | No                 |
| <b>Windows monochrome</b> (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 2-3547 and “Resolution Considerations” on page 2-3550.

---

**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.



| <b>Graphics Format</b>                                               | <b>Bitmap or Vector</b> | <b>Print Command Option String</b> | <b>MATLAB or Ghostscript</b> |
|----------------------------------------------------------------------|-------------------------|------------------------------------|------------------------------|
| <b>BMP</b> monochrome BMP                                            | Bitmap                  | -dbmpmono                          | Ghostscript                  |
| <b>BMP</b> 24-bit BMP                                                | Bitmap                  | -dbmp16m                           | Ghostscript                  |
| <b>BMP</b> 8-bit (256-color) BMP (this format uses a fixed colormap) | Bitmap                  | -dbmp256                           | Ghostscript                  |
| <b>BMP</b> 24-bit                                                    | Bitmap                  | -dbmp                              | MATLAB                       |
| <b>EMF</b>                                                           | Vector                  | -dmeta                             | MATLAB                       |
| <b>EPS</b> black and white                                           | Vector                  | -deps                              | MATLAB                       |
| <b>EPS</b> color                                                     | Vector                  | -depssc                            | MATLAB                       |
| <b>EPS</b> Level 2 black and white                                   | Vector                  | -deps2                             | MATLAB                       |
| <b>EPS</b> Level 2 color                                             | Vector                  | -depssc2                           | MATLAB                       |
| <b>HDF</b> 24-bit                                                    | Bitmap                  | -dhdf                              | MATLAB                       |
| <b>ILL</b> (Adobe Illustrator)                                       | Vector                  | -dill                              | MATLAB                       |
| <b>JPEG</b> 24-bit                                                   | Bitmap                  | -djpeg                             | MATLAB                       |
| <b>PBM</b> (plain format) 1-bit                                      | Bitmap                  | -dpbm                              | Ghostscript                  |
| <b>PBM</b> (raw format) 1-bit                                        | Bitmap                  | -dpbmraw                           | Ghostscript                  |
| <b>PCX</b> 1-bit                                                     | Bitmap                  | -dpcxmono                          | Ghostscript                  |

| <b>Graphics Format</b>                                         | <b>Bitmap or Vector</b> | <b>Print Command Option String</b> | <b>MATLAB or Ghostscript</b> |
|----------------------------------------------------------------|-------------------------|------------------------------------|------------------------------|
| <b>PCX</b> 24-bit color PCX file format, three 8-bit planes    | Bitmap                  | -dpcx24b                           | Ghostscript                  |
| <b>PCX</b> 8-bit newer color PCX file format (256-color)       | Bitmap                  | -dpcx256                           | Ghostscript                  |
| <b>PCX</b> Older color PCX file format (EGA/VGA, 16-color)     | Bitmap                  | -dpcx16                            | Ghostscript                  |
| <b>PDF</b> Color PDF file format                               | Vector                  | -dpdf                              | Ghostscript                  |
| <b>PGM</b> Portable Graymap (plain format)                     | Bitmap                  | -dpgm                              | Ghostscript                  |
| <b>PGM</b> Portable Graymap (raw format)                       | Bitmap                  | -dpgmraw                           | Ghostscript                  |
| <b>PNG</b> 24-bit                                              | Bitmap                  | -dpng                              | MATLAB                       |
| <b>PPM</b> Portable Pixmap (plain format)                      | Bitmap                  | -dppm                              | Ghostscript                  |
| <b>PPM</b> Portable Pixmap (raw format)                        | Bitmap                  | -dppmraw                           | Ghostscript                  |
| <b>SVG</b> Scalable Vector Graphics (For Simulink Models Only) | Vector                  | -dsvg                              | MATLAB                       |

| <b>Graphics Format</b>            | <b>Bitmap or Vector</b> | <b>Print Command Option String</b> | <b>MATLAB or Ghostscript</b> |
|-----------------------------------|-------------------------|------------------------------------|------------------------------|
| <b>TIFF 24-bit</b>                | Bitmap                  | -dtiff or -dtiffn                  | MATLAB                       |
| <b>TIFF preview for EPS files</b> | 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.

Naturally, the Windows only `-dwin` and `-dwinc` output formats cannot be used on UNIX or Mac platforms with or without a display.

The same holds true on Windows platforms with the `-noFigureWindows` startup option. The `-dwin`, `-dwinc`, and `-dsetup` options operate as usual under `-noFigureWindows`. However, the `printpreview` GUI does not function in this mode.

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 2-3550 for information on controlling output resolution.

| Option                  | Description                                                                                                                           |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>-adobecset</code> | PostScript devices only. Use PostScript default character set encoding. See “Early PostScript 1 Printers”.                            |
| <code>-append</code>    | PostScript devices only. Append figure to existing PostScript file. See “Settings That Are Driver Specific”.                          |
| <code>-cmyk</code>      | PostScript devices only. Print with CMYK colors instead of RGB. See “Setting CMYK Color”.                                             |
| <code>-ddriver</code>   | Printing only. Printer driver to use. See “Printer Drivers” on page 2-3540 table.                                                     |
| <code>-dformat</code>   | Exporting only. Graphics format to use. See “Graphics Format Files” table.                                                            |
| <code>-dsetup</code>    | Windows printing only. Display the (platform-specific) Print Setup dialog. Settings you make in it are saved, but nothing is printed. |

| Option        | Description                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -fhandle      | Handle of figure to print. Note that you cannot specify both this option and the <i>-swindowtitle</i> 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 <i>-zbuffer</i> or <i>-painters</i> . See “Selecting a Renderer”.                                                                                                                                                                                                                                                       |
| -painters     | Render using the Painter’s algorithm. Note that you cannot specify this method in conjunction with <i>-zbuffer</i> or <i>-opengl</i> . See “Selecting a Renderer”.                                                                                                                                                                                                                                                      |
| -Pprinter     | Specify name of printer to use. See “Selecting the Printer”.                                                                                                                                                                                                                                                                                                                                                            |
| -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 <i>-r0</i> to specify screen resolution. For details, see “Resolution Considerations” on page 2-3550 and “Setting the Resolution”. |
| -swindowtitle | Specify name of Simulink system window to print. Note that you cannot specify both this option and the <i>-fhandle</i> option. See “Which Figure Is Printed”.                                                                                                                                                                                                                                                           |

| Option   | Description                                                                                                                                         |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| -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 2-3544).

---

**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 2-3547 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          |

| Property Value | Size (Width by Height) |
|----------------|------------------------|
| 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        |
| 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 2-3548 and “Printing Interpolated Shading with PostScript Drivers” on page 2-3554 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 array
                                        % each element is a string
```

## Tiff Preview

The command

```
print('-depsec', '-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`

# printopt

---

**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

**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 `uimenu`s 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 `advprinter` on MATLAB

---

---

**Note** A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowState` in the MATLAB Figure Properties.

---

**See Also** `print` | `printopt` | `printpreview`

# printpreview

---

**Purpose** Preview figure to print

## **Contents**

“Description” on page 2-3560

“Right Pane Controls” on page 2-3561

“The Layout Tab” on page 2-3562

“The Lines/Text Tab” on page 2-3563

“The Color Tab” on page 2-3565

“The Advanced Tab” on page 2-3567

“Alternatives” on page 2-3568

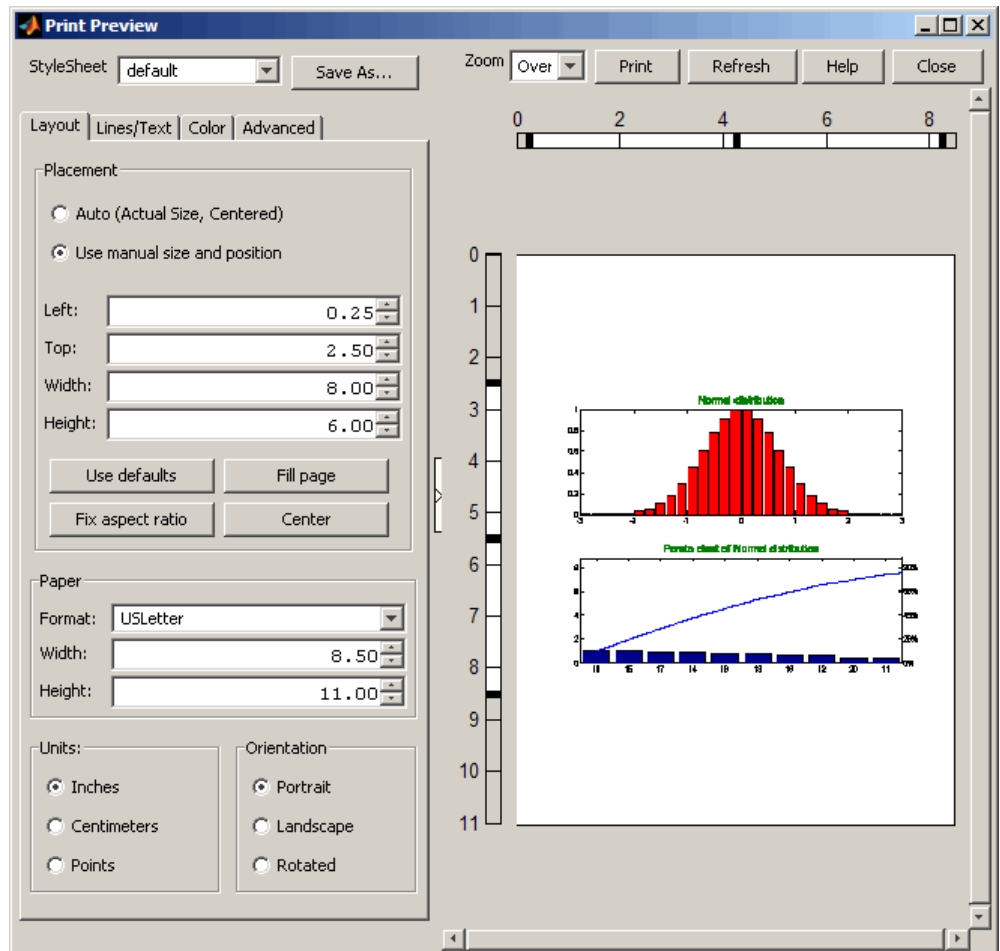
**Syntax**  
`printpreview`  
`printpreview(f)`

**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   | <b>Auto</b>                     | Let MATLAB decide placement of plot on page*          |
|             | <b>Use manual...</b>            | Specify position parameters for plot on page*         |
|             | <b>Top, Left, Width, Height</b> | Standard position parameters in current units         |
|             | <b>Use defaults</b>             | Revert to default position                            |
|             | <b>Fill page</b>                | Expand figure to fill printable area (see note below) |
|             | <b>Fix aspect ratio</b>         | Correct height/width ratio                            |
|             | <b>Center</b>                   | Center plot on printed page                           |
| Paper       | <b>Format</b>                   | U.S. and ISO® sheet size selector                     |
|             | <b>Width, Height</b>            | Sheet size in current units                           |
| Units       | <b>Inches</b>                   | Use inches as units for dimensions and positions      |
|             | <b>Centimeters</b>              | Use centimeters as units for dimensions and positions |
|             | <b>Points</b>                   | Use points as units for dimensions and positions      |
| Orientation | <b>Portrait</b>                 | Upright paper orientation                             |

| Group | Option           | Description                            |
|-------|------------------|----------------------------------------|
|       | <b>Landscape</b> | Sideways paper orientation             |
|       | <b>Rotated</b>   | Currently the same as <b>Landscape</b> |

\* 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.

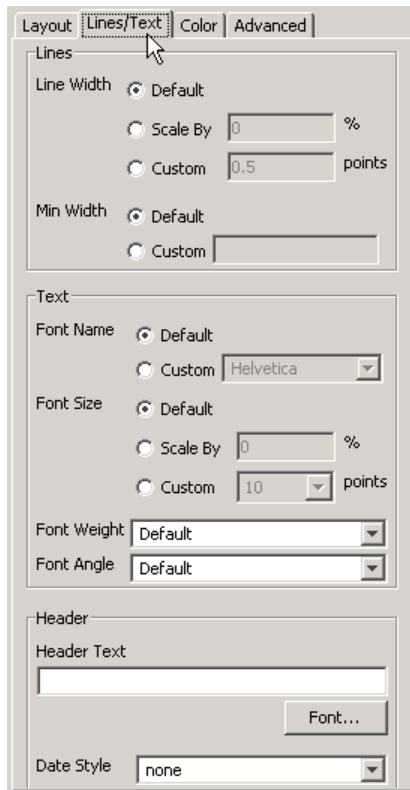
---

**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:

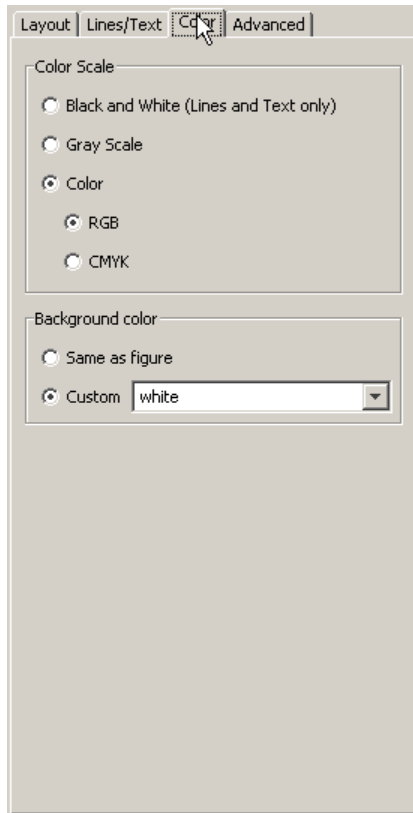


| Group | Option            | Description                                                                                                                                         |
|-------|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Lines | <b>Line Width</b> | 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 |
|       | <b>Min Width</b>  | Smallest line width (in points) to use when printing; defaults to 0.5 point                                                                         |
| Text  | <b>Font Name</b>  | Select a system font for all text on plot, or default to fonts currently used on the plot                                                           |

| Group  | Option             | Description                                                                                                                   |
|--------|--------------------|-------------------------------------------------------------------------------------------------------------------------------|
|        | <b>Font Size</b>   | Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this  |
|        | <b>Font Weight</b> | Select Normal ... Bold font styling for all text from drop-down menu or default to the font weights used on the plot          |
|        | <b>Font Angle</b>  | 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 | <b>Header Text</b> | Type the text to appear on the header at the upper left of printed pages, or leave blank for no header                        |
|        | <b>Date Style</b>  | 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:

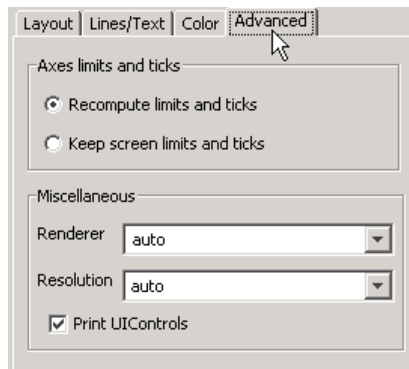


| Group       | Option                 | Description                                                                                    |
|-------------|------------------------|------------------------------------------------------------------------------------------------|
| Color Scale | <b>Black and White</b> | Select to print lines and text in black and white, but use color for patches and other objects |
|             | <b>Gray Scale</b>      | Convert colors to shades of gray on printed pages                                              |

| Group            | Option                | Description                                                                                                                      |
|------------------|-----------------------|----------------------------------------------------------------------------------------------------------------------------------|
|                  | <b>Color</b>          | Print everything in color, matching colors on plot; select RGB (default) or CMYK color model for printing                        |
| Background Color | <b>Same as figure</b> | Print the figure's background color as it is                                                                                     |
|                  | <b>Custom</b>         | 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 | <b>Recompute limits and ticks</b> | Redraw $x$ - and $y$ -axes ticks and limits based on printed plot size (default)                                      |
|                       | <b>Keep limits and ticks</b>      | Use the $x$ - and $y$ -axes ticks and limits shown on the plot when printing the previewed figure                     |
| Miscellaneous         | <b>Renderer</b>                   | Select a rendering algorithm for printing: painters, zbuffer, opengl, or auto (default)                               |
|                       | <b>Resolution</b>                 | Select resolution to print at in dots per inch: 150, 300, 600, or auto (default), or type in any other positive value |
|                       | <b>Print UIControls</b>           | 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



**Purpose**

Product of array elements

**Syntax**

```
B = prod(A)
B = prod(A,dim)
B = prod(A,'double')
B = prod(A,'native')
```

**Description**

`B = prod(A)` returns the products along different dimensions of an array. `A` can be `double` or `single`. `B` is multiplied natively, that is in the same class as `A`, and `B` has the same class as `A`.

If `A` is a vector, `prod(A)` returns the product of the elements.

If `A` is a matrix, `prod(A)` treats the columns of `A` as vectors, returning a row vector of the products of each column.

If `A` is a multidimensional array, `prod(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = prod(A,dim)` takes the products along the dimension of `A` specified by scalar `dim`.

`B = prod(A,'double')` multiplies `B` in double and `B` has class `double`, even if `A` is `single`.

`B = prod(A,'native')` multiplies `B` natively and `B` has the same class as `A`.

**Examples**

The magic square of order 3 is

```
M = magic(3)
```

```
M =
     8     1     6
     3     5     7
     4     9     2
```

The product of the elements in each column is

# prod

---

```
prod(M) =
```

```
    96    45    84
```

The product of the elements in each row can be obtained by:

```
prod(M,2) =
```

```
    48  
   105  
    72
```

## See Also

```
cumprod | diff | sum
```

**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 <code>FunctionName</code> , including local functions                                                                             |
| FunctionName       | Function name; includes local functions                                                                                                        |
| FileName           | Full path to <code>FunctionName</code> , 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           | <code>FunctionTable</code> indices to child functions                                                                                          |
| Parents            | <code>FunctionTable</code> indices to parent functions                                                                                         |

| Field         | Description                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ExecutedLines | <p>Array containing line-by-line details for the function being profiled.</p> <p>Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix.</p> <p>Column 2: Number of times the line was executed</p> <p>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.</p> |
| 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
```



end

## See Also

`depcdir` | `depfun` | `mlint` | `profsave`

## How To

- [Profiling for Improving Performance](#)
- [“Profiling Parallel Code”](#)

# profsave

---

**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.

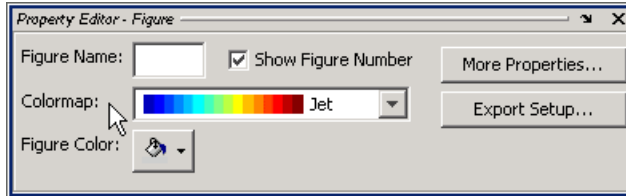
```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

**See Also** `profile`

**How To**

- [Profiling for Improving Performance](#)

**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` | `plottedit` | `propertyeditor`

# propedit (COM)

---

|                    |                                                                                                                                                                                                                                                                                 |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Open built-in property page for control                                                                                                                                                                                                                                         |
| <b>Syntax</b>      | <code>h.propedit</code><br><code>propedit(h)</code>                                                                                                                                                                                                                             |
| <b>Description</b> | <p><code>h.propedit</code> 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.</p> <p><code>propedit(h)</code> is an alternate syntax for the same operation.</p> |
| <b>Tips</b>        | COM functions are available on Microsoft Windows systems only.                                                                                                                                                                                                                  |
| <b>Examples</b>    | Create a Microsoft Calendar control and display its property page:<br><br><pre>cal = actxcontrol('mscal.calendar', [0 0 500 500]);<br/>cal.propedit</pre>                                                                                                                       |
| <b>See Also</b>    | <code>inspect   get (COM)</code>                                                                                                                                                                                                                                                |

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Class property names                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>      | <pre>properties('classname') properties(obj) p = properties(...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Description</b> | <p><code>properties('classname')</code> displays the names of the public properties for the MATLAB class named by <code>classname</code>. The <code>properties</code> function also displays inherited properties.</p> <p><code>properties(obj)</code> <code>obj</code> can be either a scalar object or an array of objects. When <code>obj</code> is scalar, <code>properties</code> also returns dynamic properties. See “Dynamic Properties — Adding Properties to an Instance” for information on using dynamic properties.</p> <p><code>p = properties(...)</code> returns the property names in a cell array of strings.</p> |
| <b>Definitions</b> | <p>A property is public when its <code>GetAccess</code> attribute value is <code>public</code> and its <code>Hidden</code> attribute value is <code>false</code> (default values for these attributes). See “Property Attributes” for a complete list of attributes.</p> <p><code>properties</code> is also a MATLAB class-definition keyword. See <code>classdef</code> for more information on class definition keywords.</p>                                                                                                                                                                                                     |
| <b>Examples</b>    | <p>Retrieve the names of the public properties of class <code>memmapfile</code> and store the result in a cell array of strings:</p> <pre>p = properties('memmapfile'); p ans =      'writable'     'offset'     'format'     'repeat'     'filename'</pre>                                                                                                                                                                                                                                                                                                                                                                         |

# properties

---

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
```

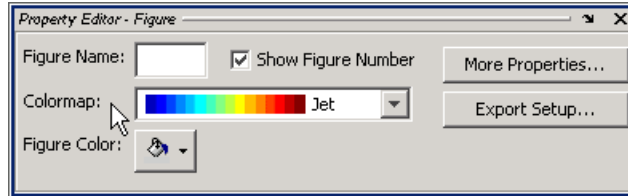
## See Also

fieldnames | events | methods

## Tutorials

- “Properties”

**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”.

# propertyeditor

---

## **See Also**

`plottools` | `plotbrowser` | `figurepalette` | `inspect`



**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  $\psi$  function for each element of array `X`. `X` must be real and nonnegative. The  $\psi$  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  $\psi$  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  $\psi$ , evaluated at `X(j)`.

**Examples** **Example 1**

Use the `psi` function to calculate Euler's constant,  $\gamma$ .

```
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

**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 the MATLAB file, `file`, in HTML format for sharing your code. For example, `publish('myfile.m')` executes the code in `myfile.m`, and saves the formatted code, MATLAB results, and comments 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. For example, 'myfile.m'

### **format - Output format of published file**

'html' (default) | 'doc' | 'latex' | 'ppt' | 'xml'

Output format of published MATLAB file, specified as a string.

**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, 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`.

**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'<br>(default) | Included           | Excluded | Matches screen   | White          |
| 'print'                        | Excluded           | Excluded | White            | White          |
| '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.

(Continued)

| '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'                                                                                                                                                                                                                                   |
| '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: <ul style="list-style-type: none"> <li>• 'getframe'</li> <li>• 'entireFigureWindow'</li> <li>• 'entireGUIWindow' and the snapped window is a GUI window</li> </ul> |

(Continued)

|       |                                                                                                                                                  | 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 'showCode' 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

---

```
'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

- “Publish MATLAB Code”
- “Publishing Markup”

## 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
```

# PutCharArray

---

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](#)

# PutFullMatrix

---

**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
```

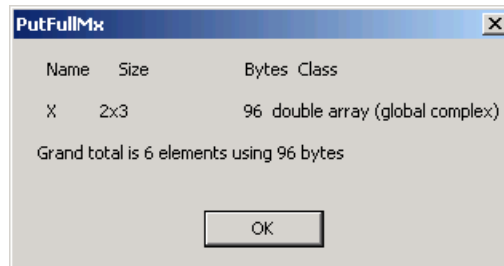
# PutFullMatrix

---

```
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”

## 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
```

# PutWorkspaceData

---

```
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”

|                     |                                                                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | Identify current folder                                                                                                                                         |
| <b>Syntax</b>       | <code>pwd</code><br><code>currentFolder = pwd</code>                                                                                                            |
| <b>Description</b>  | <code>pwd</code> displays the MATLAB current folder.<br><code>currentFolder = pwd</code> returns the current folder as a string to <code>currentFolder</code> . |
| <b>Alternatives</b> | <ul style="list-style-type: none"><li>• Use the Current Folder toolbar.</li></ul>                                                                               |
| <b>See Also</b>     | <code>cd</code>   <code>dir</code>                                                                                                                              |

**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  $\text{norm}(b-A^*x)/\text{norm}(b)$  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  $\text{inv}(M) * A^*x = \text{inv}(M) * b$  for  $x$ . If  $M$  is `[]` then `qmr` applies no



preconditioner.  $M$  can be a function handle `mfun` such that `mfun(x, 'notransp')` returns  $Mx$  and `mfun(x, 'transp')` returns  $M'x$ .

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If  $x_0$  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 <code>tol</code> within <code>maxit</code> iterations.                        |
| 1    | qmr iterated <code>maxit</code> 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 <code>qmr</code> 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  $\|b-Ax\|/\|b\|$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which  $x$  was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including  $\|b-Ax_0\|$ .

## 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 = 4 * x;
    else
        y = A * x;
    end
```

```

        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
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:

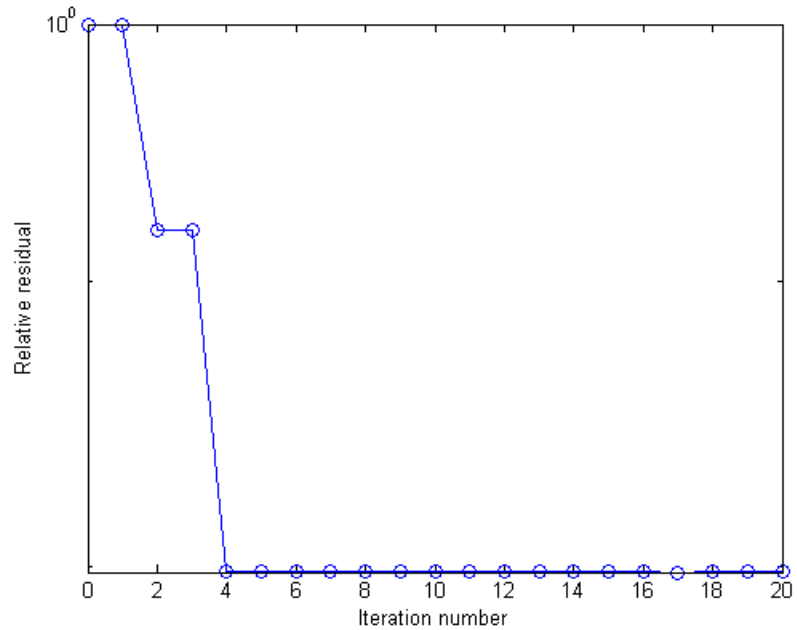
```
[x0,f10,rr0,it0,rv0] = qmr(A,b,tol,maxit);
```

`f10` 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.

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:

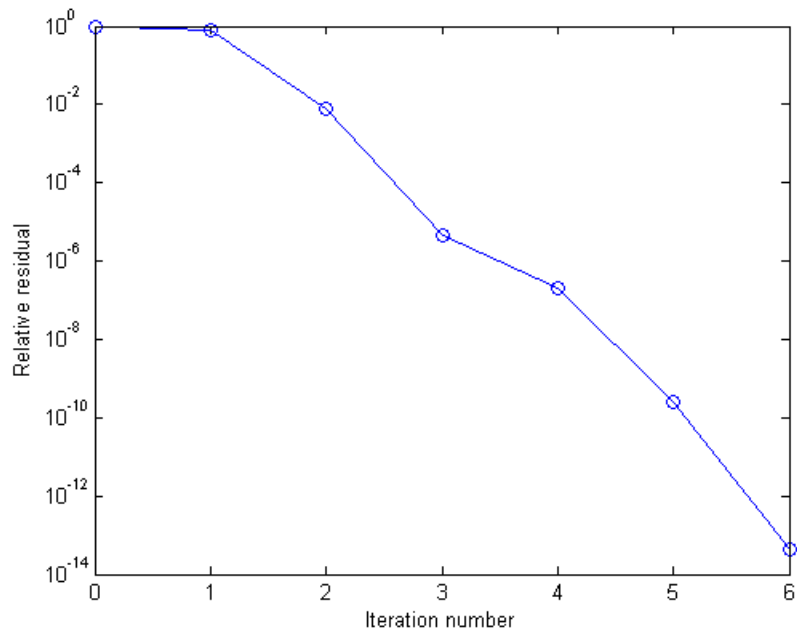
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));  
[x1,f11,rr1,it1,rv1] = qmr(A,b,tol,maxit,L,U);
```

f11 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 Noël M. Nachtigal, "QMR: A quasi-minimal residual method for non-Hermitian linear systems," *SIAM Journal: Numer. Math.* 60, 1991, pp. 315–339.

## See Also

`bigc` | `bigstab` | `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  $m \leq n$ , 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  $A(:,e) = Q*R$ .



$X = \text{qr}(A)$  and  $X = \text{qr}(A,0)$  return a matrix  $X$  such that  $\text{triu}(X)$  is the upper triangular factor  $R$ .

If  $A$  is sparse:

$R = \text{qr}(A)$  computes a Q-less QR decomposition and returns the upper triangular factor  $R$ . Note that  $R = \text{chol}(A' * A)$ . Since  $Q$  is often nearly full, this is preferred to  $[Q,R] = \text{QR}(A)$ .

$R = \text{qr}(A,0)$  produces economy-size  $R$ . If  $m > n$ ,  $R$  has only  $n$  rows. If  $m \leq n$ , this is the same as  $R = \text{qr}(A)$ .

$[Q,R,E] = \text{qr}(A)$  or  $[Q,R,E] = \text{qr}(A, 'matrix')$  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  $R$ .

$[Q,R,e] = \text{qr}(A, 'vector')$  returns the permutation information as a vector instead of a matrix. That is,  $e$  is a row vector such that  $A(:, e) = Q * R$ .

$[Q,R,e] = \text{qr}(A,0)$  produces an economy-size decomposition in which  $e$  is a permutation vector, so that  $A(:, e) = Q * R$ .

$[C,R] = \text{qr}(A,B)$ , where  $B$  has as many rows as  $A$ , returns  $C = Q' * B$ . The least-squares solution to  $A * X = B$  is  $X = R \setminus C$ .

$[C,R,E] = \text{qr}(A,B)$  or  $[C,R,E] = \text{qr}(A,B, 'matrix')$ , also returns a fill-reducing ordering. The least-squares solution to  $A * X = B$  is  $X = E * (R \setminus C)$ .

$[C,R,e] = \text{qr}(A,B, 'vector')$  returns the permutation information as a vector instead of a matrix. That is, the least-squares solution to  $A * X = B$  is  $X(e, :) = R \setminus C$ .

$[C,R] = \text{qr}(A,B,0)$  produces economy-size results. If  $m > n$ ,  $C$  and  $R$  have only  $n$  rows. If  $m \leq n$ , this is the same as  $[C,R] = \text{qr}(A,B)$ .

$[C,R,e] = \text{qr}(A,B,0)$  additionally produces a fill-reducing permutation vector  $e$ . In this case, the least-squares solution to  $A * X = B$  is  $X(e, :) = R \setminus C$ .

## 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\'\'(A'*r);  
x = x + err;
```

## See Also

[lu](#) | [chol](#) | [null](#) | [orth](#) | [qrdelete](#) | [qrinsert](#) | [qrupdate](#)

**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)
```

# qrdelete

---

```
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`

**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
```

# qrinsert

---

```
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`

**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  $A + u \cdot v'$ , 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  $A' \cdot A$ . 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
```

# qrupdate

---

In this case, the upper triangular entries of R, excluding the first row, are on the order of  $\sqrt{\text{eps}}$ .

Consider the update vectors

$$u = [-1 \ 0 \ 0 \ 0 \ 0]'; \quad v = \text{ones}(4,1);$$

Instead of computing the rather trivial QR factorization of this rank one update to A from scratch with

$$[QT,RT] = \text{qr}(A + u*v')$$

QT =

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

RT =

$$1.0\text{e-}007 * \begin{bmatrix} -0.1490 & 0 & 0 & 0 \\ 0 & -0.1490 & 0 & 0 \\ 0 & 0 & -0.1490 & 0 \\ 0 & 0 & 0 & -0.1490 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

we may use qrupdate.

$$[Q1,R1] = \text{qrupdate}(Q,R,u,v)$$

Q1 =

$$\begin{bmatrix} -0.0000 & -0.0000 & -0.0000 & -0.0000 & 1.0000 \\ 1.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0000 \end{bmatrix}$$



```

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

# quad

---

**Purpose** Numerically evaluate integral, adaptive Simpson quadrature

**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,  $[a, \text{Inf})$ , 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  $p \geq -1/2$ . 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:

# quad

---

```
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 over planar region

**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  $a \leq x \leq b$  and  $c(x) \leq y \leq d(x)$ . `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:

|                     |                          |
|---------------------|--------------------------|
| <code>AbsTol</code> | absolute error tolerance |
| <code>RelTol</code> | relative error tolerance |

`quad2d` attempts to satisfy  $ERRBND \leq \max(AbsTol, RelTol * |Q|)$ . 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  $100 * \text{eps}(\text{class}(Q))$ . This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

|                          |                                                                    |
|--------------------------|--------------------------------------------------------------------|
| <code>MaxFunEvals</code> | Maximum allowed number of evaluations of <code>fun</code> reached. |
|--------------------------|--------------------------------------------------------------------|

# quad2d

---

The `MaxFunEvals` parameter limits the number of vectorized calls to `fun`. The default is 2000.

|                          |                                                         |
|--------------------------|---------------------------------------------------------|
| <code>FailurePlot</code> | Generate a plot if <code>MaxFunEvals</code> 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`.

|                       |                                         |
|-----------------------|-----------------------------------------|
| <code>Singular</code> | 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  $\pi \leq x \leq 2\pi$ ,  $0 \leq y \leq \pi$ . 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  $0 \leq x \leq 1$  and  $0 \leq y \leq 1-x$ . 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

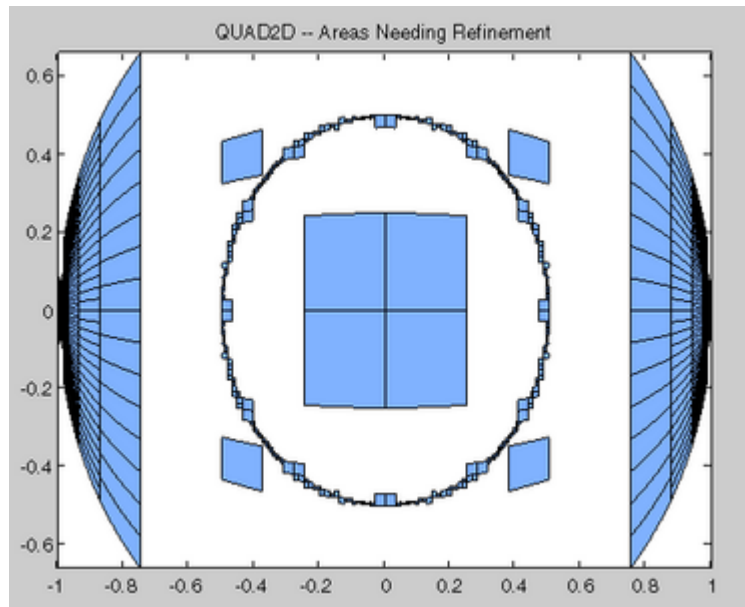
quad2d begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving `Singular` 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  $x^2 + y^2 = 0.25$ :

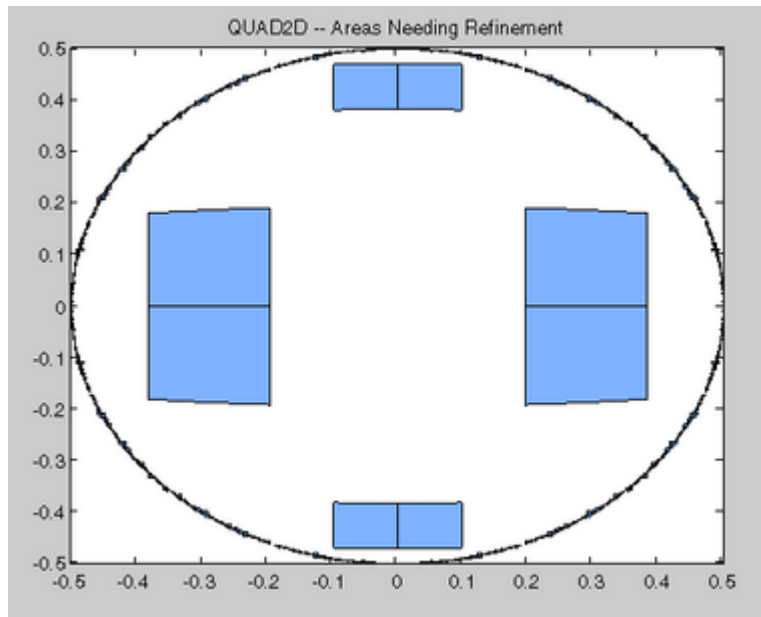
# quad2d



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:

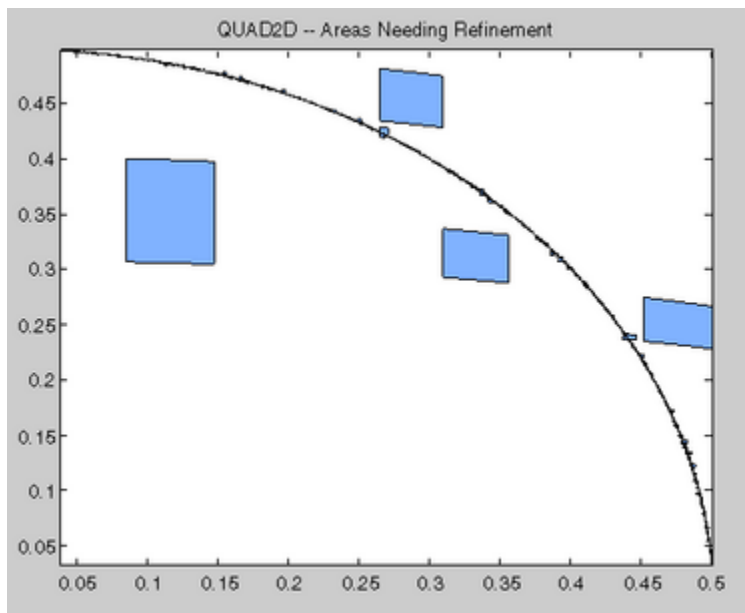
```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-8,...
             'Singular',true, 'FailurePlot',true)
```

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.
```

# quad2d



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](#) | [quad1](#) | [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.<br><br>The default value of 'AbsTol' is 1.e-10 (double), 1.e-5 (single). | quadgk attempts to satisfy $errbnd \leq \max(AbsTol, RelTol *  Q )$ . This is absolute error control when $ Q $ is sufficiently small and relative error control |

| Parameter   | Description                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'RelTol'    | <p>Relative error tolerance.</p> <p>The default value of 'RelTol' is 1.e-6 (double), 1.e-4 (single).</p> | <p>when <math> Q </math> is larger. For pure absolute error control use 'AbsTol' &gt; 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' <math>\geq 100 \cdot \text{eps}(\text{class}(Q))</math>.</p>                                                                                                                                                                                                                                                                 |
| 'Waypoints' | <p>Vector of integration waypoints.</p>                                                                  | <p>If <math>\text{fun}(x)</math> 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 <math>\text{fun}(x)</math>. Singular points should be handled by making them endpoints of separate integrations and adding the results.</p> <p>If a, b, or any entry of the waypoints vector is complex, the integration is performed over a</p> |

# quadgk

| 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.<br>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,  $[a, \text{Inf})$ , 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  $p \geq -1/2$ . 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`:

# quadgk

---

```
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 (15<sup>th</sup> and 7<sup>th</sup> 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` | `quad1` | `quadv` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

## How To

- “Anonymous Functions”

# quadl

---

**Purpose** Numerically evaluate integral, adaptive Lobatto quadrature

**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,  $[a, \text{Inf})$ , 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  $p \geq -1/2$ . 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.

# quadl

---

## 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

**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,  $[a, \text{Inf})$ , 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  $p \geq -1/2$ . 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 `WindowState` 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.
```

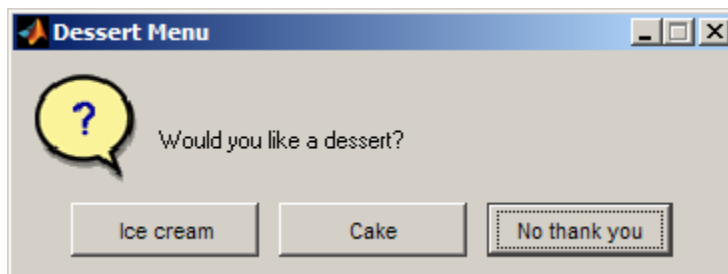
## Examples

### 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
```

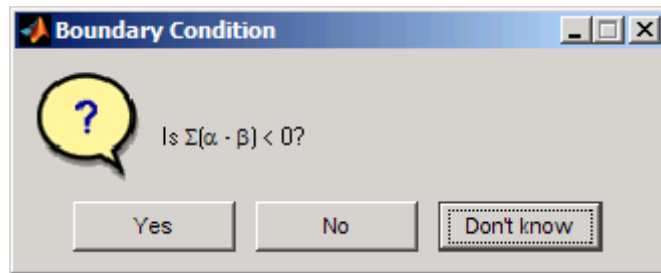
You can generalize the function by providing the cases as string or cell array calling arguments.

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
```

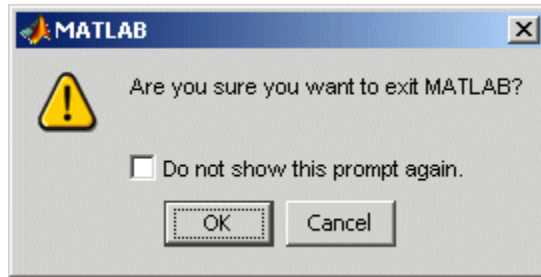
**Description** quit displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running `finish.m`, if `finish.m` exists. The workspace is not automatically saved by quit. To save 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.

- `finishesav.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

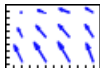
**Description** Quit terminates the MATLAB server session attached to handle h.

**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(...,LineStyle)
quiver(...,LineStyle,'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(...,LineStyle)` specifies line style, marker symbol, and color using any valid `LineStyle`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineStyle,'filled')` fills markers specified by `LineStyle`.

`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

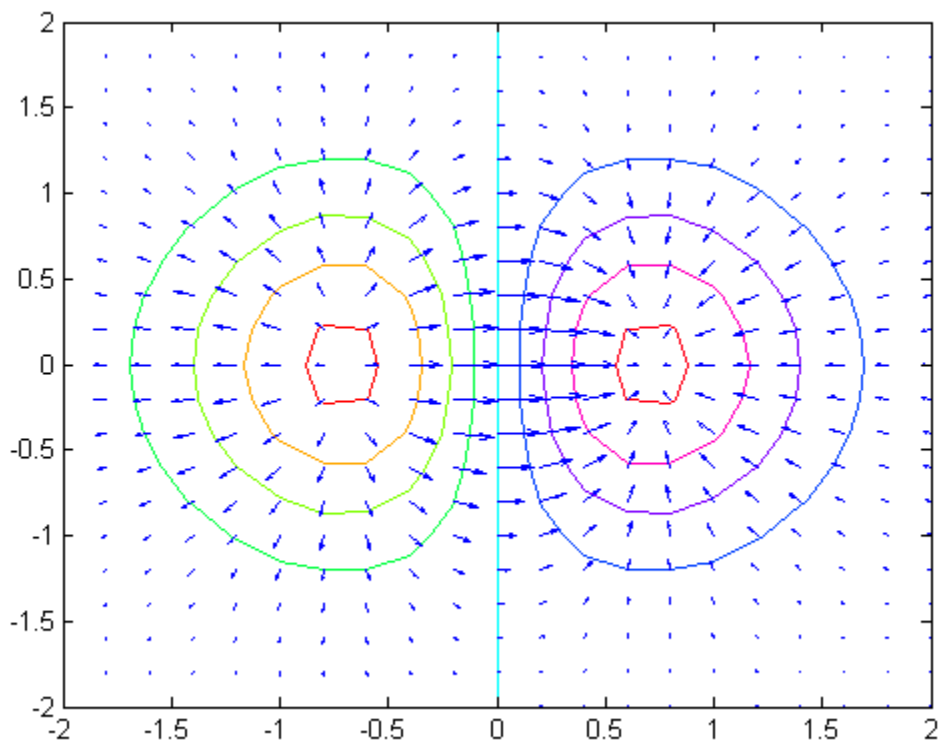
### Showing the Gradient with Quiver Plots

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](#) | [LineStyle](#) | [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(...,LineStyle)
quiver3(...,LineStyle,'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(...,LineStyle)` specifies line type and color using any valid `LineStyle`.

`quiver3(...,LineStyle,'filled')` fills markers specified by `LineStyle`.

`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.

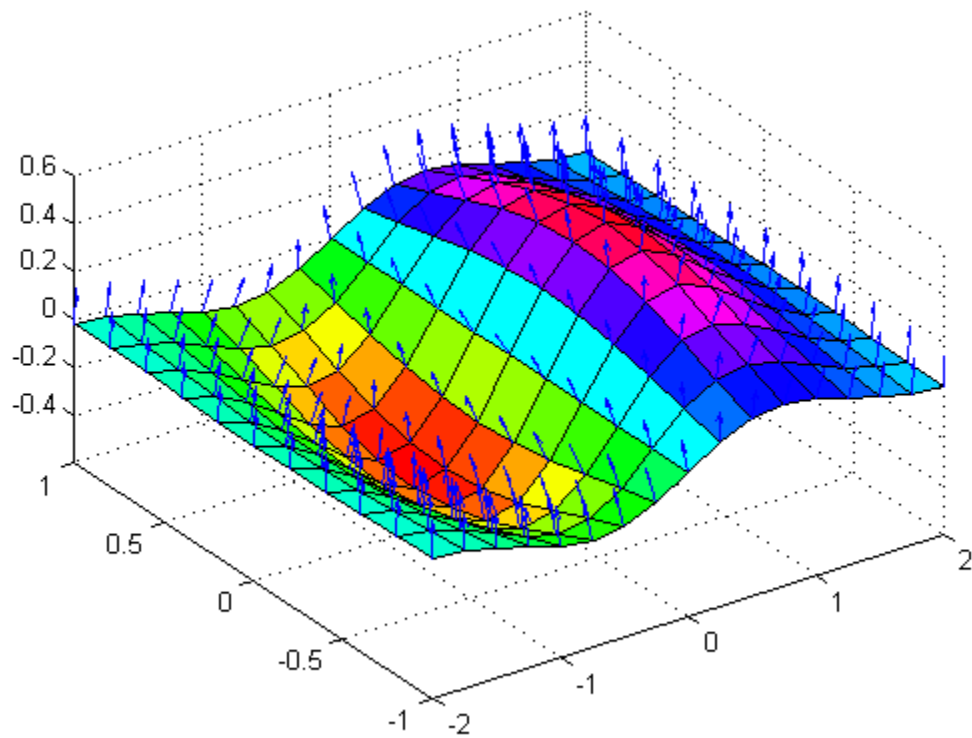
## Examples

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
```

# quiver3

---



## 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.

| <b>IconDisplayStyle Value</b> | <b>Purpose</b>                                                                        |
|-------------------------------|---------------------------------------------------------------------------------------|
| 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        |

# Quivergroup Properties

---

## 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.

# Quivergroup Properties

---

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)

# Quivergroup Properties

---

*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

# Quivergroup Properties

---

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

# Quivergroup Properties

---

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.*

# Quivergroup Properties

---

### 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).

# Quivergroup Properties

---

- `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` (the factory default for axes objects), 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, `hgroup`, or `hgtransform`

*Parent of object.* Handle of the object's parent. The parent is normally the axes, `hgroup`, 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')
```

# Quivergroup Properties

---

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 `'hgroup'`. This statement finds all the `hgroup` objects in the current axes.

```
t = findobj(gca,'Type','hgroup');
```

## 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.

---

# Quivergroup Properties

---

## 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)`.

# Quivergroup Properties

---

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

# Quivergroup Properties

---

`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  $\alpha$ s and  $\beta$ s.

$$\lambda = \alpha / \beta$$

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]. Use `r = randi(imax,m,n)` or `r = randi(imax,[m,n])` to obtain an m-by-n matrix of numbers. Use `r = randi(imax,p1,...,pn)` or `r = randi(imax,[p1,...,pn])` to obtain a n-dimensional array. Use `r = randi(imax)` to obtain a scalar value between 1 and imax. This is the same as `r = randi(imax,1)`. Use `r = randi(imax,size(A))` to obtain 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  $imin \leq imax$ . 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)
```

# randi

---

```
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

## randi (RandStream)

---

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

---

`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)

# 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 <code>RandStream.list</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 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       | <p>Internal state of the generator. You should not depend on the format of this property. The value you assign to <code>S.State</code> must be a value read from <code>S.State</code> previously. Use <code>reset</code> to return a stream to a predictable state without having previously read from the <code>State</code> property.</p> <p>The sequence of random numbers produced by a random number stream <code>s</code> is determined by the internal state of its random number generator. Saving and restoring the generator's internal state with the <code>State</code> property allows you to reproduce a sequence of random numbers.</p> |

# RandStream

---

| 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 ( <code>m1fg6331_64</code> ) and combined multiple recursive generator ( <code>mrg32k3a</code> ) support substreams. |
| NormalTransform | Transformation algorithm used by <code>randn(s, ...)</code> to generate normal pseudorandom values. Possible values are 'Ziggurat', 'Polar', or 'Inversion'.                                                                                                                                                    |
| RandnAlg        | <p>RandnAlg will be removed in a future release. Use NormalTransform instead.</p> <p>Transformation algorithm that will be used by <code>randn(S, ...)</code> to generate normal pseudorandom values. Options are 'Ziggurat', 'Polar', or 'Inversion'.</p>                                                      |
| Antithetic      | Logical value indicating whether <code>S</code> generates antithetic pseudorandom values. For uniform values, these are the usual values subtracted from 1. The default is false.                                                                                                                               |
| FullPrecision   | Logical value indicating whether <code>S</code> 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                                                                                                                                                                                                                                                                                                          |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RandStream</code>                  | Create a random number stream.                                                                                                                                                                                                                                                                                       |
| <code>RandStream.create</code>           | Create multiple independent random number streams.                                                                                                                                                                                                                                                                   |
| <code>get</code>                         | Get the properties of a random stream object.                                                                                                                                                                                                                                                                        |
| <code>list</code>                        | List available random number generator algorithms.                                                                                                                                                                                                                                                                   |
| <code>set</code>                         | Set random stream property.                                                                                                                                                                                                                                                                                          |
| <code>RandStream.getGlobalStream</code>  | Get the global random number stream.                                                                                                                                                                                                                                                                                 |
| <code>RandStream.getDefaultStream</code> | <code>RandStream.getDefaultStream</code> will be removed in a future release. Use <code>RandStream.getGlobalStream</code> instead. The shared random number stream used by <code>rand</code> , <code>randi</code> , and <code>randn</code> is now referred to as the global stream.<br>Default random number stream. |
| <code>RandStream.setGlobalStream</code>  | Set global random number stream.                                                                                                                                                                                                                                                                                     |
| <code>RandStream.setDefaultStream</code> | <code>RandStream.setDefaultStream</code> will be removed in a future release. Use <code>RandStream.setGlobalStream</code> instead.<br>Set default random number stream property.                                                                                                                                     |
| <code>reset</code>                       | Reset a stream to its initial internal state                                                                                                                                                                                                                                                                         |
| <code>rand</code>                        | Pseudorandom numbers from a uniform distribution                                                                                                                                                                                                                                                                     |

# RandStream

---

| 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);
```

# RandStream

---

## 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 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 <code>randn(s, ...)</code> to generate normal pseudorandom values.                                                                                         |

# RandStream constructor

---

| Parameter | Description                                                                                                                                                                                                       |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | Possible values are 'Ziggurat', 'Polar', or 'Inversion'.                                                                                                                                                          |
| RandnAlg  | RandnAlg will be removed in a future release. Use NormalTransform instead.<br><br>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](#)

# rank

---

**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  $\pi$  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

$$\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{\left(d_3 + \dots + \frac{1}{d_k}\right)}}$$

The  $d$ s 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  $X = \text{sqrt}(2)$ . For  $x = \text{sqrt}(2)$ , 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`

# rbbox

---

**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`



|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Right array division                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Syntax</b>          | $x = A./B$<br>$x = \text{rdivide}(A,B)$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b>     | <p><math>x = A./B</math> divides each element of <math>A</math> by the corresponding element of <math>B</math>.</p> <ul style="list-style-type: none"><li>• If <math>A</math> and <math>B</math> are arrays, then they must be the same size.</li><li>• If either <math>A</math> or <math>B</math> is a scalar, then MATLAB expands the scalar value into an appropriately sized array.</li></ul> <p><math>x = \text{rdivide}(A,B)</math> is an alternative way to divide <math>A</math> by <math>B</math>, but is rarely used. It enables operator overloading for classes.</p>                                                                                                                                                                                                                                      |
| <b>Input Arguments</b> | <p><b>A - Numerator</b><br/>numeric array   sparse numeric array</p> <p>Numerator, specified as a full or sparse numeric array. If <math>B</math> is an integer data type, <math>A</math> must be the same integer type or a scalar double.</p> <p><b>Data Types</b><br/>single   double   int8   int16   int32   int64   uint8   uint16   uint32   uint64</p> <p><b>Complex Number Support:</b> Yes</p> <p><b>B - Denominator</b><br/>numeric array   sparse numeric array</p> <p>Denominator, specified as a full or sparse numeric array. If <math>A</math> is an integer data type, <math>B</math> must be the same integer type or a scalar double.</p> <p><b>Data Types</b><br/>single   double   int8   int16   int32   int64   uint8   uint16   uint32   uint64</p> <p><b>Complex Number Support:</b> Yes</p> |

# rdivide

---

## 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)`

**Description** `video = read(obj)` reads in all video frames from the file associated with `obj`.

`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.

## 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. The dimensions are *H-by-W-by-B-by-F*, where:

|          |                                                        |
|----------|--------------------------------------------------------|
| <i>H</i> | Image frame height.                                    |
| <i>W</i> | Image frame width.                                     |
| <i>B</i> | Number of bands in the image (for example, 3 for RGB). |
| <i>F</i> | Number of frames read.                                 |

The class of *video* depends on the data in the file. For example, given a file that contains 8-bit unsigned values corresponding to three color bands (RGB24), *video* is an array of `uint8` values.

## 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);
```

# VideoReader.read

---

**See Also**      [movie](#) | [VideoReader](#)

**How To**        • [“Read Video Files”](#)

**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”

# readasync

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Read data asynchronously from device                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>      | <code>readasync(obj)</code><br><code>readasync(obj, size)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p><code>readasync(obj)</code> initiates an asynchronous read operation on the serial port object, <code>obj</code>.</p> <p><code>readasync(obj, size)</code> asynchronously reads, at most, the number of bytes given by <code>size</code>. If <code>size</code> is greater than the difference between the <code>InputBufferSize</code> property value and the <code>BytesAvailable</code> property value, an error is returned.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Tips</b>        | <p>Before you can read data, you must connect <code>obj</code> to the device with the <code>fopen</code> function. A connected serial port object has a <code>Status</code> property value of <code>open</code>. An error is returned if you attempt to perform a read operation while <code>obj</code> is not connected to the device.</p> <p>You should use <code>readasync</code> only when you configure the <code>ReadAsyncMode</code> property to <code>manual</code>. <code>readasync</code> is ignored if used when <code>ReadAsyncMode</code> is <code>continuous</code>.</p> <p>The <code>TransferStatus</code> 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 <code>stopasync</code> function.</p> <p>You can monitor the amount of data stored in the input buffer with the <code>BytesAvailable</code> property. Additionally, you can use the <code>BytesAvailableFcn</code> property to execute a callback function when the terminator or the specified amount of data is read.</p> <p><b>Rules for Completing an Asynchronous Read Operation</b></p> <p>An asynchronous read operation with <code>readasync</code> completes when one of these conditions is met:</p> <ul style="list-style-type: none"><li>• The terminator specified by the <code>Terminator</code> property is read.</li></ul> |



- 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 =
2.03999999619E0
fclose(s)
```

## See Also

`fopen` | `stopasync` | `BytesAvailable` | `BytesAvailableFcn` | `ReadAsyncMode` | `Status` | `TransferStatus`

# Tiff.readEncodedStrip

---

**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”

**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”

# Tiff.readEncodedTile

---

- “Reading Image Data and Metadata from TIFF Files”

**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

# RemoveAll

---

|                         |                                                                                                                                                                                                                                                                                     |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Convenience function for static .NET System.Delegate RemoveAll method                                                                                                                                                                                                               |
| <b>Syntax</b>           | <code>result = RemoveAll(combinedDelegate, removedDelegate)</code>                                                                                                                                                                                                                  |
| <b>Description</b>      | <code>result = RemoveAll(combinedDelegate, removedDelegate)</code> removes all instances of removedDelegate from combinedDelegate.                                                                                                                                                  |
| <b>Input Arguments</b>  | <p><b>combinedDelegate</b><br/>.NET System.Delegate object. The combined delegate from which to remove all instances of the removedDelegate delegate.</p> <p><b>removedDelegate</b><br/>.NET System.Delegate object. The delegate to remove from the combinedDelegate delegate.</p> |
| <b>Output Arguments</b> | <p><b>result</b><br/>.NET System.Delegate object. A new delegate which is the same as the combinedDelegate delegate except without all instances of the removedDelegate delegate.</p>                                                                                               |
| <b>Alternatives</b>     | Use the static RemoveAll method of the System.Delegate class.                                                                                                                                                                                                                       |
| <b>See Also</b>         | Remove   Combine                                                                                                                                                                                                                                                                    |
| <b>How To</b>           | <ul style="list-style-type: none"><li>• “Combine and Remove .NET Delegates”</li></ul>                                                                                                                                                                                               |
| <b>Related Links</b>    | <ul style="list-style-type: none"><li>• MSDN System.Delegate.RemoveAll Method reference page</li></ul>                                                                                                                                                                              |

## 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, 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`.

### **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'`.

# timeseries

---

## 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, NaNs 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 3743](#)
- [Methods to Manipulate Data and Time on page 3743](#)
- [Event Methods on page 3744](#)
- [Methods to Arithmetically Combine `timeseries` Objects on page 3745](#)
- [Methods to Calculate Descriptive Statistics for a `timeseries` Object on page 3746](#)

## Methods to Query and Set Object Properties and Plot the Data

|                                |                                                                                                                                  |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>get</code>               | Query <code>timeseries</code> object property values.                                                                            |
| <code>getdatasamplesize</code> | Return the size of each data sample in a <code>timeseries</code> object.                                                         |
| <code>getqualitydesc</code>    | Return data quality descriptions based on the <code>Quality</code> property values assigned to a <code>timeseries</code> object. |
| <code>plot</code>              | Plot the <code>timeseries</code> object.                                                                                         |
| <code>set</code>               | Set <code>timeseries</code> property values.                                                                                     |

## Methods to Manipulate Data and Time

|                             |                                                                                                                                                             |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>addsample</code>      | Add a data sample to a <code>timeseries</code> object.                                                                                                      |
| <code>append</code>         | Concatenate <code>timeseries</code> objects in the time dimension.                                                                                          |
| <code>ctranspose</code>     | Transpose a <code>timeseries</code> object.                                                                                                                 |
| <code>delsample</code>      | Delete a sample from a <code>timeseries</code> object.                                                                                                      |
| <code>detrend</code>        | Subtract the mean or best-fit line and remove all NaNs from time-series data.                                                                               |
| <code>filter</code>         | Shape frequency content of time-series data using a 1-D digital filter.                                                                                     |
| <code>getabstime</code>     | Extract a date-string time vector from a <code>timeseries</code> object into a cell array.                                                                  |
| <code>getdatasamples</code> | Extract a subset of data samples from an existing <code>timeseries</code> object into an array using a subscripted indexed array.                           |
| <code>getsamples</code>     | Extract a subset of data samples from an existing <code>timeseries</code> object into a new <code>timeseries</code> object using a subscript indexed array. |

## (Continued)

|                                  |                                                                                                                                                              |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getinterpmethod</code>     | Get the interpolation method for a <code>timeseries</code> object.                                                                                           |
| <code>getsamplingsingtime</code> | Extract data samples from an existing <code>timeseries</code> object into a new <code>timeseries</code> object based on specified start and end time values. |
| <code>idealfilter</code>         | Apply an ideal pass or notch (noncausal) filter to a <code>timeseries</code> object.                                                                         |
| <code>resample</code>            | Select or interpolate data in a <code>timeseries</code> object using a new time vector.                                                                      |
| <code>setabstime</code>          | Set the time values in the time vector as date strings.                                                                                                      |
| <code>setinterpmethod</code>     | Set interpolation method for a <code>timeseries</code> object.                                                                                               |
| <code>setuniformtime</code>      | Assign uniform time vector to <code>timeseries</code> object.                                                                                                |
| <code>synchronize</code>         | Synchronize and resample two <code>timeseries</code> objects using a common time vector.                                                                     |
| <code>transpose</code>           | Transpose a <code>timeseries</code> 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”.

|                       |                                                                  |
|-----------------------|------------------------------------------------------------------|
| <code>addevent</code> | Add one or more events to a <code>timeseries</code> object.      |
| <code>delevent</code> | Delete one or more events from a <code>timeseries</code> object. |



## (Continued)

|                                 |                                                                                                                                                      |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>gettsafteratevent</code>  | Create a new <code>timeseries</code> object by extracting the samples from an existing time series that occur after or at a specified event.         |
| <code>gettsafterevent</code>    | Create a new <code>timeseries</code> object by extracting the samples that occur after a specified event from an existing time series.               |
| <code>gettsatevent</code>       | Create a new <code>timeseries</code> object by extracting the samples that occur at the same time as a specified event from an existing time series. |
| <code>gettsbeforeatevent</code> | Create a new <code>timeseries</code> object by extracting the samples that occur before or at a specified event from an existing time series.        |
| <code>gettsbeforeevent</code>   | Create a new <code>timeseries</code> object by extracting the samples that occur before a specified event from an existing time series.              |
| <code>gettsbetweenevents</code> | Create a new <code>timeseries</code> object by extracting the samples that occur between two specified events from an existing time series.          |

## Methods to Arithmetically Combine `timeseries` Objects

|                 |                                                                                  |
|-----------------|----------------------------------------------------------------------------------|
| <code>+</code>  | Addition of the corresponding data values of <code>timeseries</code> objects.    |
| <code>-</code>  | Subtraction of the corresponding data values of <code>timeseries</code> objects. |
| <code>.*</code> | Element-by-element multiplication of <code>timeseries</code> data.               |

## (Continued)

|    |                                                                       |
|----|-----------------------------------------------------------------------|
| *  | Matrix-multiply <code>timeseries</code> data.                         |
| ./ | Right element-by-element division of <code>timeseries</code> data.    |
| /  | Right matrix division of <code>timeseries</code> data.                |
| .\ | Element-by-element left-array divide of <code>timeseries</code> data. |
| \  | Left matrix division of <code>timeseries</code> data.                 |

## Methods to Calculate Descriptive Statistics for a `timeseries` Object

|                     |                                                                 |
|---------------------|-----------------------------------------------------------------|
| <code>iqr</code>    | Return the interquartile range of <code>timeseries</code> data. |
| <code>max</code>    | Return the maximum value of <code>timeseries</code> data.       |
| <code>mean</code>   | Return the mean of <code>timeseries</code> data.                |
| <code>median</code> | Return the median of <code>timeseries</code> data.              |
| <code>min</code>    | Return the minimum of <code>timeseries</code> data.             |
| <code>std</code>    | Return the standard deviation of <code>timeseries</code> data.  |
| <code>sum</code>    | Return the sum of <code>timeseries</code> data.                 |
| <code>var</code>    | Return the variance of <code>timeseries</code> 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  $3 \times 4 \times 5 = 60$ .

## 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                |

# timeseries

---

| 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 Programming Fundamentals 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”

# timeseries.addsample

---

**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, 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`.

## 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);
```

---

# timeseries.addsample

---

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](#)



|                         |                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Concatenate time series objects in time dimension                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Syntax</b>           | <code>ts = append(ts1,ts2, ... tsn)</code>                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b>      | <code>ts = append(ts1,ts2, ... tsn)</code> creates a new <code>timeseries</code> object by concatenating <code>timeseries</code> <code>ts1</code> , <code>ts2</code> , and so on, along the time dimension.                                                                                                                                                                                                            |
| <b>Tips</b>             | <ul style="list-style-type: none"><li>• A single overlapping time between each input time series is valid, as long as the overlapping samples are identical.</li><li>• The time vectors must not overlap by a nonzero amount. That is, the last time in <code>ts1</code> must be earlier than or equal to the first time in <code>ts2</code>.</li><li>• The sample size of the time series must be the same.</li></ul> |
| <b>Input Arguments</b>  | <p><b>ts1</b><br/>The first <code>timeseries</code> object that you want to append.</p> <p><b>ts2</b><br/>The second <code>timeseries</code> object that you want to append.</p> <p><b>tsn</b><br/>The <code>n</code>th <code>timeseries</code> object that you want to append.</p>                                                                                                                                    |
| <b>Output Arguments</b> | <p><b>ts</b><br/>The <code>timeseries</code> object that results from appending the input <code>timeseries</code> objects.</p>                                                                                                                                                                                                                                                                                         |
| <b>Examples</b>         | <p>After creating <code>timeseries</code> objects, <code>ts1</code> and <code>ts2</code>, append them:</p> <pre>ts1 = timeseries(rand(5,1),[1 2 3 4 5]);<br/>ts2 = timeseries(rand(5,1),[6 7 8 9 10]);<br/>ts3 = append(ts1, ts2)</pre>                                                                                                                                                                                |
| <b>See Also</b>         | <code>timeseries</code>                                                                                                                                                                                                                                                                                                                                                                                                |

# timeseries.ctranspose

---

**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             |

## 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`

# timeseries.delsample

---

**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, 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`.

**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 NaNs 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 NaNs.

**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`

**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:

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb)x(n-nb+1) - a(2)y(n-1) - \dots - a(N_a)y(n-N_b+1).$$

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,  $x(n) x(n-1)\dots$ , and previous outputs,  $y(n-1) y(n-2)\dots$ .

- You use the discrete filter to shape the data by applying a transfer function to the input signal.

# timeseries.filter

---

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

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \dots + b(nb)x(n-nb+1) - a(2)y(n-1) - \dots - a(na)y(na+1),$$

results in the transfer function

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(nb)z^{-nb+1}}{a(1) + a(2)z^{-1} + \dots + a(na)z^{-na+1}} X(z),$$

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)
```

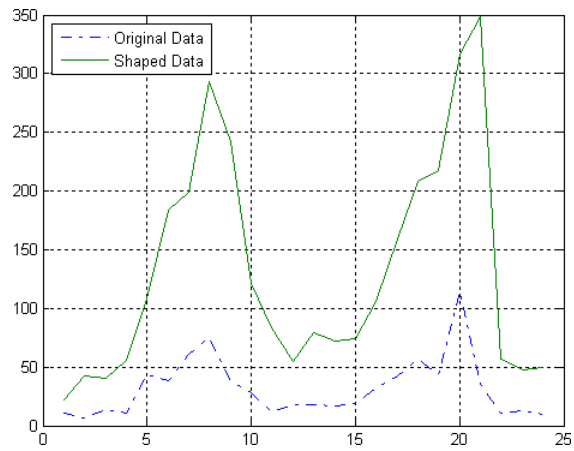
```
% Compare the original data and the shaped data with an
% overlaid plot of the two curves:
```

```
plot(count1,'-.'), grid on, hold on
```

# timeseries.filter

---

```
plot(filter_count, '- -')  
legend('Original Data', 'Shaped Data', 2)
```



## See Also

[timeseries](#) | [idealfilter](#)

|                         |                                                                                                                                                                                                                                                                                                                              |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Query <code>timeseries</code> object property values                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>           | <pre>get(ts) value = get(ts, PropertyName)</pre>                                                                                                                                                                                                                                                                             |
| <b>Description</b>      | <p><code>get(ts)</code> displays all properties and values of the <code>timeseries</code> object, <code>ts</code>.</p> <p><code>value = get(ts, PropertyName)</code> returns the property value for the specified <code>timeseries</code> object. The following syntax is equivalent:</p> <pre>value = ts.PropertyName</pre> |
| <b>Input Arguments</b>  | <p><b>ts</b><br/>A <code>timeseries</code> object.</p> <p><b>PropertyName</b><br/>String specifying the name of a <code>timeseries</code> property. For a list of <code>timeseries</code> properties, see <code>timeseries</code>.</p>                                                                                       |
| <b>Output Arguments</b> | <p><b>value</b><br/>String containing the value associated with the specified property.</p>                                                                                                                                                                                                                                  |
| <b>Examples</b>         | <p>Create a <code>timeseries</code> object, and then get the name. This example gets the name three different ways:</p> <pre>ts1 = timeseries(rand(5,1),[1 2 3 4 5], 'Name', 'MyTimeseries'); get(ts1) get(ts1, 'Name') value = ts1.Name</pre>                                                                               |
| <b>See Also</b>         | <code>timeseries</code>   <code>set</code>                                                                                                                                                                                                                                                                                   |

# timeseries.getabstime

---

**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](#)

# timeseries.getdatasamples

---

**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`

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Size of data sample in <code>timeseries</code> object                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>          | <code>getdatasamplesize(ts)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Description</b>     | <code>getdatasamplesize(ts)</code> returns the size of each data sample in a <code>timeseries</code> object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Input Arguments</b> | <b>ts</b><br>String specifying the name of a <code>timeseries</code> object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Definitions</b>     | <b>data sample</b><br>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.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Examples</b>        | <p>After loading data and creating a <code>timeseries</code> object, get the size of a data sample:</p> <pre>% Load a 24-by-3 data array:<br/>load count.dat<br/><br/>% Create a timeseries object with 24 time values:<br/>count_ts = timeseries(count,[1:24], 'Name', 'VehicleCount')<br/><br/>% Get the size of the data sample for this timeseries object:<br/>getdatasamplesize(count_ts)</pre> <p>MATLAB returns the following, which indicates that the size of each data sample in <code>count_ts</code> is 1-by-3. In other words, MATLAB stores each data sample as a row with three values.</p> <pre>ans =<br/><br/>     1     3</pre> |
| <b>See Also</b>        | <code>timeseries</code>   <code>set</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

# timeseries.getinterpmethod

---

**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`



|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Data quality descriptions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>          | <code>getqualitydesc(ts)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b>     | <code>getqualitydesc(ts)</code> returns a cell array of data quality descriptions based on the <code>Quality</code> values you assigned to a <code>timeseries</code> object, <code>ts</code> .                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Input Arguments</b> | <b>ts</b><br>A <code>timeseries</code> object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Examples</b>        | <p>Create a <code>timeseries</code> object, and then get the data quality description strings for <code>ts</code>:</p> <pre>% 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)</pre> <p>MATLAB returns:</p> <pre>ans =      'bad'     'good'     'bad'     'good'     'bad'</pre> |

# timeseries.getqualitydesc

---

## See Also

`timeseries`

|                         |                                                                                                                                                                                                                                                                                 |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Subset of time series samples using subscripted index array                                                                                                                                                                                                                     |
| <b>Syntax</b>           | <code>ts1 = getsamples(ts, i)</code>                                                                                                                                                                                                                                            |
| <b>Description</b>      | <code>ts1 = getsamples(ts, i)</code> returns a new <code>timeseries</code> object by extracting samples from <code>timeseries</code> <code>ts</code> corresponding to the time or times indicated by the subscripted index array, <code>i</code> .                              |
| <b>Input Arguments</b>  | <p><b>ts</b></p> <p>The <code>timeseries</code> object from which you want to exact samples.</p> <p><b>i</b></p> <p>A subscripted index array that specifies the time value or values for which you want to extract the corresponding samples.</p>                              |
| <b>Output Arguments</b> | <p><b>ts1</b></p> <p>The <code>timeseries</code> object that results from extracting the samples corresponding to the time value or values <code>ts.time(i)</code> .</p>                                                                                                        |
| <b>Examples</b>         | <p>After creating a <code>timeseries</code> object, <code>ts</code>, extract the data samples at times 2 and 3 into a new <code>timeseries</code> object, <code>ts1</code>:</p> <pre>ts = timeseries(rand(5,1),[1 2 3 4 5]);<br/>ts1 = getdatasamples(ts, ts.time([2 3]))</pre> |
| <b>See Also</b>         | <code>timeseries</code>   <code>getdatasamples</code>   <code>resample</code>                                                                                                                                                                                                   |

# timeseries.getsampleusingtime

---

**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 `datetime` 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 NaNs 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];
```

## timeseries.idealfilter

---

```
% Invoke an ideal notch filter:
idealfilter_countn = idealfilter(count1,interval,'notch')

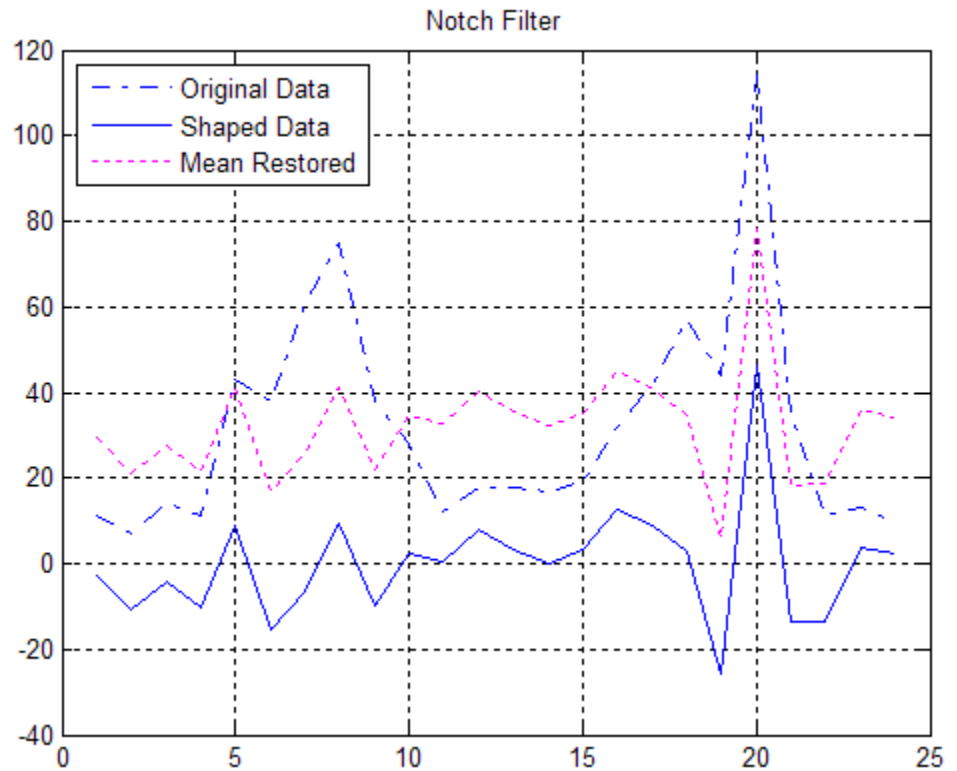
% Compare the original data and the shaped data on a line plot:

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')
```





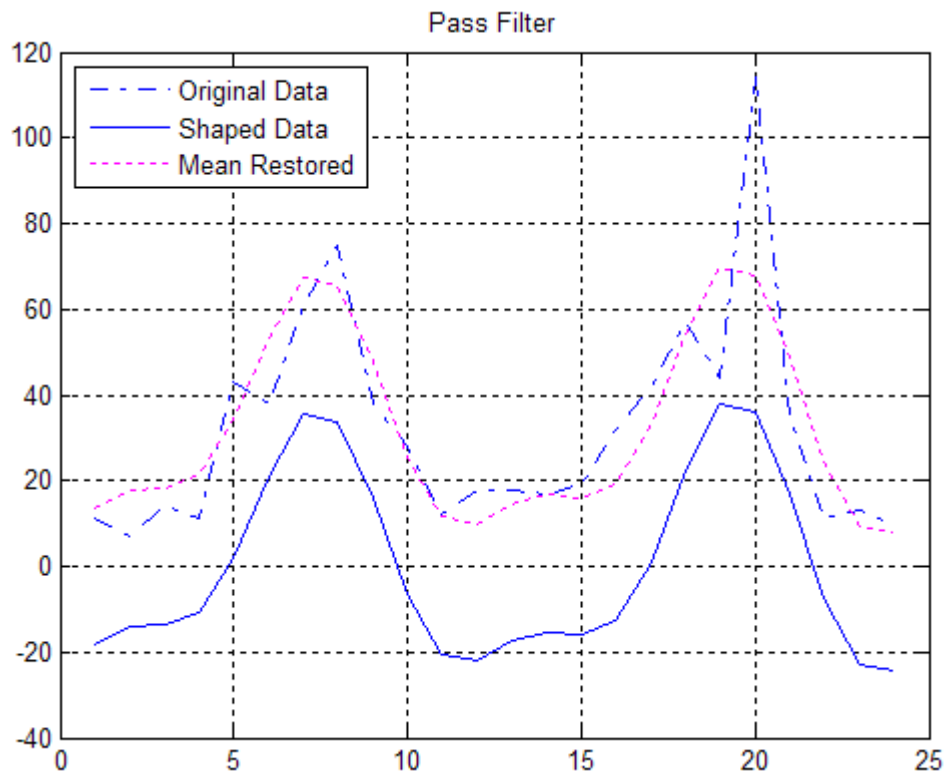
```

% 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', ...

```

# timeseries.idealfilter

'Location', 'NorthWest')



**See Also**

[timeseries | filter](#)

---

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Interquartile range of <code>timeseries</code> data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>           | <pre>ts_iqr = iqr(ts) iqr(ts, Name, Value)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>      | <p><code>ts_iqr = iqr(ts)</code> returns the interquartile range of <code>ts.Data</code>.</p> <p><code>iqr(ts, Name, Value)</code> reruns the interquartile range of <code>ts.Data</code> with the specified <code>Name, Value</code> pairs.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Input Arguments</b>  | <p><b>ts</b></p> <p>The <code>timeseries</code> object for which you want the interquartile range of <code>timeseries</code> data.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name, Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1, Value1, ..., NameN, ValueN</code>.</p> <p><b>MissingData</b></p> <p>A string specifying one of two possible values, <code>remove</code> or <code>interpolate</code>, indicating how to treat missing data during the calculation.</p> <p><b>Default:</b> <code>remove</code></p> <p><b>Quality</b></p> <p>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).</p> |
| <b>Output Arguments</b> | <p><b>ts_iqr</b></p> <p>The interquartile range of <code>ts.Data</code>, as follows:</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

# timeseries.iqr

---

- 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`

## 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, 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`.

### **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).

# timeseries.max

---

## 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`

**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, 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`.

### **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**

# timeseries.mean

---

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 ( $(t(\text{end}) - t(\text{end} - 1))$ ).
- 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`

# timeseries.median

---

**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, 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`.

## **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 ( $(t(\text{end}) - t(\text{end} - 1))$ ).
- 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`

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Minimum value of <code>timeseries</code> data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>          | <pre>ts_min = min(ts) ts_min = method(ts,Name,Value)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>     | <p><code>ts_min = min(ts)</code> returns the minimum value in the <code>timeseries</code> data.</p> <p><code>ts_min = method(ts,Name,Value)</code> uses additional options specified by one or more <code>Name,Value</code> pair arguments.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Input Arguments</b> | <p><b>ts</b></p> <p>The <code>timeseries</code> object for which you want the minimum data value.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>MissingData</b></p> <p>A string specifying one of two possible values, <code>remove</code> or <code>interpolate</code>, indicating how to treat missing data during the calculation.</p> <p><b>Default:</b> <code>remove</code></p> <p><b>Quality</b></p> <p>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).</p> |

# timeseries.min

---

## 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`

**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>)'

# timeseries.plot

---

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**



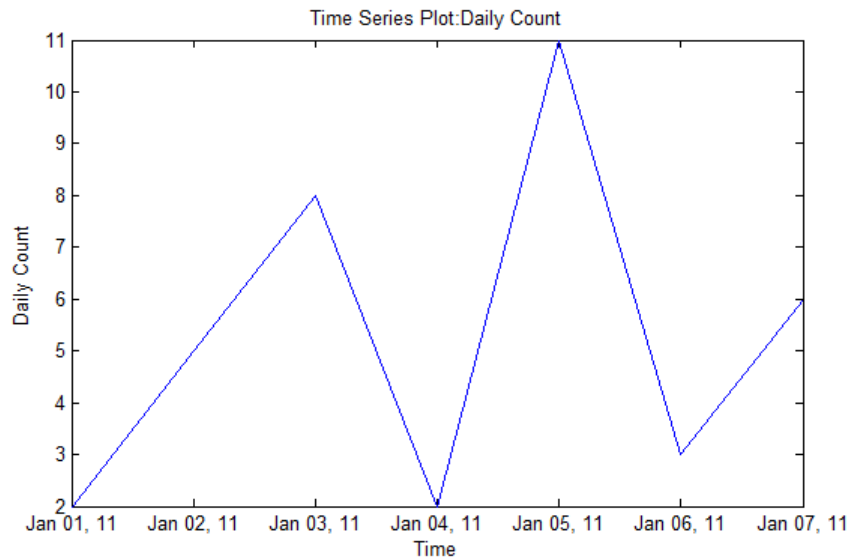
The name of a timeseries object within the tscollection .

## 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

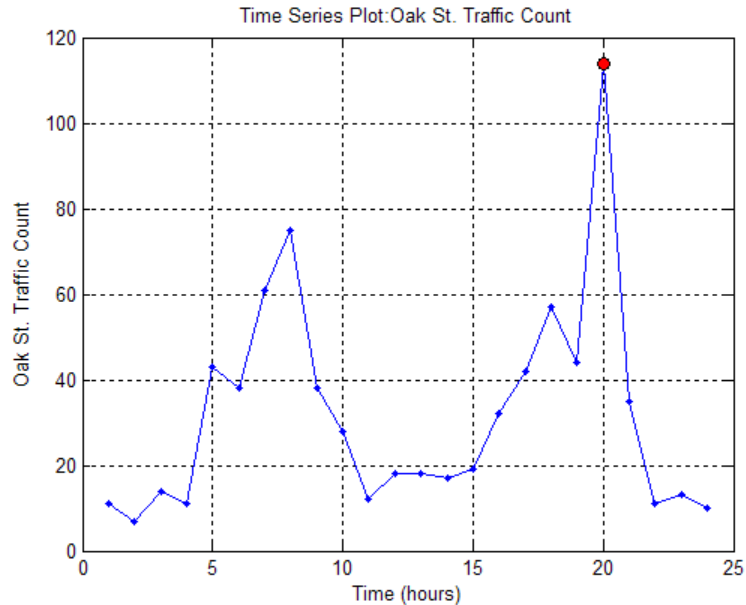
Create two time series objects from traffic count data, and then plot them in sequence on the same axes. Add an event to one series, which is automatically displayed with a red marker.

```
load count.dat;  
count1=timeseries(count(:,1),1:24);  
count1.Name = 'Oak St. Traffic Count';  
count1.TimeInfo.Units = 'hours';  
plot(count1,':b'), grid on
```



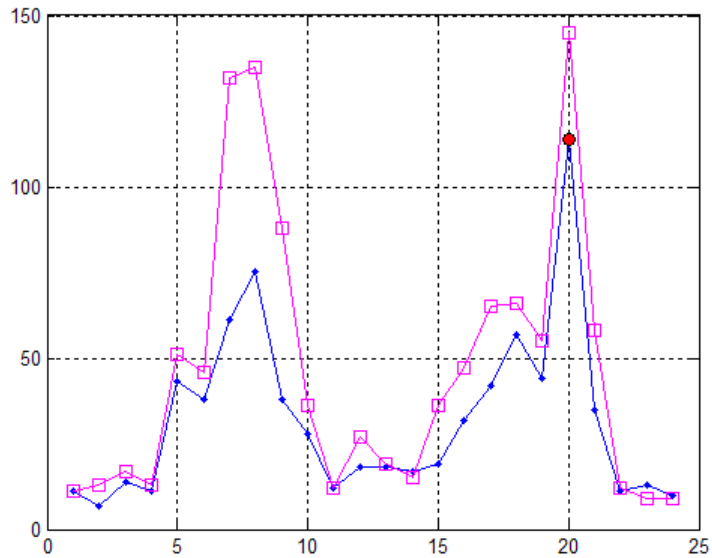
```
% Obtain time of maximum value and add it as an event:  
[~,index] = max(count1.Data);  
max_event = tsdata.event('peak',count1.Time(index));  
max_event.Units = 'hours';  
% Add the event to the time series:  
count1 = addevent(count1,max_event);
```

```
% Replace plot with new one showing the event:
plot(count1,'.-b'), grid on
```

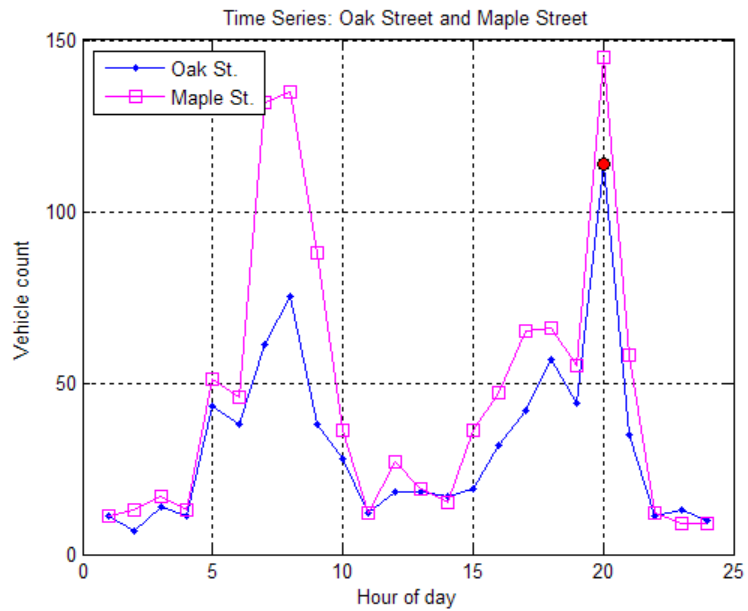


```
% Make a new ts object from column 2 of the same data source:
count2=timeseries(count(:,2),1:24);
count2.Name = 'Maple St. Traffic Count';
count2.TimeInfo.Units = 'Hours';
% Turn hold on to add the new data to the plot:
hold on
% The plot method does not add labels to a held plot.
% Use property/value pair to customize markers:
plot(count2,'s-m','MarkerSize',6),
```

# timeseries.plot



```
% 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`

# timeseries.resample

---

**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`

# timeseries.set

---

**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`



|                         |                                                                                                                                                                                                                                                                                                                                        |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Set times of timeseries object as date strings                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>           | <pre>ts1=setabstime(ts, times) ts1=setabstime(ts, times, format)</pre>                                                                                                                                                                                                                                                                 |
| <b>Description</b>      | <p><code>ts1=setabstime(ts, times)</code> sets the times in <code>ts</code> to the date strings specified in <code>times</code>.</p> <p><code>ts1=setabstime(ts, times, format)</code> explicitly specifies the date-string format, <code>format</code>, used in <code>times</code>.</p>                                               |
| <b>Input Arguments</b>  | <p><b>ts</b></p> <p>The timeseries object for which you want to set times as date strings.</p> <p><b>times</b></p> <p>A cell array of strings or a char array containing valid date or time values in the same date format.</p> <p><b>format</b></p> <p>The date-string format used for the time values.</p>                           |
| <b>Output Arguments</b> | <p><b>ts1</b></p> <p>The timeseries object that results from setting times as date strings on the original timeseries object.</p>                                                                                                                                                                                                      |
| <b>Examples</b>         | <p>Create a timeseries object, and then set the absolute time vector:</p> <pre>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'})</pre> <p>% View each timeseries object in the Variable Editor to see the % differences in the time vectors for each.</p> |
| <b>See Also</b>         | <code>datestr</code>   <code>getabstime</code>   <code>timeseries</code>                                                                                                                                                                                                                                                               |

# 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**

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);
```

# timeseries.setinterpmethod

---

## See Also

`timeseries | getinterpmethod`

## 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`. `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.

# timeseries.setuniformtime

---

## 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`

# timeseries.synchronize

---

**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, 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`.

## **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

# timeseries.synchronize

---

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`

**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.

# timeseries.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`

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Standard deviation of <code>timeseries</code> data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>          | <pre>ts_std = std(ts) ts_std = std(ts,Name,Value)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>     | <p><code>ts_std = std(ts)</code> returns the standard deviation of the <code>timeseries</code> data.</p> <p><code>ts_std = std(ts,Name,Value)</code> specifies additional options specified with one or more <code>Name,Value</code> pair arguments.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Input Arguments</b> | <p><b>ts</b></p> <p>The <code>timeseries</code> object for which you want the standard deviation of the data.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>MissingData</b></p> <p>A string specifying one of two possible values, <code>remove</code> or <code>interpolate</code>, indicating how to treat missing data during the calculation.</p> <p><b>Default:</b> <code>remove</code></p> <p><b>Quality</b></p> <p>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).</p> <p><b>Weighting</b></p> |

# timeseries.std

---

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 ( $(t(\text{end}) - t(\text{end} - 1))$ ).
- 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`

# timeseries.sum

---

**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, 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`.

## **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 ( $(t(\text{end}) - t(\text{end} - 1))$ ).
- 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`

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Variance of timeseries data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>          | <pre>ts_var = var(ts) ts_var = var(ts,Name,Value)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>     | <p><code>ts_var = var(ts)</code> returns the variance of <code>ts.data</code>.</p> <p><code>ts_var = var(ts,Name,Value)</code> uses additional options specified by one or more <code>Name,Value</code> pair arguments.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Input Arguments</b> | <p><b>ts</b></p> <p>The timeseries object for which you want the variance of the data.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments, where <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>MissingData</b></p> <p>A string specifying one of two possible values, <code>remove</code> or <code>interpolate</code>, indicating how to treat missing data during the calculation.</p> <p><b>Default:</b> <code>remove</code></p> <p><b>Quality</b></p> <p>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).</p> <p><b>Weighting</b></p> |

# timeseries.var

---

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 ( $(t(\text{end}) - t(\text{end} - 1))$ ).
- 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`

# real

---

**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`

**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`

# realmax

---

**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”



**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”

# realpow

---

**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 `.`<sup>^</sup>

**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`

# record

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Record data and event information to file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>      | <code>record(obj)</code><br><code>record(obj, 'switch')</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b> | <code>record(obj)</code> toggles the recording state for the serial port object, <code>obj</code> .<br><code>record(obj, 'switch')</code> initiates or terminates recording for <code>obj</code> . <code>switch</code> can be <code>on</code> or <code>off</code> . If <code>switch</code> is <code>on</code> , recording is initiated. If <code>switch</code> is <code>off</code> , recording is terminated.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Tips</b>        | <p>Before you can record information to disk, <code>obj</code> must be connected to the device with the <code>fopen</code> function. A connected serial port object has a <code>Status</code> property value of <code>open</code>. An error is returned if you attempt to record information while <code>obj</code> is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when <code>obj</code> is disconnected from the device with <code>fclose</code>.</p> <p>The <code>RecordName</code> and <code>RecordMode</code> properties are read-only while <code>obj</code> is recording, and must be configured before using <code>record</code>.</p> <p>For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to <a href="#">Debugging: Recording Information to Disk</a>.</p> |
| <b>Examples</b>    | <p>This example creates the serial port object <code>s</code> on a Windows platform. It connects <code>s</code> to the device, configures <code>s</code> to record information to a file, writes and reads text data, and then disconnects <code>s</code> from the device.</p> <pre>s = serial('COM1'); fopen(s) s.RecordDetail = 'verbose'; s.RecordName = 'MySerialFile.txt'; record(s, 'on') fprintf(s, '*IDN?') out = fscanf(s); record(s, 'off')</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

`fclose(s)`

## **See Also**

`fclose` | `fopen` | `RecordDetail` | `RecordMode` | `RecordName` | `RecordStatus` | `Status`

# audiorecorder.record

---

**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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Record audio to audiorecorder object, holding control until recording completes                                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <code>recordblocking(recorderObj, length)</code>                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Description</b> | <code>recordblocking(recorderObj, length)</code> records audio from an input device, such as a microphone connected to your system, for the number of seconds specified by <i>length</i> . The <code>recordblocking</code> method does not return control until recording completes. <i>recorderObj</i> is an audiorecorder object that defines the sample rate, bit depth, and other properties of the recording. |
| <b>Examples</b>    | Record 5 seconds of your speech with a microphone, and play it back:<br><br><pre>myVoice = audiorecorder;<br/><br/>disp('Start speaking.');</pre> <code>recordblocking(myVoice, 5);</code> <pre>disp('End of recording. Playing back ...');</pre><br><pre>play(myVoice);</pre>                                                                                                                                     |
| <b>See Also</b>    | <code>audiorecorder</code>   <code>getaudiodata</code>   <code>record</code>                                                                                                                                                                                                                                                                                                                                       |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Record Audio”</li></ul>                                                                                                                                                                                                                                                                                                                                                   |

# rectangle

---

**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.

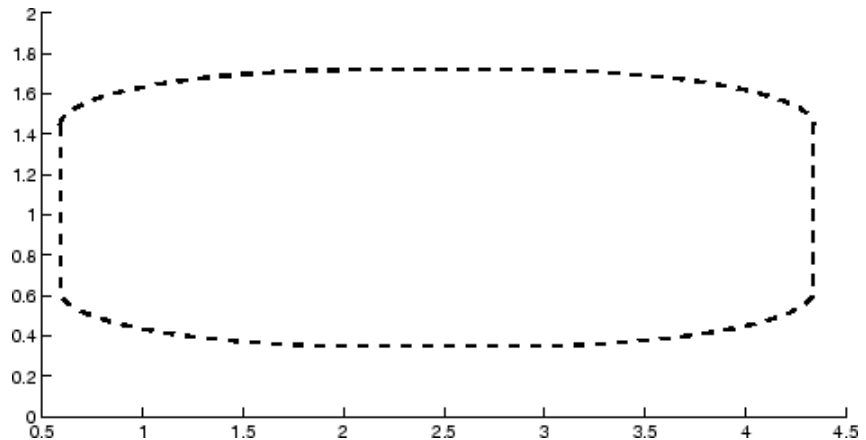
## Tips

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`.

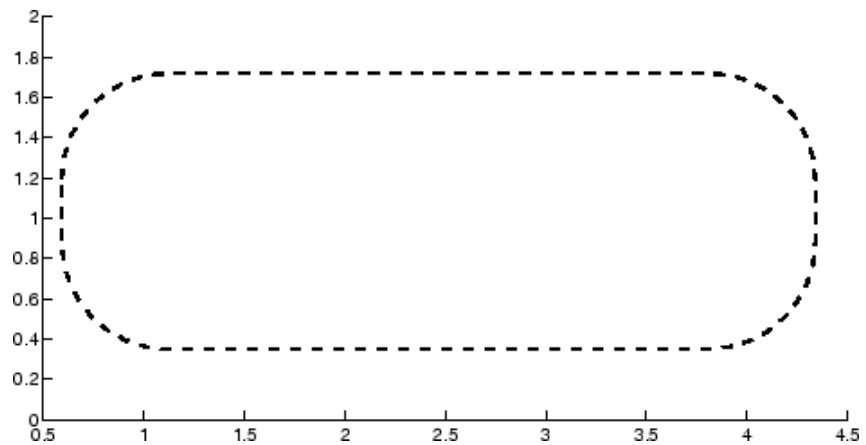
```
rectangle('Position',[0.59,0.35,3.75,1.37],...
         'Curvature',[0.8,0.4],...
         'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```



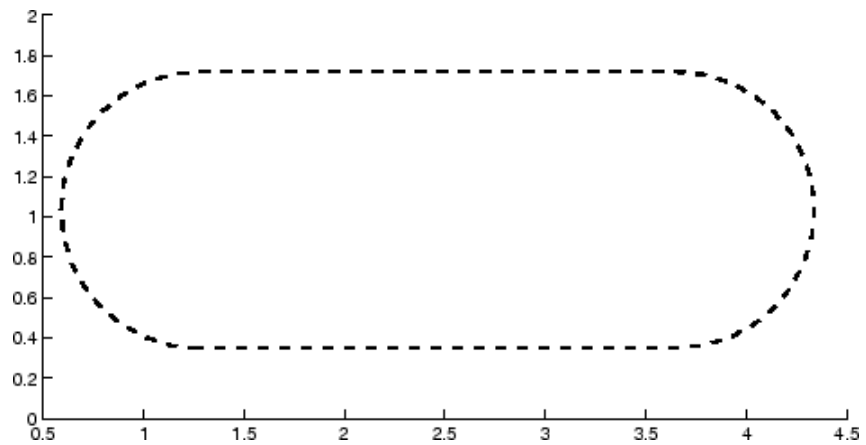
Specifying a single value of `[0.4]` for `Curvature` produces

# rectangle

---



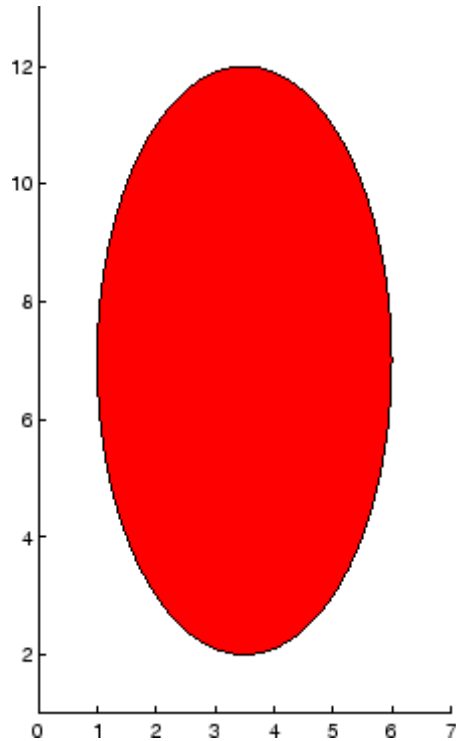
A Curvature of [1] produces a rectangle with the shortest side completely round:



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.

# rectangle

---

## **See Also**

[line](#) | [patch](#) | [annotation](#) | [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:

# Rectangle Properties

---

| <b>IconDisplayStyle Value</b> | <b>Purpose</b>                                            |
|-------------------------------|-----------------------------------------------------------|
| 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).

# Rectangle Properties

---

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(gcf, '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

# Rectangle Properties

---

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.

# Rectangle Properties

---

- 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.

# Rectangle Properties

---

- `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.

# Rectangle Properties

---

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 =  $\frac{1}{72}$  inch. The default value is 0.5 points.



## Parent

handle of axes, hgroup, or hgtransform

*Parent of rectangle object.* Contains the handle of the rectangle object's parent. The parent of a rectangle object is the axes, hgroup, 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.

# Rectangle Properties

---

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`

# recycle

---

**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”

# reducepatch

---

**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

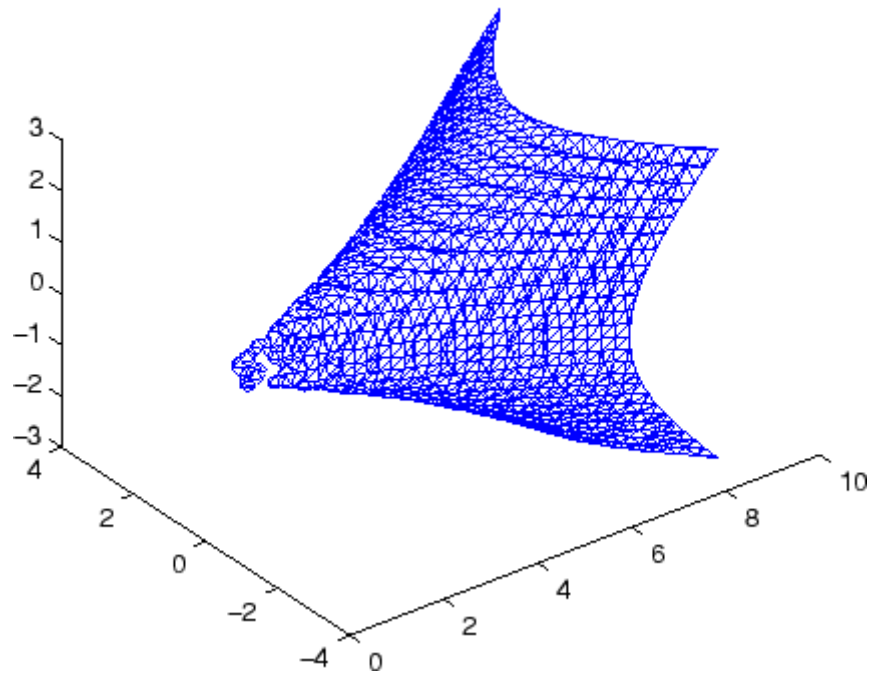
This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
[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)
```

# reducepatch

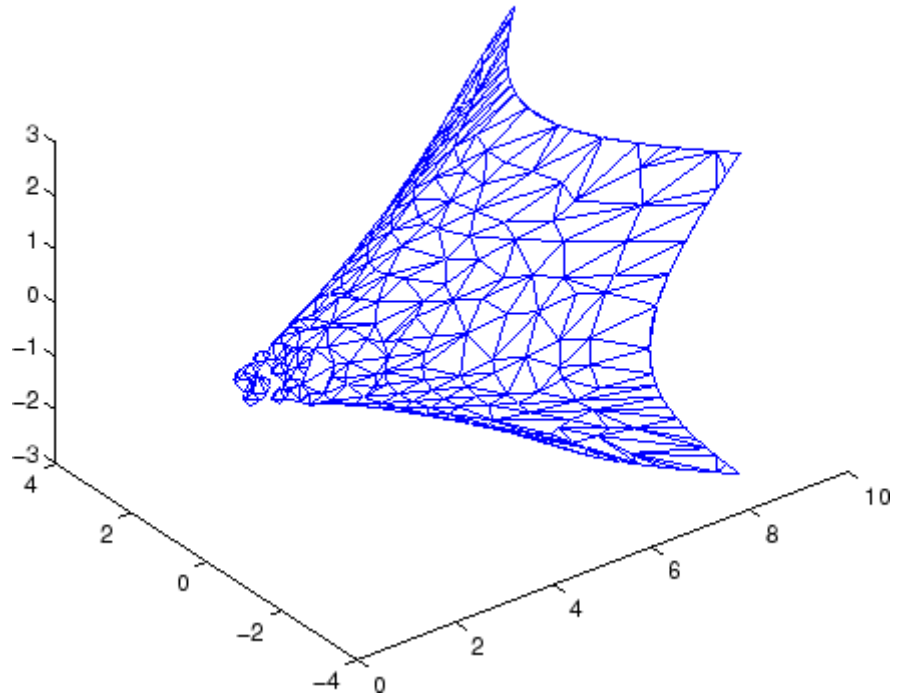
---

Before Reduction





After Reduction to 15% of Original Number of Faces



## See Also

[isosurface](#) | [isocaps](#) | [isonormals](#) | [smooth3](#) | [subvolume](#) | [reducevolume](#)

## How To

- [Vector Field Displayed with Cone Plots](#)

# reducevolume

---

**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  $R_x^{\text{th}}$  element in the  $x$  direction, every  $R_y^{\text{th}}$  element in the  $y$  direction, and every  $R_z^{\text{th}}$  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.

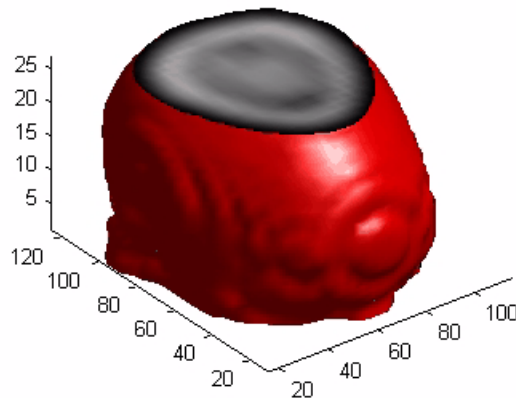
## Examples

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

---

|                    |                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Redraw current figure                                                                                                                    |
| <b>Syntax</b>      | <code>refresh</code><br><code>refresh(h)</code>                                                                                          |
| <b>Description</b> | <code>refresh</code> erases and redraws the current figure.<br><code>refresh(h)</code> redraws the figure identified by <code>h</code> . |

**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

---

`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**

```

regexp(string,expr)
[matchstart,matchend,tokenindices,matchstring,tokenstring,
 tokenname,splitstring] = regexp(string,expr)
[selected_outputs] = regexp(string,expr,outselect)
regexp(string,expr,options)

```

**Description**

regexp(string,expr) parses the input string locating those parts of string that match the character pattern(s) specified by regular expression, expr. This syntax returns the starting index of each match. If no matches are found, regexp returns an empty array.

[matchstart,matchend,tokenindices,matchstring,tokenstring,tokenname,splitstring] = regexp(string,expr) returns from one to seven output values, depending on the number of output variables you specify.

[selected\_outputs] = regexp(string,expr,outselect) returns from one to seven output values, depending on which flags you specify in outselect. The presence and ordering of the outselect inputs determine the presence and ordering of the corresponding outputs.

regexp(string,expr,options) calls regexp with one or more of the nondefault options listed in the Command Options on page 2-3866 table, below. These options must follow the string and expr inputs in the argument list.

**Tips**

- For more information on this and other regular expression functions, see “Regular Expressions” in the MATLAB Programming Fundamentals documentation.
- The string and expr inputs are required. Enter these as the first and second arguments, respectively. Any other input arguments are optional. Enter optional inputs, in any order, following the two required inputs.

## Input Arguments

### **string**

A string or cell array of strings containing the string you want to parse. This string can be of any length and can contain any characters. This argument can also be a cell array of strings.

### **expr**

A string or cell array of strings containing a MATLAB regular expression. This input consists of text and operators with which you specify character patterns to look for in `string`.

This table shows the main categories of metacharacters and operators that you can use in `expr`.

| Category                   | Metacharacters and Operators                                                                                                                                                                                              |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “Character Type Operators” | One of a certain group of characters (for example, a character in a predefined set or range, a whitespace character, an alphabetic, numeric, or underscore character, or a character that is not in one of these groups). |
| “Character Representation” | Metacharacters that represent a special character (for example, backslash, new line, tab, hexadecimal values, any untranslated literal character, etc).                                                                   |
| “Grouping Operators”       | A grouping of letters or metacharacters to apply a regular expression operator to.                                                                                                                                        |
| “Nonmatching Operators”    | Text included in an expression for the purpose of adding a comment statement, but not to be used as a pattern to find a match for.                                                                                        |
| “Positional Operators”     | Location in the string where the characters or pattern must be positioned for there to be a match (for example, start or end of the string, start or end of a word, an entire word).                                      |



| Category                      | Metacharacters and Operators                                                                                                                                                                          |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “Lookaround Operators”        | Characters or patterns that immediately precede or follow the intended match, but are not considered to be part of the match itself.                                                                  |
| “Quantifiers”                 | Various ways of expressing the number of times a character or pattern is to occur for there to be a match (for example, exact number, minimum, maximum, zero or one, zero or more, one or more, etc.) |
| “Tokens”                      | Characters or patterns selected from the string being parsed that you can use to match other characters in the string.                                                                                |
| “Named Capture”               | Operators used in assigning names to matched tokens, thus making your code more maintainable and the output easier to interpret.                                                                      |
| “Conditional Expressions”     | Operators that express conditions under which a certain match is considered to be acceptable.                                                                                                         |
| “Dynamic Regular Expressions” | Operators that include a subexpression or command that MATLAB parses or executes. MATLAB uses the result of that operation in parsing the overall expression.                                         |
| “String Replacement”          | Operators used with the <code>regexprep</code> function to specify the content of the replacement text.                                                                                               |

Any text in the expression must be an exact match for at least part of the text in the parse string. Operators, on the other hand, are symbolic. Each operator symbol stands for a *type* of character (for example, an uppercase letter ([A-Z]), a space character (\s), four characters of any type (.{4})).

**outselect**

An optional comma-separated list of one to seven keywords. The presence of any of these keywords in the input argument list tells `regexp` to return the corresponding output.

| <b>outselect Keyword</b> | <b>Selected Return Value</b> |
|--------------------------|------------------------------|
| 'start'                  | matchstart                   |
| 'end'                    | matchend                     |
| 'tokenExtents'           | tokenindices                 |
| 'match'                  | matchstring                  |
| 'tokens'                 | tokenstring                  |
| 'names'                  | tokenname                    |
| 'split'                  | splitstring                  |

You must supply one output variable for each `outselect` keyword you include as an input argument. The order of the keyword in the input argument list determines the order of the corresponding output in the output argument list.

For a description of all return values, see “Output Arguments” on page 2-3869 .

## **options**

`regexp` accepts one or more of the following options.

## **Command Options**

| <b>Option</b> | <b>Description</b>                                                                                                                    |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 'once'        | Return only the first match found.                                                                                                    |
| 'warnings'    | Display warnings regarding potential issues parsing the expression. This option only enables warnings for the command being executed. |
| mode          | Modify the parameters of a search. See the section on mode that follows this table.                                                   |

**The mode Option.** You can fine-tune your regular expression parsing using the optional mode inputs: Case Sensitivity, Empty Match, Dot Matching, Anchor Type, and Spacing. There are two ways to use the regexp modes:

- Use a mode keyword (for example, 'lineanchors') to apply the mode to the entire input string.
- Use a mode flag (for example, (?m)) to apply the mode to selected parts of the input string. (Not available for Empty Match mode.)

The mode option is available for the regexp, regexpi, and regexprep functions. For more information about regexp modes, see “Modifying Parameters of the Search (Modes)” in the MATLAB “Programming Fundamentals” documentation.

### Case Sensitivity Mode

Use the Case Sensitivity mode to control whether or not MATLAB considers letter case when matching an expression to a string.

| Mode Keyword | Flag  | Description                                                                                |
|--------------|-------|--------------------------------------------------------------------------------------------|
| 'matchcase'  | (?-i) | Letter case must match when matching patterns to a string. (The default for regexp).       |
| 'ignorecase' | (?i)  | Do not consider letter case when matching patterns to a string. (The default for regexpi). |

### Empty Match Mode

Use Empty Match mode to allow successful matches of length zero. Using this mode, you can match a location in a string in which nothing but the assertion is true.

| Mode Keyword   | Description                                        |
|----------------|----------------------------------------------------|
| 'noemptymatch' | Ignore zero length matches. (This is the default). |
| 'emptymatch'   | Allow matches of length zero.                      |

## Dot Matching Mode

Use the Dot Matching mode to control whether or not MATLAB includes the newline (`\n`) character when matching the dot (`.`) metacharacter in a regular expression.

| Mode Keyword       | Flag                 | Description                                                                                   |
|--------------------|----------------------|-----------------------------------------------------------------------------------------------|
| 'dotall'           | ( <code>?s</code> )  | Match dot ( <code>.</code> ) in the pattern string with any character. (This is the default). |
| 'dotexceptnewline' | ( <code>?-s</code> ) | Match dot in the pattern with any character that is not a newline.                            |

## Anchor Type Mode

Use the Anchor Type mode to control whether MATLAB considers the `^` and `$` metacharacters to represent the beginning and end of a string or the beginning and end of a line.

| Mode Keyword    | Flag                 | Description                                                                                                              |
|-----------------|----------------------|--------------------------------------------------------------------------------------------------------------------------|
| 'stringanchors' | ( <code>?-m</code> ) | Match the <code>^</code> and <code>\$</code> metacharacters at the beginning and end of a string. (This is the default). |
| 'lineanchors'   | ( <code>?m</code> )  | Match the <code>^</code> and <code>\$</code> metacharacters at the beginning and end of a line.                          |

### Spacing Mode

Use the Spacing mode to control how MATLAB interprets space characters and comments within the parsing string. Note that spacing mode applies to the parsing string (the second input argument that contains the metacharacters (for example, `\w`) and not the string being parsed.

| Mode Keyword              | Flag  | Description                                                                                                                                                         |
|---------------------------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ' <b>literalspacing</b> ' | (?-x) | Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (This is the default). |
| ' <b>freespacing</b> '    | (?x)  | Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)                                                   |

### Output Arguments

Each of the first seven outputs listed below is returned as a 1-by-*m* array, where *m* is the number of matches found by `regexp`.

When parsing multiple strings (i.e., a 1-by-*n* or *n*-by-1 cell array of strings), `regexp` returns a 1-by-*n* cell array for each output specified. When parsing an *m*-by-*n* cell array of strings, `regexp` returns an *m*-by-*n* cell array for each output specified.

#### **matchstart**

The starting index of each substring of `string` that matches `expression`, `expr`. The output is an array of class `double`.

#### **matchend**

The ending index of each substring of `string` that matches `expression`, `expr`. The output is an array of class `double`.

To obtain this return value, you must call `regexp` with at least two output variables, or specify the `'end'` keyword as one of the `outselect` input arguments.

## **tokenindices**

The starting and ending indices of each substring of `string` that matches a token in `expr`. The output is a cell array, each cell of which contains an array of class `double`. The size of this inner array is `m-by-2`, where `m` is the number of tokens captured by the match. Any cells that do not represent matched tokens are returned empty. (The output is a `double` when you call `regexp` with the `'once'` option.)

To obtain this return value, you must call `regexp` with at least three output variables, or specify the `'tokenExtents'` keyword as one of the `outselect` input arguments.

## **matchstring**

The text of each substring of `string` that is a match. The output is a cell array of strings, (or a single string when you call `regexp` with the `'once'` option).

To obtain this return value, you must call `regexp` with at least four output variables, or specify the `'match'` keyword as one of the `outselect` input arguments.

## **tokenstring**

The text of each token captured by `regexp`. The output is a cell array of 1-by-`n` cell arrays, where `n` is the number of token expressions specified in the `expr` input. (The output is a cell array of strings when you call `regexp` with the `'once'` option).

To obtain this return value, you must call `regexp` with at least five output variables, or specify the `'tokens'` keyword as one of the `outselect` input arguments.

## **tokenname**

The name and text of each *named* token captured by `regexp`. The output is an array of structures, each containing `n` fields, where `n` is the number of token expressions specified in the `expr` input. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression (`?<tokenname>`). If there are no named tokens in `expr`, `regexp` returns a structure array with no fields.

To obtain this return value, you must call `regexp` with at least six output variables, or specify the `'names'` keyword as one of the `outselect` input arguments.

### **splitstring**

Those parts of the input string that are delimited by substrings matched by the expression. The output is a cell array of strings. Any cells that represent strings of zero length are returned empty. When using the `'split'` keyword, `regexp` returns one more string than the number of matches in the string.

To obtain this return value, you must call `regexp` with all seven output variables, or specify the `'split'` keyword as one of `outselect` the input arguments.

### **selected\_outputs**

From one to seven of the output values listed above. Select which outputs to include using the `outselect` input arguments. Specify which outputs you want returned and in what order you want them returned using the keywords shown in the table.

For example, to have `regexp` return the starting index and matching text, specify the keywords `'start'` and `'match'`

```
[index, matstr] = regexp('Jan 24', '\d', ...
    'start', 'match')
```

| Selected Outputs | Keyword Required |
|------------------|------------------|
| matchstart       | 'start'          |
| matchend         | 'end'            |
| tokenindices     | 'tokenExtents'   |
| matchstring      | 'match'          |
| tokenstring      | 'tokens'         |
| tokenname        | 'names'          |
| splitstring      | 'split'          |

## Examples

Return a row vector of indices that match words that start with `c`, end with `t`, and contain one or more vowels between them. Make the matches sensitive to letter case (by using the case-sensitive `regexp`):

```
str = 'bat cat can car COAT court cut ct CAT-scan';
regexp(str, 'c[aeiou]+t')
ans =
     5     28
```

---

Return a cell array of row vectors of indices that match capital letters and white spaces in the cell array of strings `str`:

```
str = {'Madrid, Spain' 'Romeo and Juliet' 'MATLAB is great'};
s1 = regexp(str, '[A-Z]');
s2 = regexp(str, '\s');
```

Capital letters, '`[A-Z]`', were found at these `str` indices:

```
s1{:}
ans =
     1     9
ans =
```



```

    1    11
ans =
    1     2     3     4     5     6

```

Space characters, '\s', were found at these str indices:

```

s2{:}
ans =
    8
ans =
    6    10
ans =
    7    10

```

Return the text and the starting and ending indices of words containing the letter x:

```

str = 'REGEXP helps you relax';
[m s e] = regexp(str, '\w*x\w*', 'match', 'start', 'end')
m =
    'relax'
s =
    18
e =
    22

```

Find the substrings delimited by the ^ character:

```

s1 = ['Use REGEXP to split ^this string into ' ...
      'several ^individual pieces'];

s2 = regexp(s1, '\^', 'split');

s2{:}
ans =

```

```
'Use REGEXP to split '  
'this string into several '  
'individual pieces'
```

---

Use the **'split'** keyword to return those parts of the input string that are not returned when using **'match'**. Note that when you match the beginning or ending characters in a string (as is done in this example), the first (or last) return value is always an empty string:

```
str = 'She sells sea shells by the seashore.';  
  
[matchstr splitstr] = regexp(str, '[Ss]h.', 'match', ...  
                             'split')  
matchstr =  
    'She'    'she'    'sho'  
splitstr =  
    ''    ' sells sea '    'lls by the sea'    're.'
```

For any string that has been split, you can reassemble the pieces into the initial string using the command

```
j = [splitstr; [matchstr {''}]]; [j{:}]  
  
ans =  
    She sells sea shells by the seashore.
```

---

Search a string for opening and closing HTML tags. Use the expression `<(\w+)` to find the opening tag (for example, `<tagname'`) and to create a token for it. Use the expression `</\1>` to find another occurrence of the same token, but formatted as a closing tag (for example, `</tagname>'`):

```
str = ['if <code>A</code> == x<sup>2</sup>, ' ...  
      '<em>disp(x)</em>']  
str =  
if <code>A</code> == x<sup>2</sup>, <em>disp(x)</em>
```

```

expr = '<(\w+).*?>.*?</\1>';

[tok mat] = regexp(str, expr, 'tokens', 'match');

tok{:}
ans =
    'code'
ans =
    'sup'
ans =
    'em'

mat{:}
ans =
    <code>A</code>
ans =
    <sup>2</sup>
ans =
    <em>disp(x)</em>

```

For information on using tokens, see “Tokens” in the MATLAB Programming Fundamentals documentation.

---

- 1 Enter a string containing two names, the first and last names being in a different order:

```

str = sprintf('John Davis\nRogers, James')
str =
    John Davis
    Rogers, James

```

- 2 Create an expression that generates first and last name tokens, assigning the names `first` and `last` to the tokens. Call `regexp` to get the text and names of each token found:

```

expr = ...

```

```
'(?<first>\w+)\s+(?<last>\w+)|(?<last>\w+)\s+(?<first>\w+)';
```

```
[tokens names] = regexp(str, expr, 'tokens', 'names');
```

- 3** Examine the `tokens` cell array that was returned. The first and last name tokens appear in the order in which they were generated: first name–last name, then last name–first name:

```
tokens{:}
ans =
    'John'    'Davis'
ans =
    'Rogers'  'James'
```

- 4** Now examine the `names` structure that was returned. First and last names appear in a more usable order:

```
names(:,1)
ans =
    first: 'John'
    last:  'Davis'

names(:,2)
ans =
    first: 'James'
    last:  'Rogers'
```

---

Use Case Sensitivity mode with `regexp`.

- 1** Given a string that has both uppercase and lowercase letters, use the `regexp` default mode (case-sensitive) to locate only the lowercase instance of the word `case`:

```
str = 'A string with UPPERCASE and lowercase text.';

regexp(str, 'case', 'match')
ans =
```

```
'case'
```

**2** Now disable case-sensitive matching to find both instances of case:

```
regexp(str, 'case', 'ignorecase', 'match')
ans =
    'CASE'    'case'
```

**3** Match 5 letters that are followed by 'CASE'. Use the (?-i) flag to turn off case insensitivity for the first match and (?i) to turn insensitivity on for the second:

```
M = regexp(str, {'(?-i)\w{5}(?=CASE)', ...
                '(?i)\w{5}(?=CASE)'} , 'match');

M{:}
ans =
    'UPPER'
ans =
    'UPPER'    'lower'
```

Use Empty Match mode with `regexp` to add the characters '=>' to the beginning of each line in the following text:

```
text = {'Use the REGEXPREP function in '; ...
        'Empty Match mode to insert a right '; ...
        'arrow at the start of each line.'};

newtext = regexprep(text, '^', '=>', 'emptymatch')

newtext =
'=> Use the REGEXPREP function in '
'=> Empty Match mode to insert a right '
'=> arrow at the start of each line.'
```

The `^` operator matches the beginning of each string. Because there are no other characters to match, and the `emptymatch` mode is enabled by

the fifth input argument of the command, there is an empty match at the beginning of each string.

---

- 1 Search the string 'MATLAB' for zero or more of the characters M, A, and T. Do not include matches of zero-length. (The mode defaults to `noemptymatch`):

```
regexp('MATLAB', '[MAT]*', 'match')
ans =
    'MAT'    'A'
```

- 2 Repeat this search, but this time specify `emptymatch` mode to include matches of zero-length.

---

**Note** Note that there is also a match for the position that follows the final character.

---

```
regexp('MATLAB', '[MAT]*', 'emptymatch', 'match')
ans =
    'MAT'    ''    'A'    ''    ''
```

- 3 Now, use the + (one or more) operator instead of the \* (zero or more) operator. Continue to use `emptymatch` mode:

```
regexp('MATLAB', '[MAT]+', 'emptymatch', 'match')
ans =
    'MAT'    'A'
```

No match is found for the L and B characters this time because `regexp` needs to find a least one occurrence of these characters for a match to be found.

---

Use Dot Matching mode with `regexp` to parse the following string, which contains a newline (`\n`) character:

```
str = sprintf('abc\ndef')
str =
    abc
    def
```

When you use the default mode, `dotall`, MATLAB includes the newline in the characters matched:

```
regexp(str, '.', 'match')
ans =
    'a'    'b'    'c'    [1x1 char]    'd'    'e'    'f'
```

When you use the `dotexceptnewline` mode, MATLAB skips the newline character:

```
regexp(str, '.', 'match', 'dotexceptnewline')
ans =
    'a'    'b'    'c'    'd'    'e'    'f'
```

Use Anchor Type mode with `regexp` to control whether you parse a multiline string line by line, or once for the entire string. Use the following two-line string:

```
str = sprintf('%s\n%s', 'Here is the first line', ...
              'followed by the second line')
str =
    Here is the first line
    followed by the second line
```

In `stringanchors` mode, MATLAB interprets the `$` metacharacter as an end-of-string specifier, and thus finds the last two words of the entire *string*:

```
regexp(str, '\w+\W\w+$', 'match', 'stringanchors')
```

```
ans =  
    'second line'
```

While in `lineanchors` mode, MATLAB interprets `$` as an end-of-line specifier, and finds the last two words of each *line*:

```
regexp(str, '\w+\W\w+$', 'match', 'lineanchors')  
ans =  
    'first line'    'second line'
```

---

Use Spacing mode with `regexp` to read a regular expression from a file, retrieving all operators and metacharacters from the file, but skipping any comment text:

- 1 Create a file called `regexp_str.txt` containing the following text.

```
(?x) # turn on freespacing.  
  
# This pattern matches a string with a repeated letter.  
  
\w* # First, match any number of preceding word characters.  
  
( # Mark a token.  
. # Match a character of any type.  
) # Finish capturing said token.  
\1 # Backreference to match what token #1 matched.  
  
\w* # Finally, match the remainder of the word.
```

- 2 Use the pattern expression read from the file to find those words that have consecutive matching letters. Because the first line enables freespacing mode, MATLAB ignores all spaces and comments that appear in the file:

```
str = ['Looking for words with letters that ' ...  
      'appear twice in succession.'];  
patt = fileread('regexp_str.txt');
```



```
regexp(str, patt, 'match')
ans =
    'Looking'    'letters'    'appear'    'succession'
```

---

Debug problems in parsing a string with `regexp`, `regexpi`, or `regexprep`, using the **'warnings'** option to view all warning messages:

```
regexp('$.', '[a-]', 'warnings')
Warning: Unbound range.
[a-]
|
```

## See Also

`regexpi` | `regexprep` | `regexprtranslate` | `strfind` | `strcmp` | `strcmpi` | `strncmp` | `strncmpi` | `strrep`

## How To

- “Regular Expressions”

# regexpi

---

**Purpose** Match regular expression (case insensitive)

**Syntax**

```
regexpi(string,expr)
[matchstart,matchend,tokenindices,matchstring,tokenstring,
 tokenname,splitstring] = regexpi(string,expr)
[selected_outputs] = regexpi(string,expr,outselect)
regexpi(string,expr,options)
```

**Description** `regexpi(string,expr)` parses the input `string` without regard to letter case. The function locates those parts of `string` that match the character pattern(s) specified by regular expression, `expr`. This syntax returns the starting index of each match. If no matches are found, `regexpi` returns an empty array.

`[matchstart,matchend,tokenindices,matchstring,tokenstring,tokenname,splitstring] = regexpi(string,expr)` returns from one to seven output values, depending on the number of output variables you specify.

`[selected_outputs] = regexpi(string,expr,outselect)` returns from one to seven output values, depending on which flags you specify in `outselect`. The presence and ordering of the `outselect` inputs determine the presence and ordering of the corresponding outputs.

`regexpi(string,expr,options)` calls `regexpi` with one or more of the nondefault options listed in the Command Options on page 2-3885 table, below. These options must follow the `string` and `expr` inputs in the argument list.

## Tips

- For more information on this and other regular expression functions, see “Regular Expressions” in the MATLAB Programming Fundamentals documentation.
- The `string` and `expr` inputs are required. Enter these as the first and second arguments, respectively. Any other input arguments are optional. Enter optional inputs, in any order, following the two required inputs.

**Input Arguments**

**string**

A string or cell array of strings containing the string you want to parse. This string can be of any length and can contain any characters. This argument can also be a cell array of strings.

**expr**

A string or cell array of strings containing a MATLAB regular expression. This input consists of text and operators with which you specify character patterns to look for in `string`.

This table shows the main categories of metacharacters and operators that you can use in `expr`.

| Category                   | Metacharacters and Operators                                                                                                                                                                                              |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "Character Type Operators" | One of a certain group of characters (for example, a character in a predefined set or range, a whitespace character, an alphabetic, numeric, or underscore character, or a character that is not in one of these groups). |
| "Character Representation" | Metacharacters that represent a special character (for example, backslash, new line, tab, hexadecimal values, any untranslated literal character, etc).                                                                   |
| "Grouping Operators"       | A grouping of letters or metacharacters to apply a regular expression operator to.                                                                                                                                        |
| "Nonmatching Operators"    | Text included in an expression for the purpose of adding a comment statement, but not to be used as a pattern to find a match for.                                                                                        |
| "Positional Operators"     | Location in the string where the characters or pattern must be positioned for there to be a match (for example, start or end of the string, start or end of a word, an entire word).                                      |

# regexpi

| Category                      | Metacharacters and Operators                                                                                                                                                                          |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “Lookaround Operators”        | Characters or patterns that immediately precede or follow the intended match, but are not considered to be part of the match itself.                                                                  |
| “Quantifiers”                 | Various ways of expressing the number of times a character or pattern is to occur for there to be a match (for example, exact number, minimum, maximum, zero or one, zero or more, one or more, etc.) |
| “Tokens”                      | Characters or patterns selected from the string being parsed that you can use to match other characters in the string.                                                                                |
| “Named Capture”               | Operators used in assigning names to matched tokens, thus making your code more maintainable and the output easier to interpret.                                                                      |
| “Conditional Expressions”     | Operators that express conditions under which a certain match is considered to be acceptable.                                                                                                         |
| “Dynamic Regular Expressions” | Operators that include a subexpression or command that MATLAB parses or executes. MATLAB uses the result of that operation in parsing the overall expression.                                         |
| “String Replacement”          | Operators used with the <code>regexprep</code> function to specify the content of the replacement text.                                                                                               |

Any text in the expression must be a case-insensitive match with part of the text in the parse string. Operators, on the other hand, are symbolic. Each operator symbol stands for a *type* of character (for example, an uppercase letter ([A-Z]), a space character (\s), four characters of any type (.{4})).

## **outselect**

An optional comma-separated list of one to seven keywords. The presence of any of these keywords in the input argument list tells `regexpi` to return the corresponding output.

| outselect Keyword | Selected Return Value |
|-------------------|-----------------------|
| 'start'           | matchstart            |
| 'end'             | matchend              |
| 'tokenExtents'    | tokenindices          |
| 'match'           | matchstring           |
| 'tokens'          | tokenstring           |
| 'names'           | tokenname             |
| 'split'           | splitstring           |

You must supply one output variable for each outselect keyword you include as an input argument. The order of the keyword in the input argument list determines the order of the corresponding output in the output argument list.

For a description of all return values, see “Output Arguments” on page 2-3869 .

### options

regexp accepts one or more of the following options.

### Command Options

| Option     | Description                                                                                                                           |
|------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 'once'     | Return only the first match found.                                                                                                    |
| 'warnings' | Display warnings regarding potential issues parsing the expression. This option only enables warnings for the command being executed. |
| mode       | Modify the parameters of a search. See the section on mode that follows this table.                                                   |

**The mode Option.** You can fine-tune your regular expression parsing using the optional mode inputs: Case Sensitivity, Empty Match, Dot Matching, Anchor Type, and Spacing. There are two ways to use the `regex` modes:

- Use a mode keyword (for example, `'lineanchors'`) to apply the mode to the entire input string.
- Use a mode flag (for example, `(?m)`) to apply the mode to selected parts of the input string. (Not available for Empty Match mode.)

The mode option is available for the `regex`, `regexpi`, and `regexprep` functions. For more information about `regex` modes, see “Modifying Parameters of the Search (Modes)” in the MATLAB “Programming Fundamentals” documentation.

## Case Sensitivity Mode

Use the Case Sensitivity mode to control whether or not MATLAB considers letter case when matching an expression to a string.

| Mode Keyword              | Flag               | Description                                                                                              |
|---------------------------|--------------------|----------------------------------------------------------------------------------------------------------|
| <code>'matchcase'</code>  | <code>(?-i)</code> | Letter case must match when matching patterns to a string. (The default for <code>regex</code> ).        |
| <code>'ignorecase'</code> | <code>(?i)</code>  | Do not consider letter case when matching patterns to a string. (The default for <code>regexpi</code> ). |

## Empty Match Mode

Use Empty Match mode to allow successful matches of length zero. Using this mode, you can match a location in a string in which nothing but the assertion is true.

| Mode Keyword   | Description                                        |
|----------------|----------------------------------------------------|
| 'noemptymatch' | Ignore zero length matches. (This is the default). |
| 'emptymatch'   | Allow matches of length zero.                      |

### Dot Matching Mode

Use the Dot Matching mode to control whether or not MATLAB includes the newline (\n) character when matching the dot (.) metacharacter in a regular expression.

| Mode Keyword       | Flag  | Description                                                                      |
|--------------------|-------|----------------------------------------------------------------------------------|
| 'dotall'           | (?s)  | Match dot ('.') in the pattern string with any character. (This is the default). |
| 'dotexceptnewline' | (?-s) | Match dot in the pattern with any character that is not a newline.               |

### Anchor Type Mode

Use the Anchor Type mode to control whether MATLAB considers the ^ and \$ metacharacters to represent the beginning and end of a string or the beginning and end of a line.

| Mode Keyword    | Flag  | Description                                                                                    |
|-----------------|-------|------------------------------------------------------------------------------------------------|
| 'stringanchors' | (?-m) | Match the ^ and \$ metacharacters at the beginning and end of a string. (This is the default). |
| 'lineanchors'   | (?m)  | Match the ^ and \$ metacharacters at the beginning and end of a line.                          |

## Spacing Mode

Use the Spacing mode to control how MATLAB interprets space characters and comments within the parsing string. Note that spacing mode applies to the parsing string (the second input argument that contains the metacharacters (for example, `\w`) and not the string being parsed.

| Mode Keyword              | Flag  | Description                                                                                                                                                         |
|---------------------------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ' <b>literalspacing</b> ' | (?-x) | Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (This is the default). |
| ' <b>freespacing</b> '    | (?x)  | Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)                                                   |

## Output Arguments

Each of the first seven outputs listed below is returned as a 1-by- $m$  array, where  $m$  is the number of matches found by `regexpi`.

When parsing multiple strings (i.e., a 1-by- $n$  or  $n$ -by-1 cell array of strings), `regexpi` returns a 1-by- $n$  cell array for each output specified. When parsing an  $m$ -by- $n$  cell array of strings, `regexpi` returns an  $m$ -by- $n$  cell array for each output specified.

### **matchstart**

The starting index of each substring of `string` that matches `expression`, `expr`. The output is an array of class `double`.

### **matchend**

The ending index of each substring of `string` that matches `expression`, `expr`. The output is an array of class `double`.



To obtain this return value, you must call `regexpi` with at least two output variables, or specify the `'end'` keyword as one of the `outselect` input arguments.

### **tokenindices**

The starting and ending indices of each substring of `string` that matches a token in `expr`. The output is a cell array, each cell of which contains an array of class `double`. The size of this inner array is `m-by-2`, where `m` is the number of tokens captured by the match. Any cells that do not represent matched tokens are returned empty. (The output is a `double` when you call `regex` with the `'once'` option.)

To obtain this return value, you must call `regex` with at least three output variables, or specify the `'tokenExtents'` keyword as one of the `outselect` input arguments.

### **matchstring**

The text of each substring of `string` that is a match. The output is a cell array of strings, (or a single string when you call `regexpi` with the `'once'` option).

To obtain this return value, you must call `regexpi` with at least four output variables, or specify the `'match'` keyword as one of the `outselect` input arguments.

### **tokenstring**

The text of each token captured by `regexpi`. The output is a cell array of 1-by-`n` cell arrays, where `n` is the number of token expressions specified in the `expr` input. (The output is a cell array of strings when you call `regexpi` with the `'once'` option).

To obtain this return value, you must call `regexpi` with at least five output variables, or specify the `'tokens'` keyword as one of the `outselect` input arguments.

### **tokenname**

The name and text of each *named* token captured by `regexpi`. The output is an array of structures, each containing *n* fields, where *n* is the number of token expressions specified in the `expr` input. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression (`?<tokenname>`). If there are no named tokens in `expr`, `regexpi` returns a structure array with no fields.

To obtain this return value, you must call `regexpi` with at least six output variables, or specify the `'names'` keyword as one of the `outselect` input arguments.

## **splitstring**

Those parts of the input string that are delimited by substrings matched by the expression. The output is a cell array of strings. Any cells that represent strings of zero length are returned empty. When using the `'split'` keyword, `regexp` returns one more string than the number of matches in the string.

To obtain this return value, you must call `regexp` with all seven output variables, or specify the `'split'` keyword as one of `outselect` the input arguments.

## **selected\_outputs**

From one to seven of the output values listed above. Select which outputs to include using the `outselect` input arguments. Specify which outputs you want returned and in what order you want them returned using the keywords shown in the table.

For example, to have `regexpi` return the starting index and matching text, specify the keywords `'start'` and `'match'`.

```
[index, matstr] = regexpi('Jan 24', '\d', ...  
'start', 'match')
```

| Selected Outputs | Keyword Required |
|------------------|------------------|
| matchstart       | 'start'          |
| matchend         | 'end'            |
| tokenindices     | 'tokenExtents'   |
| matchstring      | 'match'          |
| tokenstring      | 'tokens'         |
| tokenname        | 'names'          |
| splitstring      | 'split'          |

## Examples

Return a row vector of indices that match words that start with c, end with t, and contain one or more vowels between them. Make the matches insensitive to letter case (by using `regexpi` instead of `regex`):

```
str = 'bat cat can car COAT court cut ct CAT-scan';
regexpi(str, 'c[aeiou]+t')
ans =
     5     17     28     35
```

Use Case Sensitivity mode with `regexpi` to match 5 letters that are followed by 'CASE'. The first match defaults to case insensitivity. Use the `(?-i)` flag to turn on case sensitivity for the second match:

```
str = 'A string with UPPERCASE and lowercase text.';
M = regexpi(str, {'\w{5}(?=CASE)', ...
    '(?-i)\w{5}(?=CASE)'}, 'match');

M{:}
ans =
    'UPPER'    'lower'
ans =
    'UPPER'
```

# regexpi

---

---

Debug problems in parsing a string with `regexpi`, `regextpi`, or `regexprep`, use the '**warnings**' option to view all warning messages:

```
regexpi('$.', '[a-]', 'warnings')
Warning: Unbound range.
[a-]
|
```

## See Also

`regexp` | `regexprep` | `regextpranslate` | `strfind` | `strcmpi` | `strcmp`  
| `strncmpi` | `strncmp`

## How To

- “Regular Expressions”

**Purpose** Replace string using regular expression

**Syntax** `s = regexprep('str', 'expr', 'repstr')`  
`s = regexprep('str', 'expr', 'repstr', options)`

**Description** `s = regexprep('str', 'expr', 'repstr')` replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`. You can use character representations (e.g., `'\t'` for tab, or `'\n'` for newline) in replacement string `repstr`. See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for more information.

If `str` is a cell array of strings, then the `regexprep` return value `s` is always a cell array of strings having the same dimensions as `str`.

To specify more than one expression to match or more than one replacement string, see the guidelines listed below under “Multiple Expressions or Replacement Strings” on page 2-3894.

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the `(...)` operator. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See “Tokens” and the example “Using Tokens in a Replacement String” in the MATLAB Programming Fundamentals documentation for information on using tokens.)

`s = regexprep('str', 'expr', 'repstr', options)` By default, `regexprep` replaces all matches and is case sensitive. You can use one or more of the following options with `regexprep`.

| Option              | Description                                                                              |
|---------------------|------------------------------------------------------------------------------------------|
| <code>mode</code>   | See <code>mode</code> descriptions on the <code>regexp</code> reference page.            |
| <code>N</code>      | Replace only the <code>N</code> th occurrence of <code>expr</code> in <code>str</code> . |
| <code>'once'</code> | Replace only the first occurrence of <code>expr</code> in <code>str</code> .             |

| Option         | Description                                                                                                                                                     |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'ignorecase'   | Ignore case when matching and when replacing.                                                                                                                   |
| 'preserveCase' | Ignore case when matching (as with 'ignorecase'), but override the case of replace characters with the case of corresponding characters in str when replacing.  |
| 'warnings'     | Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed. |

## Tips

See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for a listing of all regular expression metacharacters supported by MATLAB.

### Multiple Expressions or Replacement Strings

In the case of multiple expressions and/or replacement strings, `regexprep` attempts to make all matches and replacements. The first match is against the initial input string. Successive matches are against the string resulting from the previous replacement.

The `expr` and `repstr` inputs follow these rules:

- If `expr` is a cell array of strings and `repstr` is a single string, `regexprep` uses the same replacement string on each expression in `expr`.
- If `expr` is a single string and `repstr` is a cell array of N strings, `regexprep` attempts to make N matches and replacements.
- If both `expr` and `repstr` are cell arrays of strings, then `expr` and `repstr` must contain the same number of elements, and `regexprep` pairs each `repstr` element with its matching element in `expr`.

## Examples

### Example 1 – Making a Case-Sensitive Replacement

Perform a case-sensitive replacement on words starting with `m` and ending with `y`:

```
str = 'My flowers may bloom in May';
pat = 'm(\w*)y';
regexprep(str, pat, 'April')
```

```
ans =
    My flowers April bloom in May
```

Replace all words starting with `m` and ending with `y`, regardless of case, but maintain the original case in the replacement strings:

```
regexprep(str, pat, 'April', 'preserveCase')
```

```
ans =
    April flowers april bloom in April
```

### Example 2 – Using Tokens In the Replacement String

Replace all variations of the words 'walk up' using the letters following walk as a token. In the replacement string

```
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.
```

### Example 3 – Operating on Multiple Strings

This example operates on a cell array of strings. It searches for consecutive matching letters (e.g., 'oo') and uses a common replacement value ('--') for all matches. The function returns a cell array of strings having the same dimensions as the input cell array:

```
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.}';

a = regexprep(str, '(.)\1', '--', 'ignorecase')
```

```
a =
    '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.'
```

## Example 4 – Replacing Zero Length Matches

Request that `regexprep` replace text when the match is of zero length using the `emptymatch` flag.

```
regexprep('my_string', '^', 'BEGIN ', 'emptymatch')
```

```
ans =
    BEGIN my_string
```

MATLAB finds a zero length match for the `^` (beginning of string) operator just before the `my_string` input, and inserts the replacement characters `'BEGIN '` at that location.

## See Also

`regexp` | `regexpi` | `regexprtranslate` | `strfind` | `strcmp` | `strcmpi` | `strncmp` | `strncmppi` | `strep`

## How To

- “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 <code>s1</code> so that they are treated as literal characters when used in the <code>regexp</code> and <code>regexprep</code> functions. The translation inserts an escape character ('\') before each special character in <code>s1</code> . Return the new string in <code>s2</code> .               |
| 'wildcard' | Translate all wildcard and '.' characters in string <code>s1</code> so that they are treated as literal wildcards and periods when used in the <code>regexp</code> and <code>regexprep</code> functions. The translation replaces all instances of '*' with '.', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in <code>s2</code> . |

## Examples

### Example 1 – Using the 'escape' Option

Because `regexp` interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\n' and '\n' in this string:

```
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
ans =
     []
```

# regexptranslate

---

To have `regex` interpret the expression `expr` as the characters ``\`` and ``n``, first translate the expression using `regexptranslate`:

```
pat2 = regexptranslate('escape', pat)
pat2 =
    \\n

regex(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'];
regex(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](#) | [regprep](#)

## How To

- “Regular Expressions”

# registerevent

---

**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 the MATLAB Program”
- “Using the MATLAB Search Path”

# release

---

**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.

**Examples** **1** Create an instance of a Microsoft Calendar control. Get a `TitleFont` interface and use it to change the appearance of the calendar title font:

```
f = figure('position',[300 300 500 500]);  
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);
```

```
TFont = cal.TitleFont;  
TFont.Name = 'Viva BoldExtraExtended';  
TFont.Bold = 0;
```

**2** After working with the title font, release the `TitleFont` interface:

```
TFont.release;
```

**3** Delete the `cal` object and the figure window:

```
cal.delete;  
delete(f);  
clear f;
```

**See Also** `delete (COM)` | `actxcontrol` | `actxserver`



**How To**

- Releasing Interfaces

# relationaloperators (handle)

---

**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`

# rem

---

**Purpose**            Remainder after division

**Syntax**             $R = \text{rem}(X, Y)$

**Description**         $R = \text{rem}(X, Y)$  if  $Y \neq 0$ , returns  $X - n \cdot Y$  where  $n = \text{fix}(X./Y)$ . 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:

- $\text{rem}(X, 0)$  is NaN
- $\text{rem}(X, X)$  for  $X \neq 0$  is 0
- $\text{rem}(X, Y)$  for  $X \neq Y$  and  $Y \neq 0$  has the same sign as  $X$ .

**Tips**                 $\text{mod}(X, Y)$  for  $X \neq Y$  and  $Y \neq 0$  has the same sign as  $Y$ .

$\text{rem}(X, Y)$  and  $\text{mod}(X, Y)$  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  $\text{sign}(X) \cdot \text{abs}(Y)$ . If  $Y$  is zero, `rem` returns NaN.

**See Also**            `mod`

|                        |                                                                                                                                                                                                       |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Remove key-value pairs from containers.Map object                                                                                                                                                     |
| <b>Syntax</b>          | <code>remove(mapObj, keySet)</code>                                                                                                                                                                   |
| <b>Description</b>     | <code>remove(mapObj, keySet)</code> erases all specified keys, and the values associated with them, from <code>mapObj</code> . Input <code>keySet</code> can be a scalar key or a cell array of keys. |
| <b>Input Arguments</b> | <p><b>mapObj</b><br/>Object of class <code>containers.Map</code>.</p> <p><b>keySet</b><br/>Scalar value, string, or cell array that specifies keys in <code>mapObj</code> to delete.</p>              |

## 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'};
```

## containers.Map.remove

---

```
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](#)

**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:
```

# removets

---

```
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](#)



**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`

# repmat

---

**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 NaNs.

### 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](#)

# resample (tscollection)

---

**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
```

# resample (tscollection)

---

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`

**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`

# reset (RandStream)

---

**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](#)

# reshape

---

## 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 `prod(size(A))`.

`B = reshape(A,...,[ ],...)` calculates the length of the dimension represented by the placeholder `[ ]`, such that the product of the dimensions equals `prod(size(A))`. The value of `prod(size(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 `prod(size(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](#) (:)

# residue

---

**Purpose** Convert between partial fraction expansion and polynomial coefficients

**Syntax**  $[r,p,k] = \text{residue}(b,a)$   
 $[b,a] = \text{residue}(r,p,k)$

**Description** The residue function converts a quotient of polynomials to pole-residue representation, and back again.

$[r,p,k] = \text{residue}(b,a)$  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

$$\frac{b(s)}{a(s)} = \frac{b_1s^m + b_2s^{m-1} + b_3s^{m-2} + \dots + b_{m+1}}{a_1s^n + a_2s^{n-1} + a_3s^{n-2} + \dots + a_{n+1}}$$

where  $b_j$  and  $a_j$  are the  $j$ th elements of the input vectors  $b$  and  $a$ .

$[b,a] = \text{residue}(r,p,k)$  converts the partial fraction expansion back to the polynomials with coefficients in  $b$  and  $a$ .

**Definitions** If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \dots + \frac{r_n}{s-p_n} + k(s)$$

The number of poles  $n$  is

$$n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)$$

The direct term coefficient vector is empty if  $\text{length}(b) < \text{length}(a)$ ; otherwise

$$\text{length}(k) = \text{length}(b) - \text{length}(a) + 1$$

If  $p(j) = \dots = p(j+m-1)$  is a pole of multiplicity  $m$ , then the expansion includes terms of the form

$$\frac{r_j}{s-p_j} + \frac{r_j+1}{(s-p_j)^2} + \dots + \frac{r_j+m-1}{(s-p_j)^m}$$

## 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

$$\begin{aligned} \mathbf{b} &= [ 5 \ 3 \ -2 \ 7 ] \\ \mathbf{a} &= [ -4 \ 0 \ 8 \ 3 ] \end{aligned}$$

and you can calculate the partial fraction expansion as

# residue

---

```
[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. Schaffer, *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**

- “Recovering from Problems with the Search Path”
- “Using the MATLAB Search Path”



**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                                        |
|-------------------------|----------------------------------------------------|
| <code>message</code>    | Text of the error message                          |
| <code>identifier</code> | Message identifier of the error message            |
| <code>stack</code>      | 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.

# rethrow

---

## 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

**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

## rethrow (MException)

---

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                                      |
|-------------------------------|--------------------------------------------------|
| <code>line</code>             | Line number from which the exception was thrown. |
| <code>name</code>             | Name of the function being executed at the time. |
| <code>file</code>             | 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)
```

## rethrow (MException)

---

```
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)
```

**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

# Tiff.rewriteDirectory

---

**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”



**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`

# rgb2ind

---

**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** `rgb2ind` converts RGB images to indexed images using one of these methods:

- Uniform quantization
- Minimum variance quantization
- Colormap approximation

For all these methods, `rgb2ind` also dithers the image unless you specify 'nodither' for `dither_option`.

`[X,map] = rgb2ind(RGB, n)` converts the RGB image to an indexed image `X` using minimum variance quantization. `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` by matching colors in `RGB` with the nearest color in the colormap `map`. `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. `map` contains at most  $(\text{floor}(1/\text{tol})+1)^3$  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 cannot 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`.

## Tips

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

$$n = (\text{floor}(1/\text{tol})+1)^3$$

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is `n`. `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap can be much smaller than `n`.

If you specify `n`, `rgb2ind` uses minimum variance quantization. This method involves cutting 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 you specify, the output colormap is also smaller.

# rgb2ind

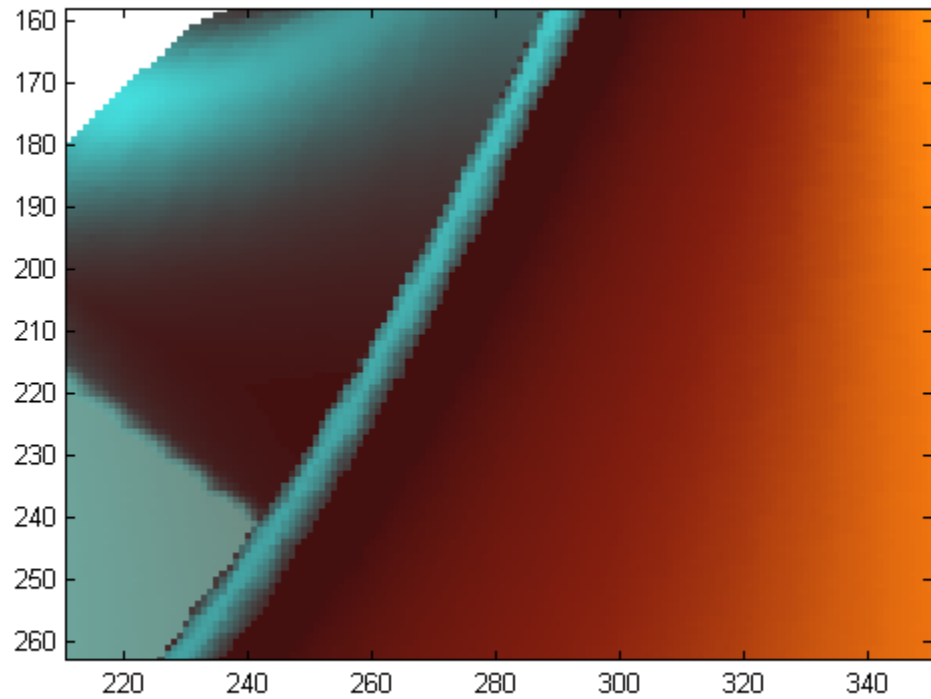
---

If you specify `map`, `rgb2ind` uses colormap mapping, which involves finding the colors in `map` that best match the colors in the RGB image.

## Examples

Read and display a truecolor uint8 PNG image of the MathWorks logo. Zoom in to see color variation better.

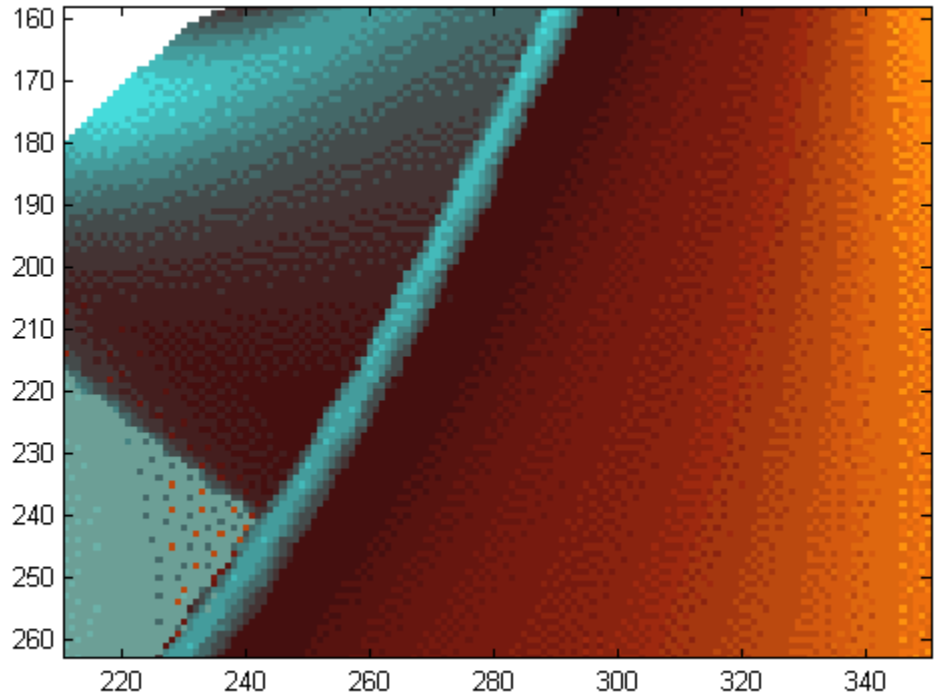
```
imfile = fullfile(matlabroot,...  
                  'toolbox','matlab','demos','html','logodemo_01.png');  
RGB = imread(imfile,'PNG');  
figure('Name','RGB Truecolor Image')  
imagesc(RGB)  
axis image  
zoom(4)
```



```
% 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)
```

# rgb2ind

---

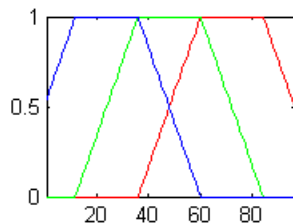


## See Also

[cunique](#) | [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.

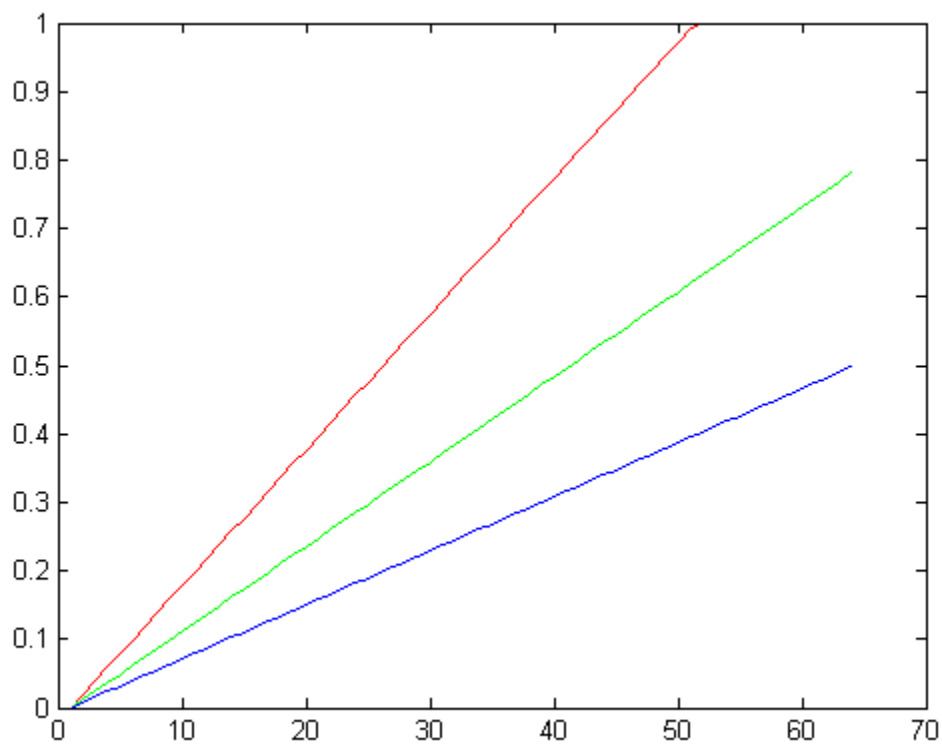
## Examples

Plot the RGB values of the copper colormap.

```
rgbplot(copper)
```

# rgbplot

---



## 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

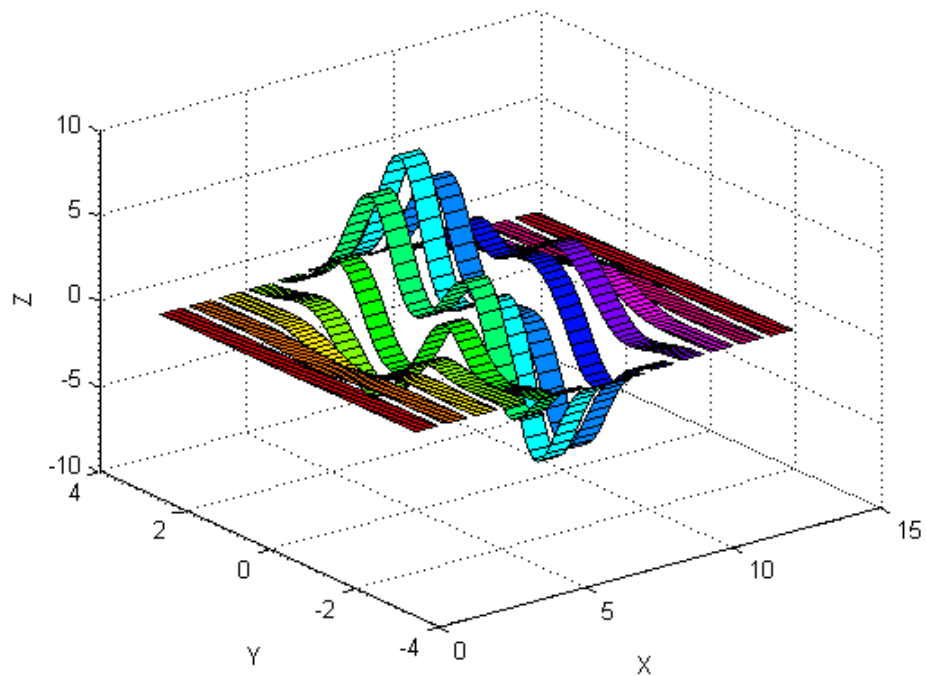
Create a ribbon plot of the peaks function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);
```

# ribbon

---

```
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, 'fieldname')`  
`s = rmfield(s, fields)`

**Description** `s = rmfield(s, 'fieldname')` removes the specified field from the structure array `s`.

`s = rmfield(s, fields)` removes more than one field at a time. `fields` is a character array of field names or cell array of strings.

**See Also** `fieldnames` | `setfield` | `getfield` | `isfield` | `orderfields`

**How To**

- dynamic field names

# rmpath

---

|                     |                                                                                                                                                                                                                                                                                                           |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | Remove folders from search path                                                                                                                                                                                                                                                                           |
| <b>Alternatives</b> | As an alternative to the <code>rmpath</code> function, use the Set Path dialog box.                                                                                                                                                                                                                       |
| <b>Syntax</b>       | <code>rmpath('folderName')</code><br><code>rmpath folderName</code>                                                                                                                                                                                                                                       |
| <b>Description</b>  | <code>rmpath('folderName')</code> removes the specified folder from the search path. Use the full path for <code>folderName</code> .<br><code>rmpath folderName</code> is the command form of the syntax.                                                                                                 |
| <b>Examples</b>     | Remove <code>/usr/local/matlab/mytools</code> from the search path:<br><br><code>rmpath /usr/local/matlab/mytools</code>                                                                                                                                                                                  |
| <b>See Also</b>     | <code>addpath</code>   <code>cd</code>   <code>dir</code>   <code>genpath</code>   <code>matlabroot</code>   <code>path</code>   <code>pathsep</code>   <code>pathtool</code>   <code>rehash</code>   <code>restoredefaultpath</code>   <code>savepath</code>   <code>userpath</code>   <code>what</code> |
| <b>How To</b>       | <ul style="list-style-type: none"><li>• “Using the MATLAB Search Path”</li></ul>                                                                                                                                                                                                                          |

**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)
sprex = 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**

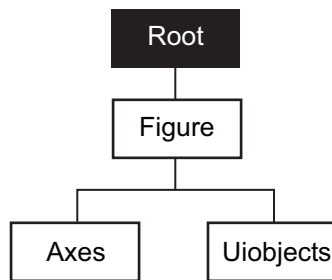
`rand` | `randi` | `randn` | `RandStream` | `now`

**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](#)

# 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)
```

# Root Properties

---

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

# Root Properties

---

- `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:

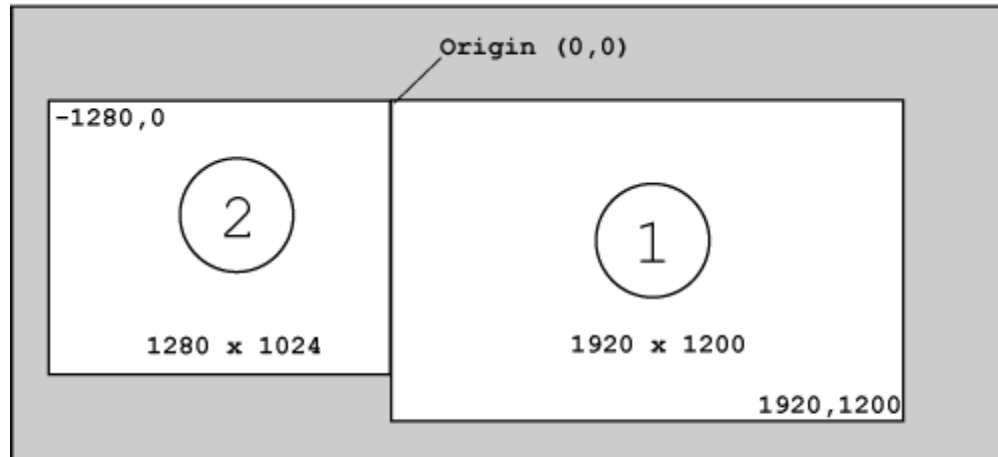
```
[xmin2,ymin2,xmax2,ymax2;  
 xmin1,ymin1,xmax1,ymax1]
```

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.

# Root Properties

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  $n^{\text{th}}$  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.

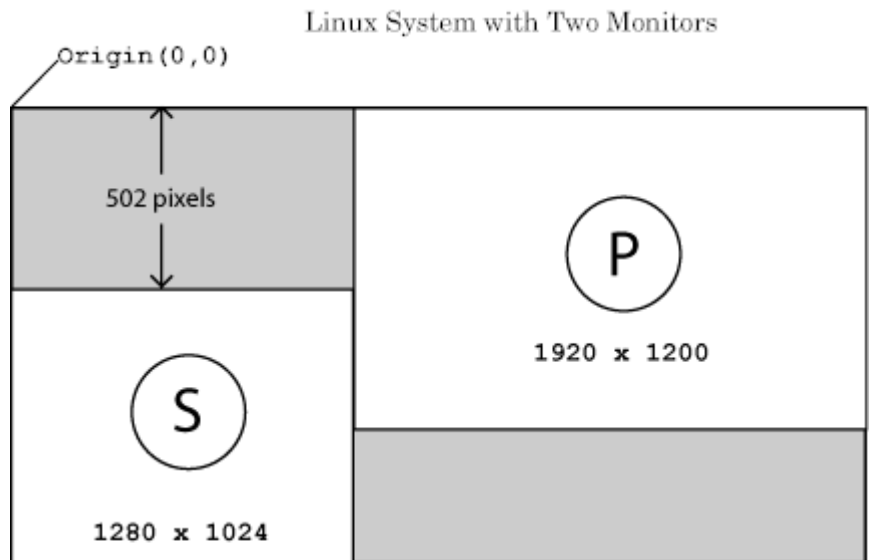
## Linux Systems

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.



```
get(0, 'MonitorPosition')
```

```
1281  0  1920  1200  
  1  503  1280  1024
```

# Root Properties

---

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-menuHeight]

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.

# Root Properties

---

`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

# Root Properties

---

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  $c_1s^n + \dots + c_n s + c_{n+1}$ .

**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)
```

## roots

---

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`).

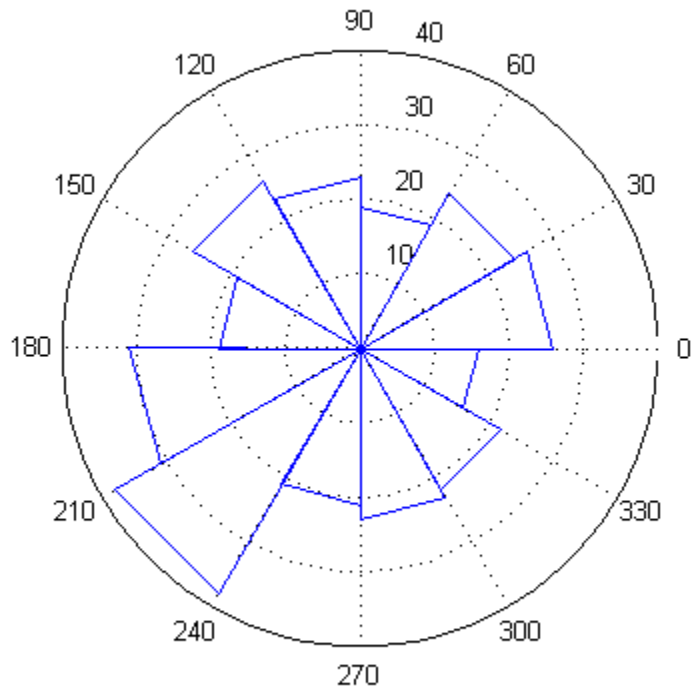
`h = rose(...)` returns the handles of the line objects used to create the graph.

`[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

# rosser

---

**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
```

**Purpose** Rotate matrix 90 degrees

**Syntax**  $B = \text{rot90}(A)$   
 $B = \text{rot90}(A, k)$

**Description**  $B = \text{rot90}(A)$  rotates matrix  $A$  counterclockwise by 90 degrees.  
 $B = \text{rot90}(A, k)$  rotates matrix  $A$  counterclockwise by  $k \cdot 90$  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](#)

# rotate

---

**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.

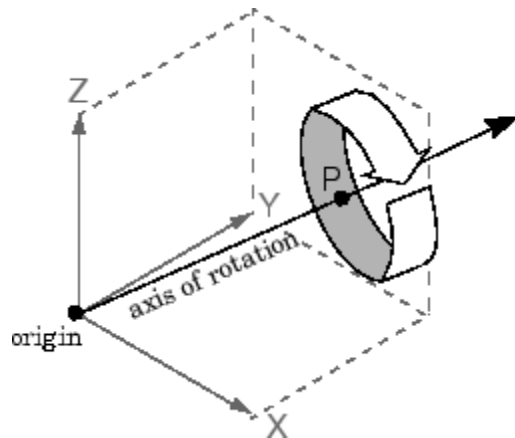
Positive `alpha` is defined as the righthand-rule angle about the direction vector as it extends from the origin of rotation.

If `h` is an array of handles, all objects must be children of the same axes.

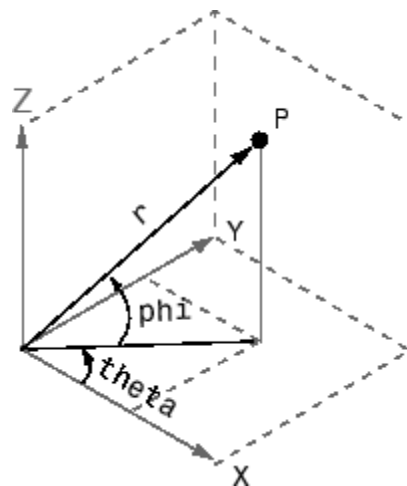
`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector `[x0,y0,z0]`.

**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 `[xp,yp,zp]`.



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.

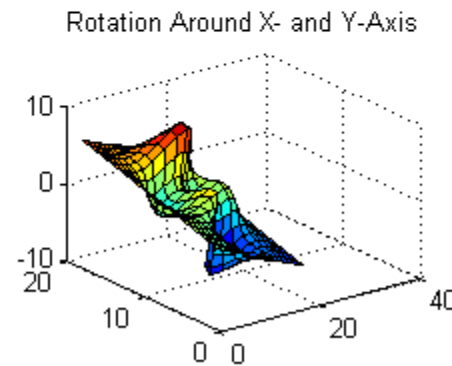
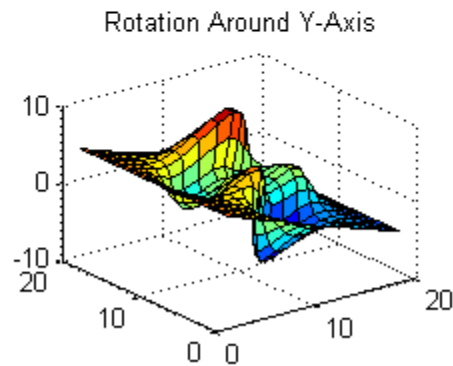
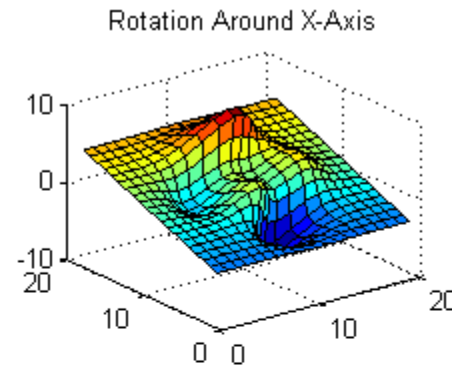
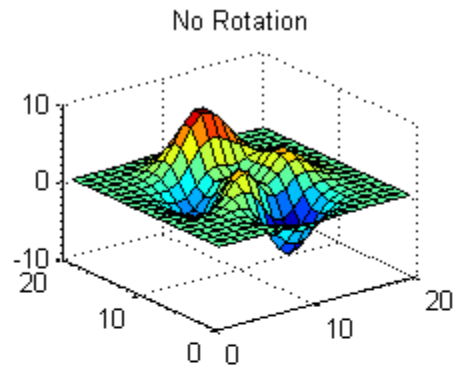
# rotate

---

## 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`

# rotate3d

---

**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

# rotate3d

---

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](#)

# round

---

**Purpose** Round to nearest integer

**Syntax**  $Y = \text{round}(X)$

**Description**  $Y = \text{round}(X)$  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  $(\max(\text{size}(A)) * \text{eps} * \text{norm}(A, \text{inf}))$  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
```

# rref

---

0 0 0 0

## See Also

[inv](#) | [lu](#) | [rank](#)



**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  $U' * U = \text{eye}(\text{size}(A))$ . 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

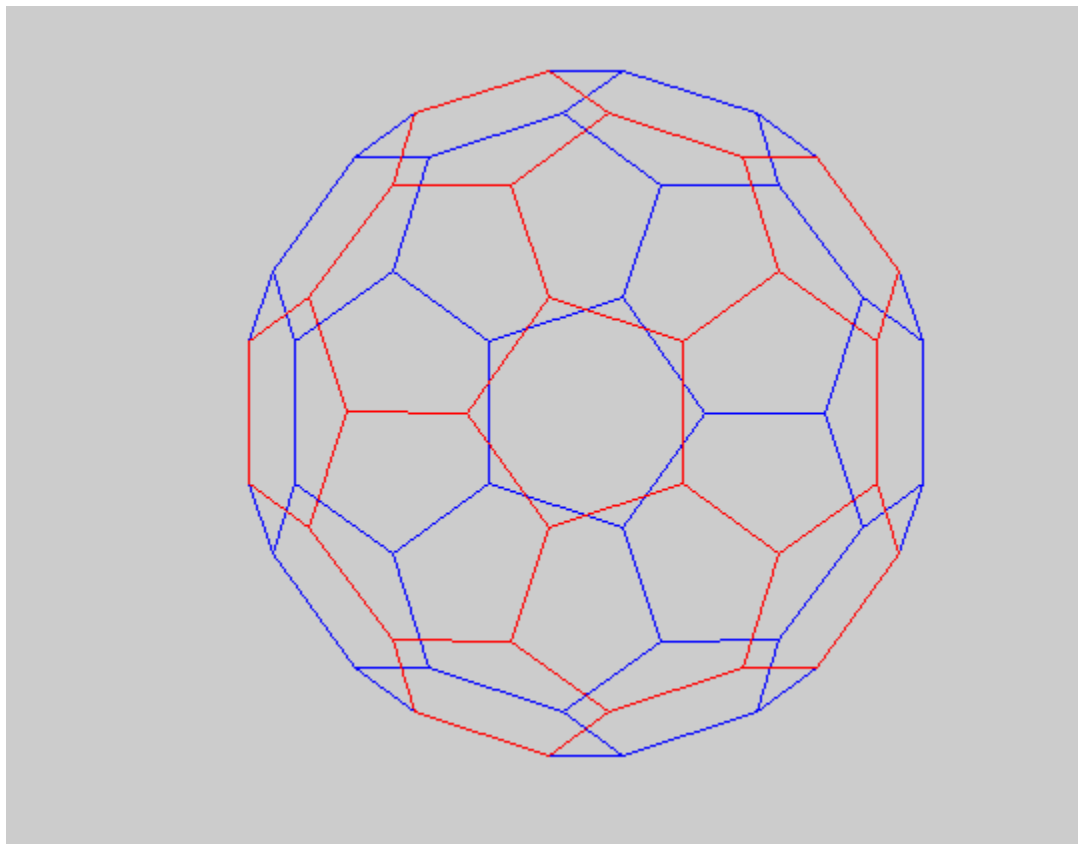
#### Run Script Not on Current Path

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

- “Making Files and Folders Accessible to MATLAB”

## 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):

|                                           |                                      |
|-------------------------------------------|--------------------------------------|
| <code>'-mat'</code>                       | Binary MAT-file format (default).    |
| <code>'-ascii'</code>                     | 8-digit ASCII format.                |
| <code>'-ascii', '-tabs'</code>            | Tab-delimited 8-digit ASCII format.  |
| <code>'-ascii', '-double'</code>          | 16-digit ASCII format.               |
| <code>'-ascii', '-double', '-tabs'</code> | 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 <code>double</code> , <code>character</code> , and <code>sparse</code> 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:

## save

---

```
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

- “Ways to Save Workspace Variables”

- “Write to Delimited Data Files”
- “Supported File Formats”

# save (COM)

---

**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)

# save (serial)

---

**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                               |
| 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)   |
| pbm        | Portable bitmap                                     |



| Text Value | Format                    |
|------------|---------------------------|
| 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_pre` using the MATLAB `fig` format. This allows you to open the file `pred_pre.fig` at a later time and continue editing it with the Plot Editor.

```
saveas(gcf, 'pred_pre.fig')
```

#### Example 2: Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file `logo`. Use the `ai` extension from the above table to specify the format. The file created is `logo.ai`.

```
saveas(gcf, 'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the `print devices` table, which is `-dill`; use `doc print` or `help print` to see the table for print device types. The file created is `logo.ai`. MATLAB automatically appends the `ai` extension for an Illustrator format file because no extension was specified.

```
saveas(gcf, 'logo', 'ill')
```

### 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](#)

# saveobj

---

**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 subj = saveobj(obj)
            % Call superclass saveobj method
            subj = 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
end
```

```
    end  
    b = a;  
end
```

See “Maintaining Class Compatibility”.

**See Also**

save | load | loadobj

**Tutorials**

- “Control Save and Load”

# savepath

---

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | Save current search path                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Alternatives</b> | As an alternative to the <code>savepath</code> function, use the Set Path dialog box.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>       | <pre>savepath savepath folderName/pathdef.m status = savepath...</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b>  | <p><code>savepath</code> updates the MATLAB search path for all users on the system so that the path can be reused in a future session. <code>savepath</code> saves the search path to the <code>pathdef.m</code> file that MATLAB located at startup, or to the current folder if a <code>pathdef.m</code> file exists there. To save the search path programmatically each time you exit MATLAB, use <code>savepath</code> in a <code>finish.m</code> 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.</p> <p><code>savepath folderName/pathdef.m</code> saves the current search path to <code>pathdef.m</code> located in <code>folderName</code>. Use this form of the syntax if you do not have write access to the current <code>pathdef.m</code>. If you do not specify <code>folderName</code>, MATLAB saves <code>pathdef.m</code> in the current folder. <code>folderName</code> can be a relative or absolute path. To use the saved search path automatically in a future session, make <code>folderName</code> be the startup folder for MATLAB.</p> <p><code>status = savepath...</code> returns 0 when <code>savepath</code> was successful and 1 when <code>savepath</code> failed.</p> |
| <b>Examples</b>     | <p>Save the current search path to <code>pathdef.m</code>, located in <code>I:/my_matlab_files</code>:</p> <pre>savepath I:/my_matlab_files/pathdef.m</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>See Also</b>     | <pre>addpath   cd   dir   finish   genpath   matlabroot   pathsep   pathtool   rehash   restoredefaultpath   rmpath   startup   userpath   what</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

**How To**

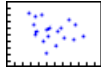
- “Running a Script When Exiting the MATLAB Program”
- “Using the MATLAB Search Path”

# scatter

---

## Purpose

Scatter plot



## Syntax

```
scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
scatter(...,'PropertyName',propertyvalue)
scatter(axes_handle,...)
h = scatter(...)
```

## Description

`scatter(X,Y,S,C)` displays colored circles at the locations specified by the vectors `X` and `Y` (which must be the same size).

`S` determines the area of each marker (specified in  $\text{points}^2$ ). `S` can be a vector the same length as `X` and `Y` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size. If `S` is empty, the default size is used.

`C` determines the color of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, `C` should be a 3-by-1 matrix. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter(X,Y)` draws the markers in the default size and color.

`scatter(X,Y,S)` draws the markers at the specified sizes (`S`) with a single color. This type of graph is also known as a bubble plot.

`scatter(...,markertype)` uses the marker type specified instead of 'o' (see `LineStyleSpec` for a list of marker specifiers).

`scatter(...,'filled')` fills the markers.



`scatter(..., 'PropertyName', propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

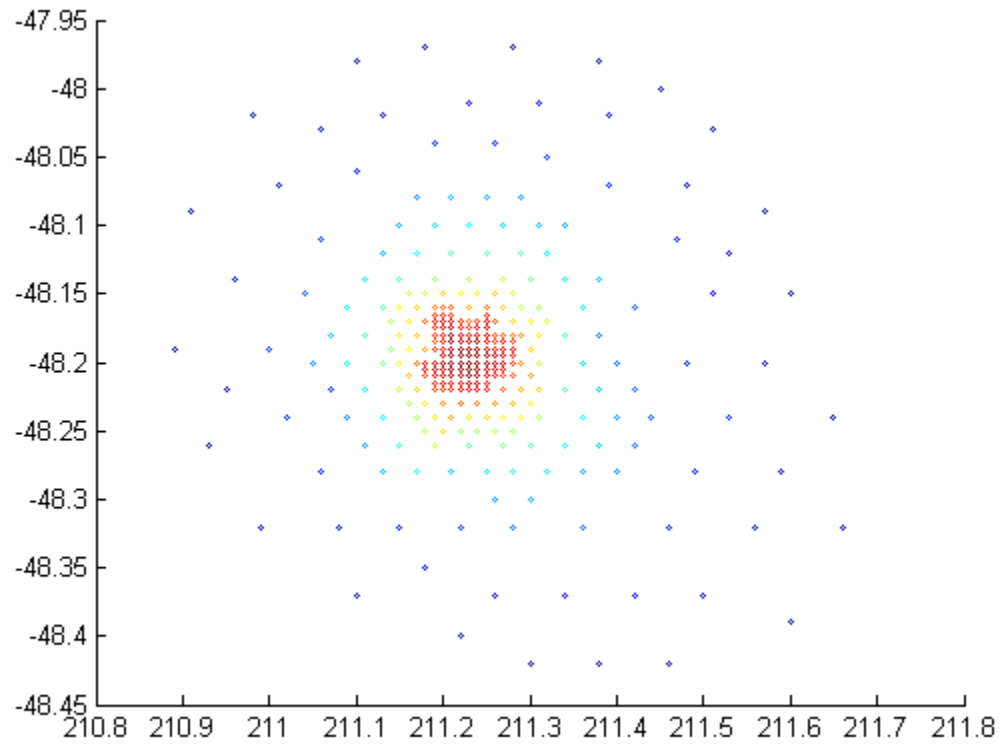
`scatter(axes_handle, ...)` plots into the axes object with handle `axes_handle` instead of the current axes object (`gca`).

`h = scatter(...)` returns the handle of the `scattergroup` object created.

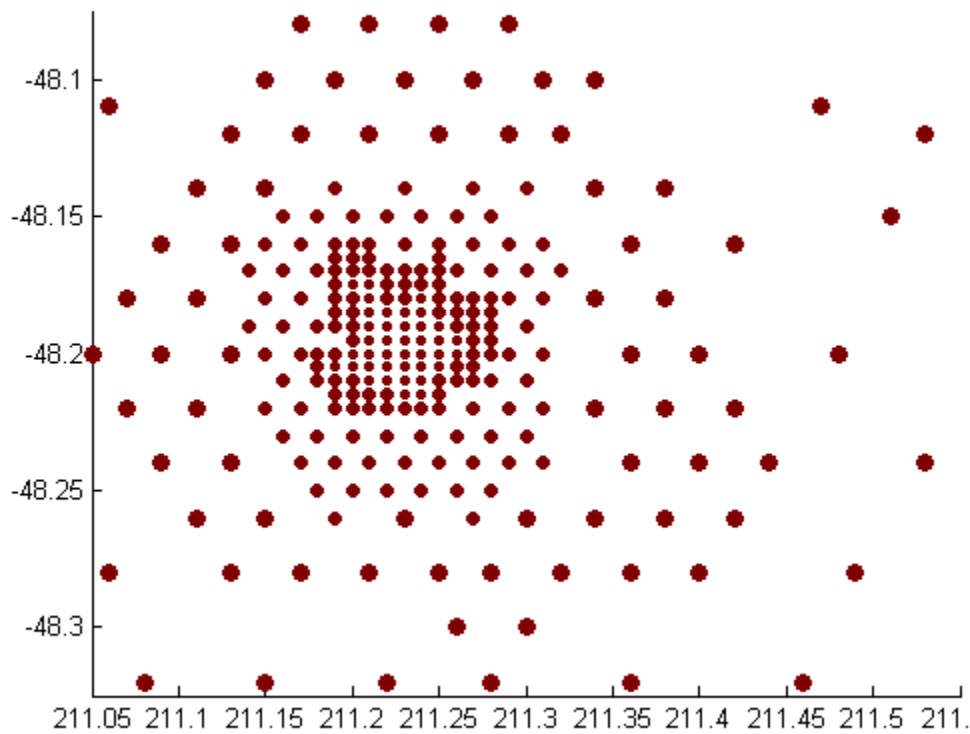
**Examples**

```
load seamount
scatter(x,y,5,z)
```

# scatter



```
% Vary the size of circles by inverse depth and fill them with dark red
figure
scatter(x,y,sqrt(-z/2),[.5 0 0],'filled')
zoom(2)
```

**See Also**

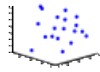
[scatter3](#) | [plot3](#) | [scattergroup](#) | [ColorSpec](#) | [LineStyle](#)

# scatter3

---

## Purpose

3-D scatter plot



## Syntax

```
scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
scatter3(...,'PropertyName',propertyvalue)
h = scatter3(...)
```

## Description

`scatter3(X,Y,Z,S,C)` displays colored circles at the locations specified by the vectors `X`, `Y`, and `Z` (which must all be the same size).

`S` determines the area of each marker (specified in points<sup>2</sup>). `S` can be a vector the same length as `X`, `Y`, and `Z` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size.

`C` determines the color of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, `C` should be a 3-by-1 matrix. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter3(X,Y,Z)` draws the markers in the default size and color.

`scatter3(X,Y,Z,S)` draws markers at the specified sizes (`S`) in a single color.

`scatter3(...,markertype)` uses the marker type specified instead of 'o' (see `LineStyle` for a list of marker specifiers).

`scatter3(...,'filled')` fills the markers.

`scatter3(..., 'PropertyName', propertyvalue)` creates the scatter graph, applying the specified property settings. See `scattergroup` properties for a description of properties.

`h = scatter3(...)` returns handles to the `scattergroup` objects created by `scatter3`. See `Scattergroup Properties` for property descriptions.

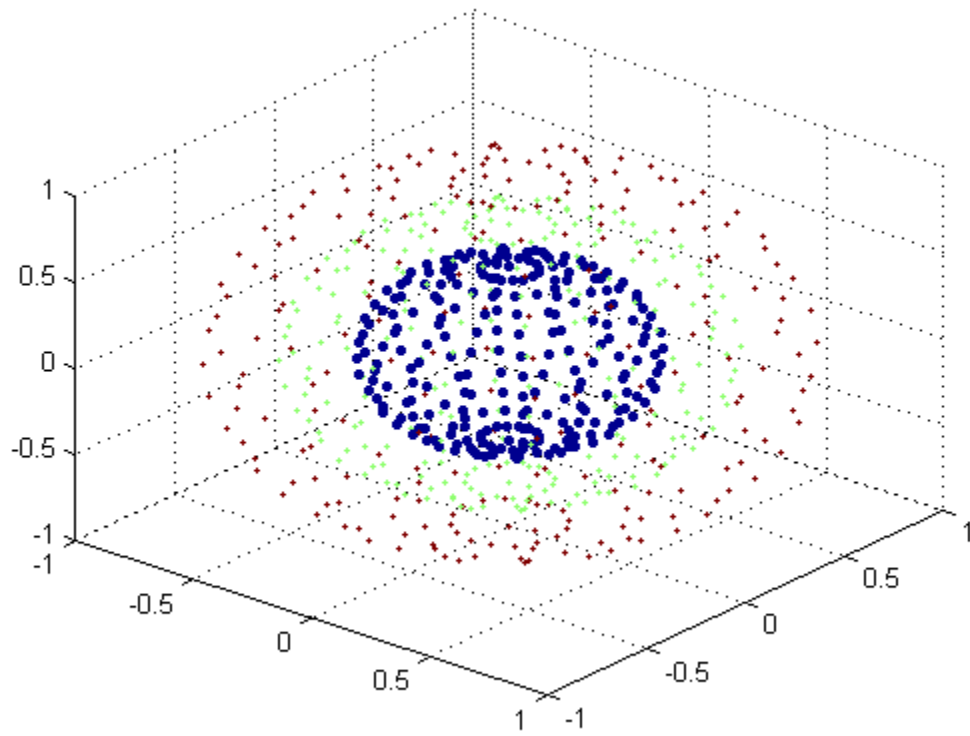
Use `plot3` for single color, single marker size 3-D scatter plots.

## Examples

```
figure
[x,y,z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,numel(x),1);
C = repmat([1 2 3],numel(x),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled'), view(-60,60)
view(40,35)
```

## scatter3

---



### See Also

[scatter](#) | [plot3](#) | [ColorSpec](#) | [LineStyle](#) | [scattergroup](#)

**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        |

# Scattergroup Properties

---

## 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.

# Scattergroup Properties

---

## 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.

# Scattergroup Properties

---

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.

# Scattergroup Properties

---

- `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

# Scattergroup Properties

---

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.

# Scattergroup Properties

---

---

**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 =  $\frac{1}{72}$  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.

# Scattergroup Properties

---

- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none` (the factory default for axes objects), sets the fill color to the figure `Color`.

## Parent

handle of parent axes, `hgggroup`, or `hgtransform`

*Parent of object.* Handle of the object's parent. The parent is normally the axes, `hgggroup`, 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.

# Scattergroup Properties

---

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 `'hggroupp'`. The following statement finds all the `hggroupp` objects in the current axes.

```
t = findobj(gca, 'Type', 'hggroupp');
```

## 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`.

# Scattergroup Properties

---

---

**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.

---

# schur

---

**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' | <code>T</code> is triangular and is complex if <code>A</code> has complex eigenvalues.                                                       |
| 'real'    | <code>T</code> has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. 'real' is the default. |

If `A` is complex, `schur` returns the complex Schur form in matrix `T`. 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`

# script

---

**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`

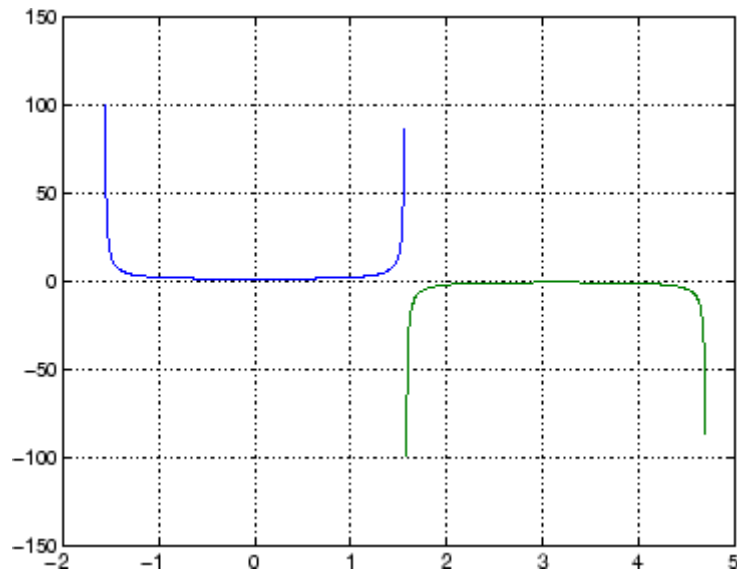
**Purpose** Secant of argument in radians

**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  $\pi$ .

**sec**

---

**See Also**

secd | sech | asec | asecd | asech

|                         |                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Secant of argument in degrees                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>           | $Y = \text{secd}(X)$                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>      | $Y = \text{secd}(X)$ is the secant of the elements of $X$ . The result, $Y$ , is expressed in degrees.                                                                                                                                                                                                                                                        |
| <b>Input Arguments</b>  | <b>X - Angle in degrees</b><br>scalar value   vector   matrix   N-D array<br><br>Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>secd</code> operation is element-wise when $X$ is nonscalar.<br><b>Example:</b><br><b>Data Types</b><br>single   double<br><b>Complex Number Support:</b> Yes |
| <b>Output Arguments</b> | <b>Y - Secant of angle</b><br>scalar value   vector   matrix   N-D array<br><br>Secant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as $X$ .                                                                                                                                                   |
| <b>Examples</b>         | <b>Secant of 90 degrees compared to secant of <math>\pi/2</math> radians</b><br>secd(90) is infinite, whereas <code>sec(pi/2)</code> is large but finite.<br><br>secd(90)<br><br>ans =<br><br>Inf<br><br>sec(pi/2)<br><br>ans =                                                                                                                               |

# secd

---

1.6331e+16

## Secant of vector of complex angles, specified in degrees

$z = [35+i \ 15+2i \ 10+3i];$

$y = \text{secd}(z)$

$y =$

1.2204 + 0.0149i    1.0346 + 0.0097i    1.0140 + 0.0094i

## See Also

[sec](#) | [asec](#) | [asecd](#)



**Purpose** Hyperbolic secant

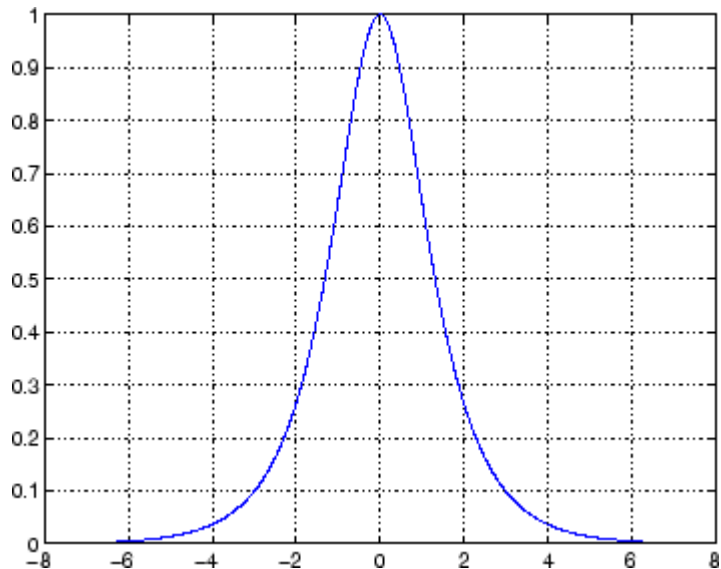
**Syntax**  $Y = \operatorname{sech}(X)$

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

$Y = \operatorname{sech}(X)$  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](#)

# selectmoveresize

---

**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.

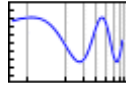
`A = selectmoveresize` returns a structure array containing

- `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`

**Purpose** Semilogarithmic plot



**Syntax**

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineStyle,...)
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 *X<sub>n</sub>* versus *Y<sub>n</sub>* pairs. If only *X<sub>n</sub>* or *Y<sub>n</sub>* 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,LineStyle,...)` plots all lines defined by the *X<sub>n</sub>*, *Y<sub>n</sub>*, *LineStyle* triples. *LineStyle* determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries` graphics objects created by `semilogx`.

`h = semilogx(...)` return a vector of handles to `lineseries` graphics objects, one handle per line.

# semilogx

---

## 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  $X_n, Y_n$  pairs with  $X_n, Y_n, \textit{LineSpec}$  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.

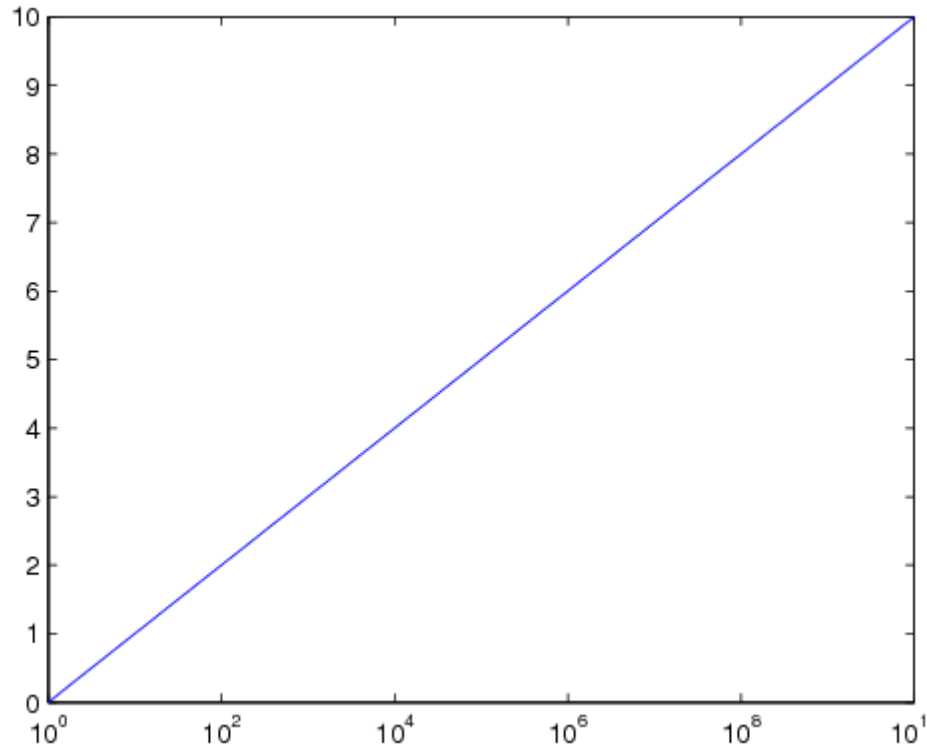
## Renderer Support

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**

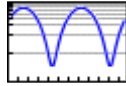
`line` | `LineStyle` | `loglog` | `plot` | `semilogy`

# semilogy

---

## Purpose

Semilogarithmic plot



## Syntax

```
semilogy(Y)
semilogy(X1,Y1,...)
semilogy(X1,Y1,LineStyle,...)
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,LineStyle,...) plots all lines defined by the Xn,Yn,LineStyle triples. LineSpec determines line style, marker symbol, and color of the plotted lines.

semilogy(...,'PropertyName',PropertyValue,...) sets property values for all lineseries 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  $X_n, Y_n$  pairs with  $X_n, Y_n, LineSpec$  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.

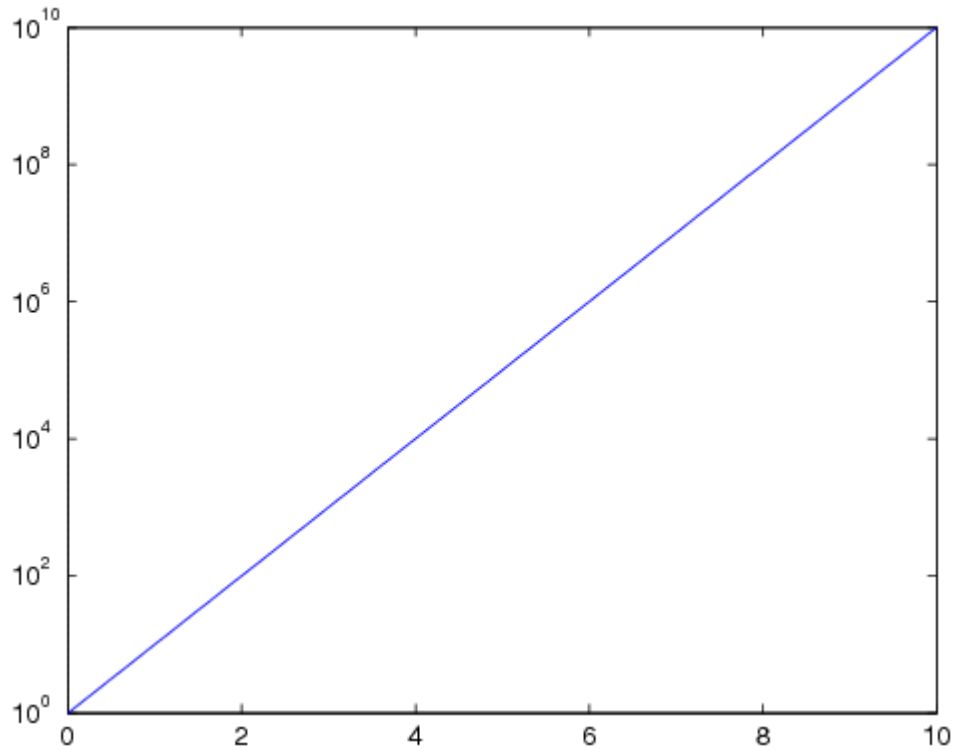
**Examples**

Create a simple `semilogy` plot.

```
x = 0:.1:10;  
semilogy(x, 10.^x)
```

# semilogy

---



## See Also

[line](#) | [LineSpec](#) | [loglog](#) | [plot](#) | [semilogx](#)



**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

# sendmail

---

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”

# serial

---

**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           | <code>serial('/dev/ttyS0');</code>          |
| Mac OS X 64                  | <code>serial('/dev/tty.KeySerial1');</code> |
| Windows 32 and<br>Windows 64 | <code>serial('com1');</code>                |

`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](#)

# serialbreak

---

**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 units — 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`

**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]};
```

# audioplayer.set

---

```
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](#)

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Set property values for audiorecorder object                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>      | <pre>set(obj,Name,Value) set(obj,cellOfNames,cellOfValues) set(obj,structOfProperties) settableProperties = set(obj)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b> | <p><code>set(obj,Name,Value)</code> sets the named property to the specified value for the object <code>obj</code>.</p> <p><code>set(obj,cellOfNames,cellOfValues)</code> sets the properties listed in the cell array <code>cellOfNames</code> to the corresponding values in the cell array <code>cellOfValues</code>. Each cell array must contain the same number of elements.</p> <p><code>set(obj,structOfProperties)</code> sets the properties identified by each field of the structure array <code>structOfProperties</code> to the values of the associated fields.</p> <p><code>settableProperties = set(obj)</code> returns the names of the properties that you can set in a structure array. The field names of <code>settableProperties</code> are the property names.</p> |
| <b>Tips</b>        | The <code>set</code> function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Examples</b>    | <p>View the list of properties that you can set for an audiorecorder object:</p> <pre>recorderObj = audiorecorder; set(recorderObj)</pre> <hr/> <p>Set the <code>Tag</code> and <code>UserData</code> properties of an audiorecorder object using a structure array:</p> <pre>newValues.Tag = 'My Tag'; newValues.UserData = {'My User Data', pi, [1 2 3 4]};</pre>                                                                                                                                                                                                                                                                                                                                                                                                                        |

# audiorecorder.set

---

```
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](#)

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Set object or interface property to specified value                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>      | <pre>h.set('pname',value) h.set('pname1',value1,'pname2',value2,...) set(h,...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b> | <p><code>h.set('pname',value)</code> sets the property specified in the string <code>pname</code> to the given value.</p> <p><code>h.set('pname1',value1,'pname2',value2,...)</code> sets each property specified in the <code>pname</code> strings to the given value.</p> <p><code>set(h,...)</code> is an alternate syntax for the same operation.</p> <p>For information on how MATLAB converts workspace matrices to COM data types, see “Handling COM Data in MATLAB Software”.</p>                           |
| <b>Tips</b>        | COM functions are available on Microsoft Windows systems only.                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Examples</b>    | <p>Create an <code>mwsamp</code> control and use <code>set</code> to change the <code>Label</code> and <code>Radius</code> properties:</p> <pre>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');</pre> <p>Here is another way to do the same thing, only without <code>set</code> and <code>invoke</code>:</p> <pre>h.Label = 'Click to fire event'; h.Radius = 40; h.Redraw;</pre> |
| <b>See Also</b>    | <code>get (COM)</code>   <code>inspect</code>   <code>isprop</code>   <code>addproperty</code>   <code>deleteproperty</code>                                                                                                                                                                                                                                                                                                                                                                                        |

# set (hgsetget)

---

**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”



**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]};
```

# VideoReader.set

---

```
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

**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

# set (serial)

---

**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`

# set (timer)

---

**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 (timer)

---

```
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)



**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)`

**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`

# RandStream.setDefaultStream

---

**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.

**Purpose**

Find set difference of two arrays

**Compatibility**

In a future release, the behavior of `setdiff` will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assessing the Impact of the Forthcoming Changes to `setdiff`” on page 2-4090. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.

**Syntax**

```
C = setdiff(A,B)
C = setdiff(A,B,'rows')
[C,ia] = setdiff(A,B)
[C,ia] = setdiff(A,B,'rows')

[C,ia] = setdiff(__,'R2012a')

[C,ia] = setdiff(__,'legacy')

[C,ia] = setdiff(A,B,setOrder)

[C,ia] = setdiff(A,B,'rows',setOrder)
```

**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')` returns the rows from A that are not in B. The rows of matrix C are in sorted order. The rows of the matrix C are in sorted order.

`[C,ia] = setdiff(A,B)` also returns the index vector `ia`, such that `C = A(ia)`.

`[C,ia] = setdiff(A,B,'rows')` also returns index vectors `ia`, such that `C = A(ia,:)`.

# setdiff

---

`[C,ia] = setdiff(___, 'R2012a')` adopts the future behavior of the `setdiff` function. `[C,ia] = setdiff(___, 'legacy')` preserves the current behavior. You can specify either flag, 'R2012a' or 'legacy', as the final input argument in any previous syntax.

`[C,ia] = setdiff(A,B,setOrder)` and `[C,ia] = setdiff(A,B,'rows',setOrder)` returns `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`.

## Input Arguments

### A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays, specified as `logical`, `char`, a cell array of strings, or any numeric class. `A` and `B` also can be any object with the class methods: `sort` (or `sortrows` for the 'rows' option), `eq`, and `ne`. This includes 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`

**Complex Number Support:** Yes

### 'rows' - Rows flag

Rows flag, specified as 'rows', identifies the rows of `A` and `B` as individual elements. When you specify this flag, `setdiff` returns the rows from `A` that are not in `B`. The arrays, `A` and `B`, must have the same

number of columns when you use the rows flag. This option does not support cell arrays.

### setOrder - Order flag

'sorted' | '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, are returned in sorted order. For example:<br>C = setdiff([4 1 3 2],[2 1],'sorted') returns<br>C = [3 4].           |
| 'stable'   | The values (or rows) in C, are returned in the same order as in A. For example:<br>C = setdiff([4 1 3 2],[2 1],'stable') returns<br>C = [4 3]. |

### 'legacy' - Legacy behavior flag

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

### 'R2012a' - Future behavior flag

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior has on your existing code.

## Output Arguments

### C - Difference of A and B

Vector | Matrix

Difference of A and B, returned as a vector or matrix. C is a vector, unless you specify the 'rows' flag. If 'rows' is specified, C is a matrix containing the rows of A that are not in B.

# setdiff

---

## **ia - Index to A**

Vector

Index to A, returned as a vector, 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 last occurrence of the value (or row).

If you call `setdiff` with the `setOrder` or `'R2012a'` argument, `ia` contains the index to the first occurrence of any repeated value (or row) in A.

## **Examples**

### **Difference of Two Vectors**

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];  
C = setdiff(A,B)
```

```
C =
```

```
     1     3     5
```

### **Find the Difference of Two Vectors and the Indices to the Different Values**

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];  
[C,ia] = setdiff(A,B)
```

```
C =
```

```
     1     3     5
```

```
ia =
```

```
     7     1     5
```

### **Difference of the Rows in Two Matrices**

```
A = [7 9 7; 0 0 0; 7 9 7; 5 5 5; 1 4 5];  
B = [0 0 0; 5 5 5];  
[C,ia] = setdiff(A,B,'rows')
```



```
C =
     1     4     5
     7     9     7
```

```
ia =
     5
     3
```

### 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
```

# setdiff

---

```
ia =  
  
    4  
    1  
    5
```

`setdiff` adopts the forthcoming behavior when you specify the `setOrder` argument. If you omit the `setOrder` argument, the output is sorted, but the size and content of `C` and `ia` conforms to the current behavior.

## Assessing the Impact of the Forthcoming Changes to `setdiff`

Use the 'R2012a' flag to assess the impact of the forthcoming behavior changes. Use the 'legacy' flag to preserve the current behavior of your existing code.

Find the difference of `A` and `B` with the current default behavior.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];  
[C1,ia1] = setdiff(A,B)
```

```
C1 =  
  
    1    3    5
```

```
ia1 =  
  
    7    1    5
```

Find the difference of `A` and `B`, and adopt the forthcoming behavior. In the future, this behavior will be the default.

```
[C2,ia2] = setdiff(A,B, 'R2012a')
```

```
C2 =  
  
    1    3    5
```

```
ia2 =
     4
     1
     5
```

Notice that `ia1` and `ia2` have different shapes and content.

Find the difference of `A` and `B`, and preserve the current behavior.

```
[C3,ia3] = setdiff(A,B,'legacy')
```

```
C3 =
     1     3     5
ia3 =
     7     1     5
```

`C3` and `ia3` match `C1` and `ia1`.

### Find the Difference of Vectors Containing NaNs

```
A = [5 NaN NaN]; B = [5 NaN];
C = setdiff(A,B)
```

```
C =
     NaN     NaN
```

`setdiff` treats NaN values as distinct.

### See Also

`unique` | `intersect` | `ismember` | `issorted` | `union` | `setxor`  
| `sort`

# Tiff.setDirectory

---

**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”

|                    |                                                                                                                                                                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Override to change command window display                                                                                                                                                                                                                                                |
| <b>Syntax</b>      | setdisp(H)                                                                                                                                                                                                                                                                               |
| <b>Description</b> | setdisp(H) called by <code>set</code> when <code>set</code> is called with no output arguments and a single input argument that is a handle array. Override this <code>hgsetget</code> class method in a subclass to change how property information is displayed in the command window. |
| <b>See Also</b>    | <code>hgsetget</code>   <code>set</code> ( <code>hgsetget</code> )                                                                                                                                                                                                                       |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Implementing a Set/Get Interface for Properties”</li></ul>                                                                                                                                                                                      |

# setenv

---

**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` | `!`

**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];
```

# setfield

---

```
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”



**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].

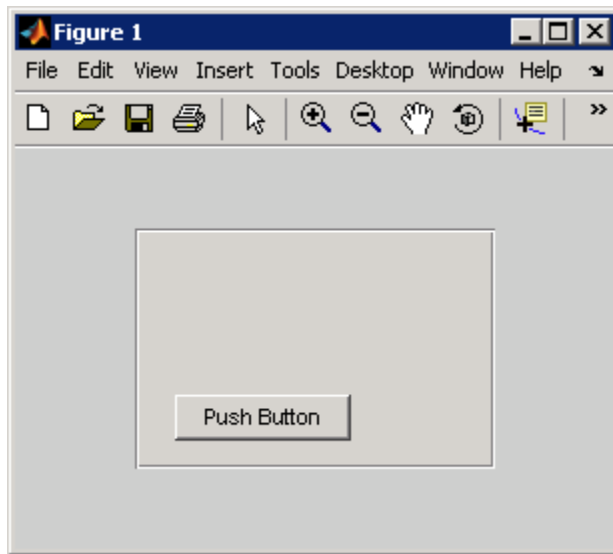
`setpixelposition(handle,position,recursive)` sets the position as above. If Boolean `recursive` is true, the position is set relative to the parent figure of `handle`.

**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]);
```

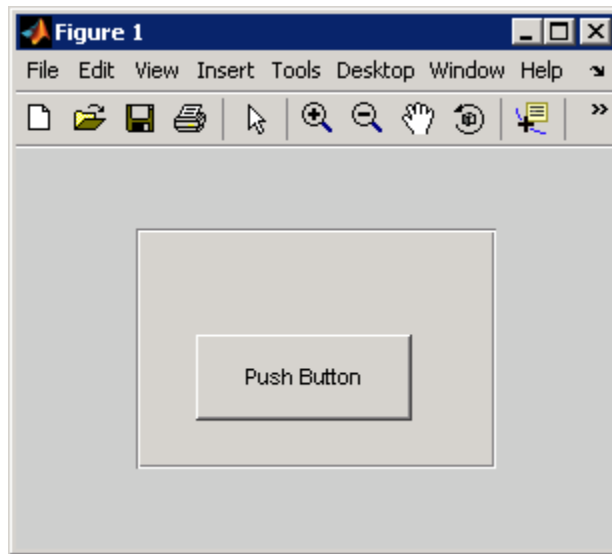
# setpixelposition

---



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](#)

# setpref

---

## 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](#)

# setstr

---

**Purpose** Set string flag

---

**Note** setstr is not recommended. Use char instead.

---

**Description** This MATLAB 4 function has been renamed char in MATLAB 5.

**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 directory.
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”

# Tiff.setTag

---

**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”

# settimeseriesnames

---

**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`

**Purpose**

Find set exclusive OR of two arrays

**Compatibility**

In a future release, the behavior of `setxor` will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assessing the Impact of the Forthcoming Changes to `setxor`” on page 2-4113. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.

**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(__,'R2012a')

[C,ia,ib] = setxor(__,'legacy')

[C,ia,ib] = setxor(A,B,setOrder)

[C,ia,ib] = setxor(A,B,'rows',setOrder)
```

**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')` returns the rows of matrices `A` and `B` that are not in their intersection. The rows of the matrix `C` are in sorted order.

`[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 elements `A(ia,:)` and `B(ib,:)`.

`[C,ia,ib] = setxor(___, 'R2012a')` adopts the future behavior of the `setxor` function. `[C,ia,ib] = setxor(___, 'legacy')` preserves the current behavior. You can specify either flag, `'R2012a'` or `'legacy'`, as the final input argument in any previous syntax.

`[C,ia,ib] = setxor(A,B,setOrder)` and `[C,ia,ib] = setxor(A,B,'rows',setOrder)` returns `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` and `B`.

## Input Arguments

### **A,B - Input arrays**

vectors | matrices | N-D arrays

Input arrays of which to find the symmetric difference, specified as `logical`, `char`, a cell array of strings, or any numeric class. `A` and `B` also can be any object with the class methods: `sort` (or `sortrows` for the `'rows'` option), `eq`, and `ne`. This includes 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`

**Complex Number Support:** Yes

**'rows' - Rows flag**

Rows flag, specified as 'rows', identifies the rows of A and B as individual elements. When you specify this flag, setxor returns the rows of A and B that are not in their intersection. A and B must have the same number of columns. This option does not support cell arrays.

**setOrder - Order flag**

'sorted' | '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, are returned in sorted order. For example:<br>C = setxor([5 1 3],[4 1 2],'sorted') returns<br>C = [2 3 4 5].                 |
| 'stable'   | The values (or rows) in C, are returned in the same order as in A and B. For example:<br>C = setxor([5 1 3],[4 1 2],'stable') returns<br>C = [5 3 4 2]. |

**'legacy' - Legacy behavior flag**

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

**'R2012a' - Future behavior flag**

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior has on your existing code.

# setxor

---

## Output Arguments

### **C - Symmetric difference array**

vector | matrix

Symmetric difference array, returned as a vector or matrix. C is a vector, unless you specify the 'rows' flag. If 'rows' is specified, C is a matrix containing the rows of A and B that are not in their intersection.

### **ia - Index to A**

vector

Index to A, returned as a vector, 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 last occurrence of the value (or row).

If you call `setxor` with the `setOrder` or 'R2012a' argument, ia contains the index to the first occurrence of any repeated value (or row) in A.

### **ib - Index to B**

vector

Index to B, returned as a vector, 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 last occurrence of the value (or row).

If you call `setxor` with the `setOrder` or 'R2012a' argument, ib contains the index to the first occurrence of any repeated value (or row) in B.

## Examples

### **Symmetric Difference of Two Vectors**

```
A = [5 1 3 3 3]; B = [4 1 2];  
C = setxor(A,B)
```

```
C =
```

```
     2     3     4     5
```

### Find Symmetric Difference of Two Vectors and the Indices to the Different Values

```
A = [5 1 3 3 3]; B = [4 1 2];
[C,ia,ib] = setxor(A,B)
```

```
C =
     2     3     4     5
```

```
ia =
     5     1
```

```
ib =
     3     1
```

### Symmetric Difference of the Rows in Two Matrices

```
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];
[C,ia,ib] = setxor(A,B,'rows')
```

```
C =
     7     7     1
     7     7     2
     7     8     9
```

```
ia =
     3
     1
```

```
ib =
     3
```

## 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
```

setxor adopts the forthcoming behavior when you specify the setOrder argument. If you omit the setOrder argument, the output is sorted, but the size and content of C, ia, and ib conforms to the current behavior.

### Assessing the Impact of the Forthcoming Changes to setxor

Use the 'R2012a' flag to assess the impact of the forthcoming behavior changes. Use the 'legacy' flag to preserve the current behavior of your existing code.

Find the symmetric difference of A and B with the current default behavior.

```
A = [5 1 3 3 3]; B = [4 1 2 2];
[C1,ia1,ib1] = setxor(A,B)
```

```
C1 =
```

```
     2     3     4     5
```

```
ia1 =
```

```
     5     1
```

```
ib1 =
```

```
     4     1
```

Find the symmetric difference and opt into the forthcoming behavior. In the future, this behavior will be the default.

```
[C2,ia2,ib2] = setxor(A,B,'R2012a')
```

```
C2 =
```

## setxor

---

```
          2     3     4     5
ia2 =
          3
          1

ib2 =
          3
          1
```

Notice that `ia1` and `ia2` have different shapes and content. This also true for `ib1` and `ib2`.

Find the symmetric difference and preserve the current behavior.

```
[C3,ia3,ib3] = setxor(A,B,'legacy')
```

```
C3 =
          2     3     4     5

ia3 =
          5     1

ib3 =
          4     1
```

Notice that `C3`, `ia3`, and `ic3` match `C1`, `ia1`, and `ic1` respectively.

### **Find Symmetric Difference of Vectors Containing NaNs**

```
A = [5 NaN NaN]; B = [5 NaN NaN];
C = setxor(A,B)
```

C =

NaN NaN NaN NaN

The `setxor` function treats NaN values as distinct.

**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')
```

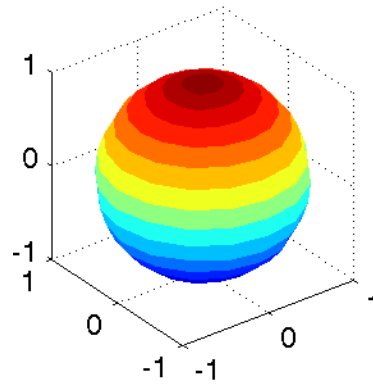
```
subplot(3,1,2)  
sphere(16)  
axis square  
shading faceted
```

```
title('Faceted Shading')  
  
subplot(3,1,3)  
sphere(16)  
axis square  
shading interp  
title('Interpolated Shading')
```

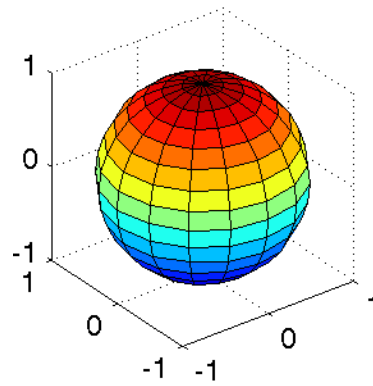
# shading

---

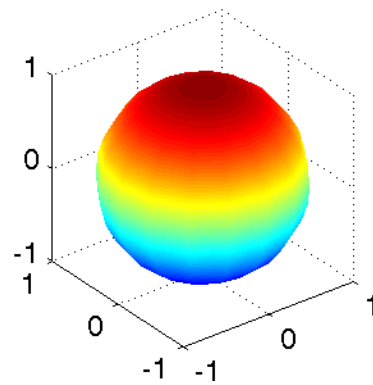
Flat Shading



Faceted Shading



Interpolated Shading



## 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.
```

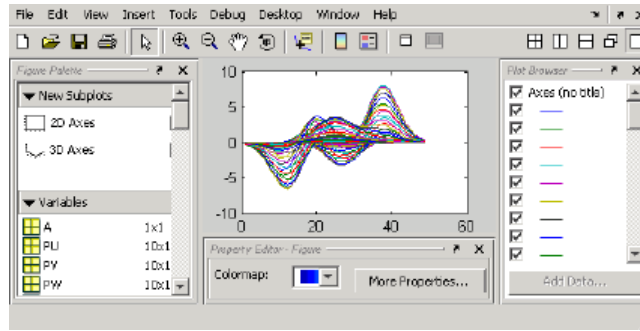
## See Also

`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.

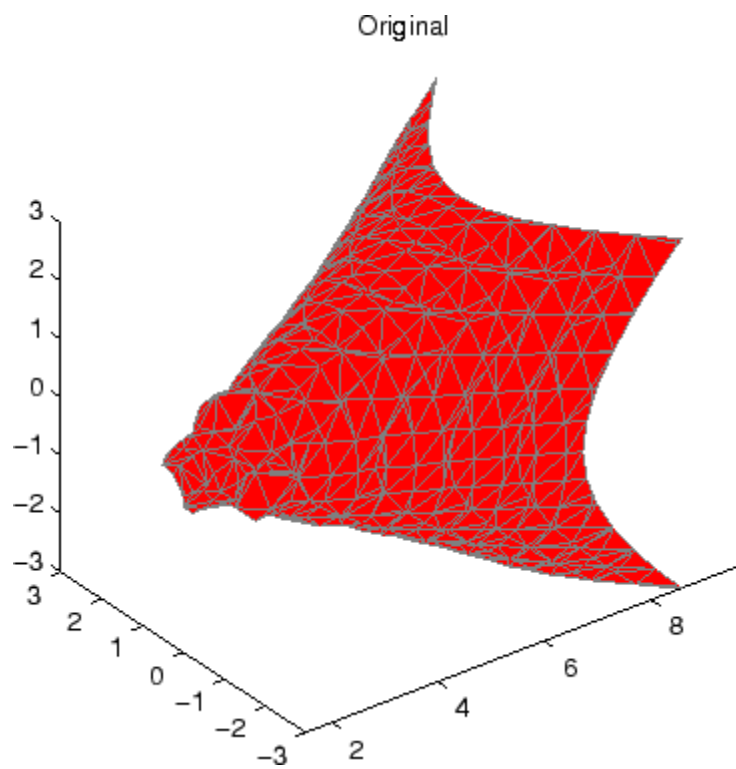
- Use the `daspect`, `view`, and `axis` commands to set up the view and then add a title.
- The `shrinkfaces` command modifies the face/vertex data and passes it directly to `patch`.

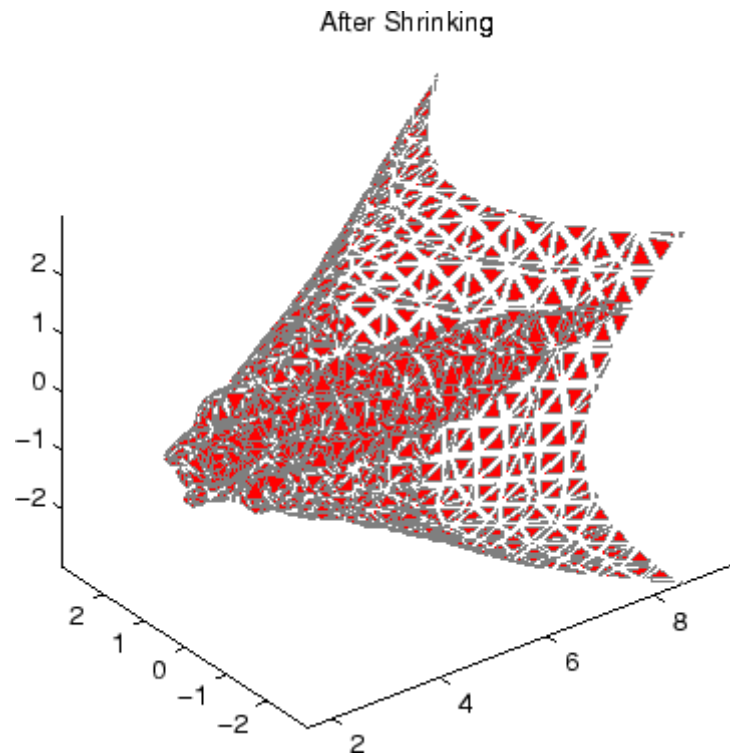
```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')
```

```
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')
```

# shrinkfaces

---





## See Also

[isosurface](#) | [patch](#) | [reducevolume](#) | [daspect](#) | [view](#) | [axis](#)

# sign

---

**Purpose** Signum function

**Syntax**  $Y = \text{sign}(X)$

**Description**  $Y = \text{sign}(X)$  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  $X$ ,  $\text{sign}(X) = X./\text{abs}(X)$ .

**See Also** `abs` | `conj` | `imag` | `real`



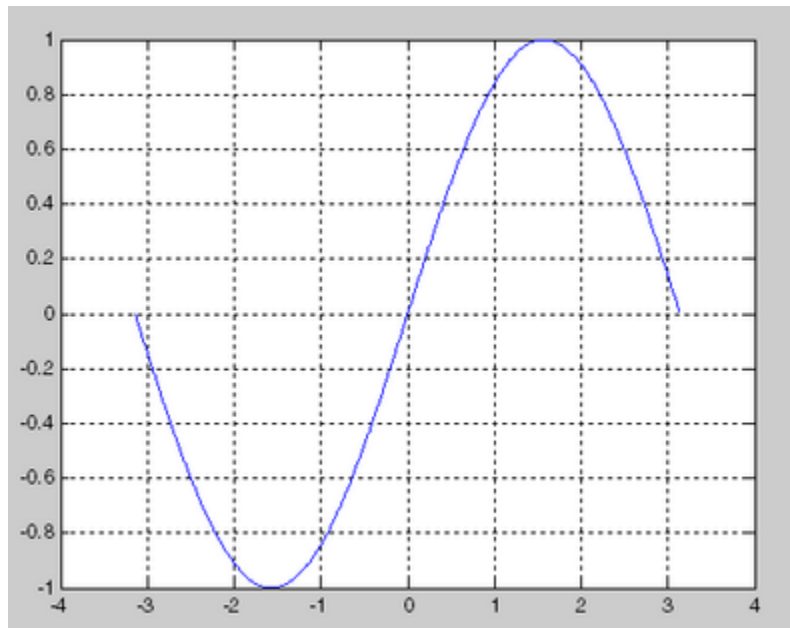
**Purpose** Sine of argument in radians

**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  $-\pi \leq x \leq \pi$ .

```
x = -pi:0.01:pi;  
plot(x,sin(x)), grid on
```



**See Also** `sind` | `asin` | `asind` | `sinh`

# sind

---

|                         |                                                                                                                                                                                                                                                                                                                                                |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Sine of argument in degrees                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>           | $Y = \text{sind}(X)$                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b>      | $Y = \text{sind}(X)$ is the sine of the elements in $X$ . The result, $Y$ , is expressed in degrees.                                                                                                                                                                                                                                           |
| <b>Input Arguments</b>  | <b>X - Angle in degrees</b><br>scalar value   vector   matrix   N-D array<br><br>Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>sind</code> operation is element-wise when $X$ is nonscalar.<br><br><b>Data Types</b><br>single   double<br><b>Complex Number Support:</b> Yes |
| <b>Output Arguments</b> | <b>Y - Sine of angle</b><br>scalar value   vector   matrix   N-D array<br><br>Sine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as $X$ .                                                                                                                                        |
| <b>Examples</b>         | <b>Sine of 180 degrees compared to sine of radians</b><br><br><code>sind(180)</code><br><br><code>ans =</code><br><br>0<br><br><code>sin(pi)</code><br><br><code>ans =</code><br><br>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](#)

# single

---

**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`

**Purpose** Hyperbolic sine of argument in radians

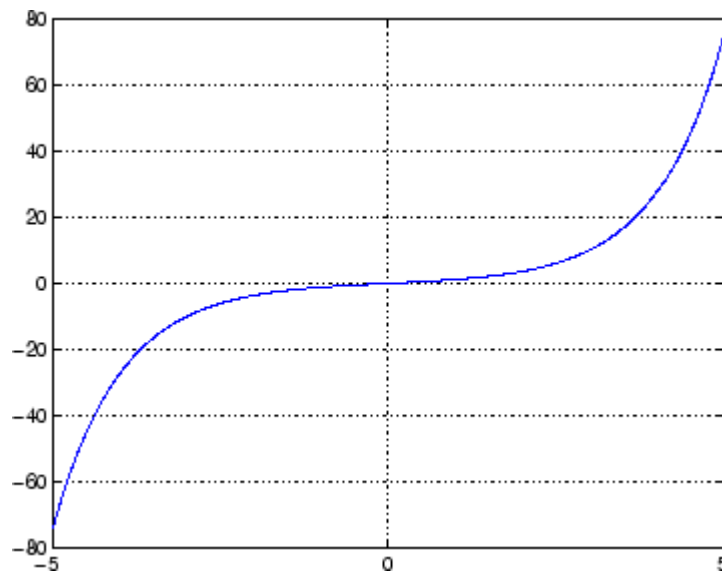
**Syntax**  $Y = \sinh(X)$

**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  $-5 \leq x \leq 5$ .

```
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  $X = \text{ones}(3,4,5)$ , 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
```

# size

---

The “extra” dimensions are collapsed into a single product.

If  $n > \text{ndims}(X)$ , 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`



|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Size of containers.Map object                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Syntax</b>           | <pre>dim = size(mapObj,1) dimVector = size(mapObj) [dim1,dim2,...,dimN] = size(mapObj)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Description</b>      | <p><code>dim = size(mapObj,1)</code> returns a scalar numeric value that indicates the number of key-value pairs in <code>mapObj</code>. If you call <code>size</code> with a numeric second input argument other than 1, the <code>size</code> method returns the scalar numeric value 1.</p> <p><code>dimVector = size(mapObj)</code> returns a two-element vector <code>[k,1]</code>, where <code>k</code> is the number of key-value pairs in <code>mapObj</code>.</p> <p><code>[dim1,dim2,...,dimN] = size(mapObj)</code> returns <code>[k,1,...,1]</code>.</p> |
| <b>Input Arguments</b>  | <p><b>mapObj</b></p> <p>Object of class <code>containers.Map</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Output Arguments</b> | <p><b>dim</b></p> <p>Scalar numeric value that indicates the number of key-value pairs in <code>mapObj</code>.</p> <p><b>dimVector</b></p> <p>Two-element numeric vector <code>[k,1]</code>, where <code>k</code> is the number of key-value pairs in <code>mapObj</code>.</p> <p><b>[dim1,dim2,...,dimN]</b></p> <p>Numeric scalar values. Variable <code>dim1</code> equals <code>k</code>, where <code>k</code> is the number of key-value pairs in <code>mapObj</code>. All other outputs equal 1.</p>                                                           |

## 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];
```

## containers.Map.size

---

```
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](#)

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Array dimensions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>           | <pre>allDims = size(matObj,variable) [dim1,...,dimN] = size(matObj,variable) selectedDim = size(matObj,variable,dim)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Description</b>      | <p><code>allDims = size(matObj,variable)</code> returns the size of each dimension of the specified variable in the file corresponding to <code>matObj</code>. Output <code>allDims</code> is a 1-by-<code>m</code> vector, where <code>m = ndims(variable)</code>.</p> <p><code>[dim1,...,dimN] = size(matObj,variable)</code> returns the sizes of each dimension in separate output variables <code>dim1,...,dimN</code>.</p> <p><code>selectedDim = size(matObj,variable,dim)</code> returns the size of the specified dimension.</p> |
| <b>Tips</b>             | <ul style="list-style-type: none"><li>• Do not call <code>size</code> with the syntax <code>size(matObj.variable)</code>. This syntax loads the entire contents of the variable into memory. For very large variables, this load operation results in Out of Memory errors.</li></ul>                                                                                                                                                                                                                                                     |
| <b>Input Arguments</b>  | <p><b>matObj</b><br/>Object created by the <code>matfile</code> function.</p> <p><b>variable</b><br/>String enclosed in single quotation marks that specifies the name of a variable in the MAT-file corresponding to <code>matObj</code>.</p> <p><b>dim</b><br/>Nonzero positive scalar integer that specifies a dimension of the variable.</p>                                                                                                                                                                                          |
| <b>Output Arguments</b> | <p><b>allDims</b><br/>1-by-<code>m</code> vector of sizes of the dimensions of the specified variable, where <code>m = ndims(variable)</code>.</p> <p><b>dim1,...,dimN</b></p>                                                                                                                                                                                                                                                                                                                                                            |

Scalar numeric values, where `dimK` contains the size of the `K`th 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** Size of triangulation matrix

**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`

## size (tscollection)

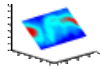
---

|                    |                                                                                                                                                                     |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Size of tscollection object                                                                                                                                         |
| <b>Syntax</b>      | <code>size(tsc)</code>                                                                                                                                              |
| <b>Description</b> | <code>size(tsc)</code> returns <code>[n m]</code> , where <code>n</code> is the length of the time vector and <code>m</code> is the number of tscollection members. |
| <b>See Also</b>    | <code>length (tscollection)</code>   <code>isempty (tscollection)</code>   <code>tscollection</code>                                                                |



**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`.

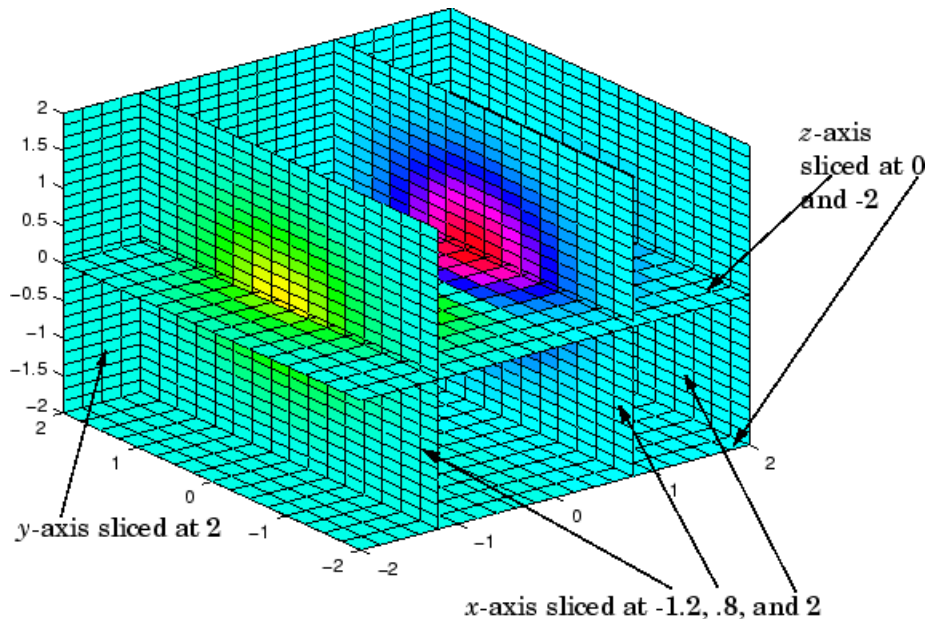
## Examples

Visualize the function

$$v = xe^{(-x^2-y^2-z^2)}$$

over the range  $-2 \leq x \leq 2$ ,  $-2 \leq y \leq 2$ ,  $-2 \leq z \leq 2$ :

```
[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.

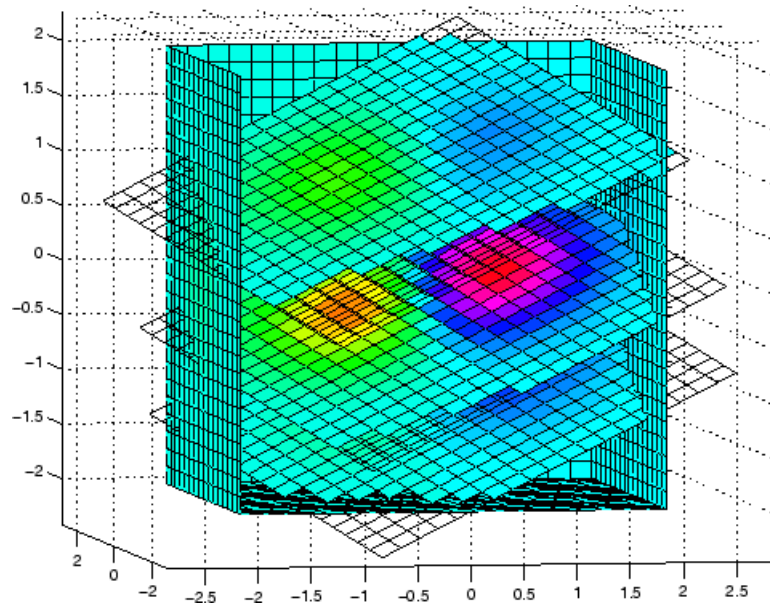
```
for i = -2:.5:2
    hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
    rotate(hsp,[1,-1,1],30)
```

# slice

---

```
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.

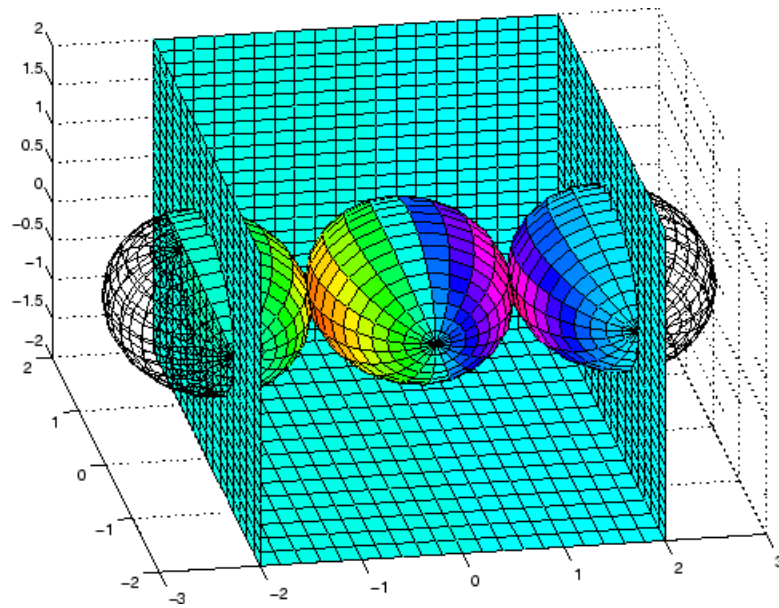
```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries

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.

# slice

---



## 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).

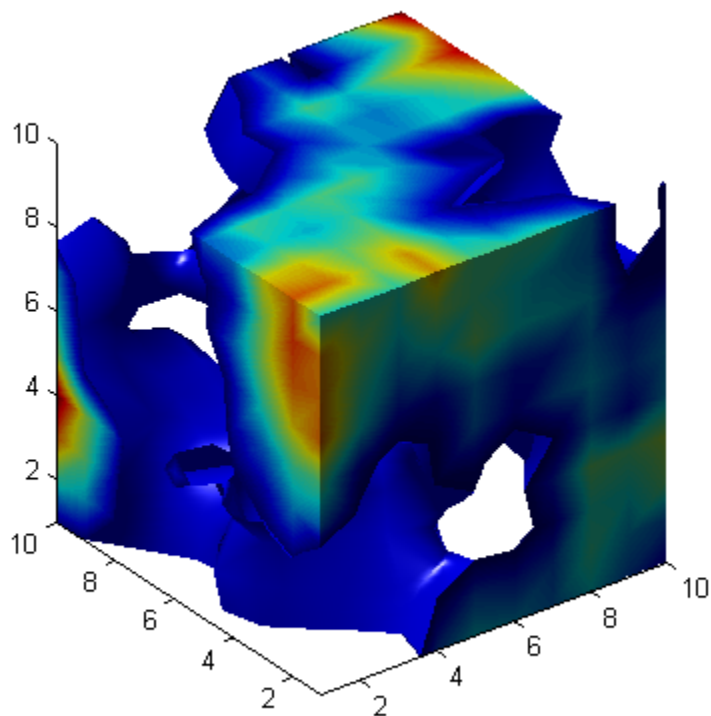
## Examples

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
```

## smooth3

---



### See Also

[isocaps](#) | [isonormals](#) | [isosurface](#) | [patch](#)

### How To

- [Displaying an Isosurface](#)



---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Force snapshot of image for inclusion in published document                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>      | snapnow                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Description</b> | 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.                                                                                                                                                                                                                  |
| <b>Examples</b>    | <p>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.</p> <p>Suppose you have a file that contains the following code:</p> <pre>%% Scale magic Data and %% Display as Image:  for i=1:3     imagesc(magic(i))     snapnow end</pre> <p>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.)</p> |


## Scale magic Data and

### Contents

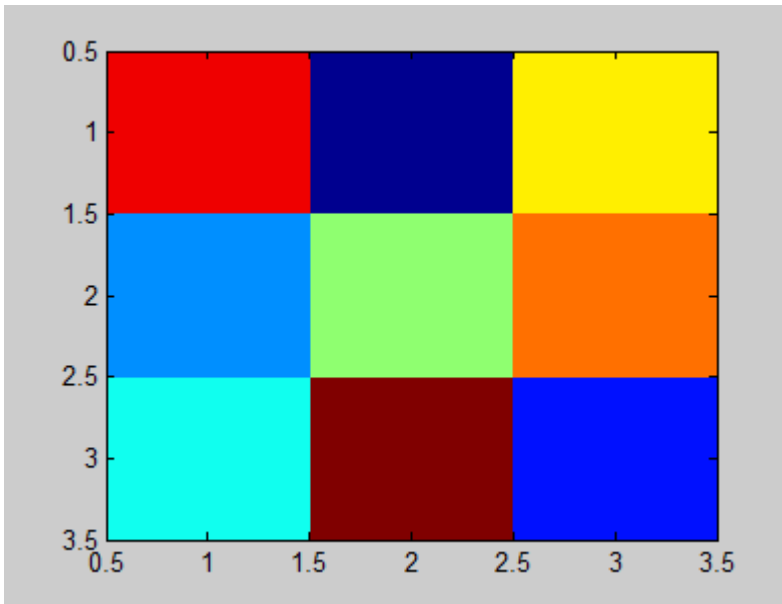
- [Display as Image:](#)

### Display as Image:

```
for i=1:3
    imagesc(magic(i))
    snapnow
end
```



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”

# sort

---

**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 <code>dim</code> or <code>mode</code> 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:

- $\text{abs}(r) < \text{abs}(s)$
- $\text{abs}(r) = \text{abs}(s)$  and  $\text{angle}(r) < \text{angle}(s)$

where  $-\pi < \text{angle}(r) \leq \pi$

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
```

# sort

---

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
```

## sortrows

---

```
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](#)

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Convert matrix of signal data to sound                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Syntax</b>          | <code>sound(y)</code><br><code>sound(y,Fs)</code><br><code>sound(y,Fs,nBits)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>     | <p><code>sound(y)</code> sends audio signal <code>y</code> to the speaker at the default sample rate of 8192 hertz.</p> <p><code>sound(y,Fs)</code> sends audio signal <code>y</code> to the speaker at sample rate <code>Fs</code>.</p> <p><code>sound(y,Fs,nBits)</code> uses <code>nBits</code> bits per sample for audio signal <code>y</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Input Arguments</b> | <p><b>y - Audio data</b><br/>column vector   m-by-2 matrix</p> <p>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 <code>m</code> is the number of audio samples. If <code>y</code> 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.</p> <p><b>Data Types</b><br/>double</p> <p><b>Fs - Sample rate</b><br/>8192 (default)   positive number</p> <p>Sample rate, in hertz, of audio data <code>y</code>, specified as a positive number between 80 and 1000000.</p> <p><b>Data Types</b><br/>single   double</p> <p><b>nBits - Bit depth of sample values</b><br/>8 (default)   16   24</p> |

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  $F_s$ , 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**  
 $S = \text{spaugment}(A, c)$   
 $S = \text{spaugment}(A)$

**Description**  $S = \text{spaugment}(A, c)$  creates the sparse, square, symmetric indefinite matrix  $S = [c \cdot I \ A; \ A' \ 0]$ . The matrix  $S$  is related to the least squares problem

$$\min \text{norm}(b - A \cdot x)$$

by

$$r = b - A \cdot x$$
$$S * [r/c; x] = [b; 0]$$

The optimum value of the residual scaling factor  $c$ , involves  $\min(\text{svd}(A))$  and  $\text{norm}(r)$ , which are usually too expensive to compute.

$S = \text{spaugment}(A)$  without a specified value of  $c$ , uses  $\max(\max(\text{abs}(A))) / 1000$ .

---

**Note** In previous versions of MATLAB product, the augmented matrix was used by sparse linear equation solvers,  $\backslash$  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,real,imag]` 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   1.0000000000000000
1   2   0.5000000000000000
2   2   0.3333333333333333
1   3   0.3333333333333333
2   3   0.2500000000000000
3   3   0.2000000000000000
1   4   0.2500000000000000
2   4   0.2000000000000000
3   4   0.1666666666666667
4   4   0.142857142857143
4   4   0.0000000000000000

```

Then the statements

# spconvert

---

```
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  $m \geq n$ , `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 2-4181 and “Example 5B” on page 2-4183, 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.

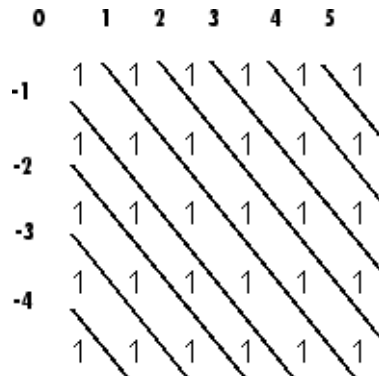
Roughly, A, B, and d are related by

```
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  $m+n-1$  diagonals. 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

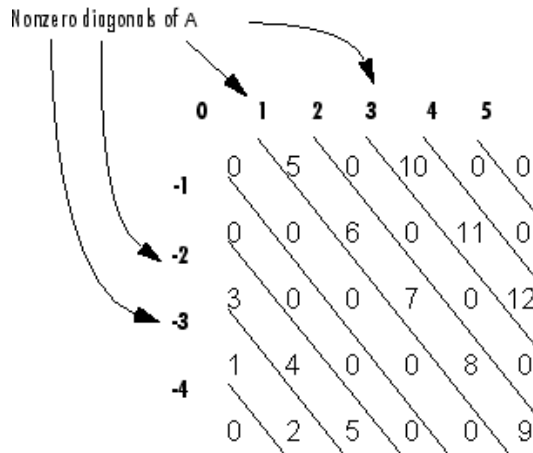
B =

|   |   |   |    |
|---|---|---|----|
| 0 | 0 | 5 | 10 |
| 0 | 0 | 6 | 11 |
| 0 | 3 | 7 | 12 |
| 1 | 4 | 8 | 0  |
| 2 | 5 | 9 | 0  |

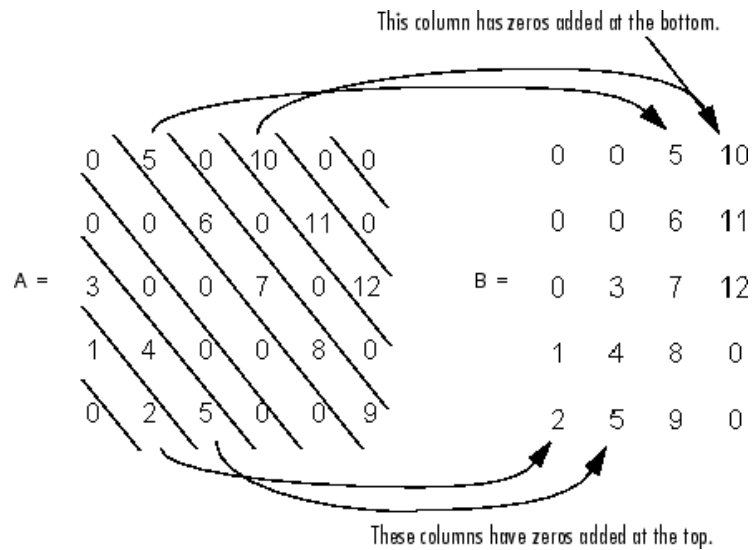
d =

```
-3
-2
1
```

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 2-4176.



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]
```

# spdiags

---

```
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 |

# spdiags

---

```
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 1s 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 1s where `S` has 0s.

```
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
```

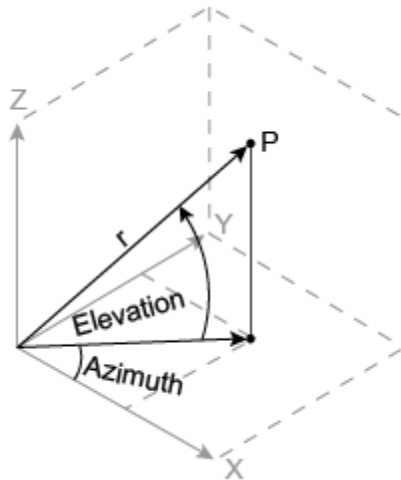
**Purpose** Transform spherical coordinates to Cartesian

**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.

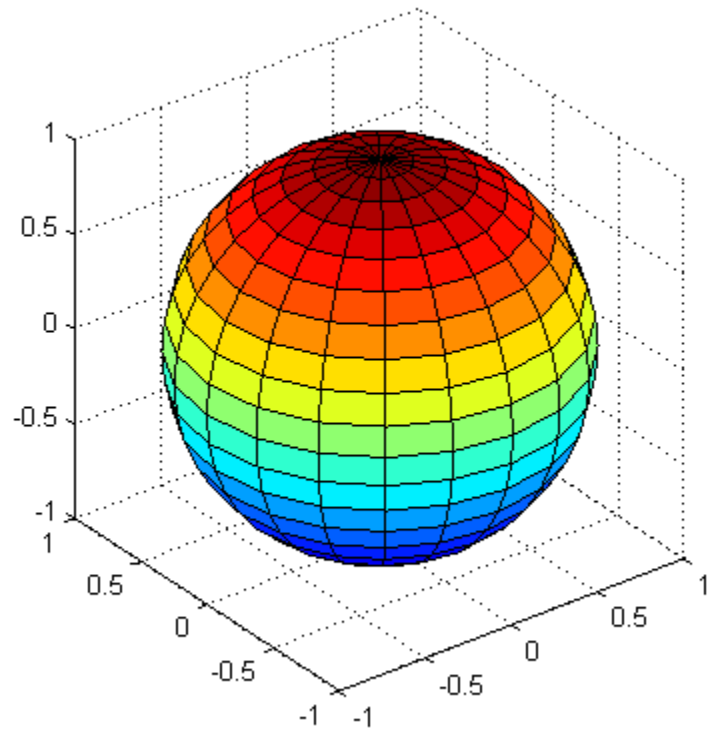
`sphere(n)` draws a `surf` plot of an  $n$ -by- $n$  sphere in the current figure.

`[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
```

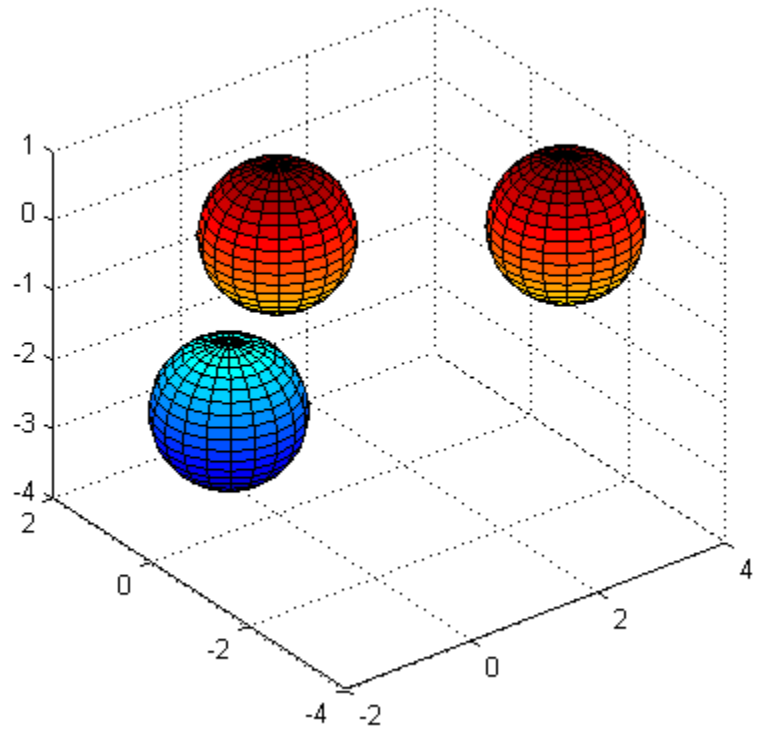


Plot multiple spheres, translating centers away from the origin:

```
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])
```

# sphere

---



**See Also**     cylinder | axis



|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Spin colormap                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>      | <code>spinmap</code><br><code>spinmap(t)</code><br><code>spinmap(t,inc)</code><br><code>spinmap('inf')</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Description</b> | <p>The <code>spinmap</code> 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.</p> <p><code>spinmap</code> cyclically rotates the colormap for approximately five seconds using an incremental value of 2.</p> <p><code>spinmap(t)</code> rotates the colormap for approximately <math>10*t</math> seconds. The amount of time specified by <code>t</code> depends on your hardware configuration (e.g., if you are running MATLAB software over a network).</p> <p><code>spinmap(t,inc)</code> rotates the colormap for approximately <math>10*t</math> seconds and specifies an increment <code>inc</code> by which the colormap shifts. When <code>inc</code> 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.</p> <p><code>spinmap('inf')</code> rotates the colormap for an infinite amount of time. To break the loop, press <b>Ctrl+C</b>.</p> |
| <b>See Also</b>    | <code>colormap</code>   <code>colormapeditor</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

# 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
  - $f(x) = Y(2:end-1)$
  - $df(\min(x)) = Y(1)$
  - $df(\max(x)) = Y(\text{end})$
- 2 If  $Y$  is a matrix or an  $N$ -dimensional array with  $\text{size}(Y,N)$  equal to  $\text{length}(x)+2$ , the following hold:
  - $f(x(j))$  matches the value  $Y(:,\dots,:,j+1)$  for  $j=1:\text{length}(x)$
  - $Df(\min(x))$  matches  $Y(:,\dots,:,1)$
  - $Df(\max(x))$  matches  $Y(:,\dots,:,\text{end})$

---

**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.

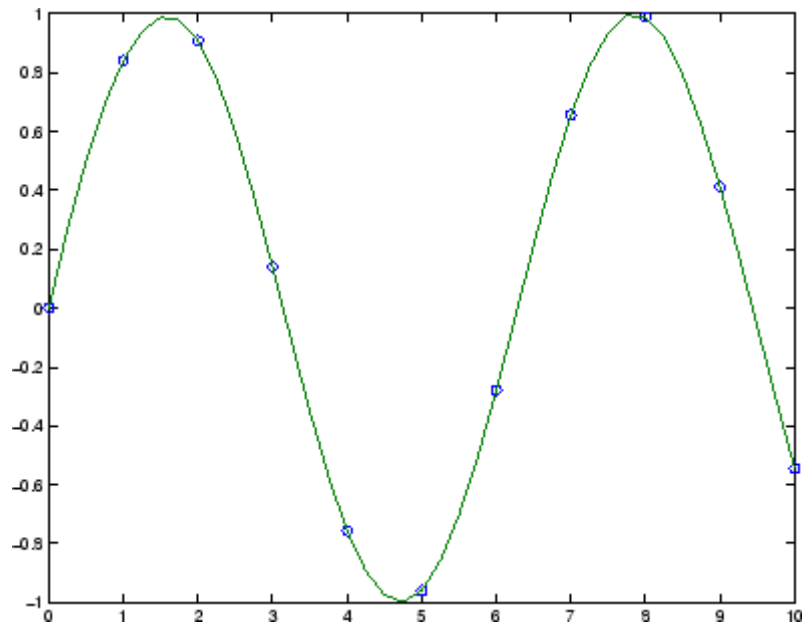
---

## Examples

### Example 1

This generates a sine curve, then samples the spline over a finer mesh.

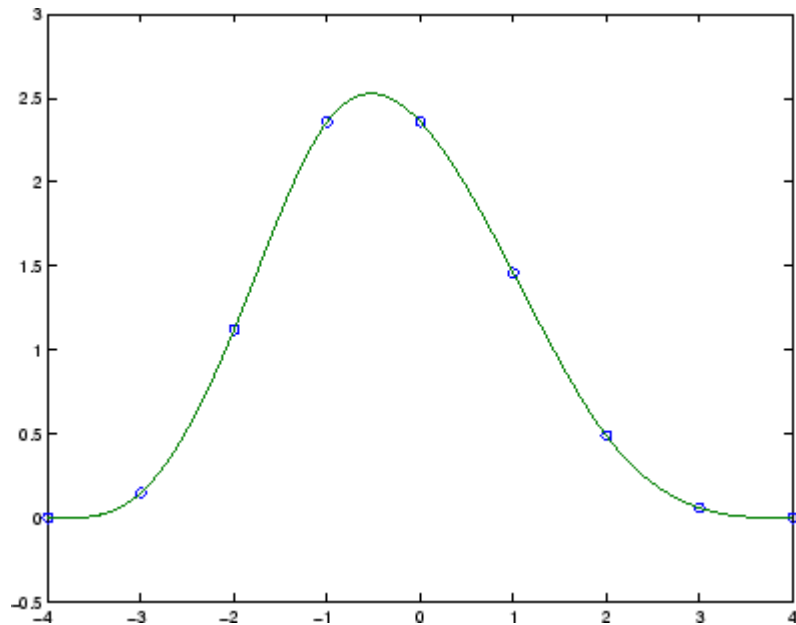
```
x = 0:10;  
y = sin(x);  
xx = 0:.25:10;  
yy = spline(x,y,xx);  
plot(x,y,'o',xx,yy)
```



## 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

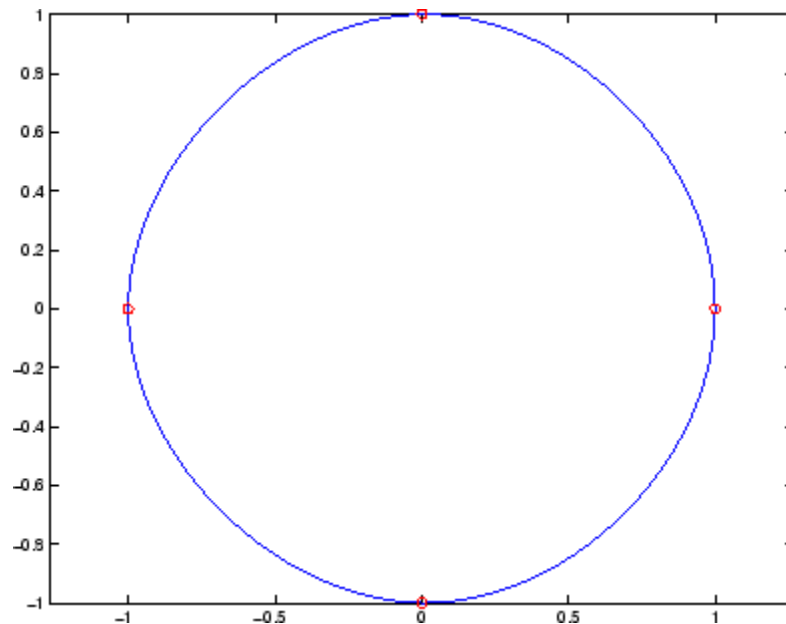
```
ans =  
    270.6060
```

## 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
```

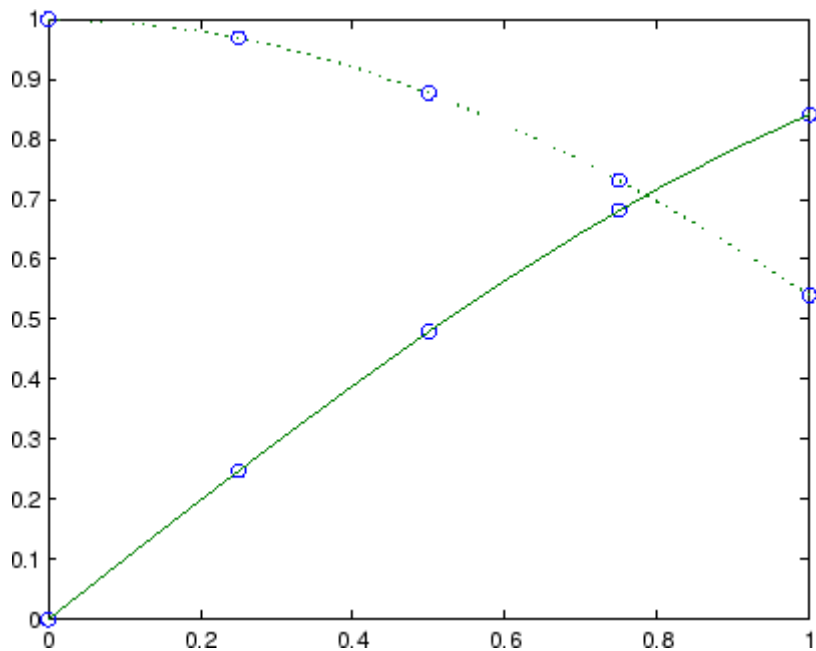
generate the plot of a circle, with the five data points  $y(:,2), \dots, y(:,6)$  marked with o's. Note that this  $y$  contains two more values (i.e., two more columns) than does  $x$ , hence  $y(:,1)$  and  $y(:,end)$  are used as endslopes.



**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

# spline

---

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`



|                    |                                                                                                                                                                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Replace nonzero sparse matrix elements with ones                                                                                                                                                                                                 |
| <b>Syntax</b>      | $R = \text{spones}(S)$                                                                                                                                                                                                                           |
| <b>Description</b> | $R = \text{spones}(S)$ generates a matrix $R$ with the same sparsity structure as $S$ , but with 1's in the nonzero positions.                                                                                                                   |
| <b>Examples</b>    | $c = \text{sum}(\text{spones}(S))$ is the number of nonzeros in each column.<br>$r = \text{sum}(\text{spones}(S'))'$ is the number of nonzeros in each row.<br>$\text{sum}(c)$ and $\text{sum}(r)$ are equal, and are equal to $\text{nnz}(S)$ . |
| <b>See Also</b>    | <a href="#">nnz</a>   <a href="#">spalloc</a>   <a href="#">spfun</a>                                                                                                                                                                            |

# 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',<br>'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 <code>density &gt; wh_frac</code> are ignored in <code>colmmd</code> .                                                                                                                                                                                                                       |
| 'autommd' | Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.                                                                                                                                                                                                                                   |
| 'autoamd' | Nonzero to use <code>colamd</code> ordering with the LU-based \ and /, and to use <code>amd</code> 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 $(\# \text{ nonzeros in the band}) / (\# \text{ nonzeros in a full band})$ . If <code>bandden = 1.0</code> , never use band solver. If <code>bandden = 0.0</code> , always use band solver. Default is <code>0.5</code> . |
| 'umfpack' | Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.                                                                                                                                                                                                                                   |
| 'sym_tol' | Symmetric pivot tolerance. See <code>lu</code> 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

# spparms

---

|            | <b>Keyword</b> | <b>Default</b> | <b>Tight</b> |
|------------|----------------|----------------|--------------|
| 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 \

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Sparse uniformly distributed random matrix                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Syntax</b>      | <pre>R = sprand(S) R = sprand(m,n,density) R = sprand(m,n,density,rc)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p><code>R = sprand(S)</code> has the same sparsity structure as <code>S</code>, but uniformly distributed random entries.</p> <p><code>R = sprand(m,n,density)</code> is a random, <code>m</code>-by-<code>n</code>, sparse matrix with approximately <code>density*m*n</code> uniformly distributed nonzero entries (<code>0 &lt;= density &lt;= 1</code>).</p> <p><code>R = sprand(m,n,density,rc)</code> also has reciprocal condition number approximately equal to <code>rc</code>. <code>R</code> is constructed from a sum of matrices of rank one.</p> <p>If <code>rc</code> is a vector of length <code>lr</code>, where <code>lr &lt;= min(m,n)</code>, then <code>R</code> has <code>rc</code> as its first <code>lr</code> singular values, all others are zero. In this case, <code>R</code> 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.</p> |
| <b>Tips</b>        | <ul style="list-style-type: none"><li>• <code>sprand</code> uses the same random number generator as <code>rand</code>, <code>randi</code>, and <code>randn</code>. You control this generator with <code>rng</code>.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>See Also</b>    | <code>sprandn</code>   <code>sprandsym</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

# 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  $\text{density} * m * n$  normally distributed nonzero entries ( $0 \leq \text{density} \leq 1$ ).

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  $lr \leq \min(m,n)$ , 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`.

# sprandsym

---

## **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`

# sprintf

---

**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

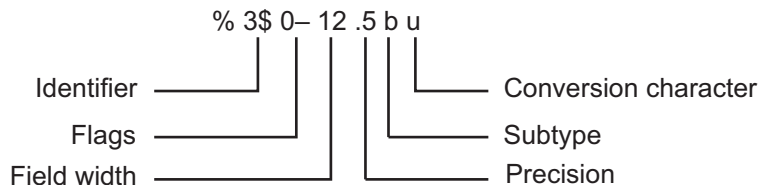
Format of the output fields, specified as a string.

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<br>%bo<br>%bu | Double-precision hexadecimal, octal, or decimal value<br>Example: %bx prints pi as 400921fb54442d18 |
|                       | %tx or %tX<br>%to<br>%tu | Single-precision hexadecimal, octal, or decimal value<br>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:

# sprintf

---

- 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.
- Escape characters, including:

|             |                              |
|-------------|------------------------------|
| '           | Single quotation mark        |
| %           | Percent character            |
| \           | Backslash                    |
| \a          | Alarm                        |
| \b          | Backspace                    |
| \f          | Form feed                    |
| \n          | New line                     |
| \r          | Carriage return              |
| \t          | Horizontal tab               |
| \v          | Vertical tab                 |
| \x <i>N</i> | Hexadecimal number, <i>N</i> |
| \o <i>N</i> | Octal number, <i>N</i>       |

The following limitations apply to conversions:

- Numeric conversions print only the real component of complex numbers.

# sprintf

---

- 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 =
00000000000000000000000123456
```

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”

**Purpose** Visualize sparsity pattern

**Syntax**

```
spy(S)
spy(S,markersize)
spy(S,'LineStyle')
spy(S,'LineStyle',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,'LineStyle')`, where `LineStyle` is a string, uses the specified plot marker type and color.

`spy(S,'LineStyle',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.

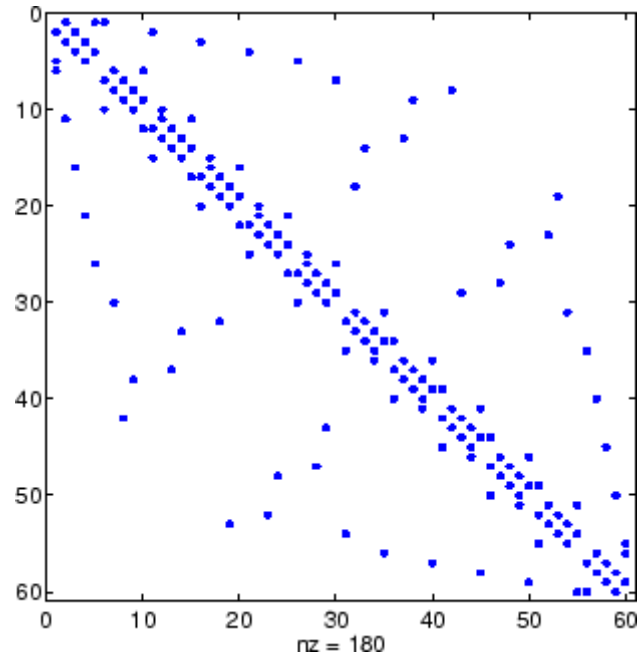
---

**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`

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Matrix square root                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Syntax</b>      | <pre>X = sqrtm(A) [X, resnorm] = sqrtm(A) [X, alpha, condest] = sqrtm(A)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p><math>X = \text{sqrtm}(A)</math> is the principal square root of the matrix <math>A</math>, i.e. <math>X^2 = A</math>. <math>X</math> is the unique square root for which every eigenvalue has nonnegative real part. If <math>A</math> has any eigenvalues with negative real parts then a complex result is produced. If <math>A</math> is singular then <math>A</math> may not have a square root. A warning is printed if exact singularity is detected.</p> <p><math>[X, \text{resnorm}] = \text{sqrtm}(A)</math> does not print any warning, and returns the residual, <math>\text{norm}(A - X^2, 'fro') / \text{norm}(A, 'fro')</math>.</p> <p><math>[X, \text{alpha}, \text{condest}] = \text{sqrtm}(A)</math> returns a stability factor <math>\text{alpha}</math> and an estimate <math>\text{condest}</math> of the matrix square root condition number of <math>X</math>. The residual <math>\text{norm}(A - X^2, 'fro') / \text{norm}(A, 'fro')</math> is bounded approximately by <math>n * \text{alpha} * \text{eps}</math> and the Frobenius norm relative error in <math>X</math> is bounded approximately by <math>n * \text{alpha} * \text{condest} * \text{eps}</math>, where <math>n = \max(\text{size}(A))</math>.</p> |
| <b>Tips</b>        | <p>If <math>A</math> is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.</p> <p>Some matrices, like <math>A = \begin{bmatrix} 0 &amp; 1 \\ 0 &amp; 0 \end{bmatrix}</math>, do not have any square roots, real or complex, and <code>sqrtm</code> cannot be expected to produce one.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Examples</b>    | <p><b>Example 1</b></p> <p>A matrix representation of the fourth difference operator is</p> <pre>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</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

This matrix is symmetric and positive definite. Its unique positive definite square root,  $Y = \text{sqrtm}(A)$ , is a representation of the second difference operator.

$$Y = \begin{bmatrix} 2 & -1 & -0 & -0 & -0 \\ -1 & 2 & -1 & 0 & -0 \\ 0 & -1 & 2 & -1 & 0 \\ -0 & 0 & -1 & 2 & -1 \\ -0 & -0 & -0 & -1 & 2 \end{bmatrix}$$

## Example 2

The matrix

$$A = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

has four square roots. Two of them are

$$Y1 = \begin{bmatrix} 1.5667 & 1.7408 \\ 2.6112 & 4.1779 \end{bmatrix}$$

and

$$Y2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The other two are  $-Y1$  and  $-Y2$ . All four can be obtained from the eigenvalues and vectors of  $A$ .

$$[V,D] = \text{eig}(A);$$
$$D = \begin{bmatrix} 0.1386 & 0 \\ 0 & 28.8614 \end{bmatrix}$$

The four square roots of the diagonal matrix D result from the four choices of sign in

$$S = \begin{pmatrix} -0.3723 & 0 \\ 0 & -5.3723 \end{pmatrix}$$

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%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

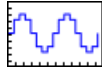
fscanf | sprintf | textscan

# stairs

---

## Purpose

Stairstep graph



## Syntax

```
stairs(Y)
stairs(X,Y)
stairs(...,LineStyle)
stairs(...,'PropertyName',propertyvalue)
stairs(axes_handle,...)
h = stairs(...)
[xb,yb] = stairs(Y,...)
```

## Description

Stairstep graphs are useful for drawing time-history graphs of digitally sampled data.

`stairs(Y)` draws a stairstep graph of the elements of `Y`, drawing one line per column for matrices. The axes `ColorOrder` property determines the color of the lines.

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 the number of rows in `Y`.

`stairs(X,Y)` plots the elements in `Y` at the locations specified in `X`.

`X` must be the same size as `Y` or, if `Y` is a matrix, `X` can be a row or a column vector such that

```
length(X) = size(Y,1)
```

`stairs(...,LineStyle)` specifies a line style, marker symbol, and color for the graph. (See `LineStyle` for more information.)

`stairs(...,'PropertyName',propertyvalue)` creates the stairstep graph, applying the specified property settings. See `Stairseries` properties for a description of properties.

`stairs(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes object (`gca`).

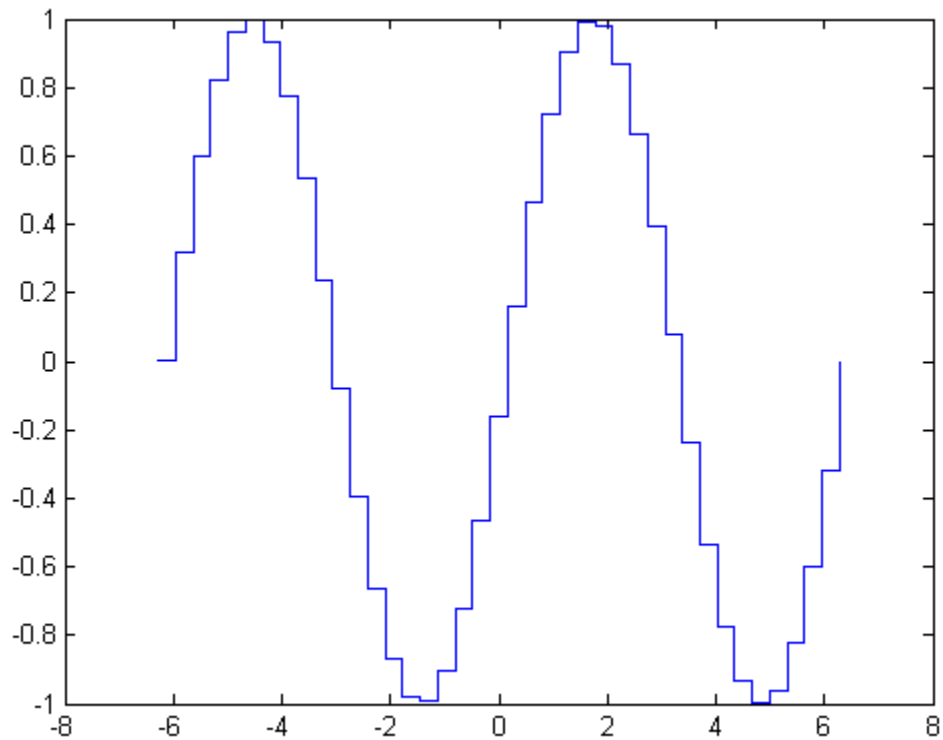
`h = stairs(...)` returns the handles of the stairseries objects created (one per matrix column).

`[xb,yb] = stairs(Y,...)` does not draw graphs, but returns vectors `xb` and `yb` such that `plot(xb,yb)` plots the stairstep graph.

## Examples

Create a stairstep plot of a sine wave.

```
x = linspace(-2*pi,2*pi,40);  
stairs(x,sin(x))
```



# stairs

---

## See Also

[bar](#) | [hist](#) | [stem](#) | [LineSpec](#) | [Stairseries](#)

## 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.

| <b>IconDisplayStyle Value</b> | <b>Purpose</b>                                                                        |
|-------------------------------|---------------------------------------------------------------------------------------|
| <code>on</code>               | Include the stairseries object in a legend as one entry, but not its children objects |
| <code>off</code>              | Do not include the stairseries or its children in a legend (default)                  |
| <code>children</code>         | Include only the children of the stairseries as separate entries in the legend        |

# Stairseries Properties

---

## 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-to-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.

# Stairseries Properties

---

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.

# Stairseries Properties

---

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

# Stairseries Properties

---

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}`

# Stairseries Properties

---

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          |

# Stairseries Properties

---

| 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 =  $\frac{1}{72}$  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.

# Stairseries Properties

---

- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is none (the factory default for axes objects), 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, `hgroup`, or `hgtransform`

*Parent of object.* Handle of the object's parent. The parent is normally the axes, `hgroup`, 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');
```

# Stairseries Properties

---

## 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,  $\text{length}(\text{XData}) == \text{size}(\text{YData}, 1)$ .

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', 'xdatabasename')
```

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

# Stairseries Properties

---

*Stairs plot data.* YData contains the data plotted in the staircase graph. Each value in YData is represented by a marker in the staircase 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`

# startat

---

**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

`datetime` | `datestr` | `now` | `timer` | `start` | `stop`

# startup

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Startup file for user-defined options                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Syntax</b>      | startup                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b> | <p>startup executes commands of your choosing when the MATLAB program starts.</p> <p>Create a <code>startup.m</code> file in your MATLAB startup folder and put in the file any commands you want executed at MATLAB startup. For example, your <code>startup.m</code> file might include physical constants, defaults for Handle Graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.</p> |
| <b>Algorithms</b>  | <p>The MATLAB program executes the <code>matlabrc.m</code> file when it starts. <code>matlabrc.m</code> invokes <code>startup.m</code>, if it exists on the MATLAB search path.</p> <p>You can extend this process to create additional startup files, if needed.</p> <p>MathWorks does not recommend modifying the <code>matlabrc.m</code> file, except perhaps by system administrators in network configurations.</p>                           |
| <b>See Also</b>    | <code>finish</code>   <code>matlabrc</code>   <code>matlabroot</code>   <code>path</code>   <code>quit</code>   <code>userpath</code>                                                                                                                                                                                                                                                                                                              |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Specifying Startup Options in the MATLAB Startup File”</li><li>• Preferences</li></ul>                                                                                                                                                                                                                                                                                                                    |

**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  $X$ .

# std

---

`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

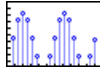
For matrix `X`

```
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(...,LineStyle)
stem(...,'PropertyName',PropertyValue,...)
stem(axes_handle,...)
h = stem(...)
```

## Description

A two-dimensional stem plot displays data as lines extending from a baseline along the  $x$ -axis. A circle (the default) or other marker whose  $y$ -position represents the data value terminates each stem.

`stem(Y)` plots the data sequence  $Y$  as stems that extend from equally spaced and automatically generated values along the  $x$ -axis. When  $Y$  is a matrix, `stem` plots all elements in a row against the same  $x$  value.

`stem(X,Y)` plots  $X$  versus the columns of  $Y$ .  $X$  and  $Y$  must be 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.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

`stem(...,LineStyle)` specifies the line style, marker symbol, and color for the stem and top marker (the baseline is not affected). See `LineStyle` for more information.

`stem(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `stemseries` objects the function creates.

`stem(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

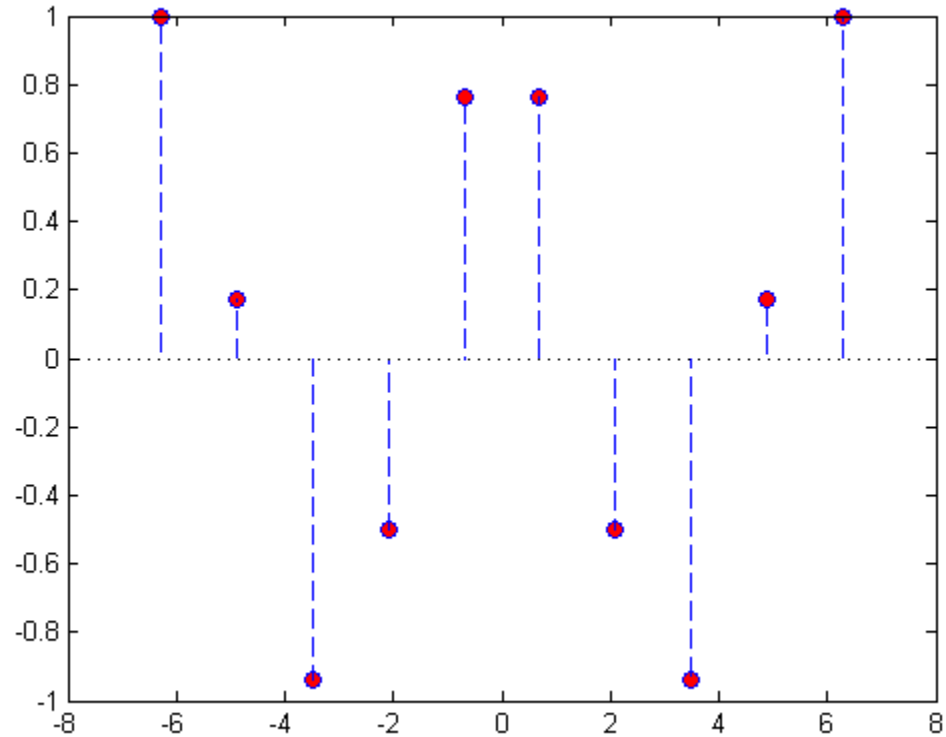
`h = stem(...)` returns a vector of `stemseries` object handles in `h`, one handle per column of data in `Y`.

## Examples

### Single Series of Data

This example creates a stem plot representing the cosine of 10 values linearly spaced between 0 and  $2\pi$ . Note that the line style of the baseline is set by first getting its handle from the `stemseries` object's `BaseLine` property.

```
figure
t = linspace(-2*pi,2*pi,10);
h = stem(t,cos(t),'fill','--');
set(get(h,'BaseLine'),'LineStyle',':')
set(h,'MarkerFaceColor','red')
```



If you do not want the baseline to show, you can make the baseline invisible:

```
set(get(h, 'Baseline'), 'Visible', 'off')
```

where `h` is the `stemseries` handle returned by the `stem` function.

Delete the baseline with this command:

```
delete(get(h, 'Baseline'))
```

Use similar code to change the color or style of the baseline, specifying any line property and value, for example,

```
set(get(h, 'Baseline'), 'LineWidth', 3)
```

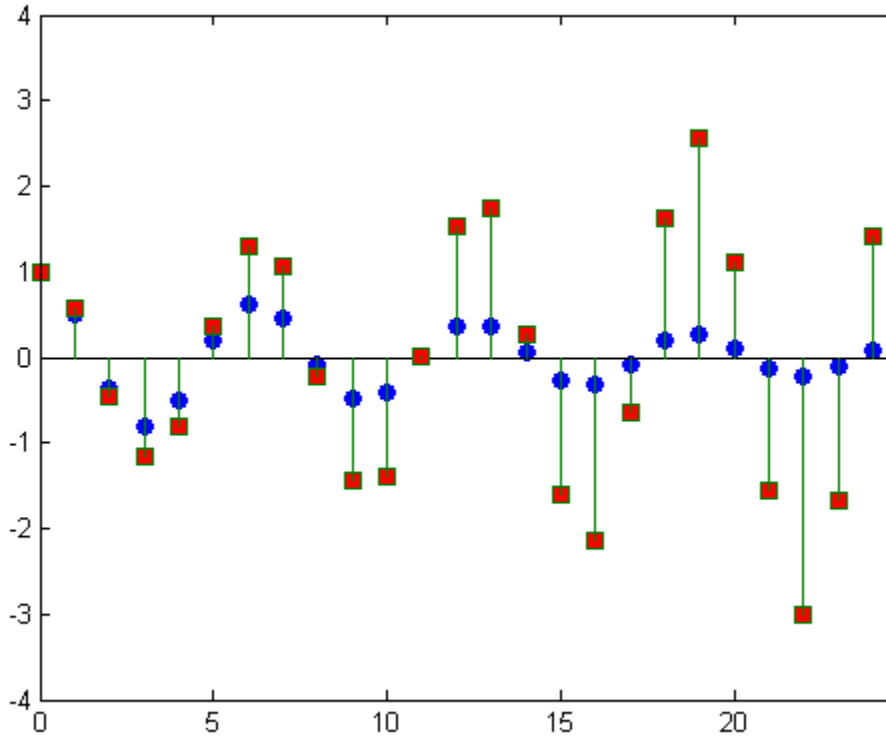
## Two Series of Data on One Graph

This example creates a stem plot from a two-column matrix. In this case, the `stem` function creates two `stemseries` objects, one for each column of data. Both objects' handles are returned in the output argument `h`.

- `h(1)` is the handle to the `stemseries` object plotting the expression  $\exp(-.07*x) \cdot \cos(x)$ .
- `h(2)` is the handle to the `stemseries` object plotting the expression  $\exp(.05*x) \cdot \cos(x)$ .

```
figure
x = 0:25;
y = [exp(-.07*x) .* cos(x); exp(.05*x) .* cos(x)]';
h = stem(x,y);
set(h(1), 'MarkerFaceColor', 'blue')
set(h(2), 'MarkerFaceColor', 'red', 'Marker', 'square')
```





**See Also**

[bar](#) | [plot](#) | [stairs](#) | [Stemseries Properties](#) | [LineStyle](#)

# stem3

---

## Purpose

Plot 3-D discrete sequence data



## Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineStyle)
stem3(...,'PropertyName',PropertyValue,...)
h = stem3(...)
```

## Description

Three-dimensional stem plots display lines extending from the  $x$ - $y$  plane. A circle (the default) or other marker symbol whose  $z$ -position represents the data value terminates each stem.

`stem3(Z)` plots the data sequence  $Z$  as stems that extend from the  $x$ - $y$  plane.  $x$  and  $y$  are generated automatically. When  $Z$  is a row vector, `stem3` plots all elements at equally spaced  $x$  values against the same  $y$  value. When  $Z$  is a column vector, `stem3` plots all elements at equally spaced  $y$  values against the same  $x$  value.

`stem3(X,Y,Z)` plots the data sequence  $Z$  at values specified by  $X$  and  $Y$ .  $X$ ,  $Y$ , and  $Z$  must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

`stem3(...,LineStyle)` specifies the line style, marker symbol, and color for the stems. See `LineStyle` for more information.

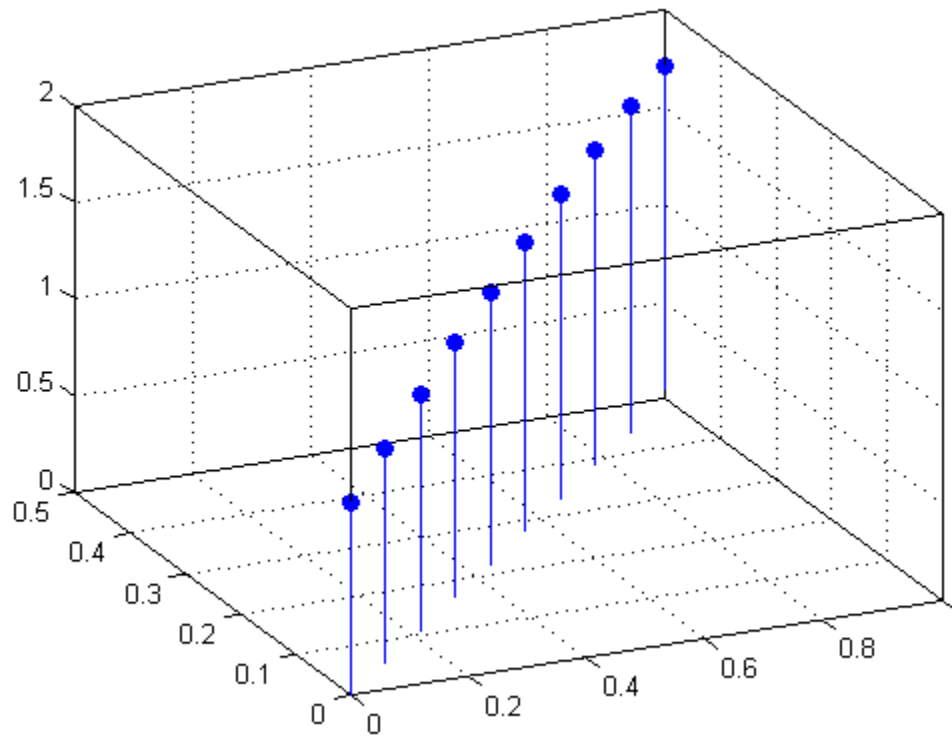
`stem3(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `stemseries` objects the function creates.

`h = stem3(...)` returns handles to `stemseries` graphics objects.

## Examples

Create a three-dimensional stem plot to visualize a function of two variables.

```
figure
X = linspace(0,1,10);
Y = X./2;
Z = sin(X) + cos(Y);
stem3(X,Y,Z,'fill')
view(-25,30)
```

**See Also**

[bar](#) | [plot](#) | [stairs](#) | [stem](#) | [Stemseries Properties](#)

# stem3

---

## How To

- Three-Dimensional Stem Plots

## 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.

| <b>IconDisplayStyle Value</b> | <b>Purpose</b>                                                                       |
|-------------------------------|--------------------------------------------------------------------------------------|
| 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        |

# Stemseries Properties

---

## 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`

# Stemseries Properties

---

*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.

# Stemseries Properties

---

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.

# Stemseries Properties

---

- `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

# Stemseries Properties

---

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.

# Stemseries Properties

---

---

**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}

# Stemseries Properties

---

*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` (the factory default for axes objects), 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, `hgggroup`, 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:

# Stemseries Properties

---

```
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

# Stemseries Properties

---

*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

# Stemseries Properties

---

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** 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

# stopasync

---

|                    |                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Stop asynchronous read and write operations                                                                                                   |
| <b>Syntax</b>      | <code>stopasync(obj)</code>                                                                                                                   |
| <b>Description</b> | <code>stopasync(obj)</code> stops any asynchronous read or write operation that is in progress for the serial port object, <code>obj</code> . |

**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`

**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`

# str2func

---

**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](#)

# str2mat

---

**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



**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,<br>67                        | 1-by-4 row vector of double    |
| '500; 250; 125;<br>62.5' | 500.0000<br>250.0000<br>125.0000<br>62.5000 | 4-by-1 column vector of double |

# str2num

| String Input       | Numeric Output           | Output Class            |
|--------------------|--------------------------|-------------------------|
| '1 23 6 21; 53:56' | 1 23 6 21<br>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

---

```
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:

## strcat

---

```
c = {'Q'};  
abc = strcat(a, b, c)
```

MATLAB returns

```
abc =  
    'abcdejk1Q'    '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` | special character

**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.

# strcmp

---

## 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.
```

# strcmp

---

```
%           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` | `regexp` |

# strcmpi

---

**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:

```
strcmapi('Yes', 'No')
ans =
     0
strcmapi('Yes', 'yes')
ans =
     1
```

Create two cell arrays of strings and call `strcmapi` to compare them:

```
A = {'Handle Graphics', 'Statistics'; ...
     'Toolboxes', 'MathWorks'};

B = {'Handle Graphics', 'Signal Processing'; ...
     'Toolboxes', 'MATHWORKS'};

match = strcmapi(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}`.

## strcmpi

---

- `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` |

**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`.

# 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](#)



|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Compute 3-D streamline data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Syntax</b>      | <pre>XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz) XYZ = stream3(U,V,W,startx,starty,startz) XYZ = stream3(...,options)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p><code>XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)</code> computes streamlines from vector data U, V, W.</p> <p>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 <code>meshgrid</code>.</p> <p><code>startx</code>, <code>starty</code>, and <code>startz</code> define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.</p> <p>The returned value <code>XYZ</code> contains a cell array of vertex arrays.</p> <p><code>XYZ = stream3(U,V,W,startx,starty,startz)</code> assumes the arrays X, Y, and Z are defined as <code>[X,Y,Z] = meshgrid(1:N,1:M,1:P)</code> where <code>[M,N,P] = size(U)</code>.</p> <p><code>XYZ = stream3(...,options)</code> specifies the options used when creating the streamlines. Define <code>options</code> as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:</p> <pre>[stepsize]</pre> <p>or</p> <pre>[stepsize, max_number_vertices]</pre> <p>If you do not specify values, MATLAB software uses the default:</p> <ul style="list-style-type: none"><li>• Step size = 0.1 (one tenth of a cell)</li><li>• Maximum number of vertices = 10000</li></ul> <p>Use the <code>streamline</code> command to plot the data returned by <code>stream3</code>.</p> |

# 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

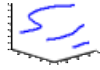
[coneplot](#) | [stream2](#) | [streamline](#)

## How To

- [Specifying Starting Points for Stream Plots](#)

**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  $[X,Y,Z] = \text{meshgrid}(1:N,1:M,1:P)$ , where  $[M,N,P] = \text{size}(U)$ .

`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`.

# streamline

---

`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](#)

# streamparticles

---

**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 <code>streamparticles</code> |
|------------------------------|-------------------------------------------|
| <code>EraseMode</code>       | <code>xor</code>                          |
| <code>LineStyle</code>       | <code>none</code>                         |
| <code>Marker</code>          | <code>o</code>                            |
| <code>MarkerEdgeColor</code> | <code>none</code>                         |
| <code>MarkerFaceColor</code> | <code>red</code>                          |

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

---

`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.

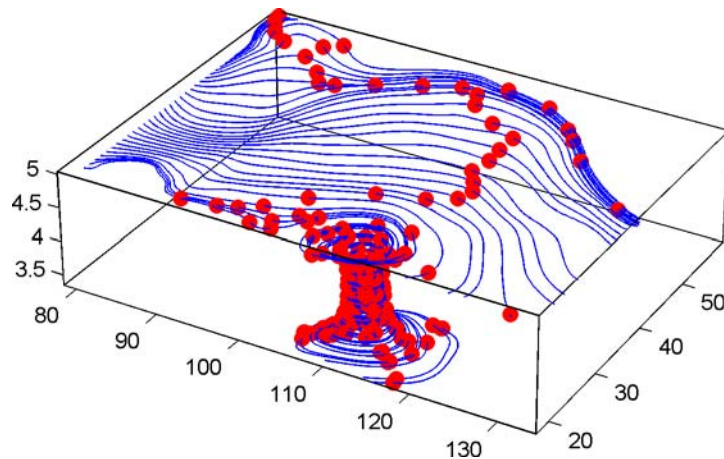
## Examples

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](#)

# streamribbon

---

**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`).

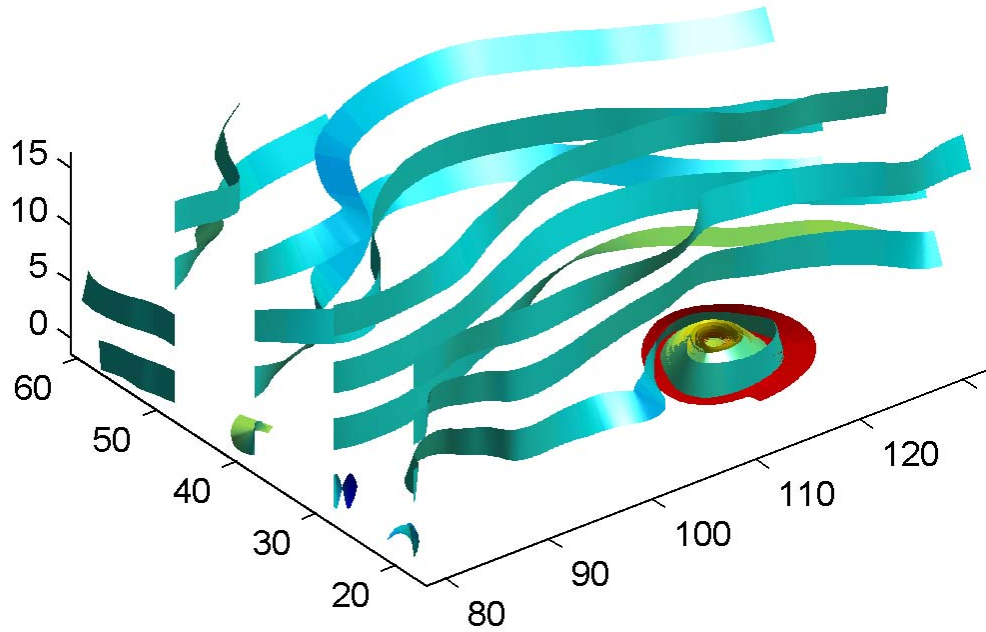
`h = streamribbon(...)` returns a vector of handles (one per start point) to surface objects.

## 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
```

# streamribbon



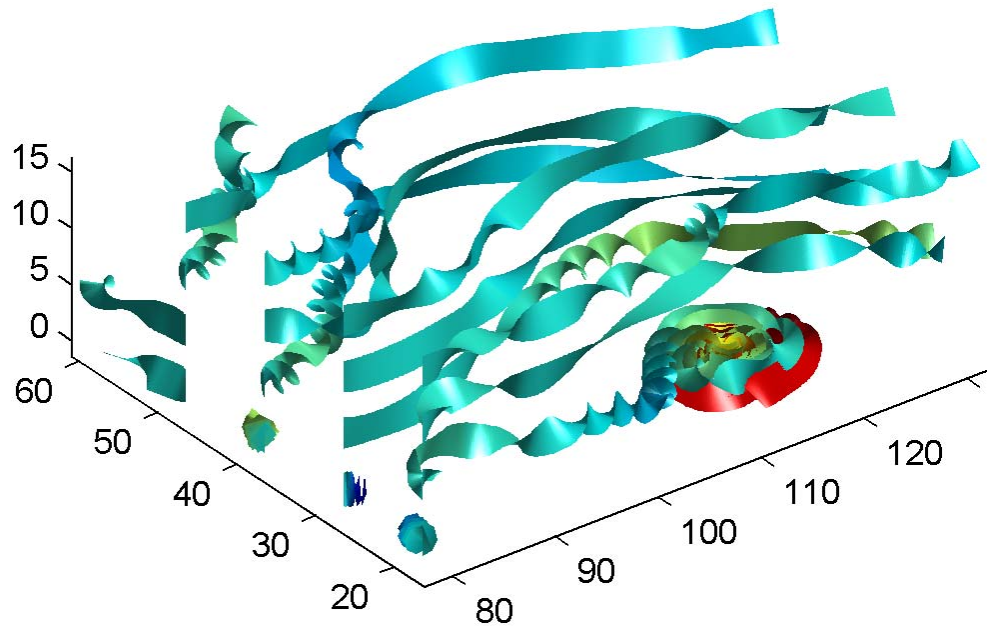
This example uses precalculated vertex data (`stream3`), curl average velocity (`curl1`), and speed  $\sqrt{u^2 + v^2 + w^2}$ . Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to 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).*0.1;
streamribbon(verts,x,y,z,cav,spd);
% Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
```

# streamribbon

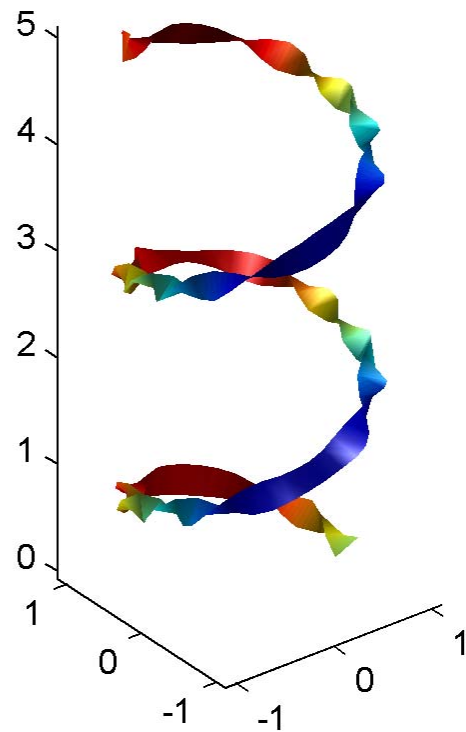
---



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.

# streamribbon

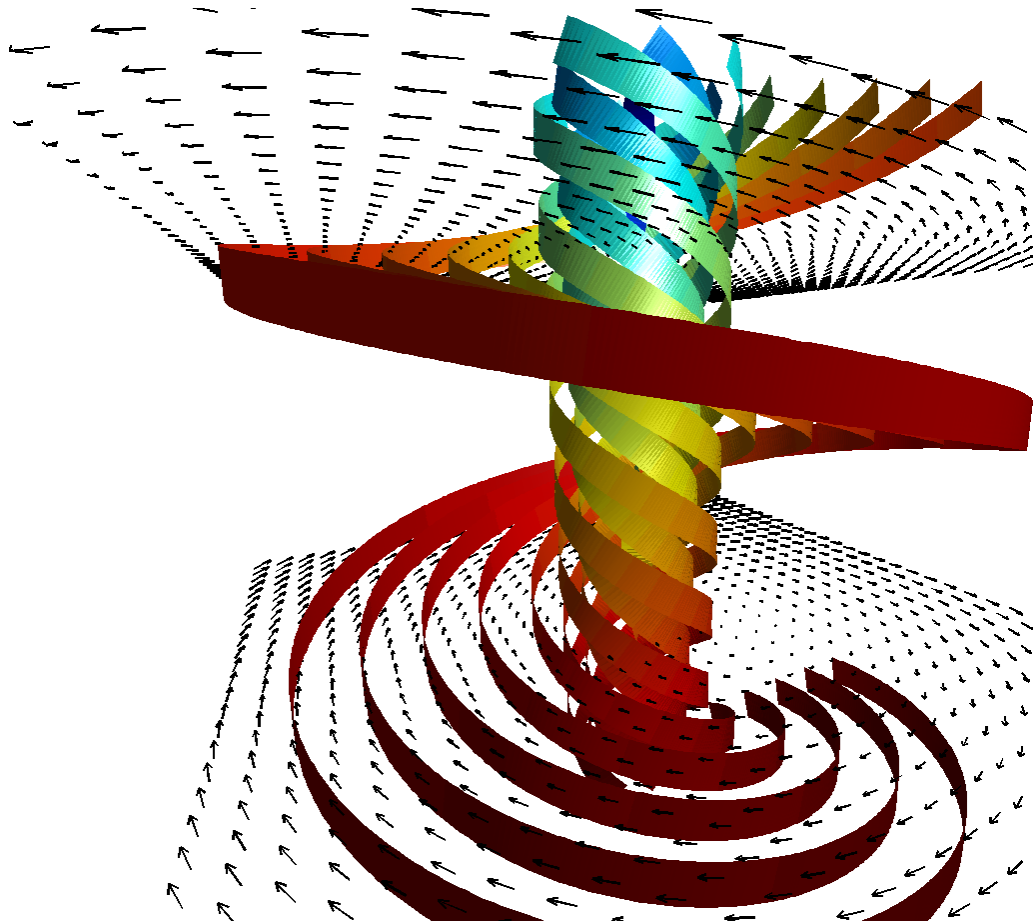
---

```
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

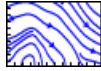
- “Volume Visualization”
- [Displaying Curl with Stream Ribbons](#)
- [Specifying Starting Points for Stream Plots](#)

# streamslice

---

## 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(...,'arrowmode')` 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`).

# streamslice

---

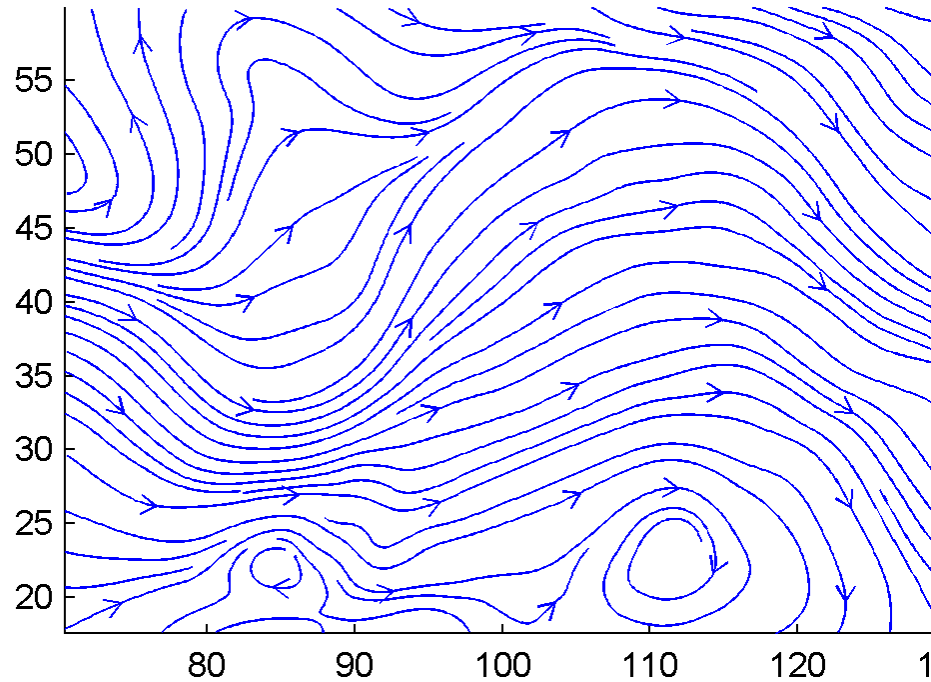
`h = streamslice(...)` returns a vector of handles to the line objects created.

`[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
```



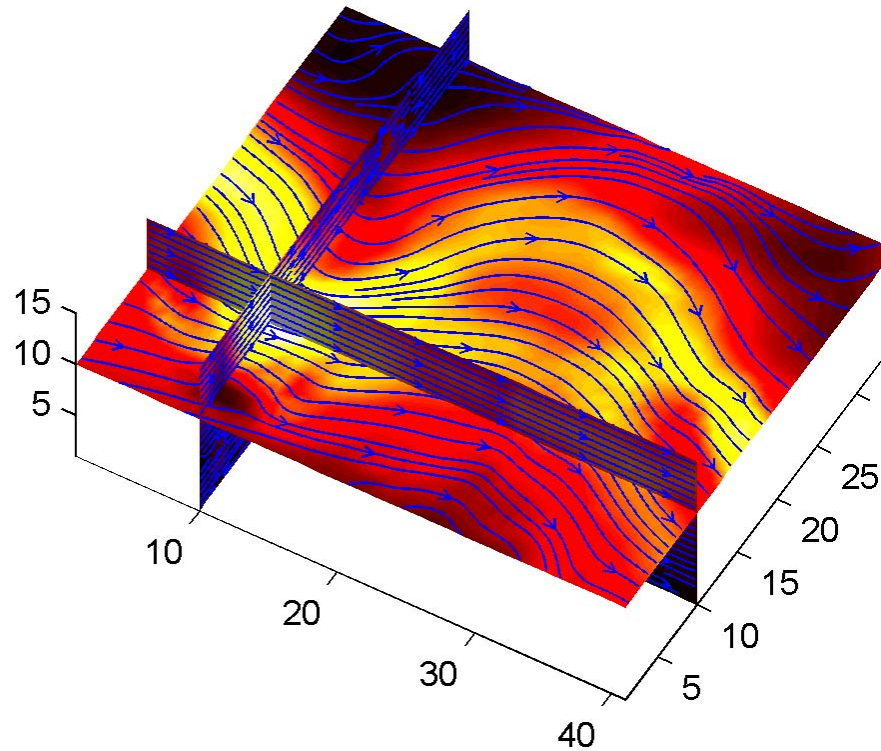
This example uses `streamslic` to calculate vertex data for the streamlines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed  $\sqrt{u^2 + v^2 + w^2}$  are drawn by `slice` in the same planes.

figure

## streamslice

---

```
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 `streamslic` 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.

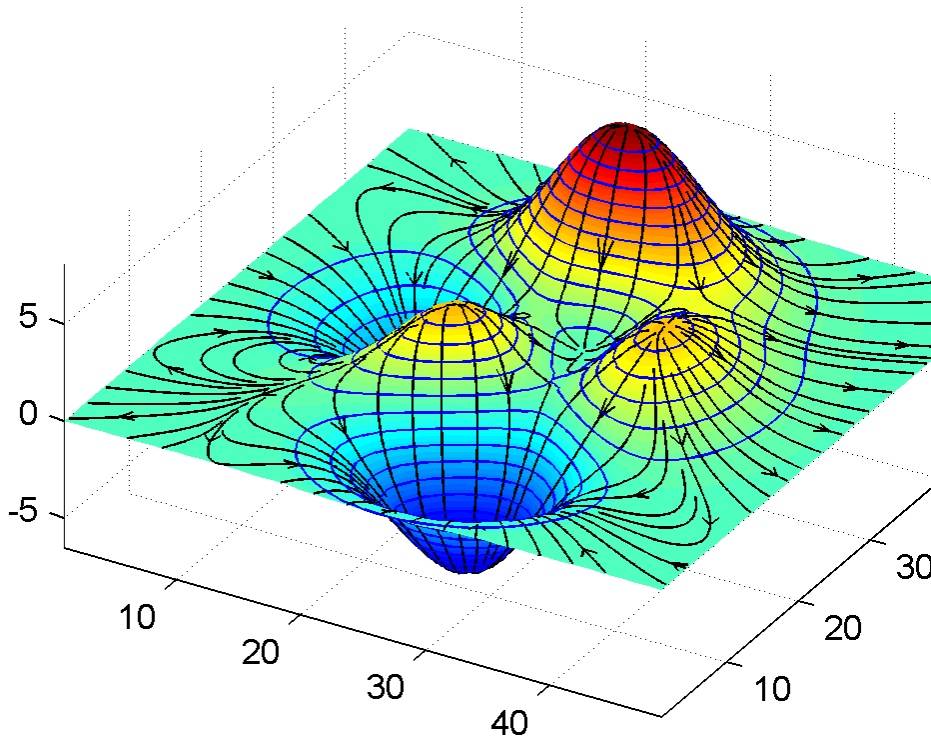
```
figure
z = peaks;
surf(z)
shading interp
```

## streamslice

---

```
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](#)

# streamtube

---

## 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.

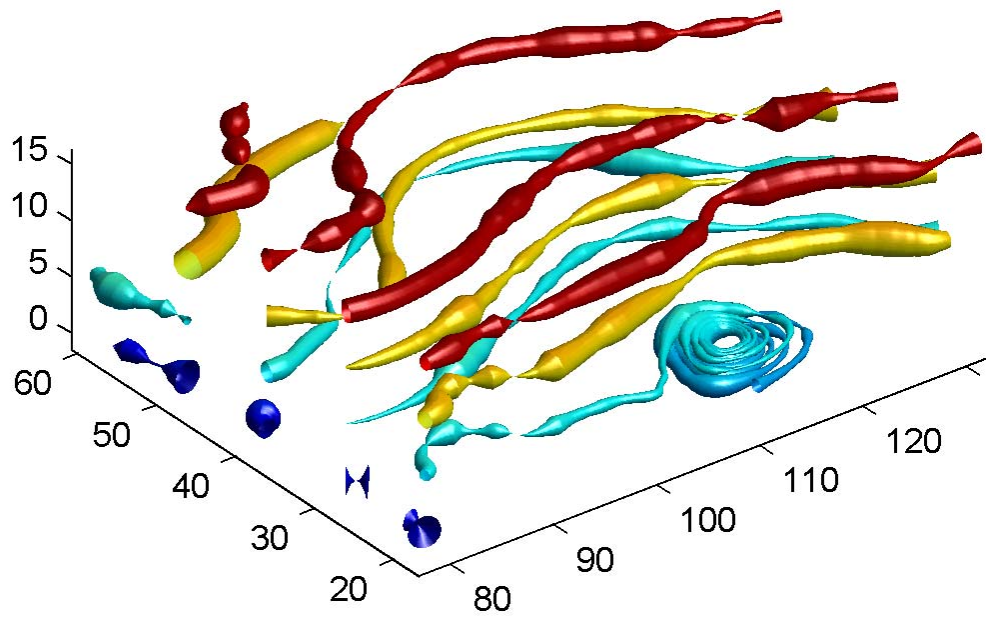
## Examples

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
```

# streamtube

```
shading interp;  
camlight; lighting gouraud
```



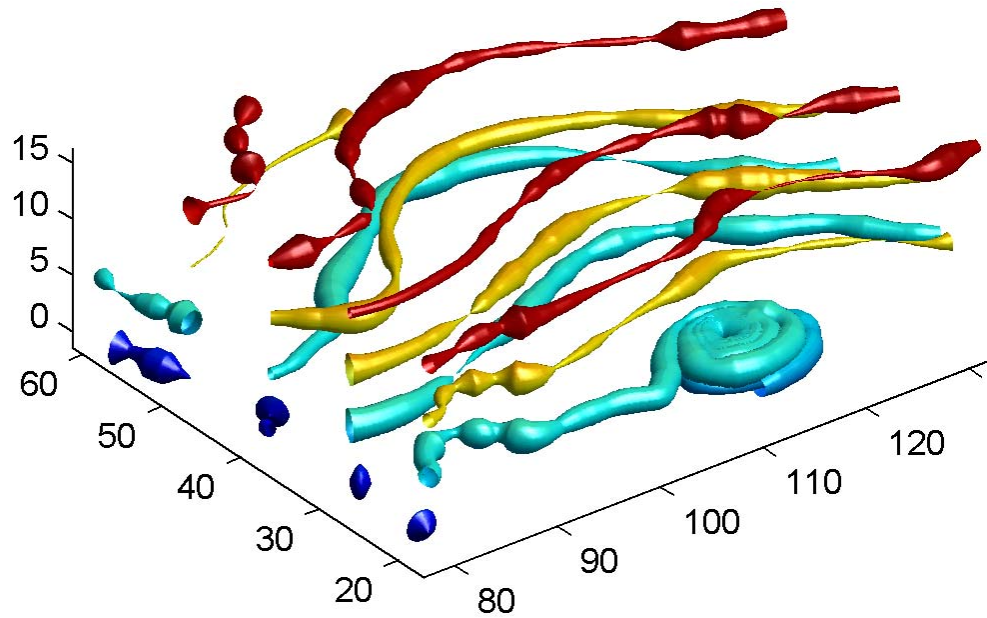
This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
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
```

# streamtube

---



## See Also

[divergence](#) | [streamribbon](#) | [streamline](#) | [stream3](#) | [meshgrid](#)  
| [stream3](#)

## How To

- [Displaying Divergence with Stream Tubes](#)
- [Specifying Starting Points for Stream Plots](#)

**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 than 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
```

# strfind

---

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` | `regexpr`  
| `regexprep` | `strrep`



**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.

# strings

---

- 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”

**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))
```

# strmatch

---

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`

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Compare first <i>n</i> characters of strings (case sensitive)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>          | TF = strncmp(string,string,n)<br>TF = strncmp(string,cellstr,n)<br>TF = strncmp(cellstr,cellstr,n)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b>     | <p>TF = strncmp(string,string,n) compares the first <i>n</i> characters of two strings for equality. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.</p> <p>TF = strncmp(string,cellstr,n) compares the first <i>n</i> characters of a string with the first <i>n</i> characters of each element of a cell array of strings. The function returns a logical array the same size as the <i>cellstr</i> input in which logical 1 represents equality. The order of the first two input arguments is not important.</p> <p>TF = strncmp(cellstr,cellstr,n) compares each element of one cell array of strings with the same element of the other. <i>strncmp</i> attempts to match only the first <i>n</i> characters of these strings. The function returns a logical array the same size as either input array.</p> |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• The <i>strncmp</i> function is intended for comparison of character data. When used to compare numeric data, it returns logical 0.</li><li>• Use <i>strncmpi</i> for case-insensitive string comparisons.</li><li>• Any leading and trailing blanks in either of the strings are explicitly included in the comparison.</li><li>• The value returned by <i>strncmp</i> is not the same as the C language convention.</li><li>• <i>strncmp</i> supports international character sets.</li></ul>                                                                                                                                                                                                                                                                                                                       |
| <b>Input Arguments</b> | <p><b>string</b><br/>Single character string or <i>n</i>-by-1 array of strings.</p> <p><b>cellstr</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

# strncmp

---

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 = ['AAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...  
        'DDDDDDDDDD'; 'EEEEEEEEEEE']
```

```
str1 =
```

```
    AAAAAAAAAA
```

```
    BBBBBBBBBB
```

```
    CCCCCCCCCC
```

```
    DDDDDDDDDD
```

```
    EEEEEEEEEE
```

```
str2 = str1;
```

```
str2(4,3) = '-'
```

```
str2 =
```

```
    AAAAAAAAAA
```

# strncmp

---

```
BBBBBBBBBB  
CCCCCCCCC  
DD-DDDDDD  
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` | `regexp` |

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Compare first <i>n</i> characters of strings (case insensitive)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>          | <pre>TF = strncmpi(string,string,n) TF = strncmpi(string,cellstr,n) TF = strncmpi(cellstr,cellstr,n)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>     | <p>TF = strncmpi(string,string,n) compares the first <i>n</i> 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.</p> <p>TF = strncmpi(string,cellstr,n) compares the first <i>n</i> characters of a string with the first <i>n</i> 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.</p> <p>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 <i>n</i> characters of these strings. The function returns a logical array the same size as either input array.</p> |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• The strncmpi function is intended for comparison of character data. When used to compare numeric data, it returns logical 0.</li><li>• Use strcmp for case-sensitive string comparisons.</li><li>• Any leading and trailing blanks in either of the strings are explicitly included in the comparison.</li><li>• The value returned by strncmpi is not the same as the C language convention.</li><li>• strncmpi supports international character sets.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Input Arguments</b> | <p><b>string</b></p> <p>A single character string or <i>n</i>-by-1 array of strings.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

# strncmpi

---

## **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 2-4362 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 2-4363 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 2-4363 and “Example 5” on page 2-4364 below.

The table [Formats for strread](#) on page 2-4359 lists the valid format specifiers. More information on using formats is available under “Formats” on page 2-4362 in the “Tips” on page 2-4361 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 2-4363 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 2-4360 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 2-4365 below.

**Formats for strread**

| <b>Format</b>                     | <b>Action</b>                                                                                                                                                                                    | <b>Output</b>         |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| Literals<br>(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 2-4365 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:<br>\b Backspace<br>\n New line<br>\r Carriage return<br>\t Horizontal tab<br>\\ Backslash<br>%% Percent sign<br>' ' 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 2-4364 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 =
```

# strread

---

```
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%" cellpadding=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:'
```

# strread

---

```
    '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](#)

# strtok

---

**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](#)

# strtrim

---

**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.

# struct

---

`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'
```

# struct

---

## 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”

**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

# structfun

---

**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, 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`.

### **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  $A_1, \dots, A_n$ , 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  $A_1, \dots, A_n$ , 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:
 

|                         |                                                                                             |
|-------------------------|---------------------------------------------------------------------------------------------|
| <code>identifier</code> | Error identifier.                                                                           |
| <code>message</code>    | Error message text.                                                                         |
| <code>index</code>      | 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

### $A_1, \dots, A_n$

Arrays that collect the  $n$  outputs from function `func`.

If `UniformOutput` is `true` (the default):

# structfun

---

- 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”

# strvcat

---

**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
```

# sub2ind

---

**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 equivalent 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 equivalent 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:

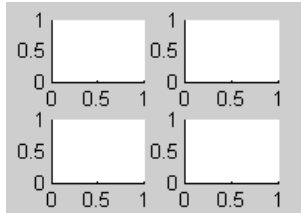
```
[A(3,3,2), A(2,4,1), A(3,1,2), A(1,3,2), A(2,4,1)]
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576
```

```
A([21, 11, 15, 19, 11])
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576
```

## See Also

`ind2sub` | `find` | `size`

**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 pth 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

---

`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`.)

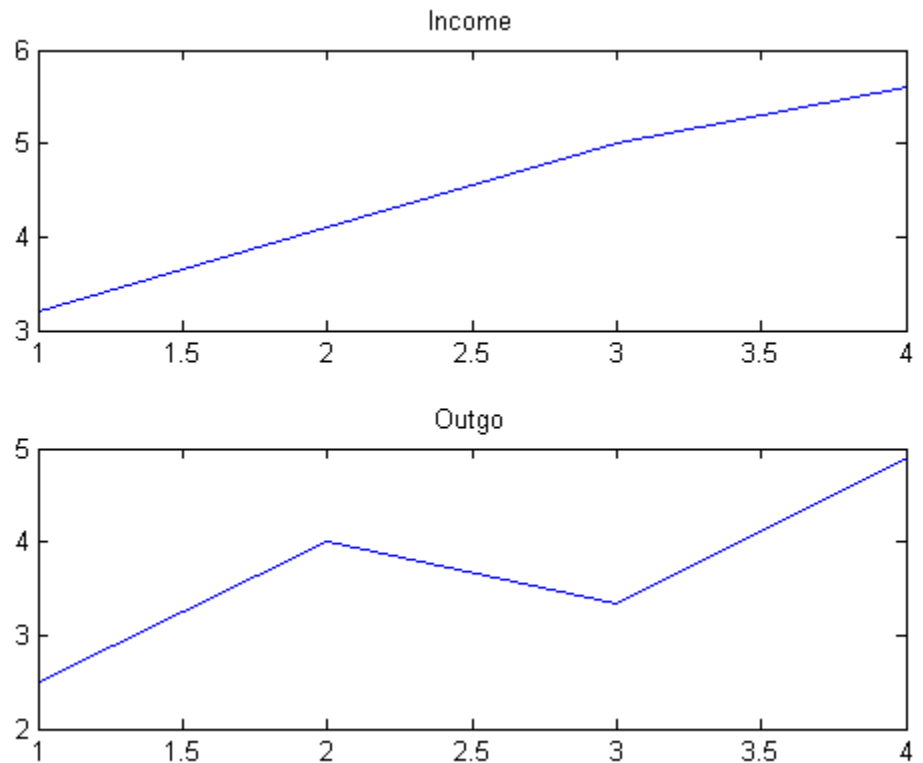
## Examples

### 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')
```

# subplot

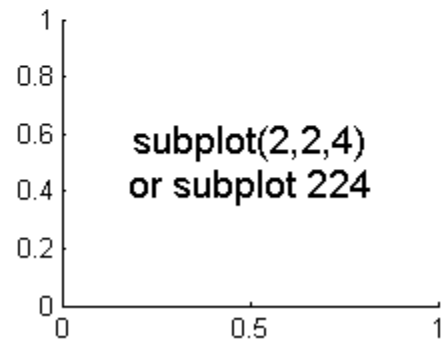
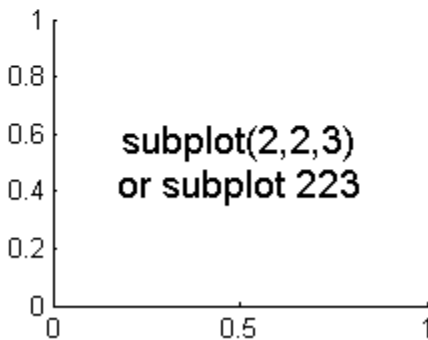
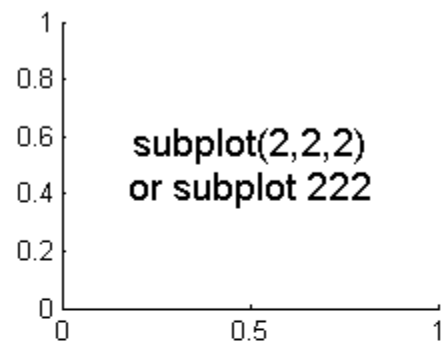
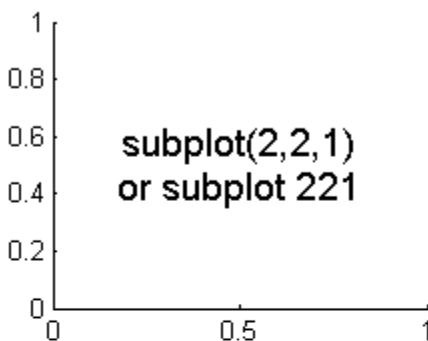


## Subplots in Quadrants

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'},...
```

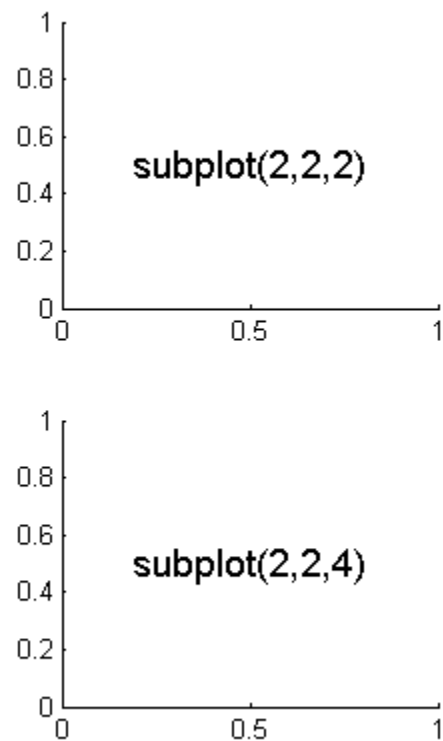
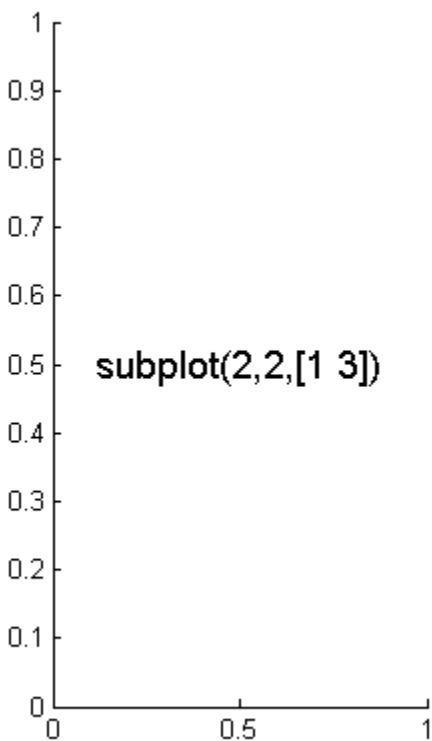
```
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,3)
text(.5,.5,{'subplot(2,2,3)';'or subplot 223'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,4)
text(.5,.5,{'subplot(2,2,4)';'or subplot 224'},...
    'FontSize',14,'HorizontalAlignment','center')
```



## 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')
```



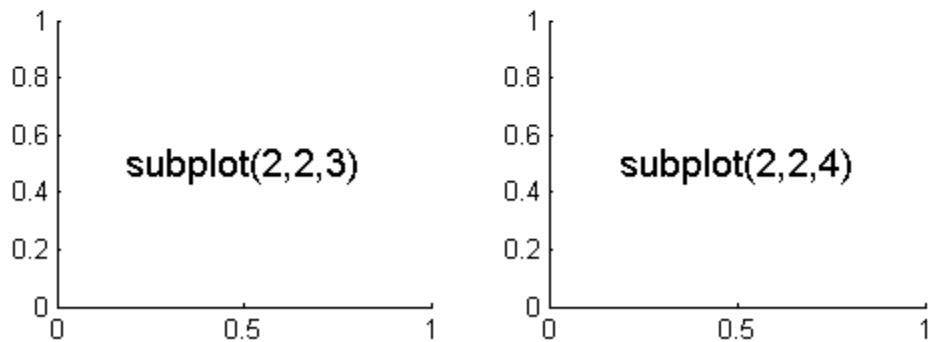
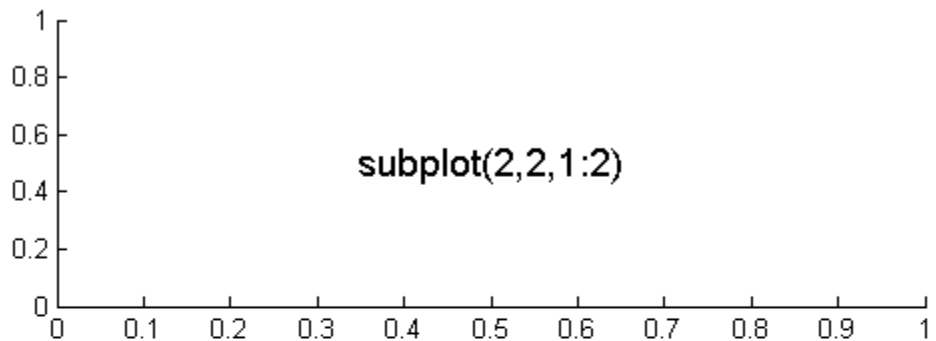
You can also use the colon operator to specify multiple locations if they are in sequence.

```
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

---

```
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',[]);
```

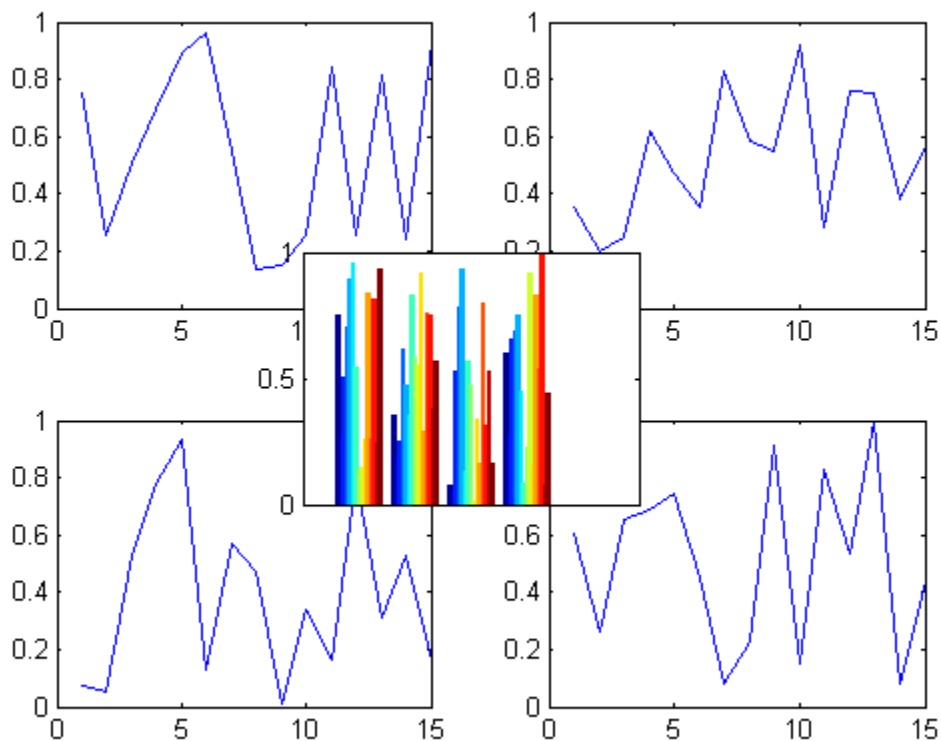
If the subplots are stacked, you can then display tick labels for the bottom axis.

### 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',[])
```

# subplot



## 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](#)”



|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Subscripted assignment                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>          | <code>A = subsasgn(A, S, B)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Description</b>     | <p><code>A = subsasgn(A, S, B)</code> is called by MATLAB for the syntax <code>A(i) = B</code>, <code>A{i} = B</code>, or <code>A.i = B</code> when <code>A</code> is an object.</p> <p>MATLAB uses the built-in <code>subsasgn</code> function to interpret indexed assignment statements. Modify the indexed assignment behavior of classes by overloading <code>subsasgn</code> in the class.</p> <p>If <code>A</code> is a fundamental class (see “Fundamental MATLAB Classes”), then an indexed reference to <code>A</code> calls the built-in <code>subsasgn</code> function. It does not call a <code>subsasgn</code> method that you have overloaded for that class. Therefore, if <code>A</code> is an array of class <code>double</code>, and there is an <code>@double/subsasgn</code> method on your MATLAB path, the statement <code>A(I) = B</code> calls the MATLAB built-in <code>subsasgn</code> function.</p> |
| <b>Tips</b>            | <p>Within a class’s own methods, MATLAB calls the built-in <code>subsasgn</code>, not the class defined <code>subsasgn</code>. This behavior enables to use the default <code>subsasgn</code> behavior when defining specialized indexing for your class. See “<code>subsref</code> and <code>subsasgn</code> Within Class Methods — Built-In Called” for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Input Arguments</b> | <p><b>A</b><br/>Object</p> <p><b>S</b><br/>struct array with two fields, <code>type</code> and <code>subs</code>.</p> <ul style="list-style-type: none"><li><code>type</code> is a string containing <code>'()''</code>, <code>'{}''</code>, or <code>'.'</code>, where <code>'()''</code> specifies integer subscripts, <code>'{}''</code> specifies cell array subscripts, and <code>'.'</code> specifies subscripted structure fields.</li><li><code>subs</code> is a cell array or string containing the actual subscripts.</li></ul> <p><b>B</b></p>                                                                                                                                                                                                                                                                                                                                                                       |

# subsasgn

---

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  $A(J,K,\dots) = B(M,N,\dots)$ , 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”

# subsindex

---

**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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Angle between two subspaces                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <code>theta = subspace(A,B)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b> | <code>theta = subspace(A,B)</code> 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 <code>acos(abs(A'*B))</code> .                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Tips</b>        | 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, <code>subspace(A,B)</code> gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.                                                                                                                                                                                                                                     |
| <b>Examples</b>    | <p>Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.</p> <pre>H = hadamard(8); A = H(:,2:4); B = H(:,5:8);</pre> <p>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.</p> <pre>theta = subspace(A,B) theta =     1.5708</pre> <p>That A and B are orthogonal is shown by the fact that theta is equal to <math>\pi/2</math>.</p> <pre>theta - pi/2 ans =     0</pre> |

# subsref

---

**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:

|                              |                               |                                  |
|------------------------------|-------------------------------|----------------------------------|
| <code>S(1).type='()'</code>  | <code>S(2).type='.'</code>    | <code>S(3).type='()'</code>      |
| <code>S(1).subs={1,2}</code> | <code>S(2).subs='name'</code> | <code>S(3).subs={[3 4 5]}</code> |

## See Also

`numel` | `subsasgn` | `substruct`

## Tutorials

- “Indexed Reference and Assignment”

# substruct

---

**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”

# subvolume

---

**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 NaNs 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.

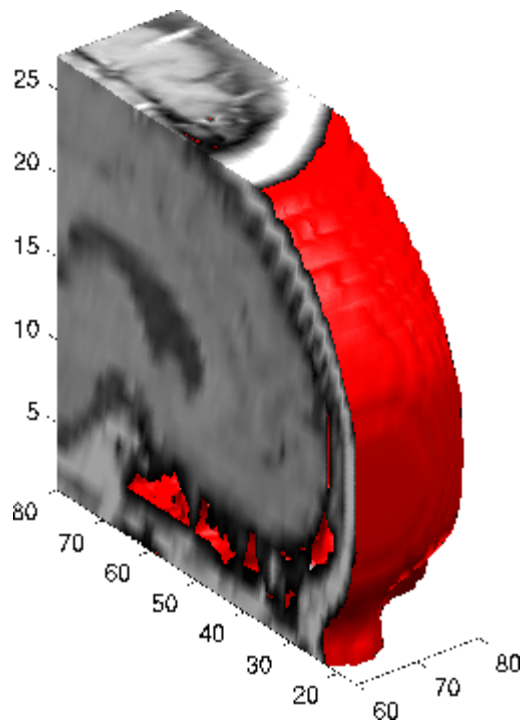
## Examples

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
```



# subvolume

---

## **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)`.

# sum

---

## 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`

# superclasses

---

**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”



|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Establish superior class relationship                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Syntax</b>      | <code>superiorto('class1', 'class2', ...)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Description</b> | <p><code>superiorto('class1', 'class2', ...)</code> establishes that the class invoking this function in its constructor has higher precedence than the classes in the argument list.</p> <p>The <code>superiorto</code> function establishes a precedence that determines which object method MATLAB calls. Use this function only from a constructor that calls the <code>class</code> function to create an object. For classes defined with <code>classdef</code> statements, see “Class Precedence”.</p>                                                                                                                                                                                                                                                  |
| <b>Examples</b>    | <p>Show function dispatching:</p> <p>a is an object of class <code>class_a</code>, b is an object of class <code>class_b</code>, and c is an object of class <code>class_c</code>. The constructor method for <code>class_c</code> contains the statement <code>superiorto('class_a')</code>. Then, either of the following two statements:</p> <pre>e = fun(a,c);<br/>e = fun(c,a);</pre> <p>invokes <code>class_c/fun</code>.</p> <p>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 <code>fun(b,c)</code> calls <code>class_b/fun</code>, while <code>fun(c,b)</code> calls <code>class_c/fun</code>.</p> |
| <b>See Also</b>    | <code>inferiorto</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

# support

---

**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.

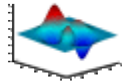
This Web page contains resources including

- 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  $[m,n] = \text{size}(Z)$ . 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,  $\text{length}(X) = n$  and  $\text{length}(Y) = m$ , where  $[m,n] = \text{size}(Z)$ . 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,  $1 \leq i \leq m$  and  $1 \leq j \leq n$ . 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.

$$\begin{array}{c} i-1, j \\ | \\ i, j-1 - i, j - i, j+1 \\ | \\ i+1, j \end{array}$$

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:

$$\begin{array}{ccc} (i, j) & - & (i, j+1) \\ | & C(i, j) & | \\ (i+1, j) & - & (i+1, j+1) \end{array}$$

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.

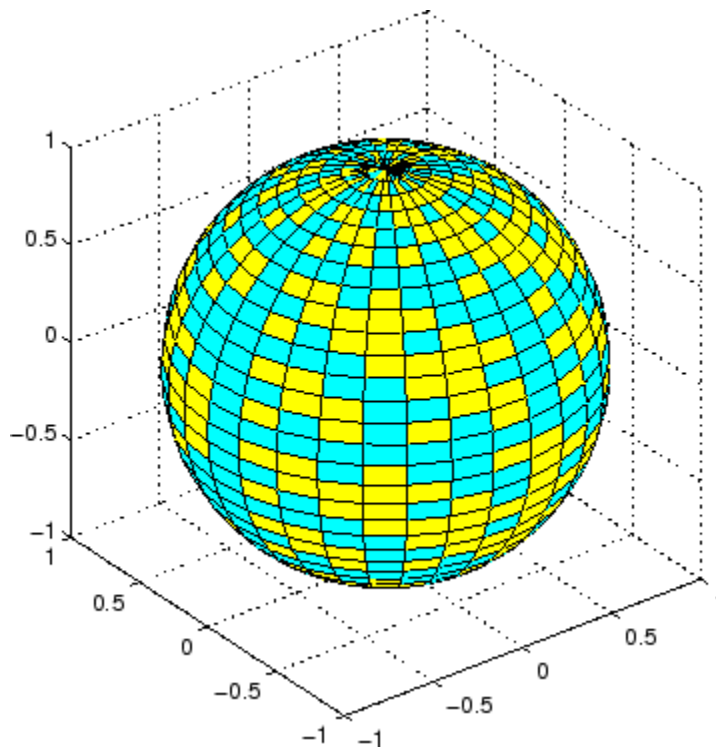
## Examples

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
```

# surf

---



## See Also

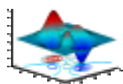
[surfc](#) | [axis](#) | [colormap](#) | [griddata](#) | [imagesc](#) | [mesh](#) | [meshgrid](#) | [pcolor](#) | [shading](#) | [trisurf](#) | [TriScatteredInterp](#) | [view](#) | [Surfaceplot Properties](#)

## How To

- “Representing Data as a Surface”
- “Surface Plots of Nonuniformly Sampled Data”
- Coloring Mesh and Surface Plots

**Purpose**

Contour plot under a 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 contour plot under the three-dimensional shaded surface from the  $z$  components in matrix  $Z$ , using  $x = 1:n$  and  $y = 1:m$ , where  $[m,n] = \text{size}(Z)$ . 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,  $\text{length}(X) = n$  and  $\text{length}(Y) = m$ , where  $[m,n] = \text{size}(Z)$ . 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,  $1 \leq i \leq m$  and  $1 \leq j \leq n$ . 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.

$$\begin{array}{c} i-1, j \\ | \\ i, j-1 - i, j - i, j+1 \\ | \\ i+1, j \end{array}$$

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:

$$\begin{array}{ccc} (i, j) & - & (i, j+1) \\ | & C(i, j) & | \\ (i+1, j) & - & (i+1, j+1) \end{array}$$

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 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.

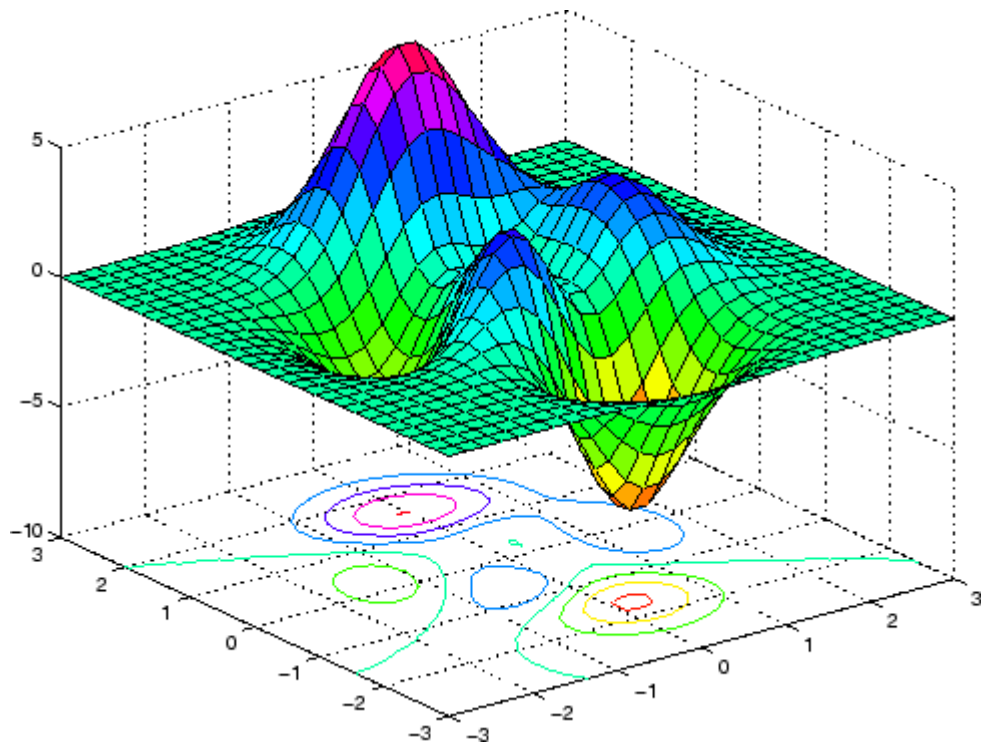
## Examples

Display a surface plot and a contour plot of the peaks surface.

```
[X,Y,Z] = peaks(30);
surf(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
```

# surf

---



## 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](#)

**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

# surf2patch

---

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`

**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

# surface

---

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

---

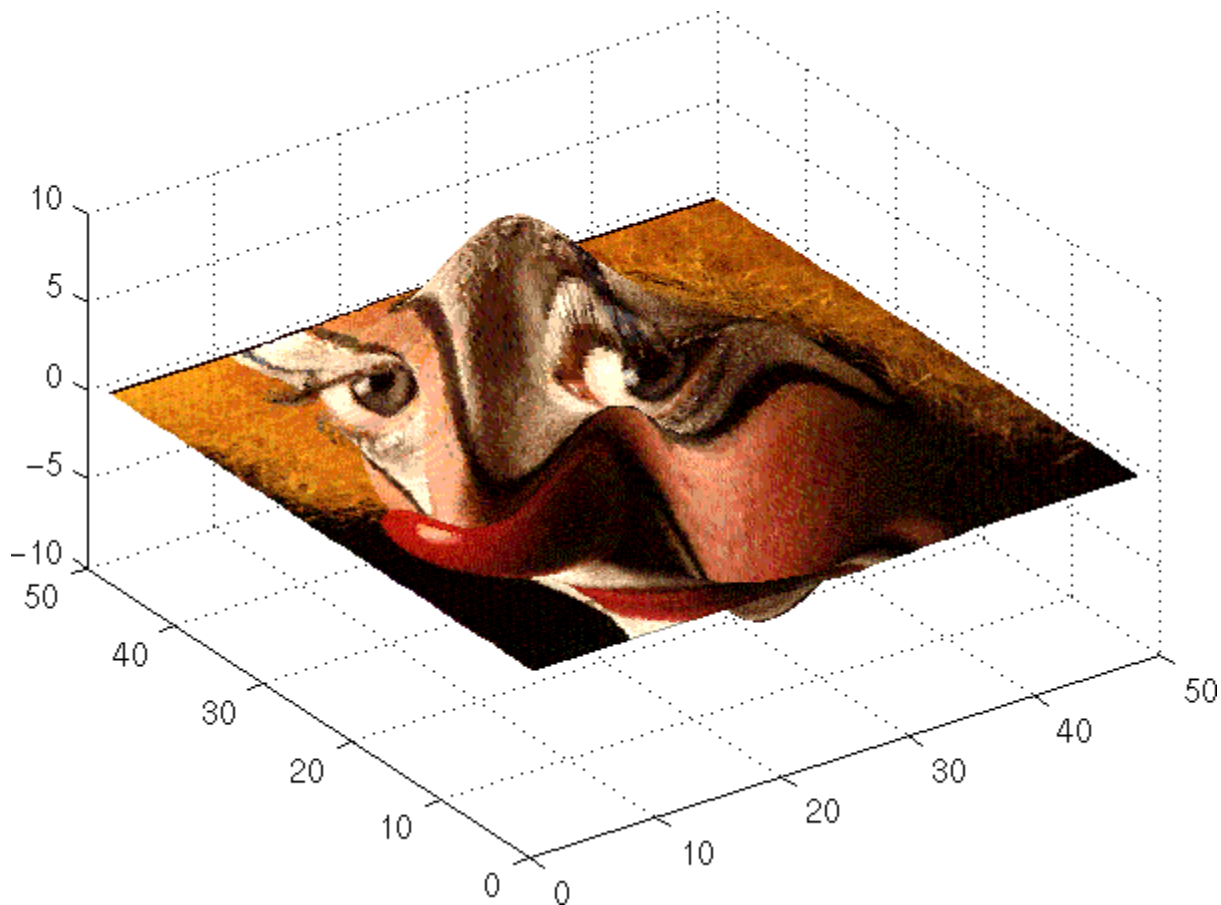
surface property values after creating them using the set and get commands.

## 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)
```

## surface



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](#)

# 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  $\geq 0$  and  $\leq 1$

*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.

# Surface Properties

---

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:

| <b>IconDisplayStyle Value</b> | <b>Purpose</b>                                          |
|-------------------------------|---------------------------------------------------------|
| 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*

# Surface Properties

---

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
```

```
% evt - empty for this property
sel_typ = get(gcf,'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.

# Surface Properties

---

## 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.



## 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)

# Surface Properties

---

*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  $\geq 0$  and  $\leq 1$

*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.

# Surface Properties

---

- `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.

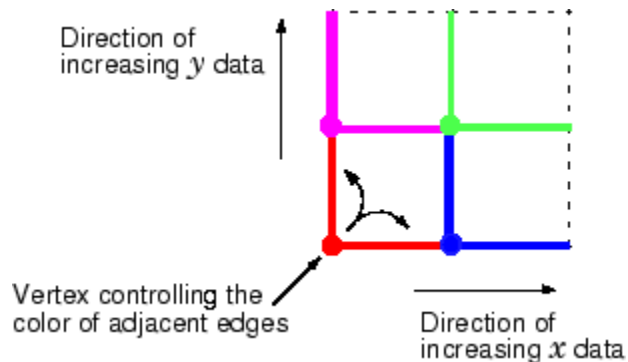
Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

## 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.

# Surface Properties

---

- `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.

# Surface Properties

---

- `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

# Surface Properties

---

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.

# Surface Properties

**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' '1' | 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.

# Surface Properties

---

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 of the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

**SpecularExponent**  
scalar  $\geq 1$

*Harshness of specular reflection.* Controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**SpecularStrength**  
scalar  $\geq 0$  and  $\leq 1$

*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)

# Surface Properties

---

*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`

# Surfaceplot Properties

---

## 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  $\geq 0$  and  $\leq 1$

*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)

# Surfaceplot Properties

---

*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.

# Surfaceplot Properties

---

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

# Surfaceplot Properties

---

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.

# Surfaceplot Properties

---

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  $\geq 0$  and  $\leq 1$

*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.

# Surfaceplot Properties

---

- `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`.

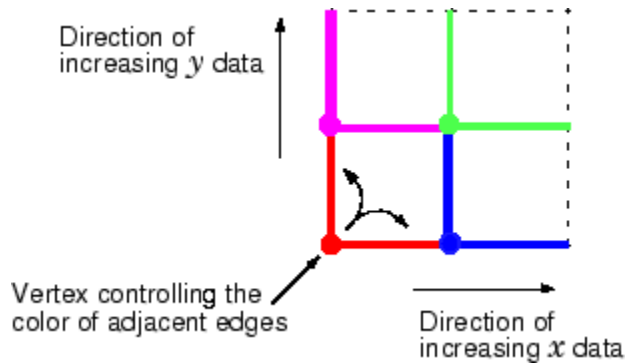
Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

## `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`

# Surfaceplot Properties

---

*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.

# Surfaceplot Properties

---

- `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`.

# Surfaceplot Properties

---

## 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

# Surfaceplot Properties

---

*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

# Surfaceplot Properties

---

*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.

# Surfaceplot Properties

---

**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 of the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

**SpecularExponent**  
scalar  $\geq 1$

*Harshness of specular reflection.* Controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**SpecularStrength**  
scalar  $\geq 0$  and  $\leq 1$

*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

# Surfaceplot Properties

---

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

# Surfaceplot Properties

---

*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.

---

# surf1

---

## Purpose

Surface plot with colormap-based lighting



## Syntax

```
surf1(Z)
surf1(...,'light')
surf1(...,s)
surf1(X,Y,Z,s,k)
h = surf1(...)
```

## Description

The `surf1` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surf1(Z)` and `surf1(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.

`surf1(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surf1(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surf1(...,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  $45^\circ$  counterclockwise from the current view direction.

`surf1(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 = surf1(...)` returns a handle to a surface graphics object.

**Tips**

surf1 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 surf1(X',Y',Z'). Because of the way surface normal vectors are computed, surf1 requires matrices that are at least 3-by-3.

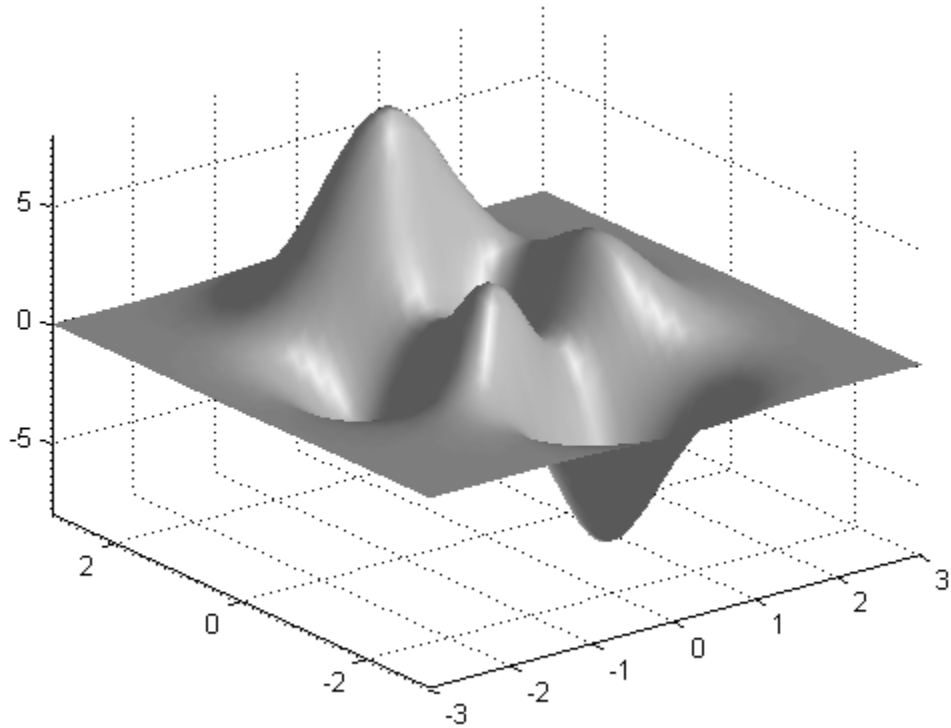
**Examples**

View peaks using colormap-based lighting.

```
[x,y] = meshgrid(-3:1/8:3);  
z = peaks(x,y);  
surf1(x,y,z);  
shading interp  
colormap(gray);  
axis([-3,3,-3,3,-8,8])
```

# surf1

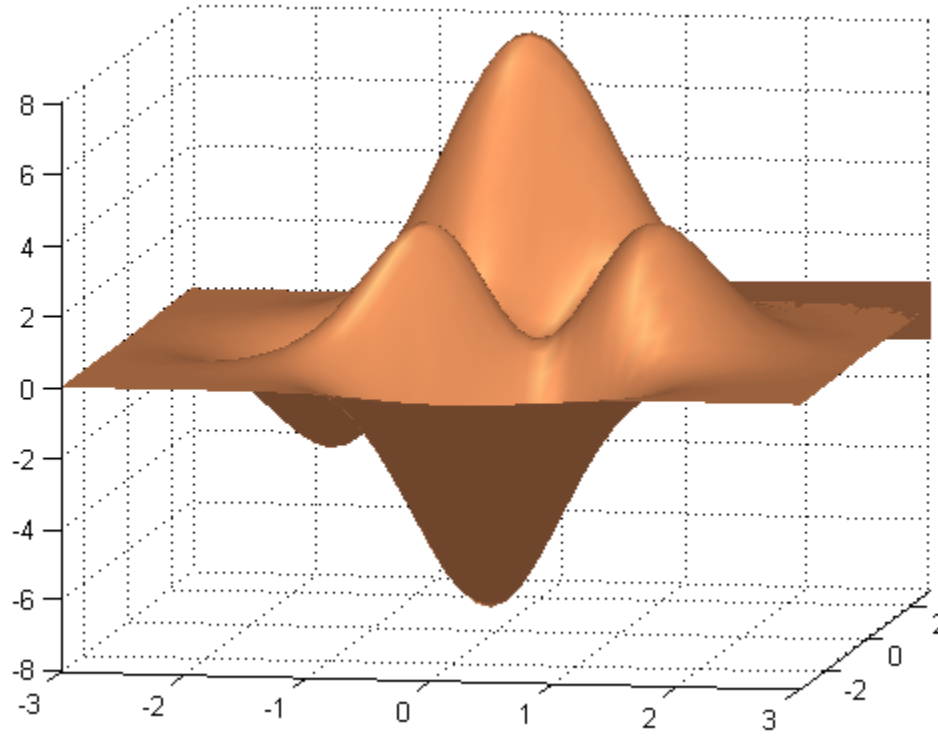
---



To plot a lighted surface from a view direction other than the default,

```
view([10 10])
grid on
hold on
surf1(peaks)
shading interp
colormap copper
hold off
```





**See Also**

[colormap](#) | [shading](#) | [light](#)

# surfnorm

---

**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.
- `surf1` 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

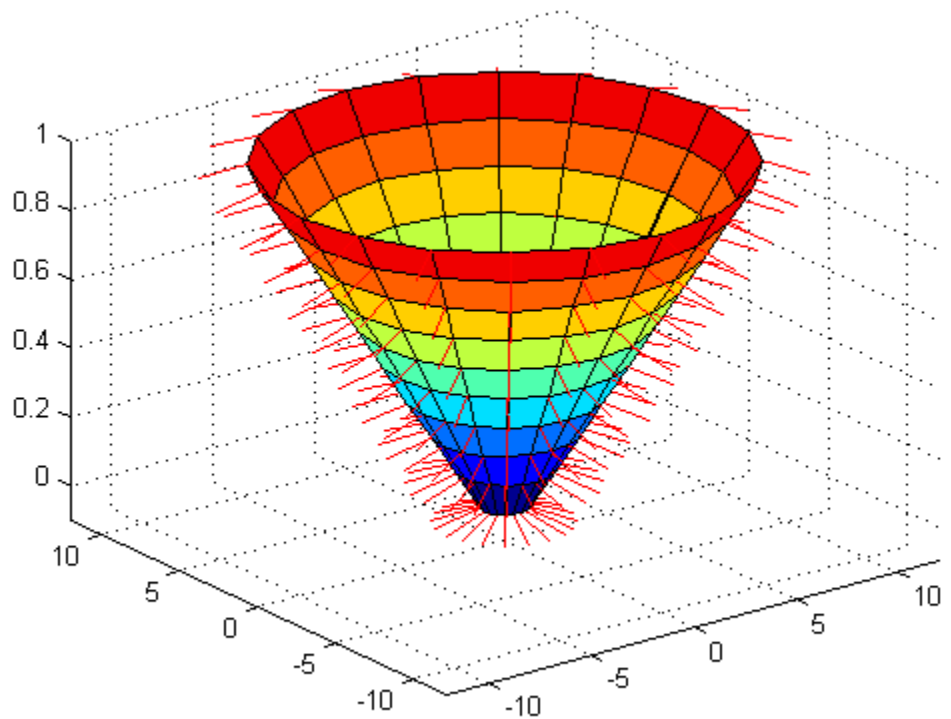
# surfnorm

---

## 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])
```



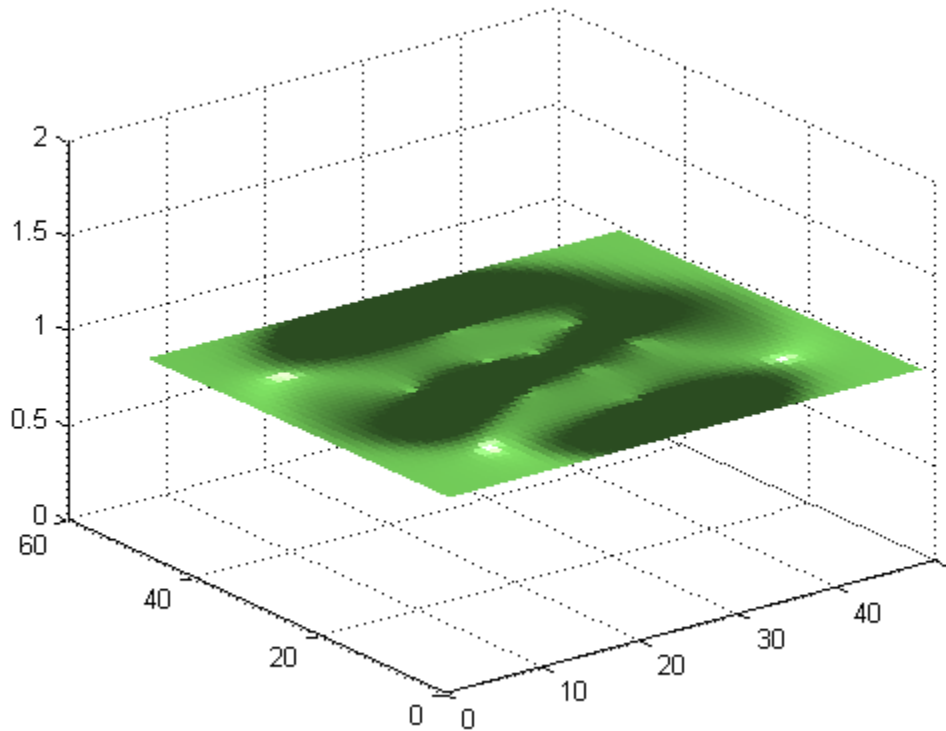
---

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 to the image  
shading interp % remove edge definition from the image  
camlight
```



# surfnorm

---

## 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` | `surf1`

**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  $m \geq n$ , 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
```

# svd

---

```
-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.
```



**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 |
|----------------------------|----------------------------------------------------------------------------|---------|
| <code>options.tol</code>   | Convergence tolerance:<br><code>norm(AV-US,1) &lt;= tol * norm(A,1)</code> | 1e-10   |
| <code>options.maxit</code> | Maximum number of iterations                                               | 300     |
| <code>options.disp</code>  | 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]`.

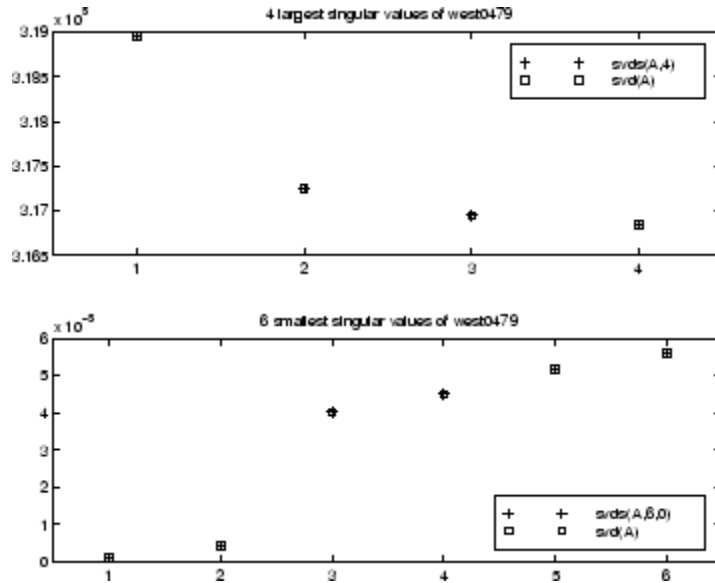
`svds(A,k,0)` uses `eigs` to find the `2k` smallest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`, and then selects the `k` positive eigenvalues and their eigenvectors.

## 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))
s1 = 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

# swapbytes

---

**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`:

## switch/case/otherwise

---

```
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.');
```

---

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
```



**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`.

|                       |                                                                                                                                                                           |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>stats(1)</code> | Number of dense or empty rows ignored by <code>symamd</code>                                                                                                              |
| <code>stats(2)</code> | Number of dense or empty columns ignored by <code>symamd</code>                                                                                                           |
| <code>stats(3)</code> | Number of garbage collections performed on the internal data structure used by <code>symamd</code> (roughly of size $8.4 * \text{nnz}(\text{tril}(S, -1)) + 9n$ integers) |
| <code>stats(4)</code> | 0 if the matrix is valid, or 1 if invalid                                                                                                                                 |
| <code>stats(5)</code> | Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists                                                                      |
| <code>stats(6)</code> | Last seen duplicate or out-of-order row index in the column index given by <code>stats(5)</code> , or 0 if no such row index exists                                       |
| <code>stats(7)</code> | 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:

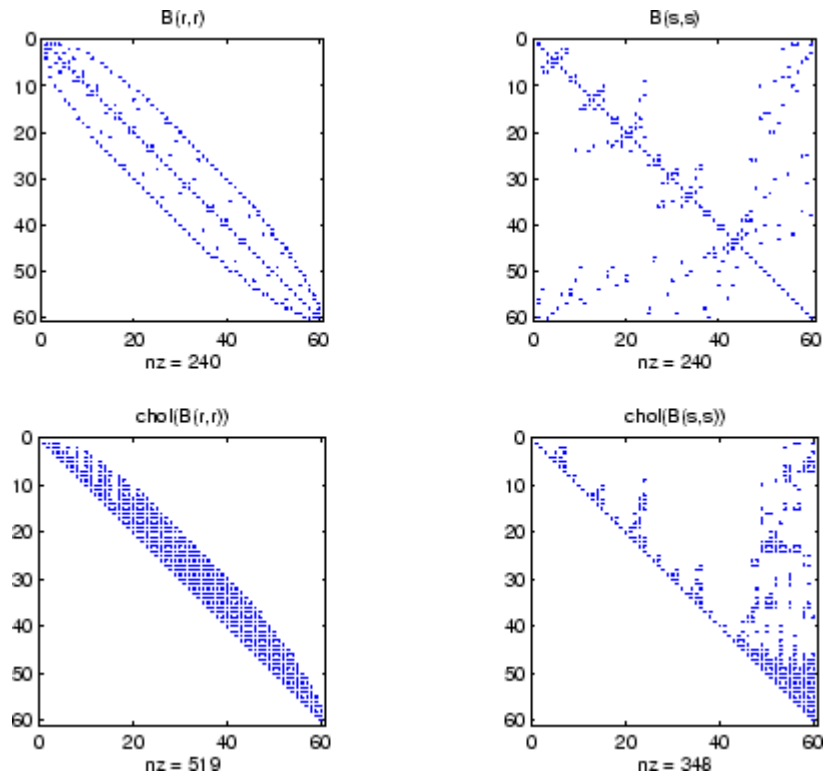
- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `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](mailto: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.

# **symamd**

---

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  $R = \text{chol}(A)$ . `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  $R = \text{chol}(A' * A)$  (without forming it explicitly).

`count = symbfact(A, 'row')` returns row counts of  $R = \text{chol}(A * A')$ .

`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 <code>chol(A)</code> for the symmetric case, <code>chol(A'*A)</code> for the 'col' case, or <code>chol(A*A')</code> for the 'row' case. |

# **symbfact**

---

`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`

**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  $\text{norm}(b-A*x)/\text{norm}(b)$  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  $\text{inv}(\text{sqrt}(M))*A*\text{inv}(\text{sqrt}(M))*y =$

# symmlq

---

$\text{inv}(\text{sqrt}(M)) * b$  for  $y$  and then return  $x = \text{in}(\text{sqrt}(M)) * y$ . If  $M$  is  $[]$  then `symmlq` applies no preconditioner.  $M$  can be a function handle `mfun` such that `mfun(x)` returns  $M \backslash x$ .

`symmlq(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If  $x_0$  is  $[]$ , then `symmlq` uses the default, an all-zero vector.

`[x,flag] = symmlq(A,b,...)` also returns a convergence flag.

| Flag | Convergence                                                                                                             |
|------|-------------------------------------------------------------------------------------------------------------------------|
| 0    | <code>symmlq</code> converged to the desired tolerance <code>tol</code> within <code>maxit</code> iterations.           |
| 1    | <code>symmlq</code> iterated <code>maxit</code> times but did not converge.                                             |
| 2    | Preconditioner $M$ was ill-conditioned.                                                                                 |
| 3    | <code>symmlq</code> stagnated. (Two consecutive iterates were the same.)                                                |
| 4    | One of the scalar quantities calculated during <code>symmlq</code> 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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, `relres`  $\leq$  `tol`.

`[x,flag,relres,iter] = symmlq(A,b,...)` also returns the iteration number at which  $x$  was computed, where  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including  $\text{norm}(b-A*x_0)$ .

`[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 reordering 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 `democs` 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

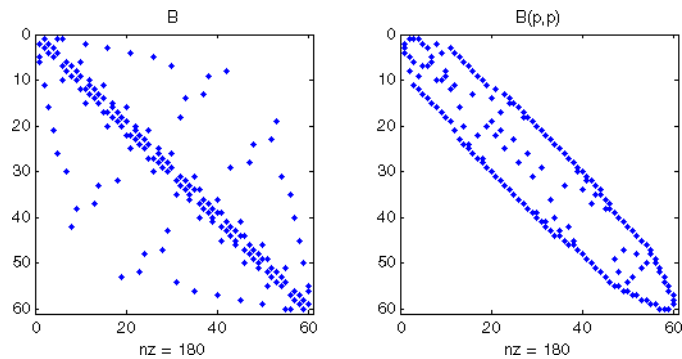
```
subplot(1,2,1),spy(B),title('B')
```

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`

# symvar

---

**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 2-4520.

### **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 2-4520.

## 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”

# system

---

**Purpose** Execute operating system command and return result

**Syntax**

```
system('command')  
[status, result] = system('command')  
[status,result] = system('command', '-echo')
```

**Description** `system('command')` calls the operating system to run the given command and directs the output to the MATLAB software. The command executes in a system shell, which might not be the shell from which you launched MATLAB.

`[status, result] = system('command')` returns completion status to the `status` variable and returns the result of the command to the `result` variable.

`[status,result] = system('command', '-echo')` also forces the output to the Command Window.

The ampersand, `&`, character has special meaning. For console programs this causes the console to open. Omitting this character causes console programs to run iconically. For GUI programs, appending this character causes the application to run in the background. MATLAB continues processing.

This function is interchangeable with the `dos` and `unix` functions.

---

**Note** Running `system` on a Microsoft Windows platform with a command that relies on the current folder fails when the current folder is specified using a UNC pathname because DOS does not support UNC pathnames. When this happens, MATLAB displays:

Error using `system`

DOS commands may not be executed when the current directory is a UNC path

To work around this limitation, change the folder to a mapped drive prior to running `system` or a function that calls `system`.

---

**Examples**

On a Windows system, display the current folder by accessing the operating system.

```
[status currdir] = system('cd')
status =
    0
currdir =
    D:\work\matlab\test
```

---

On a UNIX system, if you redirect `stdin` in the script and type at the keyboard while the script executes, MATLAB does not display the type-ahead characters.

```
[a,b] = system('cmd < /dev/null');
```

**See Also**

`computer` | `dos` | `perl` | `unix` | ! (exclamation point)

**How To**

- “Running External Commands, Scripts, and Programs”

# tan

**Purpose** Tangent of argument in radians

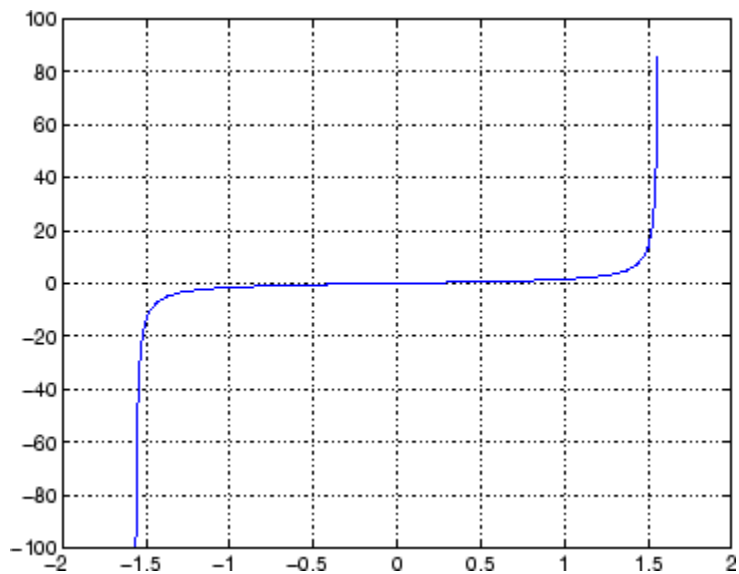
**Syntax**  $Y = \tan(X)$

**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  $\pi$ .

**See Also** `tand` | `tanh` | `atan` | `atan2` | `atand` | `atan2d`

|                         |                                                                                                                                                                                                                                                                                                                                                                        |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Tangent of argument in degrees                                                                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>           | $Y = \text{tand}(X)$                                                                                                                                                                                                                                                                                                                                                   |
| <b>Description</b>      | $Y = \text{tand}(X)$ is the tangent of the elements of $X$ . The result, $Y$ , is expressed in degrees.                                                                                                                                                                                                                                                                |
| <b>Input Arguments</b>  | <p><b>X - Angle in degrees</b><br/>scalar value   vector   matrix   N-D array</p> <p>Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The <code>tand</code> operation is element-wise when <math>X</math> is nonscalar.</p> <p><b>Data Types</b><br/>single   double</p> <p><b>Complex Number Support:</b> Yes</p> |
| <b>Output Arguments</b> | <p><b>Y - Tangent of angle</b><br/>scalar value   vector   matrix   N-D array</p> <p>Tangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as <math>X</math>.</p>                                                                                                                                        |
| <b>Examples</b>         | <p><b>Tangent of 90 degrees compared to tangent of <math>\pi/2</math> radians</b></p> <p><code>tand(90)</code> is infinite, whereas <code>tan(pi/2)</code> is large but finite.</p> <pre>tand(90)  ans =       Inf  tan(pi/2)  ans =   1.6331e+16</pre>                                                                                                                |

# tand

---

## 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](#)

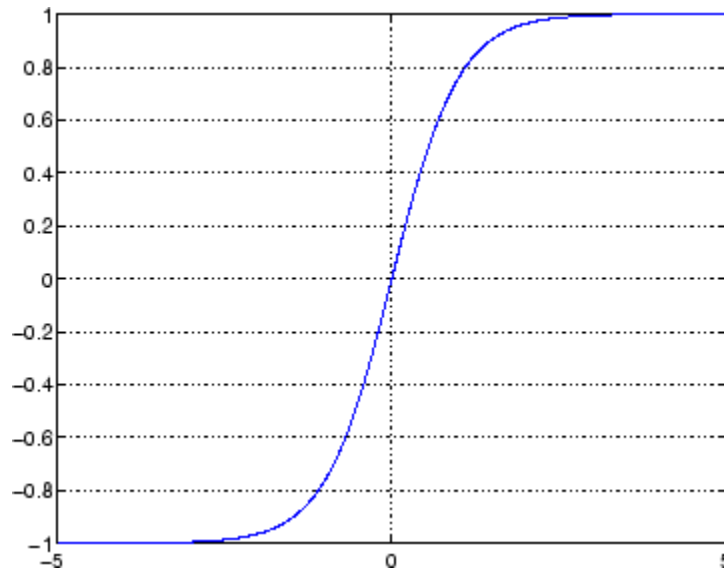
**Purpose** Hyperbolic tangent

**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  $-5 \leq x \leq 5$ .

```
x = -5:0.01:5;  
plot(x,tanh(x)), grid on
```



**See Also** [atan](#) | [atan2](#) | [tan](#) | [atanh](#) | [sinh](#) | [cosh](#)

# tar

---

## 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

# tempdir

---

|                    |                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Name of system's temporary folder                                                                                                               |
| <b>Syntax</b>      | <code>tmp_folder = tempdir</code>                                                                                                               |
| <b>Description</b> | <code>tmp_folder = tempdir</code> returns the name of the system's temporary folder, if one exists. This function does not create a new folder. |
| <b>See Also</b>    | <code>delete</code>   <code>recycle</code>   <code>tempname</code>                                                                              |
| <b>How To</b>      | <ul style="list-style-type: none"><li>• “Creating Temporary Files”</li></ul>                                                                    |

**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”

# tetramesh

---

**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.

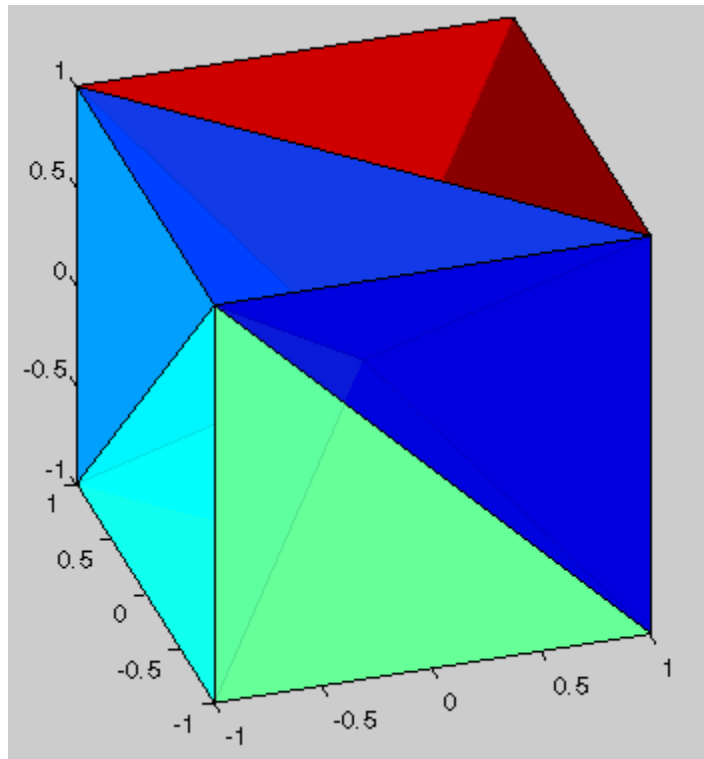
`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of  $h$  is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch 'Visible' property 'on' or 'off'.

`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 that form the corresponding simplex.

```
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 = DelaunayTri(x,y,z);
Tes = dt(:,:,);
X = [x(:) y(:) z(:)];
tetramesh(Tes,X);
camorbit(20,0)
```



You can also plot the Delaunay triangulation directly.

# tetramesh

---

```
close(gcf);  
tetramesh(dt);
```

## See Also

[trimesh](#) | [trisurf](#) | [patch](#) | [delaunayn](#) | [TriRep](#) | [TriRep.freeBoundary](#)

**Purpose**

Produce TeX format from character string

**Syntax**

```
texlabel(f)
texlabel(f, 'literal')
```

**Description**

`texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. It processes Greek variable names (e.g., `lambda`, `delta`, etc.) into a string that is displayed as actual Greek letters.

`texlabel(f, 'literal')` prints Greek variable names as literals.

If the string is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

**Examples**

You can use `texlabel` as an argument to the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` commands. For example,

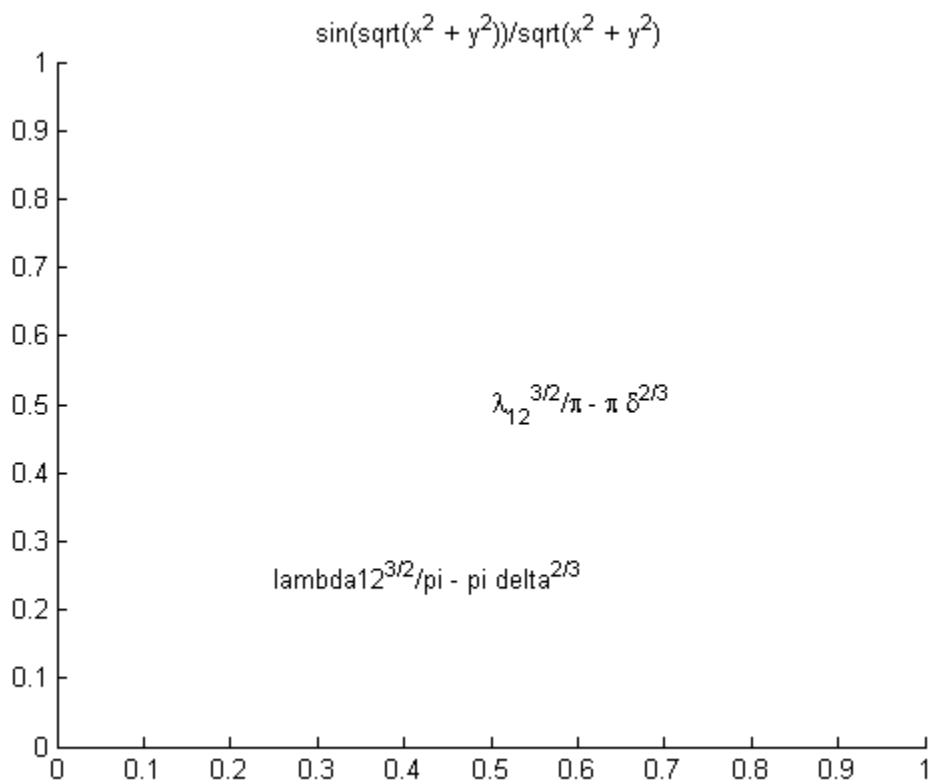
```
figure
title(texlabel('sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'))
```

By default, `texlabel` translates Greek variable names to the equivalent Greek letter. You can select literal interpretation by including the `literal` argument. For example, compare these two commands.

```
text(.5,.5,...
      texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)'))
text(.25,.25,...
      texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)', 'literal'))
```

# textlabel

---



## See Also

text | title | xlabel | ylabel | zlabel | String



**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 2-4539). 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')
```

---

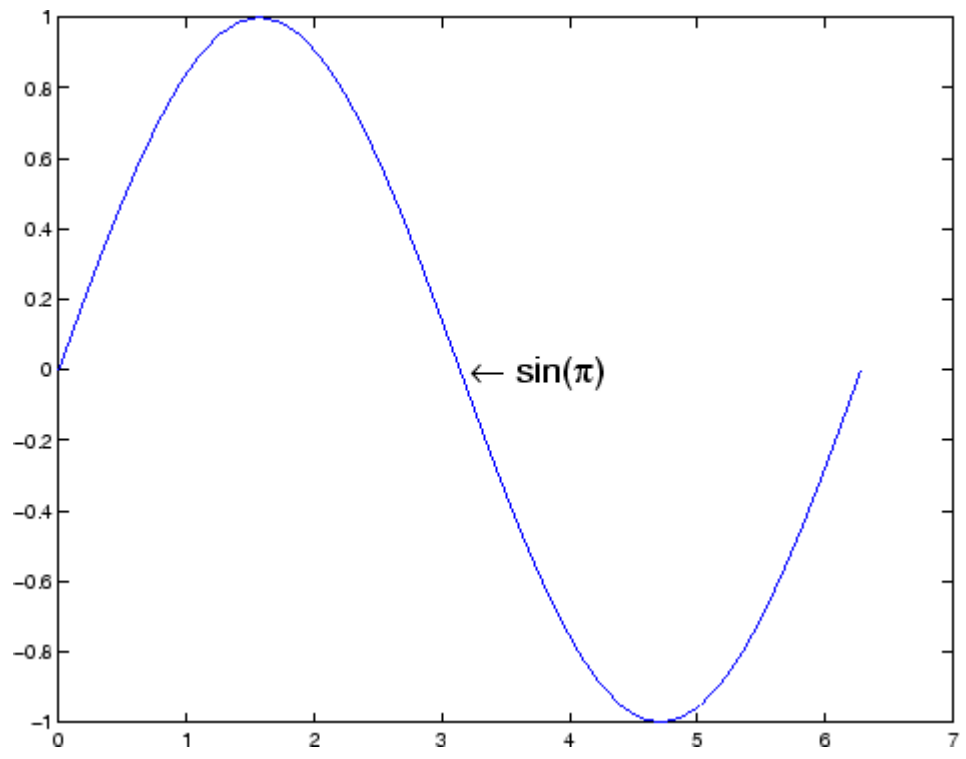
You can specify other properties only as property name/property value pairs. For a description of each property, see `Text Properties`. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify 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  $\sin(\pi)$  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](#)

# 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.

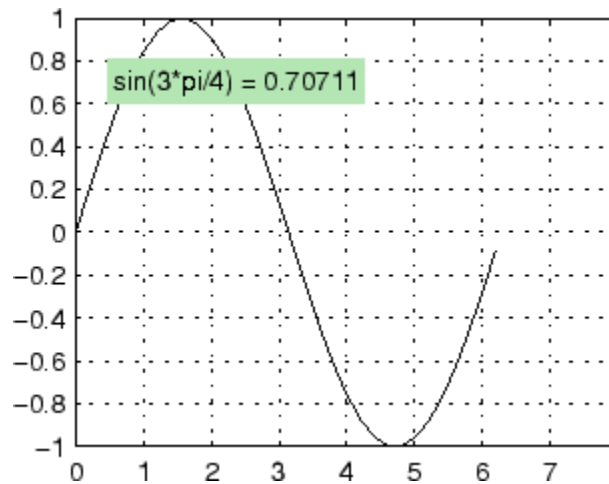
`BackgroundColor`  
`ColorSpec | {none}`

*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]);
```

# Text Properties

---



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.

# Text Properties

---

## 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(gcf, '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)
```

# Text Properties

---

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(gcf,'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
```

```
% evt - 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.

# Text Properties

---

- 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.

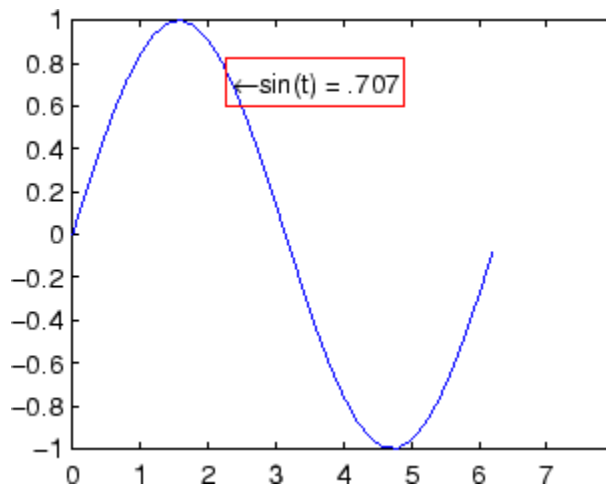
See “Controlling Legends” for more information and examples.

## 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),...  
'\leftarrow sin(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 this property is on, MATLAB places an insert cursor at the end of the text string and enables editing. To apply the new text string,

- 1 Press the **Esc** key.
- 2 Click in any figure window (including the current figure).
- 3 Reset the **Editing** property to off.

MATLAB then 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 =  $\frac{1}{72}$  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.

# Text Properties

---

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.

# Text Properties

---

## Interpreter

latex | {tex} | none

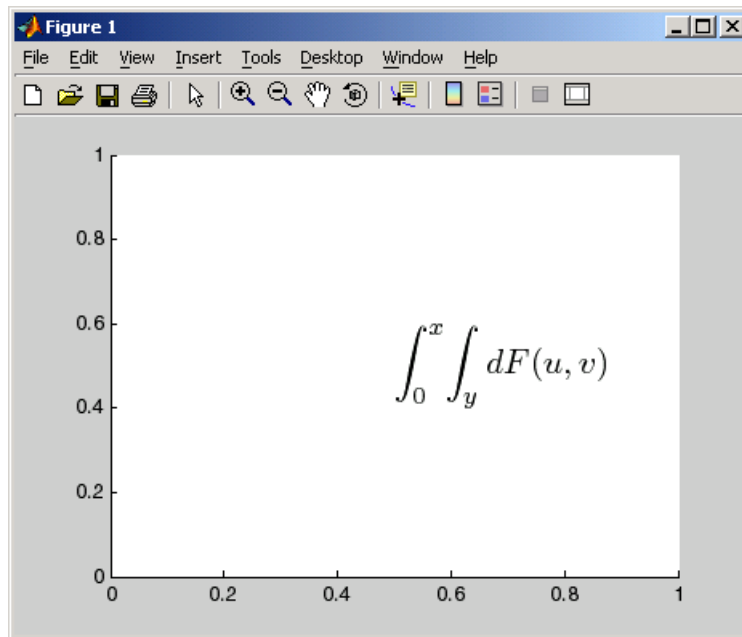
*Interpret TeXinstructions.* Controls whether MATLAB interprets certain characters in the **String** property as TeX instructions (default) or displays all characters literally.

- **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

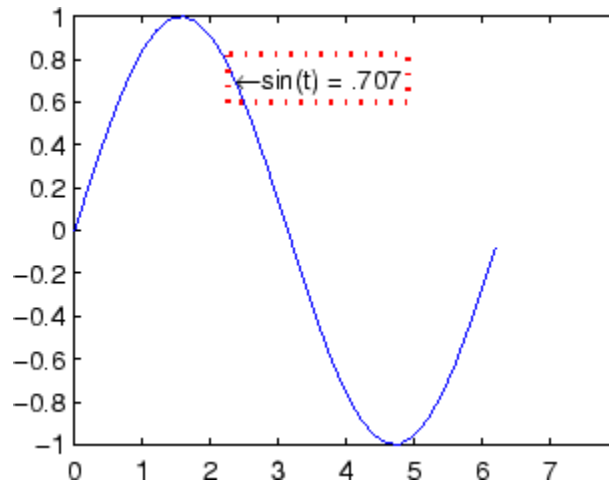
| Specifier | Line Style           |
|-----------|----------------------|
| '-'       | Solid line (default) |
| '--'      | Dashed line          |
| ':'       | Dotted line          |
| '-.'      | Dash-dot line        |
| 'none'    | No line              |

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),...
     '\leftarrow sin(t) = .707',...
     'EdgeColor','red',...
     'LineWidth',2,...
     'LineStyle',':');
```

# Text Properties

---



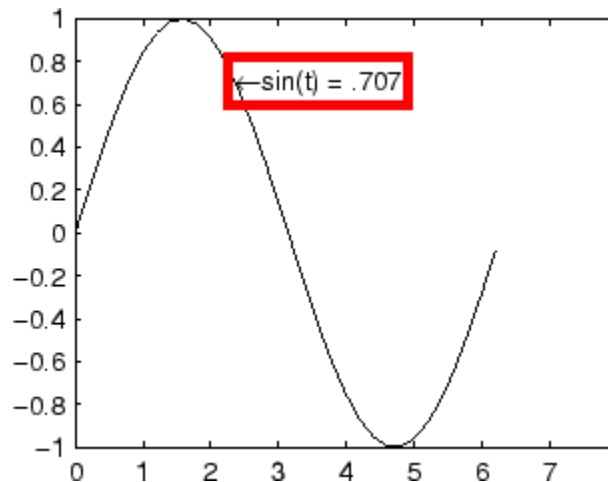
For additional features, see the following properties:

- `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),...  
'\leftarrow sin(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`)

# Text Properties

---

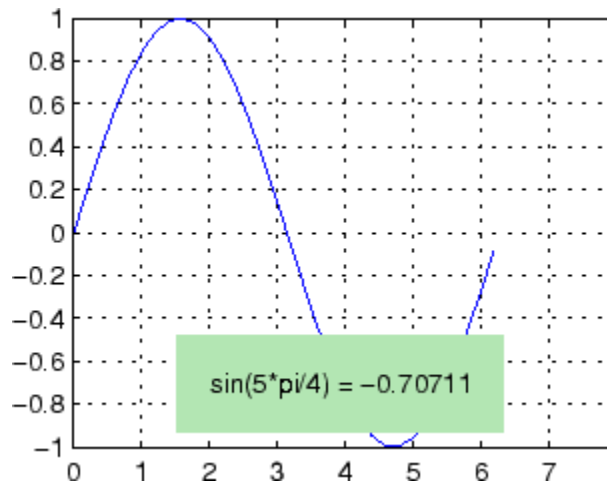
- **Margin** — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the 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**.

# Text Properties

---

## 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      |
|---------------------|----------|-----------------------|------------|------------------------|-------------|
| <code>\alpha</code> | $\alpha$ | <code>\upsilon</code> | $\upsilon$ | <code>\sim</code>      | $\sim$      |
| <code>\angle</code> | $\angle$ | <code>\phi</code>     | $\Phi$     | <code>\leq</code>      | $\leq$      |
| <code>\ast</code>   | $*$      | <code>\chi</code>     | $\chi$     | <code>\infty</code>    | $\infty$    |
| <code>\beta</code>  | $\beta$  | <code>\psi</code>     | $\psi$     | <code>\clubsuit</code> | $\clubsuit$ |

# Text Properties

| Character Sequence     | Symbol      | Character Sequence     | Symbol      | Character Sequence            | Symbol            |
|------------------------|-------------|------------------------|-------------|-------------------------------|-------------------|
| <code>\gamma</code>    | $\Upsilon$  | <code>\omega</code>    | $\omega$    | <code>\diamondsuit</code>     | $\blacklozenge$   |
| <code>\delta</code>    | $\delta$    | <code>\Gamma</code>    | $\Gamma$    | <code>\heartsuit</code>       | $\heartsuit$      |
| <code>\epsilon</code>  | $\epsilon$  | <code>\Delta</code>    | $\Delta$    | <code>\spadesuit</code>       | $\spadesuit$      |
| <code>\zeta</code>     | $\zeta$     | <code>\Theta</code>    | $\Theta$    | <code>\leftrightharrow</code> | $\leftrightarrow$ |
| <code>\eta</code>      | $\eta$      | <code>\Lambda</code>   | $\Lambda$   | <code>\leftarrow</code>       | $\leftarrow$      |
| <code>\theta</code>    | $\Theta$    | <code>\Xi</code>       | $\Xi$       | <code>\Leftarrow</code>       | $\Leftarrow$      |
| <code>\vartheta</code> |             | <code>\Pi</code>       | $\Pi$       | <code>\uparrow</code>         | $\uparrow$        |
| <code>\iota</code>     | $\iota$     | <code>\Sigma</code>    | $\Sigma$    | <code>\rightarrow</code>      | $\rightarrow$     |
| <code>\kappa</code>    | $\kappa$    | <code>\Upsilon</code>  |             | <code>\Rightarrow</code>      | $\Rightarrow$     |
| <code>\lambda</code>   | $\lambda$   | <code>\Phi</code>      | $\Phi$      | <code>\downarrow</code>       | $\downarrow$      |
| <code>\mu</code>       | $\mu$       | <code>\Psi</code>      | $\Psi$      | <code>\circ</code>            | $\circ$           |
| <code>\nu</code>       | $\nu$       | <code>\Omega</code>    | $\Omega$    | <code>\pm</code>              | $\pm$             |
| <code>\xi</code>       | $\xi$       | <code>\forall</code>   | $\forall$   | <code>\geq</code>             | $\geq$            |
| <code>\pi</code>       | $\pi$       | <code>\exists</code>   | $\exists$   | <code>\propto</code>          | $\propto$         |
| <code>\rho</code>      | $\rho$      | <code>\ni</code>       | $\ni$       | <code>\partial</code>         | $\partial$        |
| <code>\sigma</code>    | $\sigma$    | <code>\cong</code>     | $\cong$     | <code>\bullet</code>          | $\bullet$         |
| <code>\varsigma</code> | $\varsigma$ | <code>\approx</code>   | $\approx$   | <code>\div</code>             | $\div$            |
| <code>\tau</code>      | $\tau$      | <code>\Re</code>       | $\Re$       | <code>\neq</code>             | $\neq$            |
| <code>\equiv</code>    | $\equiv$    | <code>\oplus</code>    | $\oplus$    | <code>\aleph</code>           |                   |
| <code>\Im</code>       | $\Im$       | <code>\cup</code>      | $\cup$      | <code>\wp</code>              | $\wp$             |
| <code>\otimes</code>   | $\otimes$   | <code>\subseteq</code> | $\subseteq$ | <code>\oslash</code>          | $\oslash$         |



| Character Sequence   | Symbol    | Character Sequence   | Symbol    | Character Sequence      | Symbol       |
|----------------------|-----------|----------------------|-----------|-------------------------|--------------|
| <code>\cap</code>    | $\cap$    | <code>\in</code>     | $\in$     | <code>\supseteq</code>  | $\supseteq$  |
| <code>\supset</code> | $\supset$ | <code>\lceil</code>  | $\lceil$  | <code>\subset</code>    | $\subset$    |
| <code>\int</code>    | $\int$    | <code>\cdot</code>   | $\cdot$   | <code>\o</code>         | $\circ$      |
| <code>\rfloor</code> | $\rfloor$ | <code>\neg</code>    | $\neg$    | <code>\nabla</code>     | $\nabla$     |
| <code>\lfloor</code> | $\lfloor$ | <code>\times</code>  | $\times$  | <code>\ldots</code>     | $\dots$      |
| <code>\perp</code>   | $\perp$   | <code>\surd</code>   | $\surd$   | <code>\prime</code>     | $'$          |
| <code>\wedge</code>  | $\wedge$  | <code>\varpi</code>  | $\varpi$  | <code>\0</code>         | $\emptyset$  |
| <code>\rceil</code>  | $\rceil$  | <code>\rangle</code> | $\rangle$ | <code>\mid</code>       | $ $          |
| <code>\vee</code>    | $\vee$    |                      |           | <code>\copyright</code> | $\copyright$ |
| <code>\langle</code> | $\langle$ |                      |           |                         |              |

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.

# Text Properties

---

- `\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:

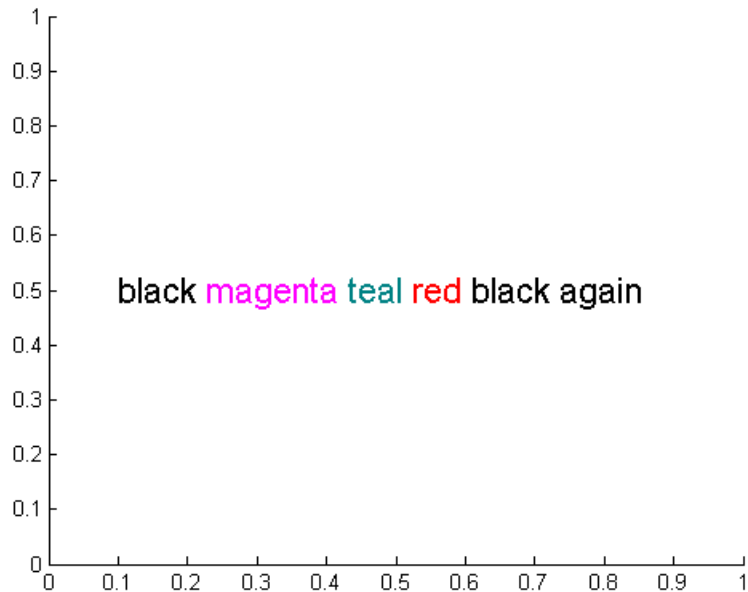
- `\color{colorname}` — Use for the eight basic named colors (red, green, yellow, magenta, blue, black, white), and plus the four Simulink colors (gray, darkGreen, orange, and lightBlue).

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 2-4539 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.

# Text Properties

---

## 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 =  $\frac{1}{72}$  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.

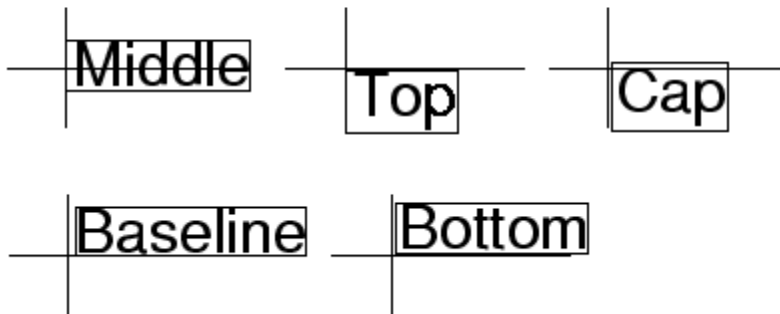
# Text Properties

---

- `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`

**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<br>(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 |
| %*...<br>instead of %             | Ignore the matching characters specified by *.                                                                                                                                                 | No output             |
| %w...<br>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:<br>' '           Space<br>\b            Backspace<br>\n            Newline<br>\r            Carriage return<br>\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 =
    12.340000000000000
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 =  
    12.340000000000000  
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(fileID,formatSpec,N,Name,Value)

C = textscan(str,formatSpec,N,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`, 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(fileID,formatSpec,N,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. You can append `Name,Value` pair arguments to any of the other syntaxes in this syntax group.

`C = textscan(str,formatSpec,N,Name,Value)` reads data from a string, `str`. As with previous syntaxes, use `formatSpec` and optional `N`, and `Name,Value` pair arguments with this syntax. However, 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.

[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        |

| <b>Numeric Input Type</b> | <b>Conversion Specifier</b> | <b>Output Class</b> |
|---------------------------|-----------------------------|---------------------|
| 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

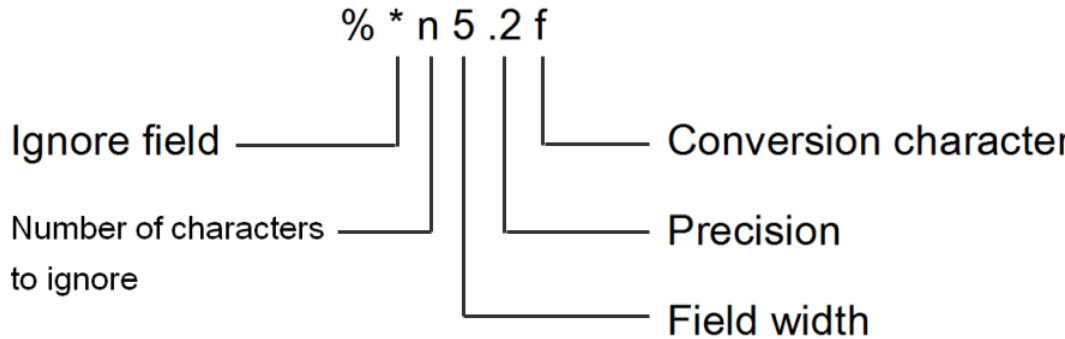
This table lists available conversion specifiers for character inputs.

| <b>Character Strings</b> | <b>Conversion Specifier</b> | <b>Details</b>                                                      |
|--------------------------|-----------------------------|---------------------------------------------------------------------|
| 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 | <code>%[...]</code>  | Read only the characters inside the brackets up to the first nonmatching character. To include <code>]</code> in the set, specify it first: <code>%[...]</code> .<br><br>Example: <code>%[mus]</code> reads 'summer ' as 'summ'. |
|                          | <code>%[^...]</code> | Exclude characters inside the brackets, reading until the first matching character. To exclude <code>]</code> , specify it first: <code>%[^...]</code> .<br><br>Example: <code>%[^xrg]</code> 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                                                                                                                                                                                                                                                                                                                                                                |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%*</code>  | <p>Skip the field. textscan does not create an output cell for any field that it skips.</p> <p>Example: <code>'%s %*s %s %s %*s %*s %s'</code> (spaces are optional) converts the string <code>'Blackbird singing in the dead of night'</code> to four output cells with the strings <code>'Blackbird'</code> <code>'in'</code> <code>'the'</code> <code>'night'</code></p> |
| <code>%*n</code> | <p>Ignore <math>n</math> characters of the field, where <math>n</math> is an integer less than or equal to the number of characters in the field.</p> <p>Example: <code>%*4s</code> reads <code>'summer '</code> as <code>'er'</code>.</p>                                                                                                                                  |

- 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, 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**.

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:

|                 |                              |
|-----------------|------------------------------|
| <code>\b</code> | Backspace                    |
| <code>\n</code> | Newline                      |
| <code>\r</code> | Carriage return              |
| <code>\t</code> | Tab                          |
| <code>\\</code> | Backslash ( <code>\</code> ) |

## **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:

|                 |                              |
|-----------------|------------------------------|
| <code>\b</code> | Backspace                    |
| <code>\n</code> | Newline                      |
| <code>\r</code> | Carriage return              |
| <code>\t</code> | Tab                          |
| <code>\\</code> | Backslash ( <code>\</code> ) |

## 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} =

## textscan

---

```
    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 ui
```

```
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', ',', 'EmptyValue',  
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', ',', 'MultipleDelimsAsOne', 1);
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\f\singing\f\in\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:

|                                                                |                   |
|----------------------------------------------------------------|-------------------|
| $\pm\langle\text{real}\rangle\pm\langle\text{imag}\rangle i j$ | Example: 5.7-3.1i |
| $\pm\langle\text{imag}\rangle i j$                             | Example: -7j      |

Do not include embedded white space in a complex number. textscan interprets embedded white space as a field delimiter.

# textscan

---

## 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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Wrapped string matrix for given uicontrol                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>      | <pre>outstring = textwrap(h,instring) outstring = textwrap(h,instring,columns) [outstring,position] = textwrap(...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Description</b> | <p><code>outstring = textwrap(h,instring)</code> returns a wrapped string cell array, <code>outstring</code>, that fits inside the uicontrol with handle <code>h</code>. <code>instring</code> is a cell array, with each cell containing a single line of text. <code>outstring</code> is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.</p> <p><code>outstring = textwrap(h,instring,columns)</code> returns an <code>outstring</code> with each line wrapped at <code>columns</code> characters. Spaces are included in the character count.</p> <p><code>[outstring,position] = textwrap(...)</code> returns the recommended position of the uicontrol in the units of the uicontrol. <code>position</code> considers the extent of the multiline text in the <code>x</code> and <code>y</code> directions.</p> <p><code>textwrap</code> 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.</p> |
| <b>Tips</b>        | When programming a GUI, do not call <code>copyobj</code> or <code>textwrap</code> (which calls <code>copyobj</code> ) inside a <code>CreateFcn</code> . The act of copying the uicontrol object fires the <code>CreateFcn</code> repeatedly, which raises a series of error messages after exceeding the root object's <code>RecursionLimit</code> property.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Examples</b>    | <p>Place two text-wrapped strings in text uicontrols. The left one has a <code>Position</code> calculated by <code>textwrap</code> in Units of pixels; the right one's <code>Position</code> is calculated manually in Units of characters:</p> <pre>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);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## textwrap

---

```
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)

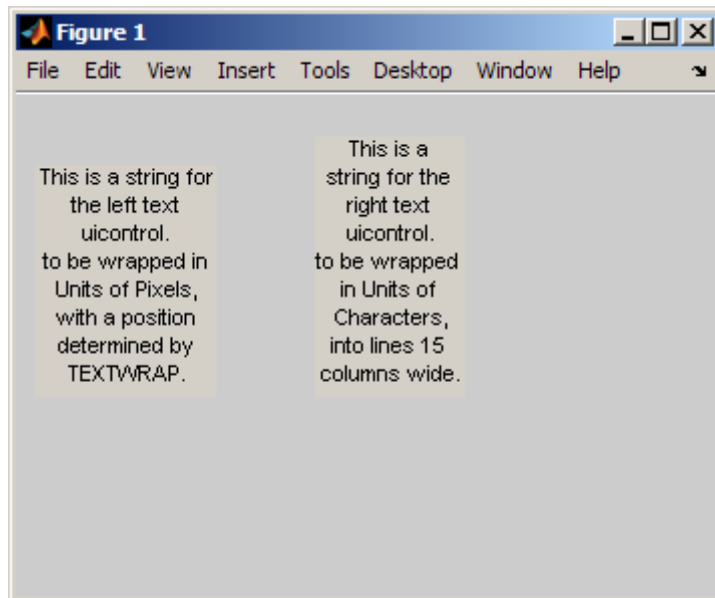
% Make another text uicontrol to wrap to a column width of 15
colwidth = 15;
% Create it in Units of Pixels, 100 wide, 10 high
pos1 = [150 100 100 10];
ht1 = uicontrol('Style','Text','Position',pos1);
string1 = {'This is a string for the right text uicontrol.',...
          'to be wrapped in Units of 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  $m \setminus x$ .

`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 <code>tol</code> within <code>maxit</code> iterations.           |
| 1    | tfqmr iterated <code>maxit</code> times but did not converge.                                             |
| 2    | Preconditioner <code>m</code> 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  $\text{norm}(b-A*x)/\text{norm}(b)$ . If `flag` is 0, then `relres`  $\leq$  `tol`.

`[x,flag,relres,y]y(A,b,...)` also returns the iteration number at which `x` was computed:  $0 \leq \text{iter} \leq \text{maxit}$ .

`[x,flag,relres,iter,resvec] = tfqmr(A,b,...)` also returns a vector of the residual norms at each iteration, including  $\text{norm}(b-A*x_0)$ .

## 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,f10,rr0,it0,rv0] = tfqmr(A,b,tol,maxit);
```

`f10` is 1 because `tfqmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the

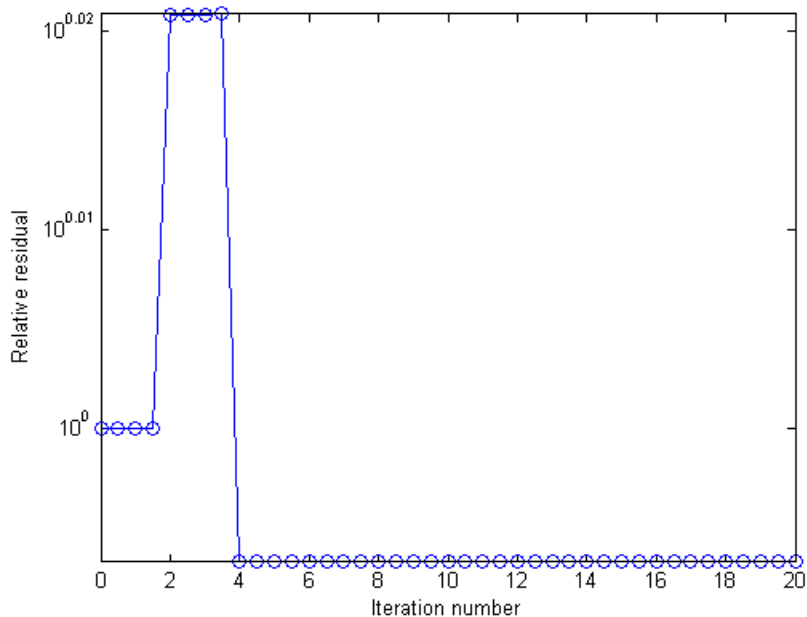


best approximate solution and is the one returned as indicated by `it0 = 17`. MATLAB stores the residual history in `rv0`.

## 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.
```

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:

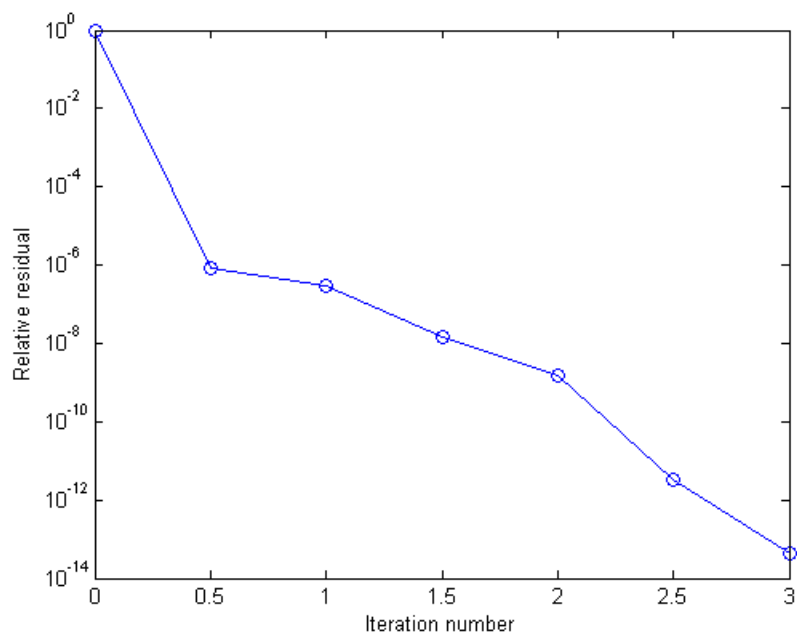
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));  
[x1,f11,rr1,it1,rv1] = tfqmr(A,b,tol,maxit,L,U);
```

`f11` 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

# throw (MException)

---

**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');
```

## throw (MException)

---

```
        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)

# throwAsCaller (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`:

## throwAsCaller (MException)

---

```
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)
```

---

|                         |                                                                                                                                                                                                                                                                                                                                  |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Start clock to measure performance                                                                                                                                                                                                                                                                                               |
| <b>Syntax</b>           | <pre>tic ticID = tic</pre>                                                                                                                                                                                                                                                                                                       |
| <b>Description</b>      | <p><code>tic</code> starts a stopwatch timer. Stop the timer with the <code>toc</code> function.</p> <p><code>ticID = tic</code> stores an identifier for the <code>tic</code> command, so that you can nest timing operations.</p>                                                                                              |
| <b>Tips</b>             | <ul style="list-style-type: none"><li>• Consecutive <code>tic</code> commands overwrite the internally recorded starting time.</li><li>• The <code>clear</code> function does not reset the starting time recorded by a <code>tic</code> command.</li></ul>                                                                      |
| <b>Output Arguments</b> | <p><b>ticID</b></p> <p>Value to use as an input argument for a subsequent call to <code>toc</code>. The value has no independent meaning.</p>                                                                                                                                                                                    |
| <b>Examples</b>         | <p>Measure how the time required to solve a linear system varies with the order of a matrix:</p> <pre>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)</pre> <hr/> <p>Measure the minimum and average time to compute a summation of Bessel functions:</p> |

# tic

---

```
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](#)

---

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Stop clock to measure performance                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Syntax</b>           | <pre>toc toc(ticID) elapsedTime = toc elapsedTime = toc(ticID)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Description</b>      | <p><code>toc</code> stops a stopwatch timer started by the <code>tic</code> function, and displays the time elapsed in seconds.</p> <p><code>toc(ticID)</code> displays the time elapsed since the <code>tic</code> command corresponding to <code>ticID</code>.</p> <p><code>elapsedTime = toc</code> stores the elapsed time in a variable.</p> <p><code>elapsedTime = toc(ticID)</code> stores in a variable the time elapsed since the <code>tic</code> command corresponding to <code>ticID</code>.</p> |
| <b>Tips</b>             | <ul style="list-style-type: none"><li>• Consecutive <code>toc</code> commands return the increasing time that has elapsed since the most recent <code>tic</code>. Therefore, you can take multiple measurements from a single point in time.</li><li>• The <code>toc</code> function displays the time elapsed unless you specify an output argument.</li></ul>                                                                                                                                              |
| <b>Input Arguments</b>  | <p><b>ticID</b></p> <p>Identifier generated by a previous call to <code>tic</code> with an output argument.</p>                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Output Arguments</b> | <p><b>elapsedTime</b></p> <p>Scalar <code>double</code> representing the time elapsed between <code>tic</code> and <code>toc</code> commands, in seconds.</p>                                                                                                                                                                                                                                                                                                                                                |
| <b>Examples</b>         | <p>Measure how the time required to solve a linear system varies with the order of a matrix:</p> <pre>t = zeros(1,100); for n = 1:100</pre>                                                                                                                                                                                                                                                                                                                                                                  |

```
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                                           |
| '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.

### Supported Values

|                   |
|-------------------|
| Unspecified       |
| AssociatedAlpha   |
| UnassociatedAlpha |

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);
```

## 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 <code>DotRange</code> . When you specify CMYK, do not set the <code>InkNames</code> tag. |
| MultiInk | Any ordering other than CMYK. Consult the TIFF specification for <code>InkNames</code> field for a description of the inks used.                                                                                                                                                                        |

Example:

```
tiffobj.setTag('InkSet', Tiff.InkSet.CMYK);
```

## 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.CH
```

### 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

|               |
|---------------|
| Uint          |
| Int           |
| IEEEFP        |
| Void          |
| ComplexInt    |
| ComplexIEEEFP |

Example:

```
tiffobj.setTag('SampleFormat', Tiff.SampleFormat.IEEFP);
```

## 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

|                   |
|-------------------|
| 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                        |

# Tiff

---

|                                |                                                  |
|--------------------------------|--------------------------------------------------|
| <code>readEncodedTile</code>   | Read data from specified tile                    |
| <code>rewriteDirectory</code>  | Write modified metadata to existing IFD          |
| <code>setDirectory</code>      | Make specified IFD current IFD                   |
| <code>setSubDirectory</code>   | Make subIFD specified by byte offset current IFD |
| <code>setTag</code>            | Set value of tag                                 |
| <code>write</code>             | Write entire image                               |
| <code>writeDirectory</code>    | Create new IFD and make it current IFD           |
| <code>writeEncodedStrip</code> | Write data to specified strip                    |
| <code>writeEncodedTile</code>  | Write data to specified tile                     |

## Examples

Create a new TIFF file using the `Tiff` object. To run this example, your directory must be writable.

```
t = Tiff('myfile.tif', 'w');  
%  
% Close the Tiff object  
t.close();
```

## See Also

`imread` | `imwrite`

## 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 2-4635 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.

# timer

| Property Name | Property Description                                                                                                                                                                                                                                                            | Data Types, Values, Defaults, Access |                                             |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|---------------------------------------------|
| AveragePeriod | Average time between TimerFcn executions since the timer started.<br><br>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.<br><br>'drop' — Do not execute the function.<br><br>'error' — Generate an error. Requires ErrorFcn to be set.<br><br>'queue' — Execute function at next opportunity. | Data type                            | Enumerated string                           |
|               |                                                                                                                                                                                                                                                                                 | Values                               | 'drop'<br>'error'<br>'queue'                |
|               |                                                                                                                                                                                                                                                                                 | Default                              | 'drop'                                      |
|               |                                                                                                                                                                                                                                                                                 | Read only                            | While Running = 'on'                        |
| 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                                       |

| Property Name | Property Description                                                                                             | Data Types, Values, Defaults, Access |                                                                                                                                                                                      |
|---------------|------------------------------------------------------------------------------------------------------------------|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ExecutionMode | Determines how the timer object schedules timer events. See “Timer Object Execution Modes” for more information. | Data type                            | Enumerated string                                                                                                                                                                    |
|               |                                                                                                                  | Values                               | 'singleShot'<br>'fixedDelay'<br>'fixedRate'<br>'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                                                                                                                                                                               |
| Name          | User-supplied name.                                                                                              | Data type                            | Text string                                                                                                                                                                          |
|               |                                                                                                                  | Default                              | 'timer- <i>i</i> ', where <i>i</i> is a number indicating the <i>i</i> th timer object created this session. To reset <i>i</i> to 1, execute the <code>clear classes</code> command. |
|               |                                                                                                                  | Read only                            | Never                                                                                                                                                                                |

# timer

| Property Name    | Property Description                                                                                                                                                                                                                                                                                                                                                                                                | Data Types, Values, Defaults, Access |                         |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|-------------------------|
| ObjectVisibility | Provides a way for application developers to prevent end-user access to the timer objects created by their application. The <code>timerfind</code> function does not return an object whose <code>ObjectVisibility</code> 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'<br>'on'           |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Default                              | 'on'                    |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Read only                            | Never                   |
| Period           | Specifies the delay, in seconds, between executions of <code>TimerFcn</code> .                                                                                                                                                                                                                                                                                                                                      | Data type                            | double                  |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Value                                | Any number $\geq 0.001$ |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Default                              | 1.0                     |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Read only                            | While Running = 'on'    |
| Running          | Indicates whether the timer is currently executing.                                                                                                                                                                                                                                                                                                                                                                 | Data type                            | Enumerated string       |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Values                               | 'off'<br>'on'           |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Default                              | 'off'                   |
|                  |                                                                                                                                                                                                                                                                                                                                                                                                                     | Read only                            | Always                  |

| Property Name | Property Description                                                                                                                                                                                                                                                                                                                                              | Data Types, Values, Defaults, Access |                                             |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|---------------------------------------------|
| 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 $\geq 0$                         |
|               |                                                                                                                                                                                                                                                                                                                                                                   | 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                                       |
| StopFcn       | Function the timer calls when it stops. The timer stops when <ul style="list-style-type: none"> <li>• You call the timer stop function</li> <li>• The timer finishes executing TimerFcn, i.e., the value of TasksExecuted reaches the limit set by TasksToExecute.</li> <li>• An error occurs (The ErrorFcn is called first, followed by the StopFcn.)</li> </ul> | Data type                            | Text string, function handle, or cell array |
|               |                                                                                                                                                                                                                                                                                                                                                                   | Default                              | None                                        |
|               |                                                                                                                                                                                                                                                                                                                                                                   | Read only                            | Never                                       |

# timer

| Property Name  | Property Description                                                                                    | Data Types, Values, Defaults, Access |                                             |
|----------------|---------------------------------------------------------------------------------------------------------|--------------------------------------|---------------------------------------------|
|                | See “Creating Callback Functions” for more information.                                                 |                                      |                                             |
| Tag            | User supplied label.                                                                                    | Data type                            | Text string                                 |
|                |                                                                                                         | Default                              | Empty string ( ' ' )                        |
|                |                                                                                                         | Read only                            | Never                                       |
| 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                                       |



| Property Name | Property Description        | Data Types, Values, Defaults, Access |              |
|---------------|-----------------------------|--------------------------------------|--------------|
| 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('string')
title(fname)
title(...,'PropertyName',PropertyValue,...)
title(axes_handle,...)
h = title(...)
```

**Description**

Each axes graphics object can have one title. The title is located at the top and in the center of the axes.

`title('string')` outputs the string at the top and in the center of the current axes.

`title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current axes.

`title(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the text graphics object that `title` creates. Do not use the 'String' text property to set the title string; the content of the title should be given by the first argument.

`title(axes_handle,...)` adds the title to the specified axes.

`h = title(...)` returns the handle to the text object used as the title.

---

**Note** The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

---

**Examples**

Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

# title

---

```
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),'C'])
```

Make a multi-colored title:

```
title(['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```

Include a variable's value in a title and set the color of the title to yellow:

```
n = 3;
title(['Case number #',int2str(n)], 'Color', 'y')
```

Include Greek symbols in a title:

```
title('\ite^{\omega\tau} = cos(\omega\tau) + isin(\omega\tau)')
```

Include a superscript character in a title:

```
title('\alpha^2')
```

Include a subscript character in a title:

```
title('X_1')
```

The text object String property lists the available symbols.

Create a multiline title using a multiline cell array.

```
title({'First line';'Second line'})
```

To change the default interpreter from Tex, set the Interpreter property as:

```
title('X_1', 'Interpreter', 'none');
```




This displays the string `X_1` in the title of the figure instead of making `1` subscript of `X`. See `Interpreter` for other options for the property and also `Text Properties` for other text properties.

**Tips**

`title` sets the `Title` property of the current axes graphics object to a new text graphics object. See the text `String` property for more information.

**Alternatives**

To create or modify a plot's title from a GUI, use **Insert Title** from the figure menu. Use the `Property Editor`, one of the plotting tools , to modify the position, font, and other properties of a legend. For details, see `The Property Editor`.

**See Also**

`gtext` | `int2str` | `num2str` | `text` | `xlabel` | `ylabel` | `zlabel` | `Text Properties`

**How To**

- `Adding Titles to Graphs`

# 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` | `cdfwrite` | `datenum`

**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`

# toolboxdir

---

**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 *toolboxfcn*:

```
n = 'toolboxfcn';
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](#)

# trace

---

|                    |                                                                                |
|--------------------|--------------------------------------------------------------------------------|
| <b>Purpose</b>     | Sum of diagonal elements                                                       |
| <b>Syntax</b>      | <code>b = trace(A)</code>                                                      |
| <b>Description</b> | <code>b = trace(A)</code> is the sum of the diagonal elements of the matrix A. |
| <b>Algorithms</b>  | <code>t = sum(diag(A));</code>                                                 |
| <b>See Also</b>    | <code>det</code>   <code>eig</code>                                            |

**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`

# trapz

---

**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](#)

# treelayout

---

**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` | `treepplot` | `etreeplot` | `symbfact`

**Purpose**

Plot picture of tree

**Syntax**

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

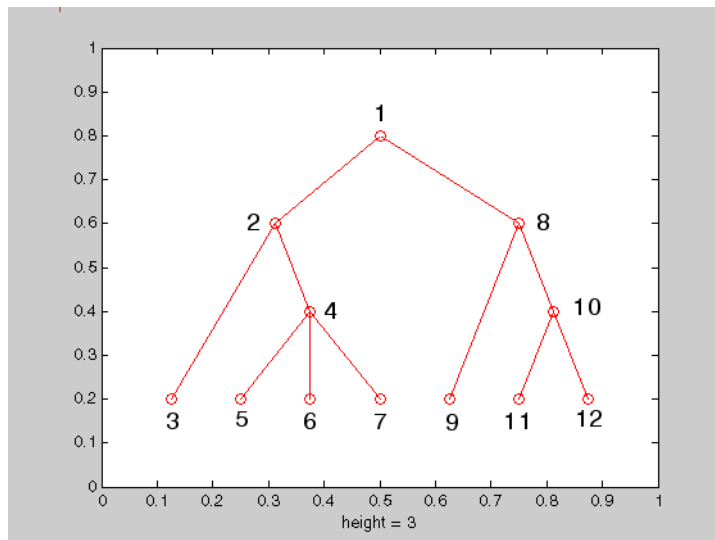
**Description**

`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).

# treeplot

---

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)
```

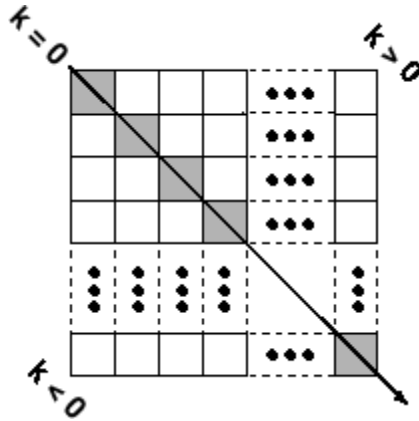
## See Also

[etree](#) | [etreeplot](#) | [treelayout](#)

**Purpose** Lower triangular part of matrix

**Syntax**  $L = \text{tril}(X)$   
 $L = \text{tril}(X,k)$

**Description**  $L = \text{tril}(X)$  returns the lower triangular part of  $X$ .  
 $L = \text{tril}(X,k)$  returns the elements on and below the  $k$ th 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`

# trimesh

---

**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 `TriRep` 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.

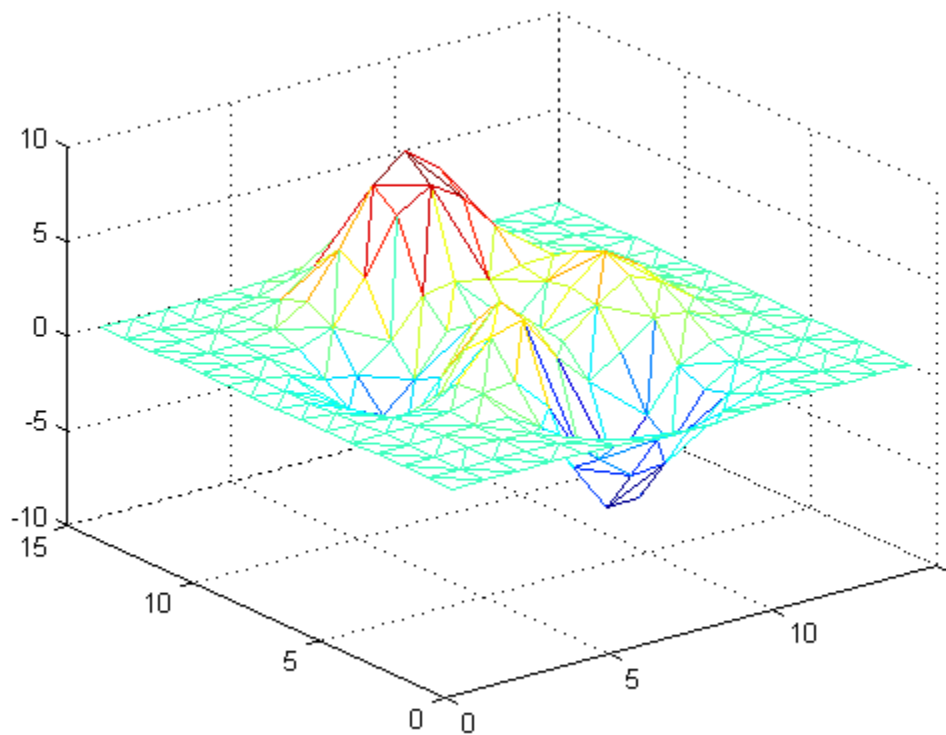
**Examples** Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
[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 it may be plotted as follows:

```
tr = TriRep(tri, x(:), y(:), z(:));
```

```
trimesh(tr)
```



## See Also

[patch](#) | [trisurf](#) | [delaunay](#) | [DelaunayTri](#) | [TriRep](#)

# triplequad

---

**Purpose** Numerically evaluate triple integral

**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  $\text{fun}(x,y,z)$  over the three dimensional rectangular region  $x_{\min} \leq x \leq x_{\max}$ ,  $y_{\min} \leq y \leq y_{\max}$ ,  $z_{\min} \leq z \leq z_{\max}$ . 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 `@quad1` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quad1`.

**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  $0 \leq x \leq \pi, 0 \leq y \leq 1, -1 \leq z \leq 1$ . Note that the integrand can be evaluated with a vector  $x$  and scalars  $y$  and  $z$ .

## See Also

[dblquad](#) | [quad2d](#) | [quad](#) | [quadgk](#) | [quad1](#) | [function\\_handle](#) | [integral](#) | [integral2](#) | [integral3](#)

## How To

- “Anonymous Functions”

# triplot

---

**Purpose** 2-D triangular plot

**Syntax**

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
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.

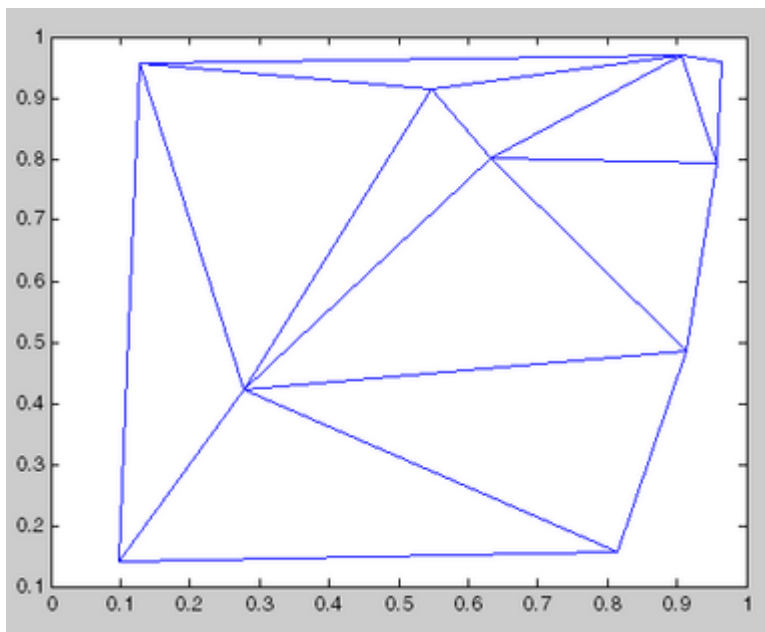
`triplot(TRI,x,y,color)` uses the string color as the line color. color can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineStyle` for information about line specifications.

`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.

```
X = rand(10,2);
dt = DelaunayTri(X);
triplot(dt)
```



Plot the Delaunay triangulation in face-vertex format.

```
tri = dt(:,:,);
triplot(tri, X(:,1), X(:,2));
```

**See Also**

[DelaunayTri](#) | [delaunay](#) | [trimesh](#) | [trisurf](#)

**Purpose** Triangulation representation

**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 Triangulation representation

|                |                              |                                                         |
|----------------|------------------------------|---------------------------------------------------------|
| <b>Methods</b> | <code>baryToCart</code>      | Convert point coordinates from barycentric to Cartesian |
|                | <code>cartToBary</code>      | Convert point coordinates from cartesian to barycentric |
|                | <code>circumcenters</code>   | Circumcenters of specified simplices                    |
|                | <code>edgeAttachments</code> | Simplices attached to specified edges                   |
|                | <code>edges</code>           | Triangulation edges                                     |
|                | <code>faceNormals</code>     | Unit normals to specified triangles                     |
|                | <code>featureEdges</code>    | Sharp edges of surface triangulation                    |
|                | <code>freeBoundary</code>    | Facets referenced by only one simplex                   |
|                | <code>incenters</code>       | Incenters of specified simplices                        |
|                | <code>isEdge</code>          | Test if vertices are joined by edge                     |

|                   |                                                 |
|-------------------|-------------------------------------------------|
| neighbors         | Simplex neighbor information                    |
| size              | Size of triangulation matrix                    |
| vertexAttachments | 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.

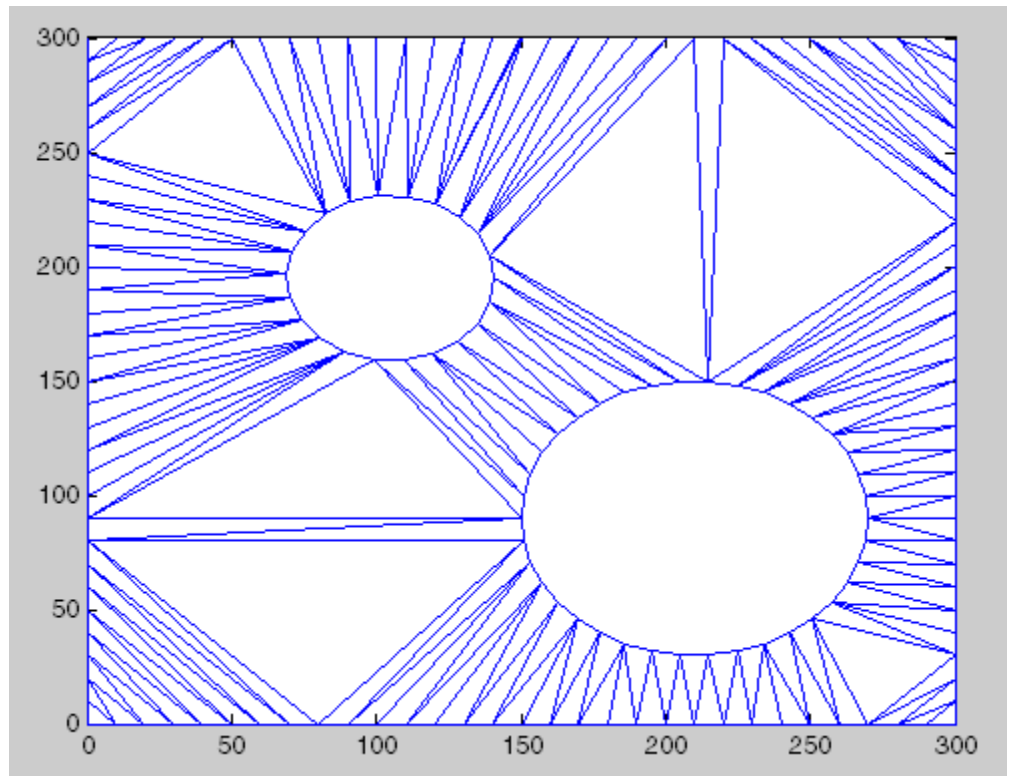
## Examples

Load a 2-D triangulation and use the TriRep constructor to build an array of the free boundary edges:

```
load trimesh2d
```

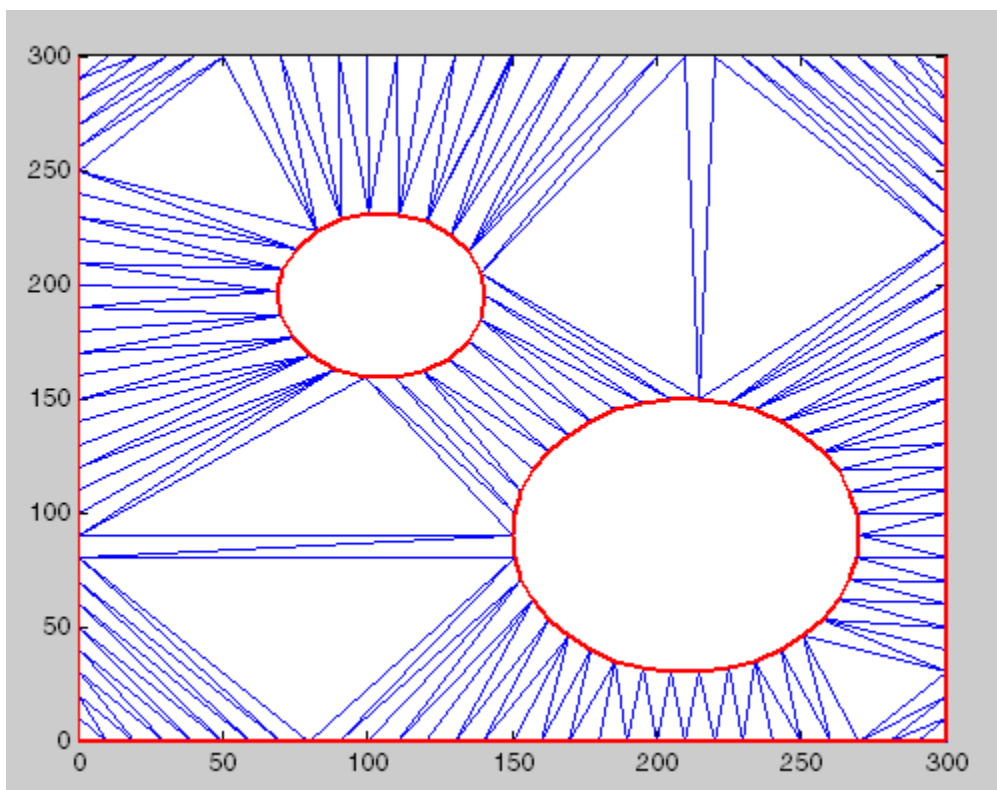
This loads triangulation `tri` and vertex coordinates `x, y`:

```
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**

[DelaunayTri](#) | [TriScatteredInterp](#)

**Purpose**            Triangulation representation

**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  $m$  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  $2 \leq ndim \leq 3$ .

**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**

TriScatteredInterp

# TriScatteredInterp

---

**Purpose** Interpolate scattered data

**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 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.                  |                                |
| Method | Defines method used to interpolate the data .                   |                                |
|        | natural                                                         | Natural neighbor interpolation |
|        | linear                                                          | Linear interpolation (default) |
|        | nearest                                                         | Nearest neighbor interpolation |

## 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:

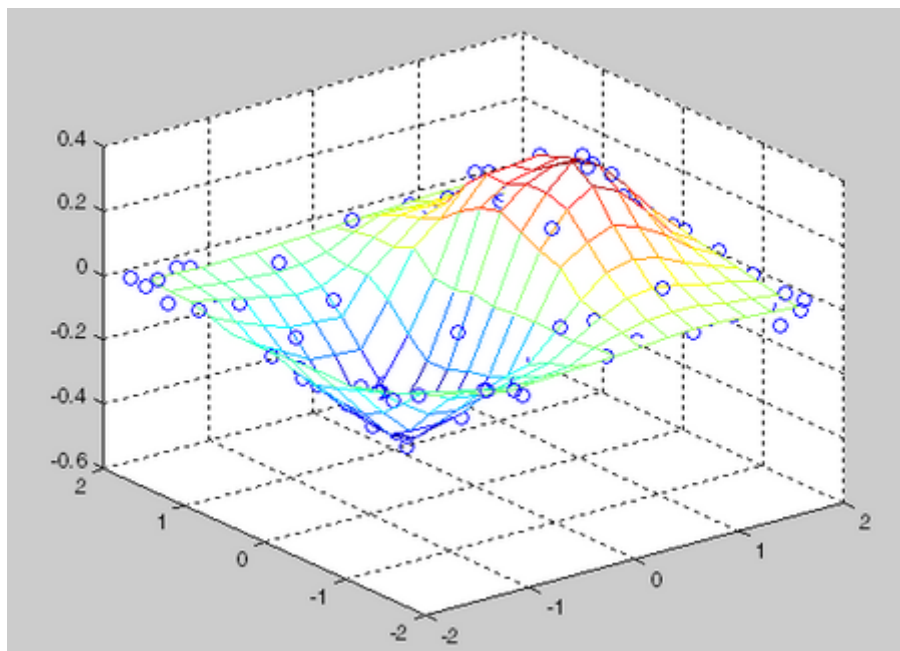
```
F = TriScatteredInterp(x,y,z);
```

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');
```

# TriScatteredInterp

---



## See Also

[DelaunayTri](#) | [interp1](#) | [interp2](#) | [interp3](#) | [meshgrid](#)

## Purpose

Interpolate scattered data

## 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,  $2 \leq \text{ndim} \leq 3$ . `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.

# TriScatteredInterp

## Input Arguments

|        |                                                                                                                                                                                                                                           |                                |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------|
| X      | Matrix of size <code>mpts-by-ndim</code> , where <code>mpts</code> is the number of points and <code>ndim</code> 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 <code>mpts</code> .                                                                                                                                              |                                |
| 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  $V_q = F(X_q)$ ,  $V_q = F(X_q, Y_q)$ , or  $V_q = F(X_q, Y_q, Z_q)$  where  $V_q$  is the value of the interpolant at the query location and  $X_q$ ,  $Y_q$ , and  $Z_q$  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:

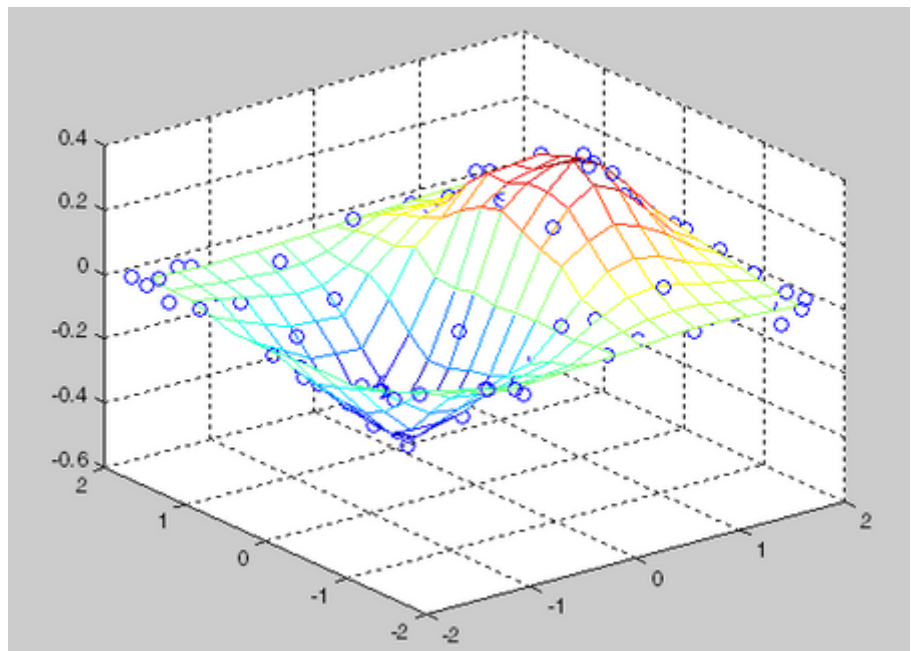
```
x = rand(100,1)*4-2;  
y = rand(100,1)*4-2;  
z = x.*exp(-x.^2-y.^2);
```

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

[DelaunayTri](#) | [interp1](#) | [interp2](#) | [interp3](#) | [meshgrid](#)

# trisurf

---

**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 `TriRep` triangulation representation. It uses `C = TR.X(:,3)` so 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.

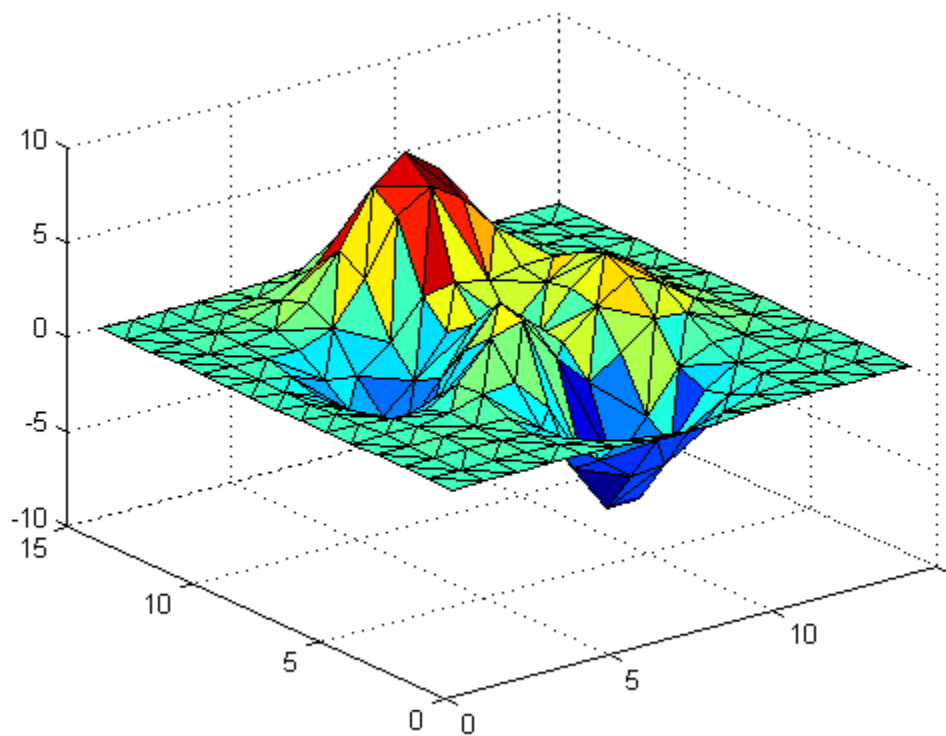
**Examples** Create vertex vectors and a face matrix, then create a triangular surface plot.

```
[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 `TriRep` triangulation representation, plot it as follows:

```
tr = TriRep(tri, x(:), y(:), z(:));
trisurf(tr)
```



**See Also**

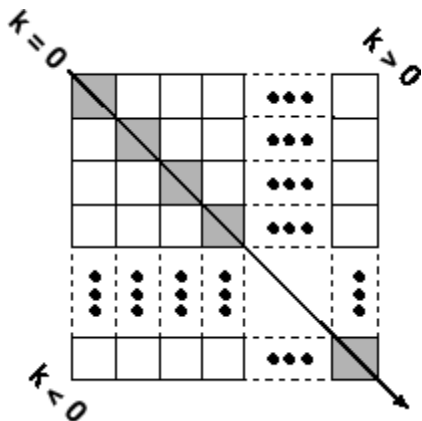
`patch` | `surf` | `tetramesh` | `trimesh` | `triplot` | `delaunay` | `TriRep`  
| `DelaunayTri`

# triu

**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 `k`th 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`

**Purpose** Logical 1 (true)

**Syntax**

```
true
true(n)
true(m, n)
true(m, n, p, ...)
true(size(A))
```

**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  $m, n, p, \dots$  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`.

**Tips**

`true(n)` is much faster and more memory efficient than `logical(ones(n))`.

**See Also** `false` | `logical`

# try/catch

---

**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 occurred: 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 modified filename.
        try
            imageData = imread(altFilename);
        catch exception2
```

# try/catch

---

```
        % Rethrow original error.  
        rethrow(exception)  
    end  
  
    else  
        rethrow(exception)  
    end  
  
end
```

## See Also

error | assert | MException

**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.

# tscollection

---

| Property | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Name     | tscollection name as a string. This can differ from the tscollection name in the MATLAB workspace.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Time     | <p>When <code>TimeInfo.StartDate</code> is empty, values are measured relative to 0 . When <code>TimeInfo.StartDate</code> is defined, values represent date strings measured relative to the <code>StartDate</code>.</p> <p>The length of <code>Time</code> must be the same as the first or the last dimension of <code>Data</code> for each collection .</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| TimeInfo | <p>Contains fields for contextual information about <code>Time</code>:</p> <ul style="list-style-type: none"><li>• <code>Units</code> — Time units with any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', 'nanoseconds'</li><li>• <code>Start</code> — Start time</li><li>• <code>End</code> — End time (read only)</li><li>• <code>Increment</code> — Interval between subsequent time values. NaN when times are not uniformly sampled.</li><li>• <code>Length</code> — Length of the time vector (read only)</li><li>• <code>Format</code> — String defining the date string display format. See <code>datestr</code>.</li><li>• <code>StartDate</code> — Date string defining the reference date. See <code>setabstime (tscollection)</code>.</li><li>• <code>UserData</code> — Any additional user-defined information</li></ul> |

## 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`

# tsdata.event

---

**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.

**Purpose** Search for enclosing Delaunay triangle

---

**Note** tsearch has been removed. Use DelaunayTri/pointLocation instead.

---

**Syntax** `T = tsearch(x,y,TRI,xi,yi)`

**Description** `T = tsearch(x,y,TRI,xi,yi)` returns an index into the rows of TRI for each point in xi, yi. The tsearch command returns NaN for all points outside the convex hull. Requires a triangulation TRI of the points x,y obtained from delaunay.

**See Also** DelaunayTri | delaunay | delaunayn | tsearchn

# 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** [DelaunayTri](#) | [tsearch](#)

**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 ( $3 \times 256 + 232 = 1000$ ). 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 =
```

# typecast

---

12345678

## **See Also**

cast | class | swapbytes

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Create container object to exclusively manage radio buttons and toggle buttons                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Syntax</b>      | <pre>uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,     ...) handle = uibuttongroup(...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p>A <code>uibuttongroup</code> 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, <code>uipanel</code>s, and <code>uibuttongroups</code>. It cannot contain ActiveX controls.</p> <pre>uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)</pre> creates a visible container component in the current figure window. This component manages exclusive selection behavior for <code>uicontrol</code> s of style <code>radiobutton</code> and <code>togglebutton</code> . |
|                    | <pre>handle = uibuttongroup(...)</pre> creates a <code>uibuttongroup</code> object and returns a handle to it in <code>handle</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                    | A <code>uibuttongroup</code> object can have axes, <code>uicontrol</code> , <code>uipanel</code> , and <code>uibuttongroup</code> objects as children. However, only <code>uicontrol</code> s of style <code>radiobutton</code> and <code>togglebutton</code> are managed by the component.                                                                                                                                                                                                                                                                                                                   |
|                    | When programming a button group, you do not code callbacks for the individual buttons; instead, use its <code>SelectionChangeFcn</code> callback to manage responses to selections. The following example illustrates how you use <code>uibuttongroup</code> event data to do this.                                                                                                                                                                                                                                                                                                                           |
|                    | For the children of a <code>uibuttongroup</code> object, the <code>Position</code> 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:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|                    | <ul style="list-style-type: none"><li>• 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 <code>SelectionChangeFcn</code> callback function,</li></ul>                                                                                                                                                                                                                                                                                                                                   |

# uibbuttongroup

---

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 `uibbuttongroup`. If you do not specify a parent, `uibbuttongroup` adds the button group to the current figure. If no figure exists, one is created.

See the `Uibbuttongroup Properties` reference page for more information.

After creating a `uibbuttongroup`, 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 `uibbuttongroup` object to `'off'`, any child objects it contains (buttons, button groups, etc.) become invisible along with the `uibbuttongroup` 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 `uibbuttongroup` with three radio buttons. It manages the radio buttons with the `SelectionChangeFcn` callback, `selcbk`.

When you select a new radio button, `selcbk` displays the `uibbuttongroup` 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.

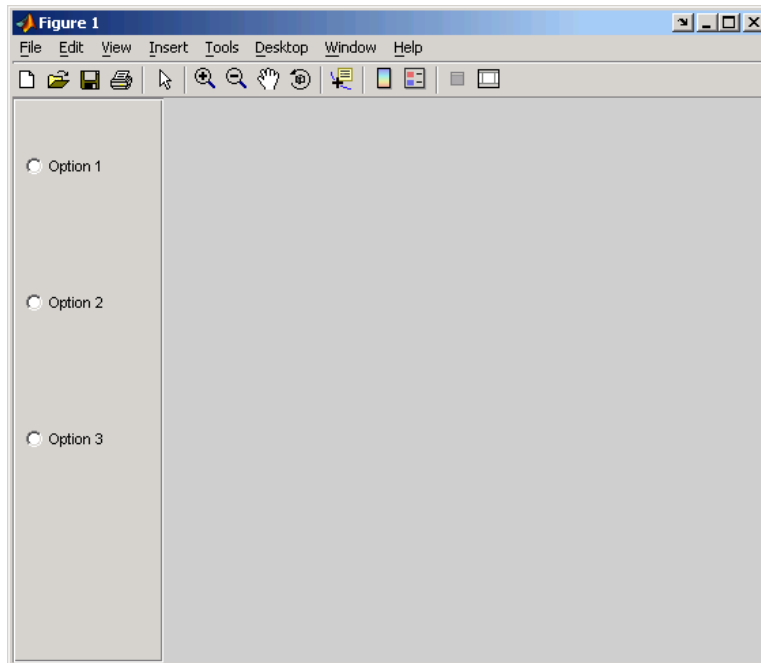
```
% Create the button group.
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
% Create three radio buttons in the button group.
u0 = uicontrol('Style','radiobutton','String','Option 1',...
    'pos',[10 350 100 30],'parent',h,'HandleVisibility','off');
u1 = uicontrol('Style','radiobutton','String','Option 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'));
```

# uibuttongroup

---



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                                |
|------------------------------|--------------------------------------------|
| <code>BackgroundColor</code> | Color of the button group background       |
| <code>BeingDeleted</code>    | This object is being deleted               |
| <code>BorderType</code>      | Type of border around the button group     |
| <code>BorderWidth</code>     | Width of the button group border in pixels |
| <code>BusyAction</code>      | Interruption of other callback routines    |
| <code>ButtonDownFcn</code>   | Button-press callback routine              |
| <code>Children</code>        | All children of the button group           |



# Uibuttongroup Properties

| 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                                                                                                 |

# Uibuttongroup Properties

---

| 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<br><br><b>Note</b> Controls the visibility of a uibuttongroup and of its child axes, uibuttongroups, uipanel, 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:

# Uibuttongroup Properties

---

- '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, `uipanel`s, `uibuttongroup`s, and other `uicontrol`s. You can use this property to reorder the children.

## Clipping

{on} | off

*Clipping mode.* By default, MATLAB clips a `uibuttongroup`'s child axes, `uipanel`s, and `uibuttongroup`s 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 `gcb0` 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 `uipanel`s. 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(gcb0, ...  
    ' '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(gcb0, ...  
    ' 'FontName', 'times', 'FontSize', 14)')
```

# Uibuttongroup Properties

---

---

**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).

# Uibuttongroup Properties

---

## 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:

# Uibuttongroup Properties

---

- 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

*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

# Uibuttongroup Properties

---

*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

# Uibuttongroup Properties

---

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                                                                            |
|----------------------------|----------------------------------------------------------------------------------------|
| <code>EventName</code>     | 'SelectionChanged'                                                                     |
| <code>OldValue</code>      | Handle of the object selected before this event. <code>[]</code> if none was selected. |
| <code>NewValue</code>      | 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 of the 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 `uipanel`.

## UIContextMenu

handle

*Associate a context menu with a uibuttongroup.* Assign this property the handle of a `Uicontextmenu` object. MATLAB displays



# Uibuttongroup Properties

---

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 Properties

---

*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, `uipanel`s, and other `uibuttongroup`s, 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.

---

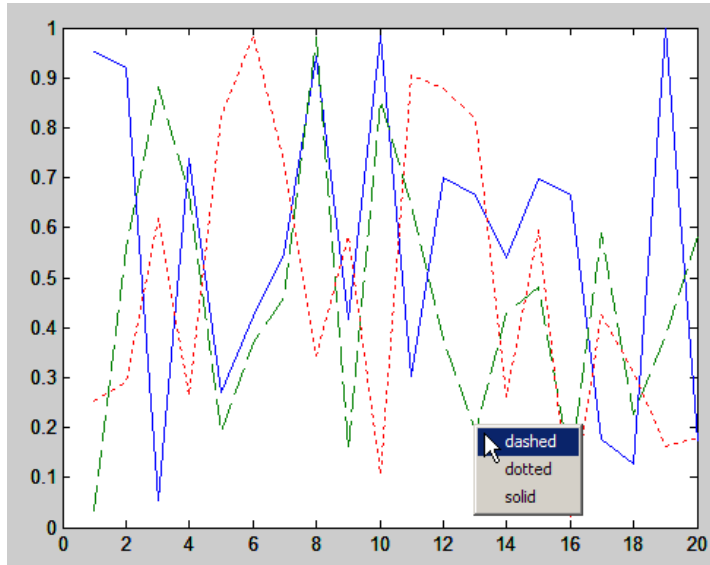
|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Create context menu                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <code>handle = uicontextmenu('PropertyName',PropertyValue,...)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p><code>handle = uicontextmenu('PropertyName',PropertyValue,...)</code> 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.</p> <p>In its initial state, a context menu has no menu items. You create menu items within the context menu using the <code>uimenu</code> function. Menu items appear in the order in which the <code>uimenu</code> statements appear. You then associate a context menu with an object by specifying the handle of the context menu as the value for its <code>UIContextMenu</code> property.</p>                                                                                                                                                                                                                                                                                                                  |
| <b>Examples</b>    | <p>The following statements define a context menu associated with a line on a graph. The menu items enable you to change the line style.</p> <pre>% 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(gcf, ''LineStyle'', ''--'')']; hcb2 = ['set(gcf, ''LineStyle'', '':''')']; hcb3 = ['set(gcf, ''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)</pre> |

## uicontextmenu

---

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**

`uibbuttongroup` | `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.

## Uicontextmenu 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                                                              |
|-------------------------------|----------------------------------------------------------------------|
| <code>BeingDeleted</code>     | This object is being deleted                                         |
| <code>BusyAction</code>       | Callback routine interruption                                        |
| <code>Callback</code>         | Control action                                                       |
| <code>Children</code>         | The <code>uimenu</code> s defined for the <code>uicontextmenu</code> |
| <code>CreateFcn</code>        | Callback routine executed during object creation                     |
| <code>DeleteFcn</code>        | Callback routine executed during object deletion                     |
| <code>HandleVisibility</code> | Whether handle is accessible from command line and GUIs              |
| <code>Interruptible</code>    | Callback routine interruption mode                                   |
| <code>Parent</code>           | <code>Uicontextmenu</code> 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:

# Uicontextmenu Properties

---

- '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 `gcb0` 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.

# Uicontextmenu Properties

---

## 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`

# Uicontextmenu Properties

---

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, panel, or button group to the top left corner of the context menu.

## Tag

string

# Uicontextmenu Properties

---

*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 **Ctrl+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 theClick 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.

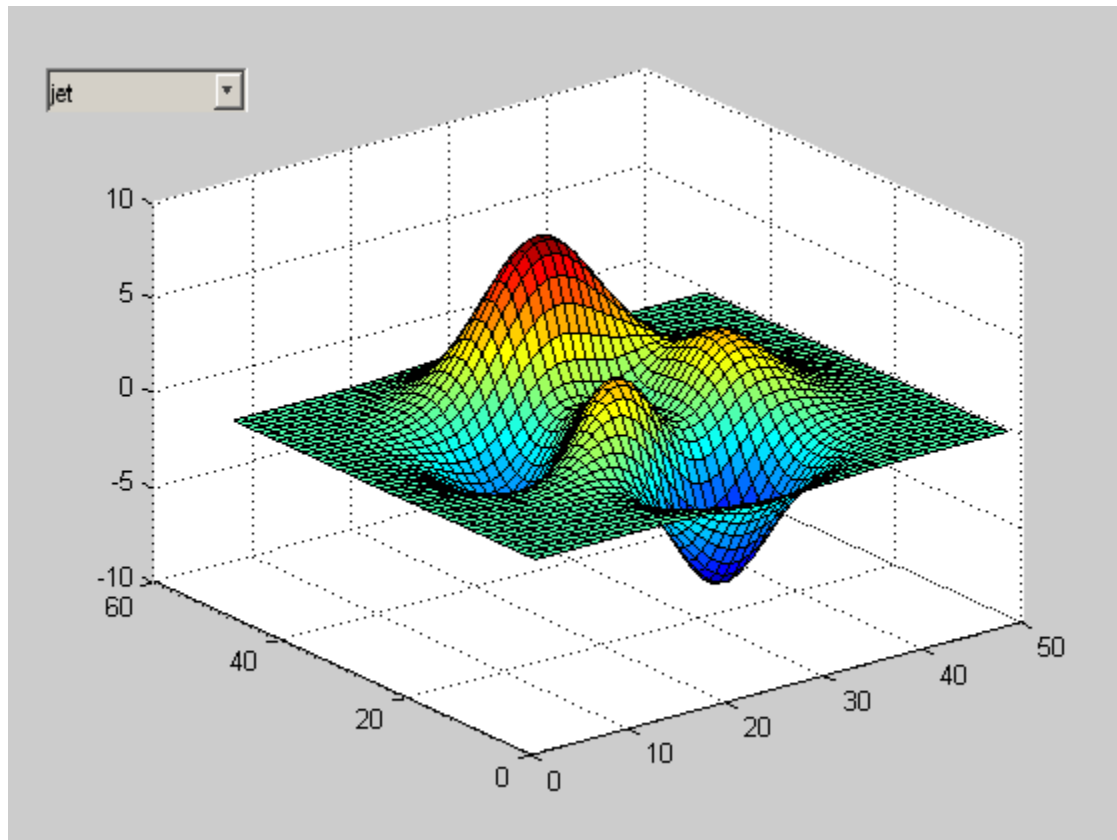
## Examples

Create a figure and an axes to contain a 3-D surface plot.

```
figure
hax = axes('Units','pixels');
surf(peaks)
```

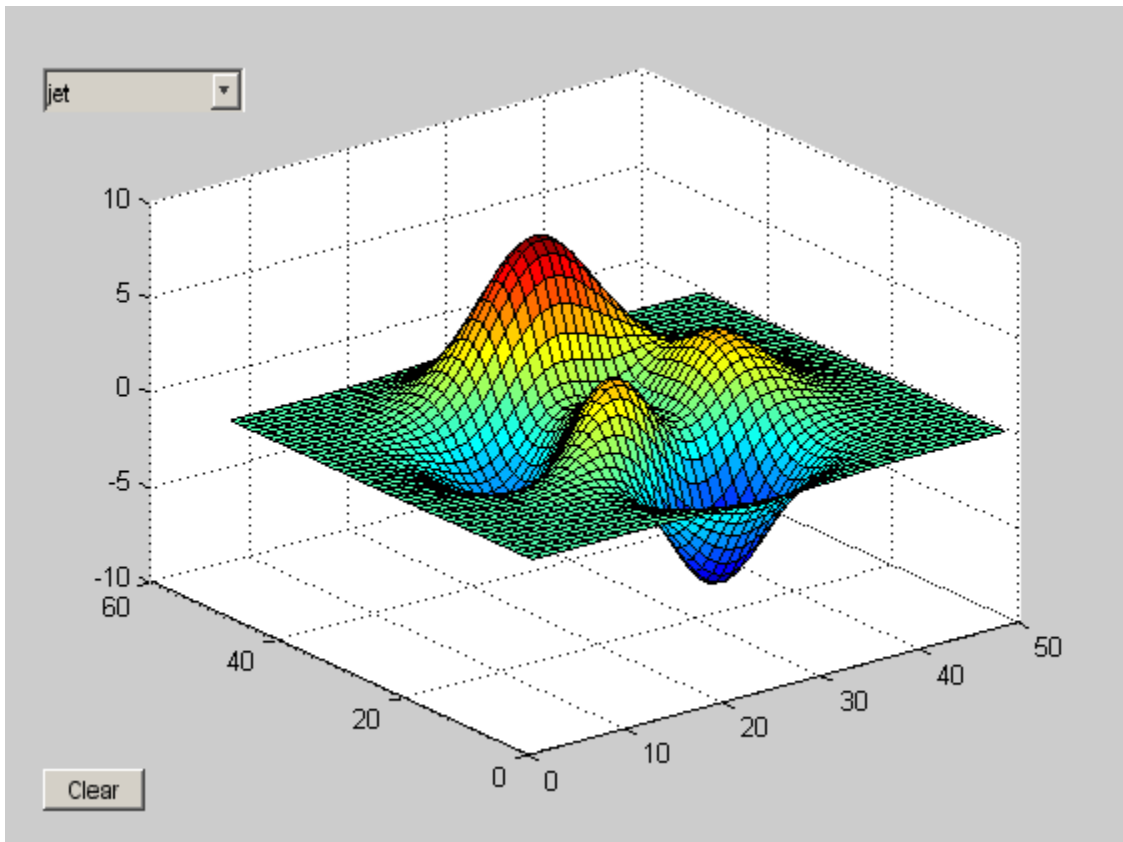
Place a uicontrol object to let users change the colormap with a pop-up menu. Supply a function handle as the object's `Callback` property:

```
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.

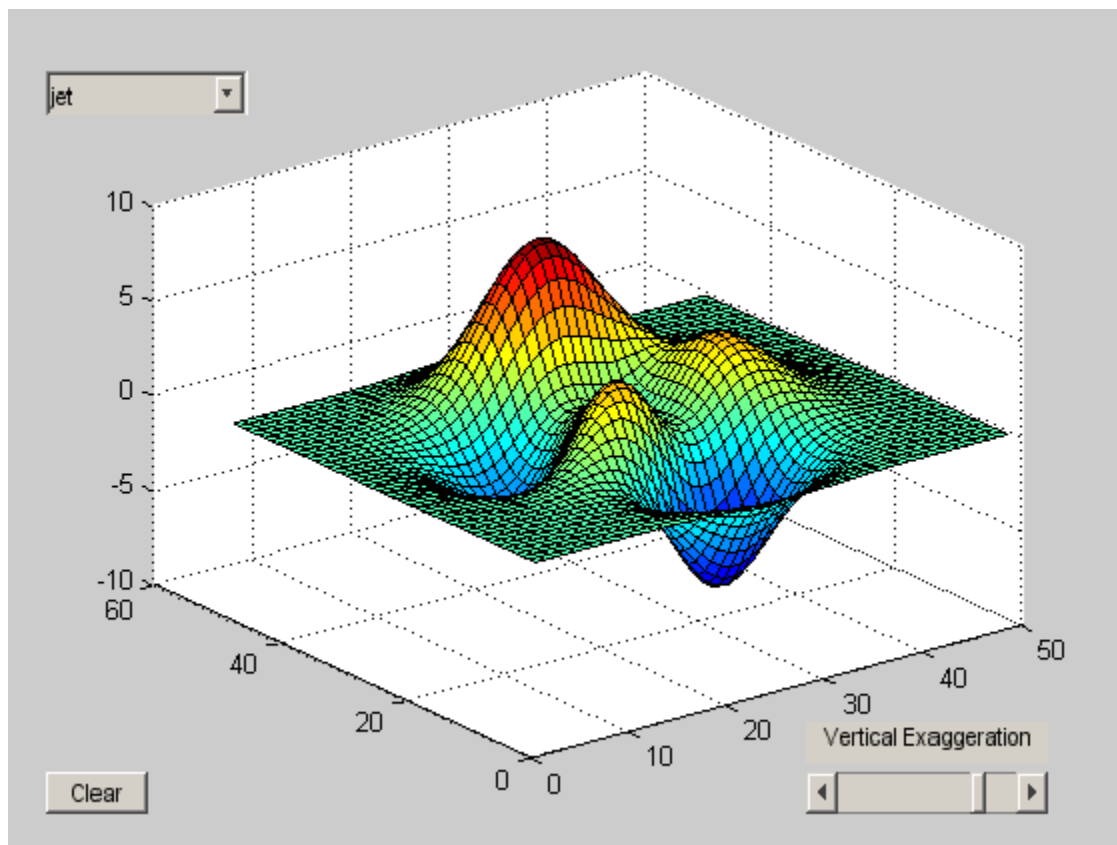
```
uicontrol('Style', 'pushbutton', 'String', 'Clear',...  
         'Position', [20 20 50 20],...  
         'Callback', 'cla');           % Pushbutton string callback  
                                       % that calls a MATLAB function
```



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(hObject,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(hObject,event,ax) %#ok<INUSL>
% Called to set zlim of surface in figure axes
% when user moves the slider control
val = 51 - get(hObject,'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                       |
|------------------------------|-------------------------------|
| <code>BackgroundColor</code> | Object background color       |
| <code>BeingDeleted</code>    | This object is being deleted  |
| <code>BusyAction</code>      | Callback routine interruption |
| <code>ButtonDownFcn</code>   | Button-press callback routine |

# Uicontrol Properties

| <b>Property</b>     | <b>Purpose</b>                                          |
|---------------------|---------------------------------------------------------|
| 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)             |

# Uicontrol Properties

---

| 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.

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)

# Uicontrol Properties

---

*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

# Uicontrol Properties

---

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 `uicontrol`s. 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 `uicontrol`s.

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

# Uicontrol Properties

---

*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

# Uicontrol Properties

---

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 =  $\frac{1}{72}$  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

# Uicontrol Properties

---

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.

# Uicontrol Properties

---

---

**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”.

# Uicontrol Properties

---

---

**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`

# Uicontrol Properties

---

$\leq 1$ , 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  $\text{Max} - \text{Min} > 1$ , then list boxes allow multiple item selection. If  $\text{Max} - \text{Min} \leq 1$ , 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  $\text{Max} - \text{Min} > 1$ , then editable text boxes accept multiline input. If  $\text{Max} - \text{Min} \leq 1$ , 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  $\text{Max} - \text{Min} > 1$ , then list boxes allow multiple item selection. If  $\text{Max} - \text{Min} \leq 1$ , 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.

# Uicontrol Properties

---

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.

# Uicontrol 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                                      | <code>{'one' 'two' 'three'}</code>         |
| Padded string column matrix                                | <code>['one ' '; 'two ' '; 'three']</code> |
| String vector using vertical slash ( ) as a line separator | <code>['one two three']</code>             |

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.

# Uicontrol Properties

---

---

**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.

# Uicontrol 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 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)
```

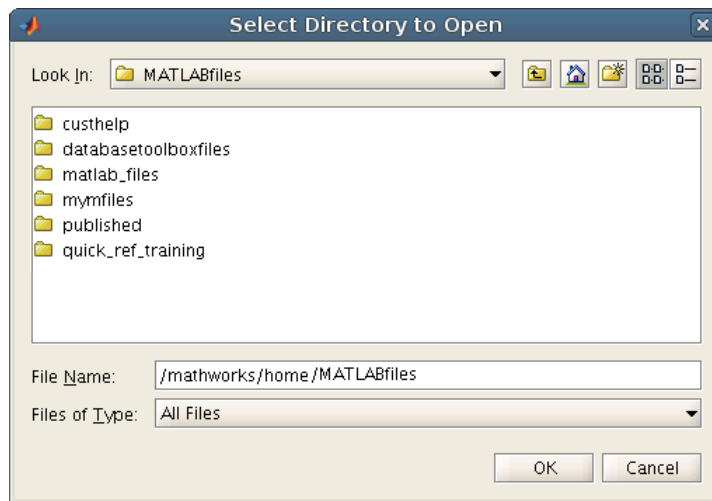
**Description** `folder_name = uigetdir` displays a modal dialog box enabling you to navigate the folder hierarchy and select a folder or type the name of a folder. If the folder exists, `uigetdir` returns the selected path when you click **OK**. If you type the name of a folder that does not exist, `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.

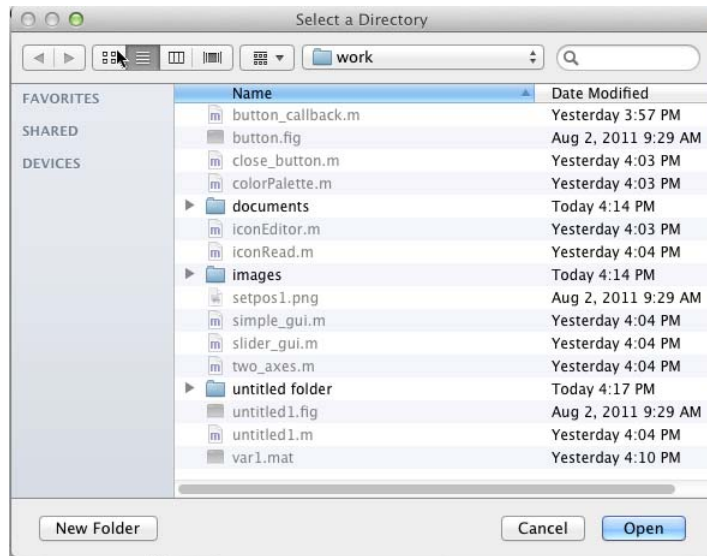
`folder_name = uigetdir(start_path,dialog_title)` opens a dialog box with the specified title. On Windows and UNIX platforms, the string replaces the default caption inside the dialog box for specifying instructions to the user. The default `dialog_title` is **Select folder to Open**.

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 `WindowState` in the MATLAB Figure Properties.

---

The `pwd` and `cd` functions return the name of the current folder.

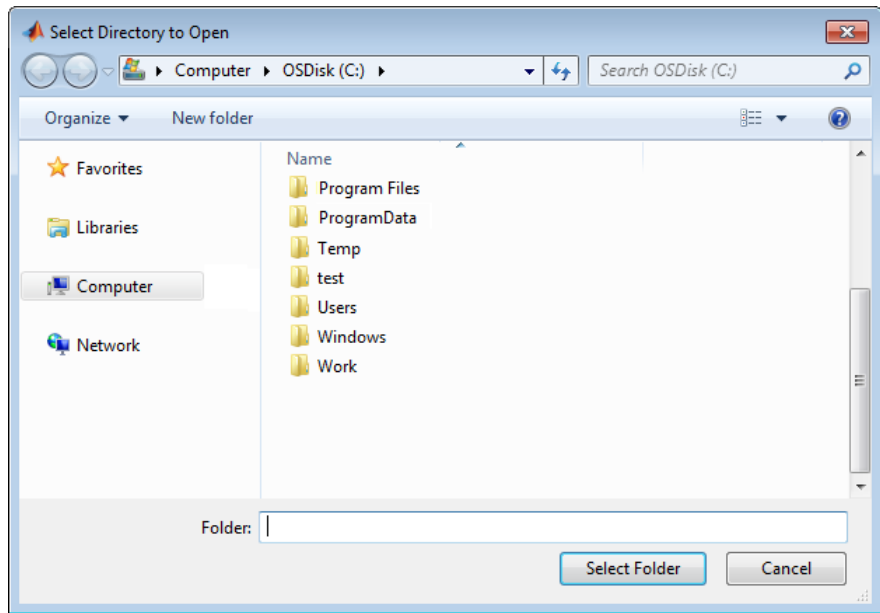
## Examples

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.
```

# uigetfile

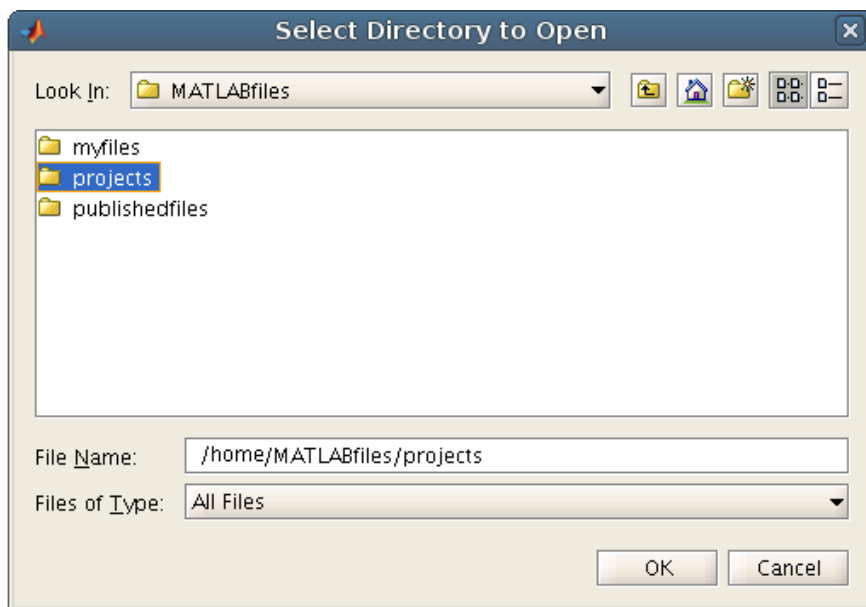
---

`[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.

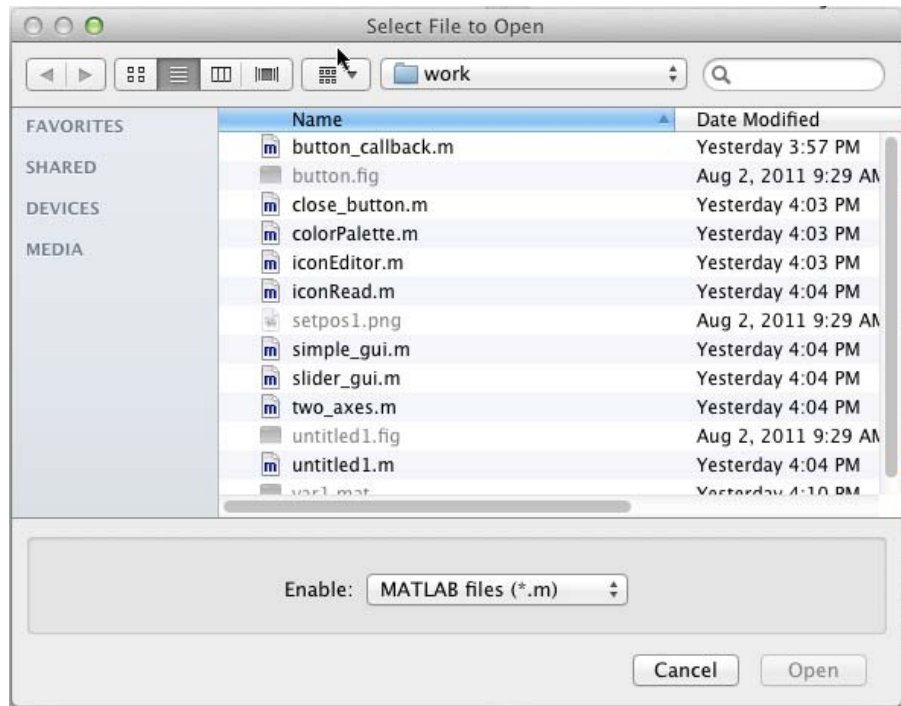
For Windows platforms, the dialog box is the Windows dialog box native to your platform. Depending on your version of Windows, dialogs you see can differ from the figures shown in following examples.

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.

# uigetfile



---

**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 `WindowState` in the MATLAB Figure Properties.

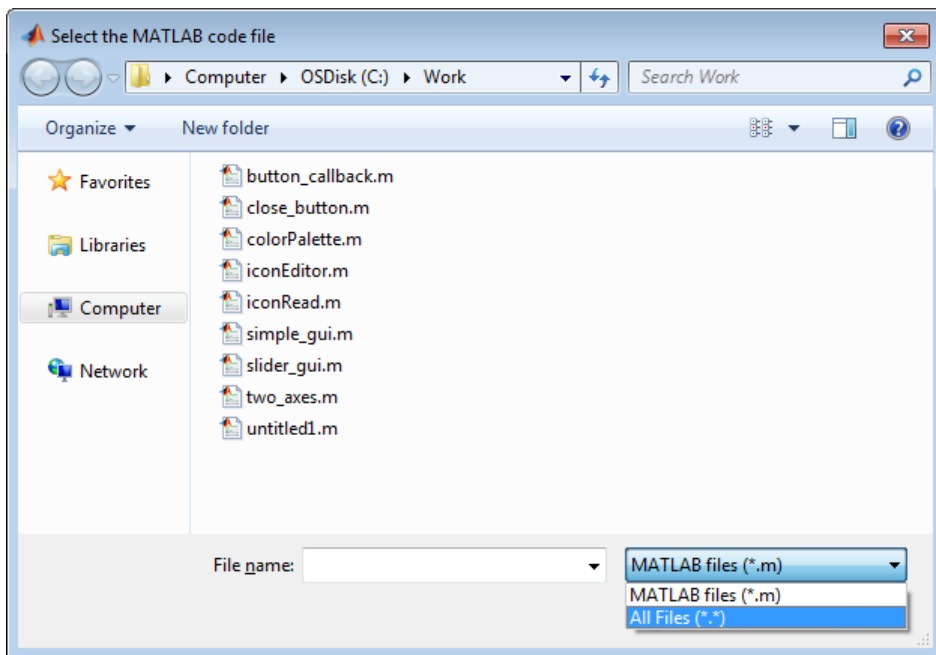
---

## 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');
```

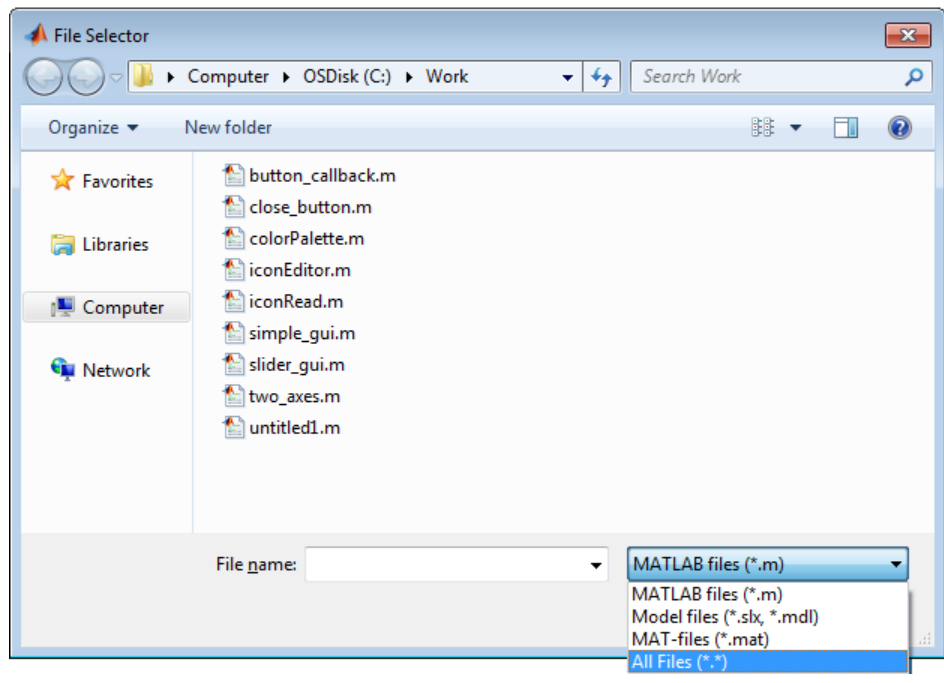
The following figure shows the dialog box with the file type drop-down list open.



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');
```

# uigetfile

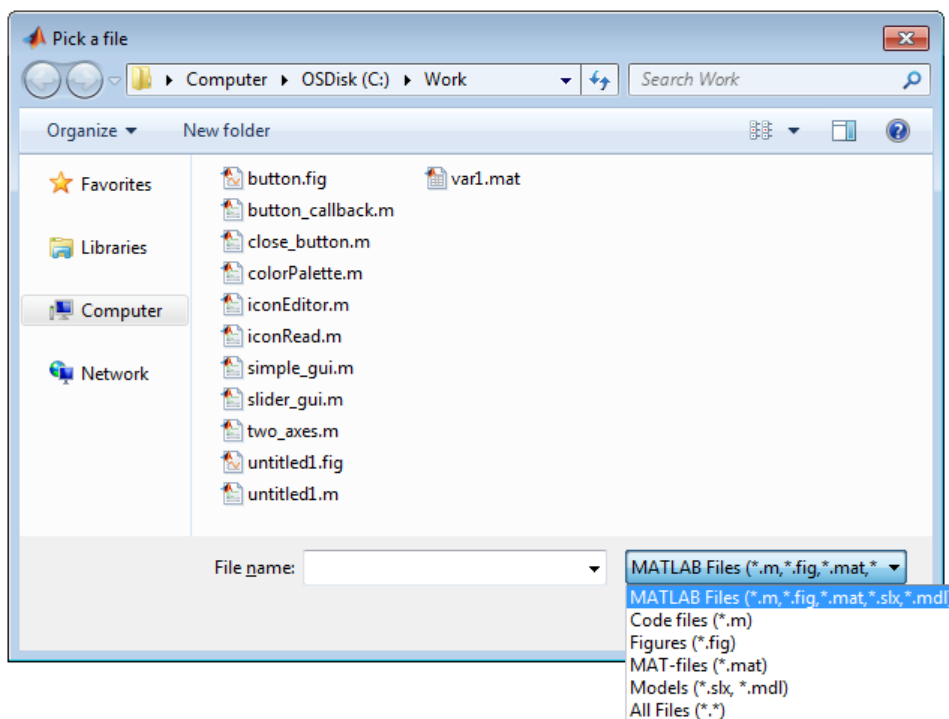


If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' and 'Models' descriptions.

```
[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('*.*', 'Select a MATLAB code file');
if isequal(filename,0)
    disp('User selected Cancel')
```

# uigetfile

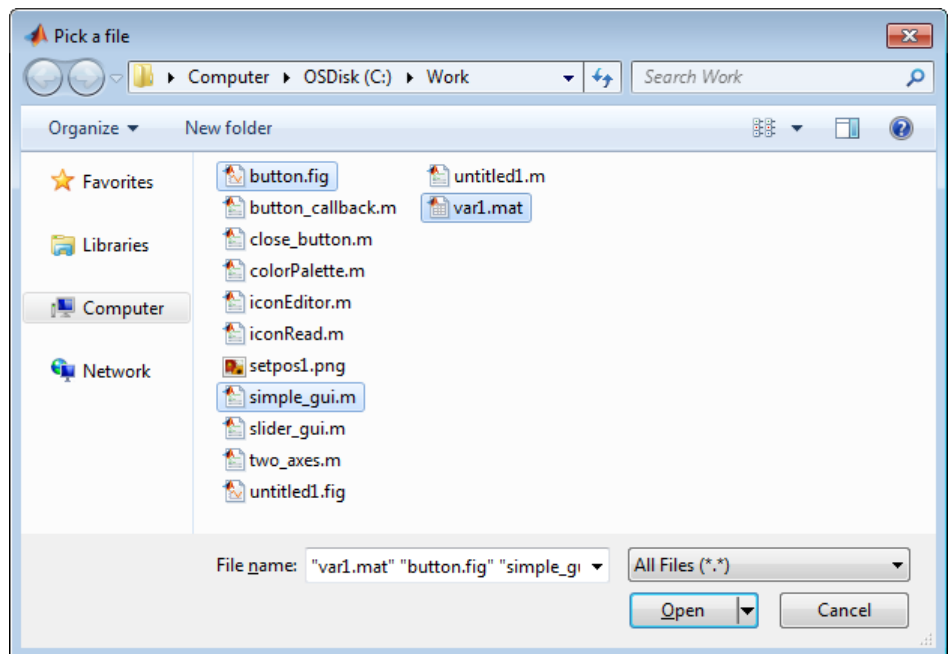
---

```
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');
```

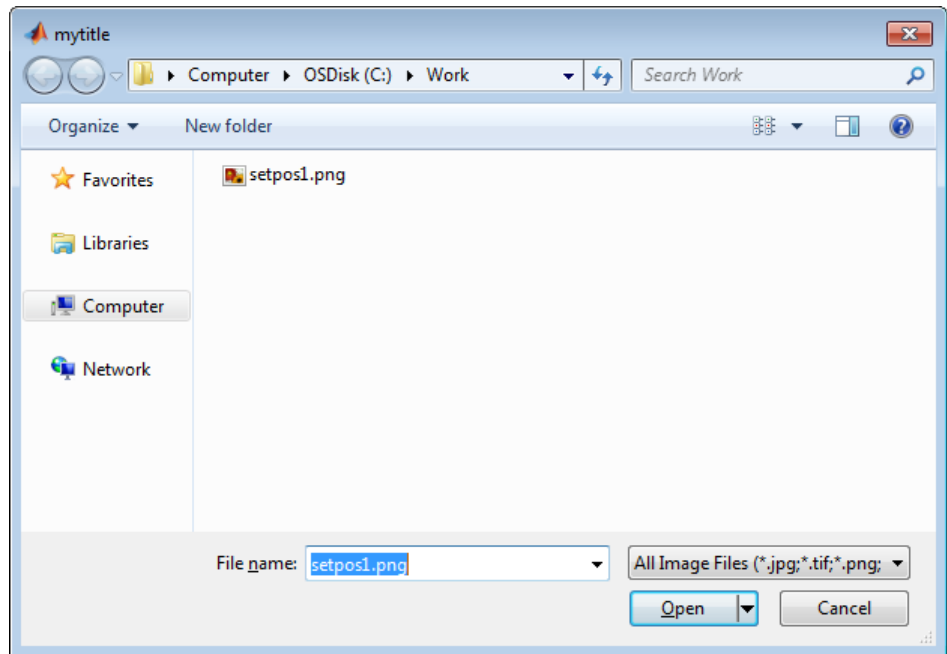


As mentioned previously, `uigetfile` does not open the file or files you select.

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')
```

# uigetfile



## 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`

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Specify and conditionally open dialog box according to user preference                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>      | <pre>pref_value = uigetpref(group,pref,title,question,     pref_choices) [pref_value,dlgshown] = uigetpref(...) [...] = uigetpref(... 'Name',value)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b> | <p><code>pref_value = uigetpref(group,pref,title,question,pref_choices)</code> returns one of the strings in <code>pref_choices</code> in either of two ways:</p> <ul style="list-style-type: none"><li>• Displays a multiple-choice dialog box that prompts you to answer a question. The dialog box includes a check box with the label <b>Do not show this dialog again</b>.</li><li>• 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 <b>Do not show this dialog again</b>.</li></ul> <p><code>[pref_value,dlgshown] = uigetpref(...)</code> also returns in <code>dlgshown</code> whether or not the dialog displayed.</p> <p><code>[...] = uigetpref(... 'Name',value)</code> specify optional name-value pairs to control how dialogs display.</p> |
| <b>Tips</b>        | <ul style="list-style-type: none"><li>• You must supply all input arguments up to and including <code>pref_choices</code>.</li><li>• <code>uigetpref</code> creates specified groups and preferences, if they do not currently exist. To delete a preference group you no longer need, use <code>rmpref</code>.</li><li>• The string returned in <code>pref_value</code> is a preference name (as specified in <code>pref</code>), not its button label (as specified in <code>pref_choices</code>).</li><li>• After you select the check box <b>Do not show this dialog again</b> 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:</li></ul>                                                                                                             |

# uigetpref

---

```
uigetpref('clearall')
```

Executing `uigetpref` 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, 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`.

### **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

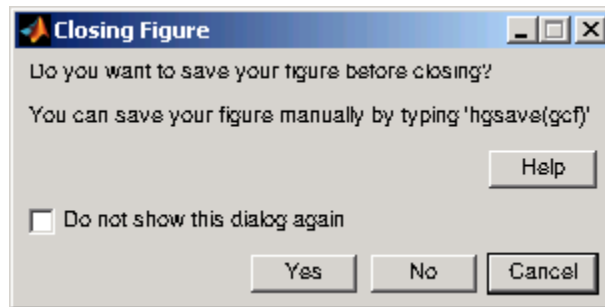
Preferences provide the ability to specify how applications behave and how users interact with them. For example, you can set a preference for which products display their documentation in the Help browser, or which messages the Code Analyzer displays. In MathWorks software products, preferences persist 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] = uinputfile('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”

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Create menus and menu items on figure windows                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Syntax</b>      | <pre>handle = uimenu('PropertyName',PropertyValue,...) handle = uimenu(parent,'PropertyName',PropertyValue,...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b> | <p><code>handle = uimenu('PropertyName',PropertyValue,...)</code> creates a menu in the current figure's menu bar using the values of the specified properties and assigns the menu handle to <code>handle</code>.</p> <p><code>handle = uimenu(parent,'PropertyName',PropertyValue,...)</code> creates a submenu of a parent menu or a menu item on a context menu specified by <code>parent</code> and assigns the menu handle to <code>handle</code>. If <code>parent</code> refers to a figure instead of another <code>uimenu</code> object or a <code>uicontextmenu</code>, MATLAB software creates a new menu on the referenced figure's menu bar.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Tips</b>        | <p>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. <code>uimenu</code> accepts property name/property value pairs as well as structures and cell arrays of properties as input arguments.</p> <p>Use the <code>uimenu</code> <code>Callback</code> property to define the action taken when you activate the created menu item.</p> <p>Uimenu only appear in figures whose <code>Window Style</code> is <code>normal</code>. If a figure containing <code>uimenu</code> children is changed to <code>modal</code>, the <code>uimenu</code> children still exist and are contained in the <code>Children</code> list of the figure, but are not displayed until the <code>WindowStyle</code> is changed to <code>normal</code>.</p> <p>The value of the figure <code>MenuBar</code> property affects the content of the figure menu bar. When <code>MenuBar</code> is <code>figure</code>, a set of built-in menus precedes any user-created <code>uimenu</code>s on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user). When <code>MenuBar</code> is <code>none</code>, <code>uimenu</code>s are the only items on the menu bar (that is, the built-in menus do not appear).</p> |

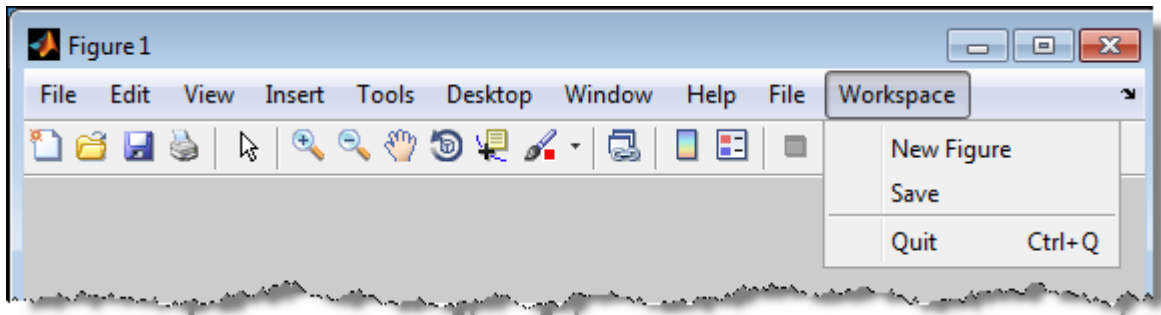
# uimenu

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.

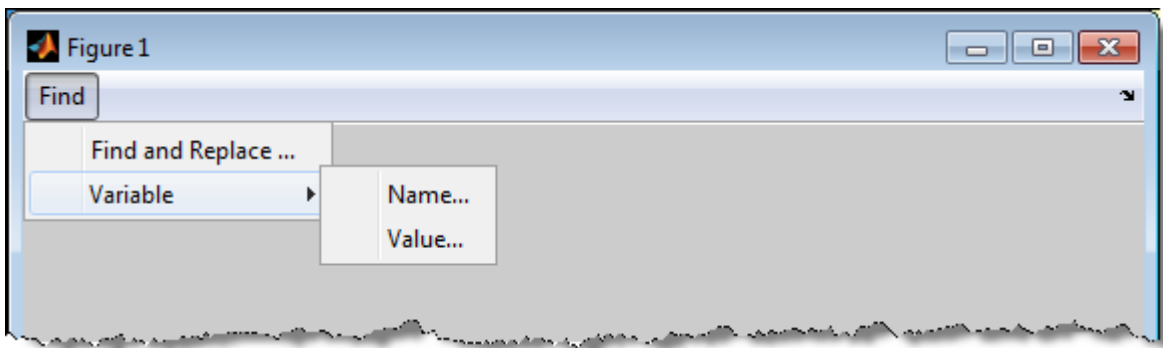
```
f = uimenu('Label', 'Workspace');
    uimenu(f, 'Label', 'New Figure', 'Callback', 'figure');
    uimenu(f, 'Label', 'Save', 'Callback', 'save');
    uimenu(f, 'Label', 'Quit', 'Callback', 'exit', ...
           'Separator', 'on', 'Accelerator', 'Q');
```



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.

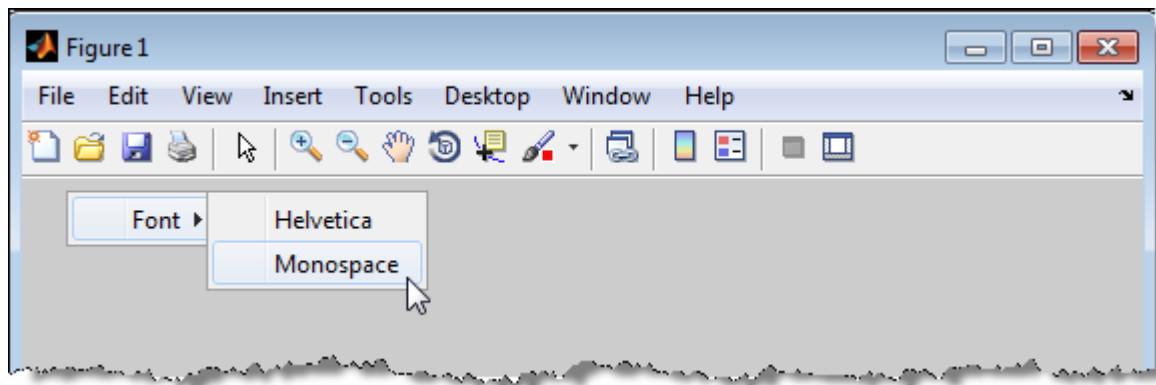
```
% Create the UICONTEXTMENU
cmenu = uicontextmenu;

% 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);
```

# uimenu

---



## 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           |

# Uimenu Properties

---

| 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.

# Uimenu Properties

---

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.

# Uimenu Properties

---

---

**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 uimenu are other uimenu objects, 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 uimenu objects. 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.

# Uimenu Properties

---

## 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

# Uimenu Properties

---

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.

# Uimenu Properties

---

## 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.

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Convert to 8-bit unsigned integer                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Syntax</b>           | <code>intArray = uint8(array)</code>                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b>      | <code>intArray = uint8(array)</code> converts the elements of an array into unsigned 8-bit (1-byte) integers of class <code>uint8</code> .                                                                                                                                                                                                                                                                     |
| <b>Input Arguments</b>  | <p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If <code>array</code> is already of class <code>uint8</code>, the <code>uint8</code> function has no effect.</p>                                                                                                                                                                                        |
| <b>Output Arguments</b> | <p><b>intArray</b></p> <p>Array of class <code>uint8</code>. Values range from 0 to <math>2^8 - 1</math>.</p> <p>The <code>uint8</code> function maps any values in <code>array</code> that are outside the limit to the nearest endpoint. For example,</p> <pre>uint8(2^8)    % 2^8 = 256</pre> <p>returns</p> <pre>ans =     255</pre>                                                                       |
| <b>Examples</b>         | <p>Convert a double array to <code>uint8</code>:</p> <pre>mydata = uint8(magic(10));</pre>                                                                                                                                                                                                                                                                                                                     |
| <b>Alternatives</b>     | <p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = uint8(zeros(100));    % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'uint8');  % Preferred</pre> |

# uint8

---

## See Also

[double](#) | [single](#) | [uint16](#) | [uint32](#) | [uint64](#) | [int8](#) | [int16](#) | [int32](#) | [int64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)

|                         |                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Convert to 16-bit unsigned integer                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>           | <code>intArray = uint16(array)</code>                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b>      | <code>intArray = uint16(array)</code> converts the elements of an array into unsigned 16-bit (2-byte) integers of class <code>uint16</code> .                                                                                                                                                                                                                                                                |
| <b>Input Arguments</b>  | <p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If array is already of class <code>uint16</code>, the <code>uint16</code> function has no effect.</p>                                                                                                                                                                                                 |
| <b>Output Arguments</b> | <p><b>intArray</b></p> <p>Array of class <code>uint16</code>. Values range from 0 to <math>2^{16} - 1</math>.</p> <p>The <code>uint16</code> function maps any values in array that are outside the limit to the nearest endpoint. For example,</p> <pre>uint16(2^16) % 2^16 = 65536</pre> <p>returns</p> <pre>ans =     65535</pre>                                                                         |
| <b>Examples</b>         | <p>Convert a double array to <code>uint16</code>:</p> <pre>mydata = uint16(magic(100));</pre>                                                                                                                                                                                                                                                                                                                |
| <b>Alternatives</b>     | <p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = uint16(zeros(100)); % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'uint16'); % Preferred</pre> |

# uint16

---

## See Also

[double](#) | [single](#) | [uint8](#) | [uint32](#) | [uint64](#) | [int8](#) | [int16](#) | [int32](#) | [int64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)



|                         |                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Convert to 32-bit unsigned integer                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Syntax</b>           | <code>intArray = uint32(array)</code>                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b>      | <code>intArray = uint32(array)</code> converts the elements of an array into unsigned 32-bit (4-byte) integers of class <code>uint32</code> .                                                                                                                                                                                                                                                                   |
| <b>Input Arguments</b>  | <p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If array is already of class <code>uint32</code>, the <code>uint32</code> function has no effect.</p>                                                                                                                                                                                                    |
| <b>Output Arguments</b> | <p><b>intArray</b></p> <p>Array of class <code>uint32</code>. Values range from 0 to <math>2^{32} - 1</math>.</p> <p>The <code>uint32</code> function maps any values in array that are outside the limit to the nearest endpoint. For example,</p> <pre>uint32(2^32)    % 2^32 = 4294967296</pre> <p>returns</p> <pre>ans =     4294967295</pre>                                                               |
| <b>Examples</b>         | <p>Convert a double array to <code>uint32</code>:</p> <pre>mydata = uint32(magic(100));</pre>                                                                                                                                                                                                                                                                                                                   |
| <b>Alternatives</b>     | <p>When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as <code>zeros</code>, <code>ones</code> or <code>eye</code>), rather than calling an integer conversion function. For example,</p> <pre>I = uint32(zeros(100));    % Creates an intermediate array</pre> <p>is not as efficient as</p> <pre>I = zeros(100, 'uint32'); % Preferred</pre> |

# uint32

---

## See Also

[double](#) | [single](#) | [uint8](#) | [uint16](#) | [uint64](#) | [int8](#) | [int16](#) | [int32](#) | [int64](#) | [intmax](#) | [intmin](#)

## Tutorials

- [“Integers”](#)
- [“Arithmetic Operations on Integer Classes”](#)

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Convert to 64-bit unsigned integer                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Syntax</b>           | <code>intArray = uint64(array)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Description</b>      | <code>intArray = uint64(array)</code> converts the elements of an array into unsigned 64-bit (8-byte) integers of class <code>uint64</code> .                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Tips</b>             | <p>Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, <code>double</code> values cannot represent all integers greater than <math>2^{53}</math> correctly. Before performing arithmetic operations on values larger than <math>2^{53}</math> in magnitude, convert the values to 64-bit integers. For example,</p> <pre>x = uint64(2^53+1);    % Floating-point arithmetic, loses precision</pre> <p>is not as accurate as the 64-bit integer arithmetic operation:</p> <pre>x = uint64(2^53) + 1; % Preferred</pre> |
| <b>Input Arguments</b>  | <p><b>array</b></p> <p>Array of any numeric class, such as <code>single</code> or <code>double</code>. If <code>array</code> is already of class <code>uint64</code>, the <code>uint64</code> function has no effect.</p>                                                                                                                                                                                                                                                                                                                               |
| <b>Output Arguments</b> | <p><b>intArray</b></p> <p>Array of class <code>uint64</code>. Values range from 0 to <math>2^{64} - 1</math>.</p> <p>The <code>uint64</code> function maps any values in <code>array</code> that are outside the limit to the nearest endpoint. For example,</p> <pre>uint64(2^64)    % 2^64 = 18446744073709551616</pre> <p>returns</p> <pre>ans =<br/>    18446744073709551615</pre>                                                                                                                                                                  |

# uint64

---

## 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                                                                                                                                    |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>matlab</code>   | All MATLAB files (with file extensions <code>*.m</code> , <code>*.mat</code> , <code>*.fig</code> , <code>*.mdl</code> , and <code>*.slx</code> .) |
| <code>load</code>     | All MAT-files ( <code>*.mat</code> )                                                                                                               |
| <code>figure</code>   | All figure files ( <code>*.fig</code> )                                                                                                            |
| <code>simulink</code> | All Simulink model files ( <code>*.mdl</code> and <code>*.slx</code> )                                                                             |
| <code>editor</code>   | All MATLAB files except for <code>.mat</code> , <code>.fig</code> , and <code>.slx</code> 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

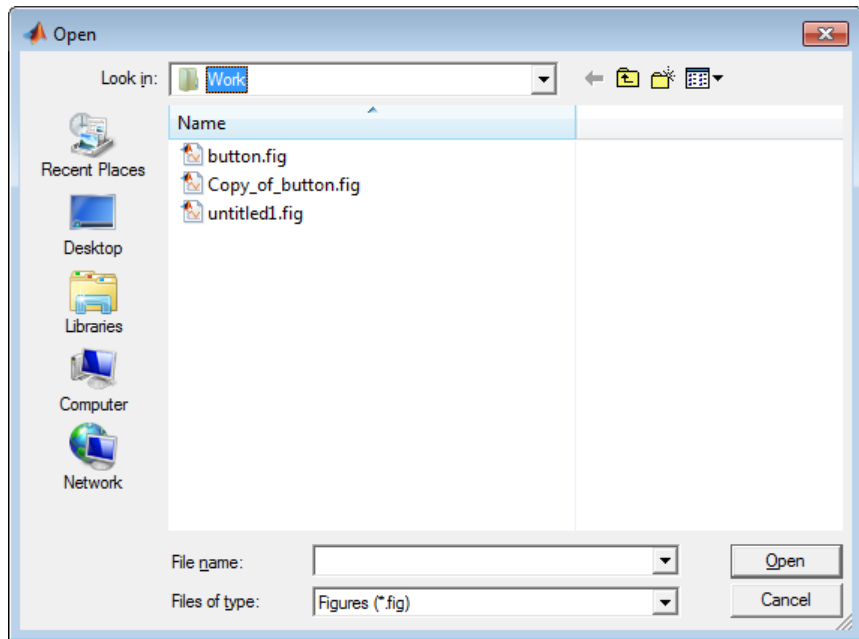
### **Modal Dialog**

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowState` 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.

# uiopen

---

## **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

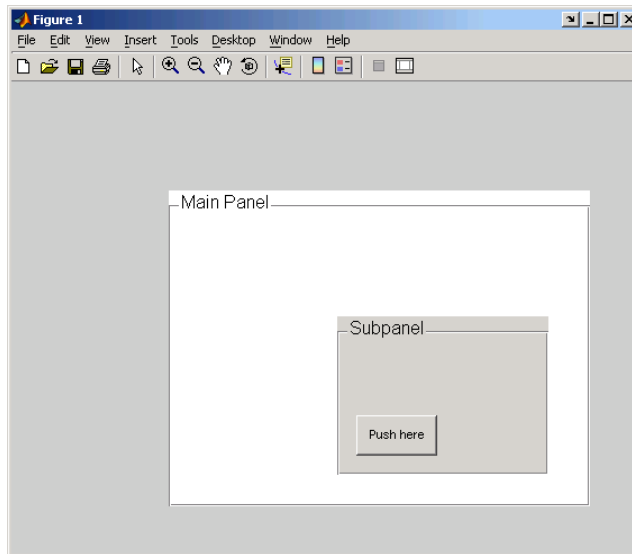
# uipanel

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 uipanel 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                             |
|------------------------------|-----------------------------------------|
| <code>BackgroundColor</code> | Color of the uipanel background         |
| <code>BeingDeleted</code>    | This object is being deleted            |
| <code>BorderType</code>      | Type of border around the uipanel area. |

# Uipanel Properties

| 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, uipanel, 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                                                                         |

# Uipanel Properties

---

| 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.<br><hr/> <b>Note</b> Controls the visibility of a uipanel and of its child axes, uibuttongroups, uipanel, and child uicontrols. Setting it does not change their Visible property. <hr/> |

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

# Uipanel Properties

---

*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, uipanel, 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 uipanel. 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 uipanel defines the same default property for all `uibuttongroups`.

---

# Uipanel Properties

---

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

# Uipanel Properties

---

*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

# Uipanel Properties

---

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”.

# Uipanel Properties

---

---

**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);
```

# Uipanel Properties

---

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 `uipanel`s resize before the figure itself does. Nested `uipanel`s resize from inner to outer, with child `ResizeFcns` being called before parent `ResizeFcns`.

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

# Uipanel Properties

---

*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 `uipanel`s, 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.

---

# uipushtool

---

**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

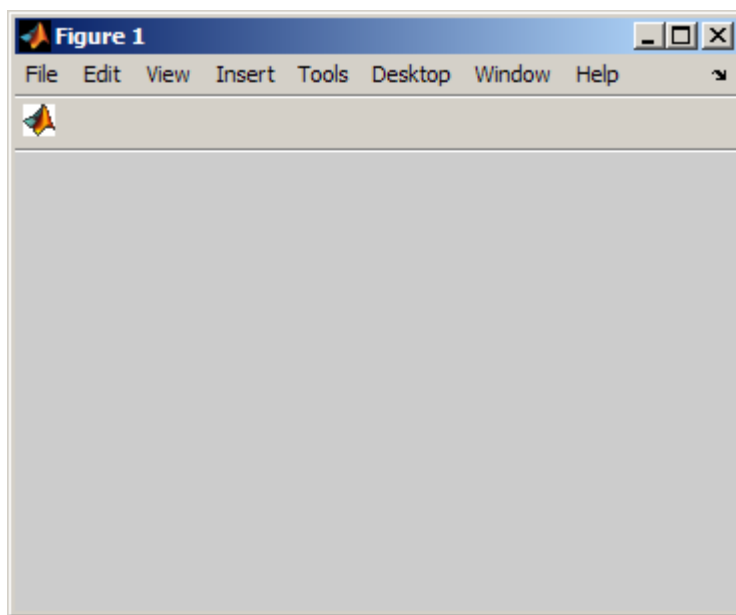
not display until you change the figure `WindowState` to `'normal'` or `'docked'`.

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 uicontrol 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.               |

# Uipushtool Properties

---

| 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

# Uipushtool Properties

---

*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.

# Uipushtool Properties

---

## 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`

# Uipushtool Properties

---

- 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

*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

# Uipushtool Properties

---

*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 2-4863 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.
```

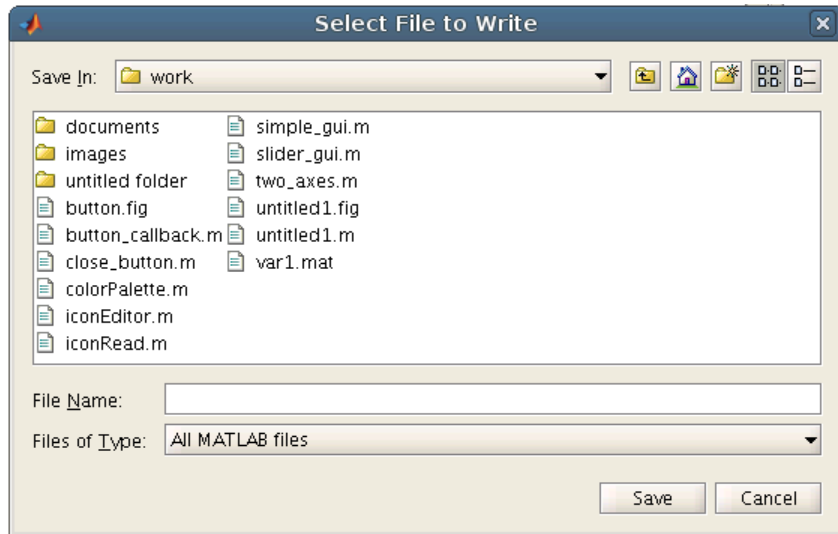
When typing into the dialog box, if you include either of the wildcard characters `'*'` or `'?'` in a file name, `uiputfile` does not respond to clicking **Save**. The dialog box remains open until you cancel it or remove the wildcard characters. This restriction applies to all platforms, even to file 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.

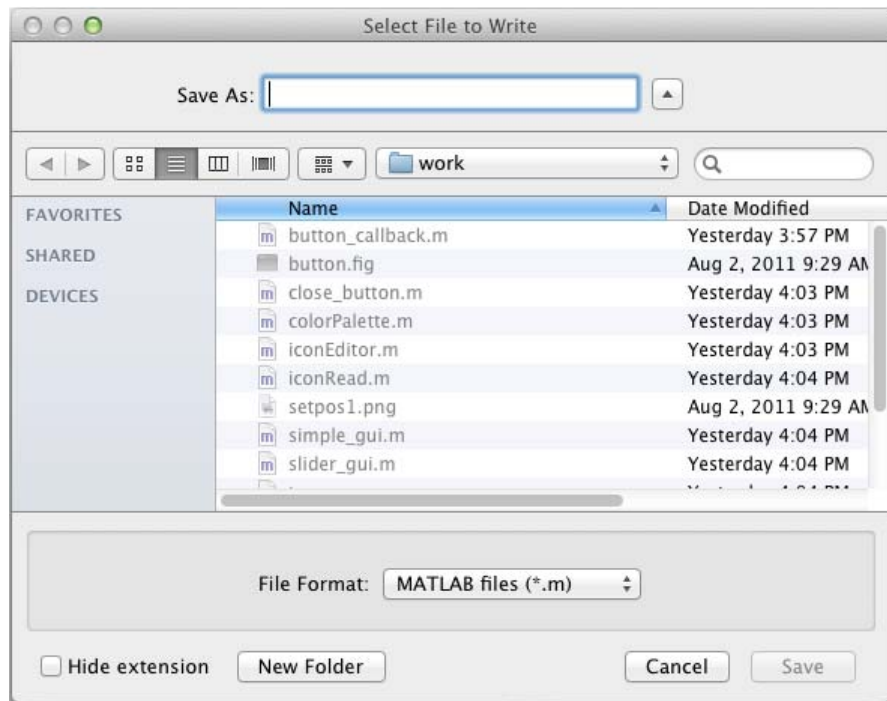
Select **Yes** to replace the existing file or **No** to return to the dialog to select another file name. Selecting **Yes** returns the name of the file. Selecting **No** returns 0.

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 `WindowState` in MATLAB Figure Properties.

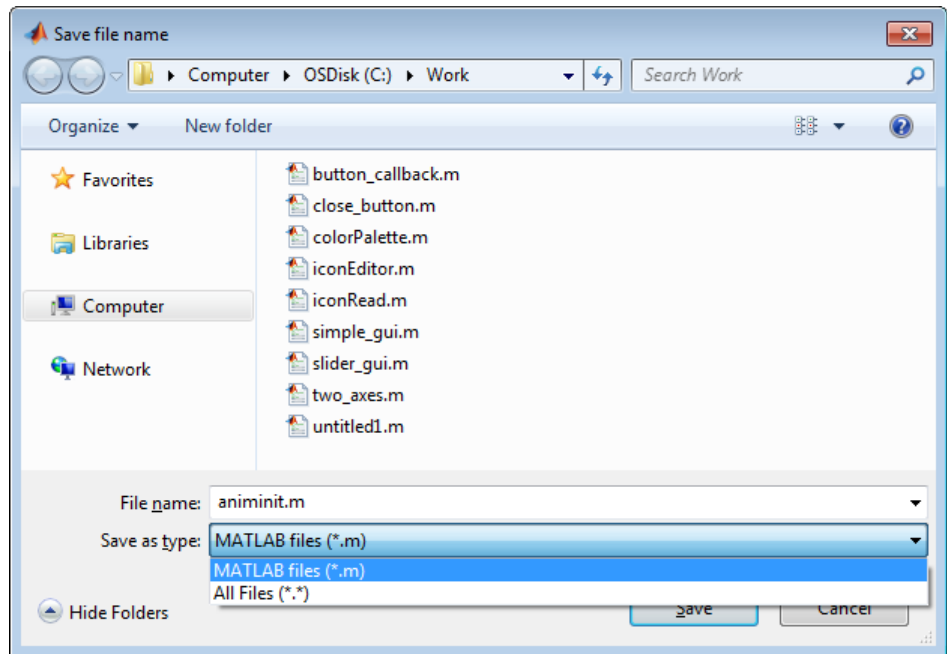
---

## 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');
```

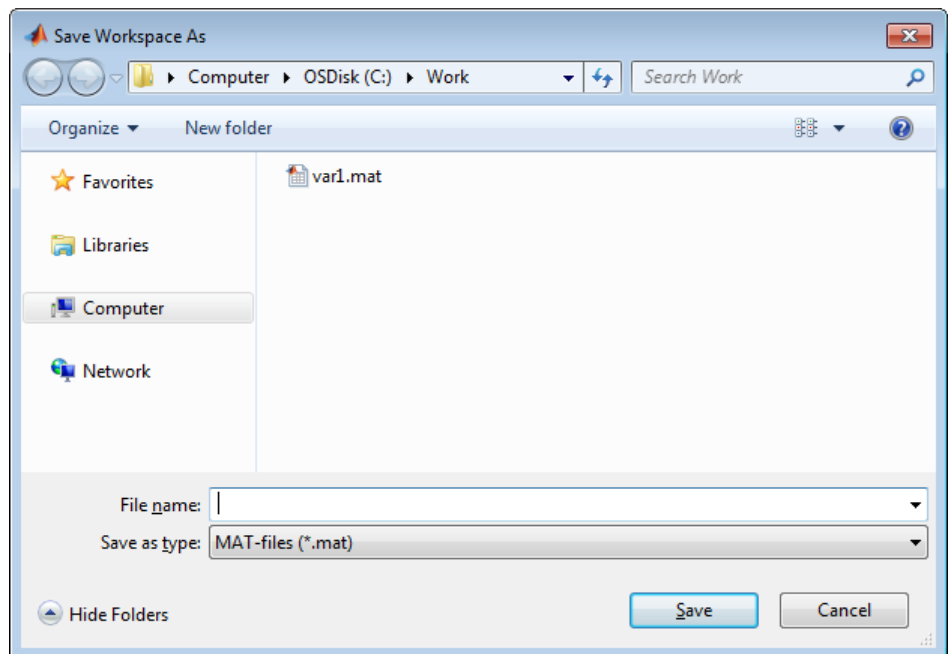
# uiputfile



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');
```

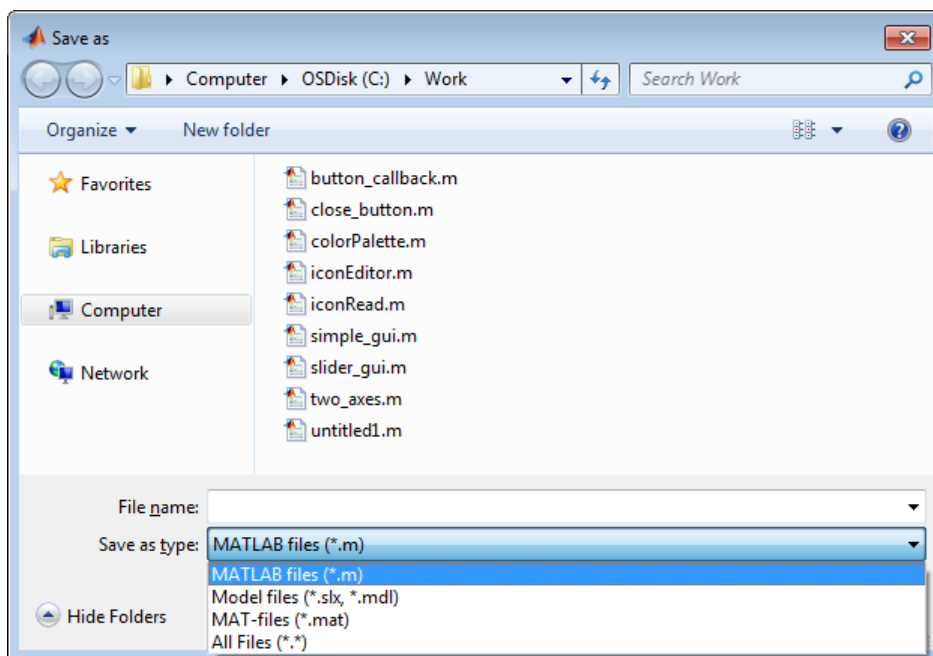




To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. `uiputfile` displays a default description for each known file type, such as "Model files" for Simulink `.mdl` and `.slx` files.

```
[filename, pathname] = uiputfile(...  
    {'*.m'; '*.slx'; '*.mat'; '*..*'},...  
    'Save as');
```

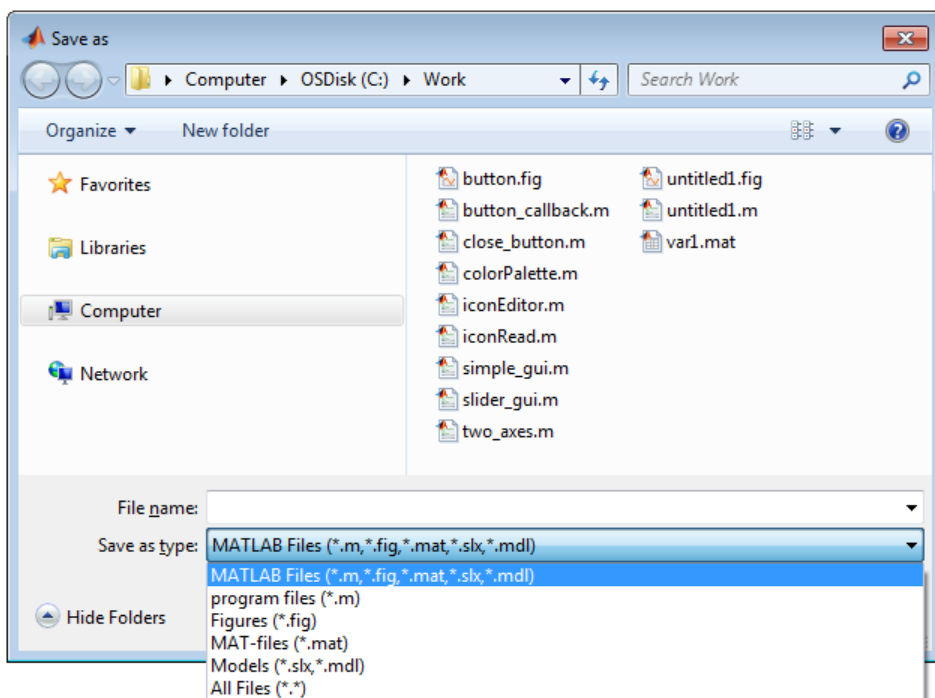
# uiputfile



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.

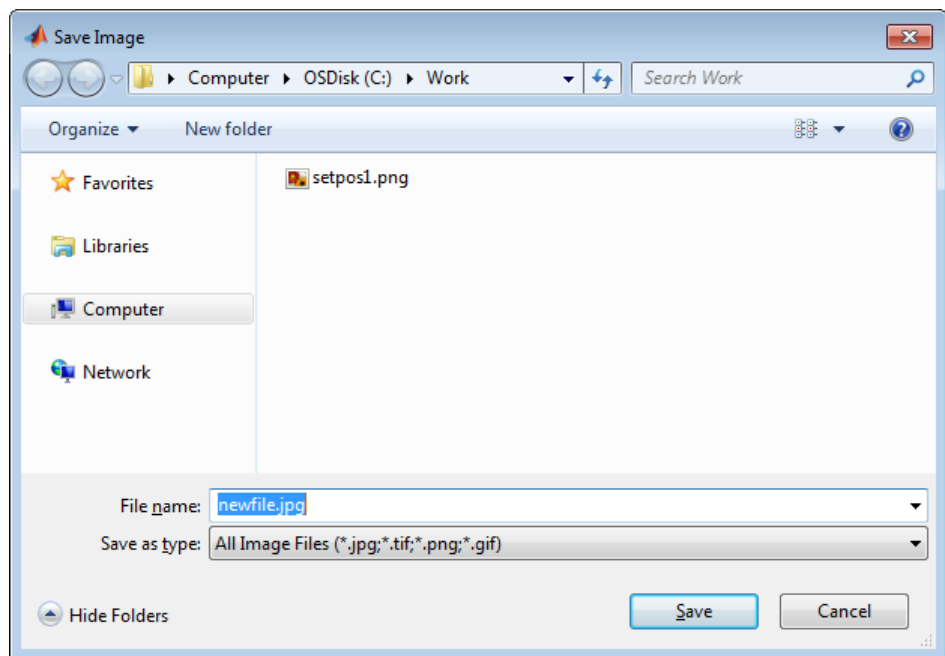
```
[filename, pathname] = uiputfile('*.m','Pick a MATLAB program file');
if isequal(filename,0) || isequal(pathname,0)
    disp('User selected Cancel')
else
```

# uiputfile

```
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](#)

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Resume execution of blocked program                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Syntax</b>      | <code>uiresume(h)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Description</b> | <code>uiresume(h)</code> resumes the program execution that <code>uiwait</code> suspended.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Tips</b>        | <p>The <code>uiwait</code> and <code>uiresume</code> functions block and resume MATLAB program execution. When creating a dialog box, you should have a <code>uicontrol</code> component with a callback that calls <code>uiresume</code> or a callback that destroys the dialog box. These are the only methods that resume program execution after the <code>uiwait</code> function blocks execution.</p> <p>When used in conjunction with a modal dialog box, <code>uiresume</code> can resume the execution of the program that <code>uiwait</code> suspended while presenting a dialog box.</p>                                                                                                                                                                                                                    |
| <b>Examples</b>    | <p>This example creates a GUI with a <b>Continue</b> push button. The example calls <code>uiwait</code> to block MATLAB execution until <code>uiresume</code> is called. This happens when the user clicks the <b>Continue</b> push button because the push button's <code>Callback</code>, which responds to the click, calls <code>uiresume</code>.</p> <pre>f = figure;<br/>h = uicontrol('Position',[20 20 200 40],'String','Continue',...<br/>            'Callback','uiresume(gcf)');<br/>disp('This will print immediately');<br/>uiwait(gcf);<br/>disp('This will print after you click Continue');<br/>close(f);</pre> <p><code>gcbf</code> is the handle of the figure that contains the object whose callback is executing.</p> <p>“Modal Dialog Box (GUIDE)” is a more complex example for a GUIDE GUI.</p> |
| <b>See Also</b>    | <code>dialog</code>   <code>figure</code>   <code>uicontrol</code>   <code>uimenu</code>   <code>uiwait</code>   <code>waitfor</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

# 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

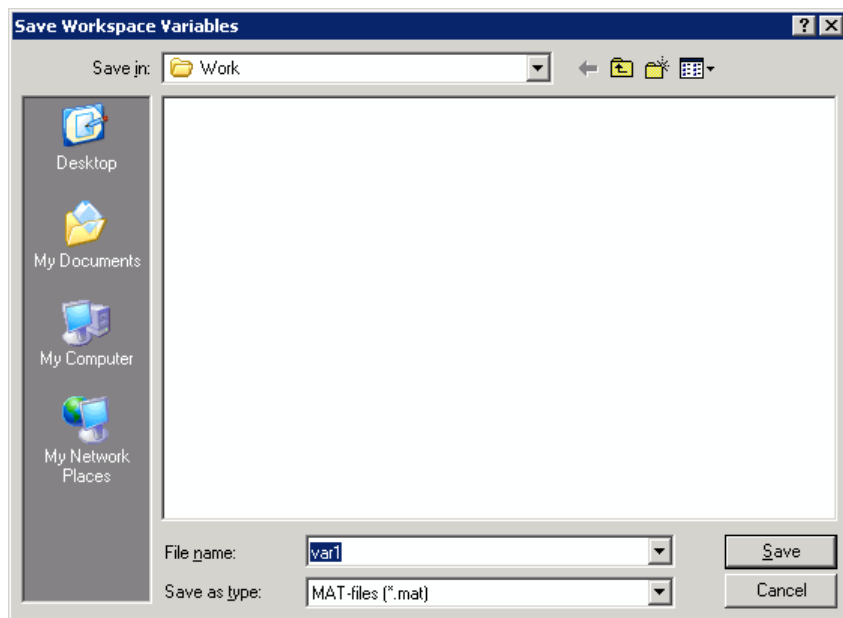
### Modal Dialog

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowState` 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 = uigetcolor
c = uigetcolor([r g b])
c = uigetcolor(h)
c = uigetcolor(...,'dialogTitle')
```

**Description**

`c = uigetcolor` 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 = uigetcolor([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 = uigetcolor(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 = uigetcolor(...,'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 `WindowState` in the MATLAB Figure Properties.

---

**See Also** ColorSpec

# uifont

---

**Purpose** Open standard dialog box for setting object's font characteristics

**Syntax**

```
uifont
uifont(h)
uifont(S)
uifont(..., 'DialogTitle')
S = uifont(...)
```

**Description** `uifont` 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.

`uifont` 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 `WindowState` in the MATLAB Figure Properties.

---

`uifont(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.

`uifont(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 `uisetpref`

**Syntax** `uisetpref('clearall')`

**Description** `uisetpref('clearall')` resets the value of all preferences registered through `uisetpref` to 'ask'. This causes the dialog box to display when you call `uisetpref`.

---

**Note** Use `setpref` to set the value of a particular preference to 'ask'.

---

**See Also** `setpref` | `uisetpref`

**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<sup>10</sup> system without a Java virtual machine (`matlab -nojvm`) or without a display (`matlab nodisplay`), no table generates and you receive an error.

10. 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

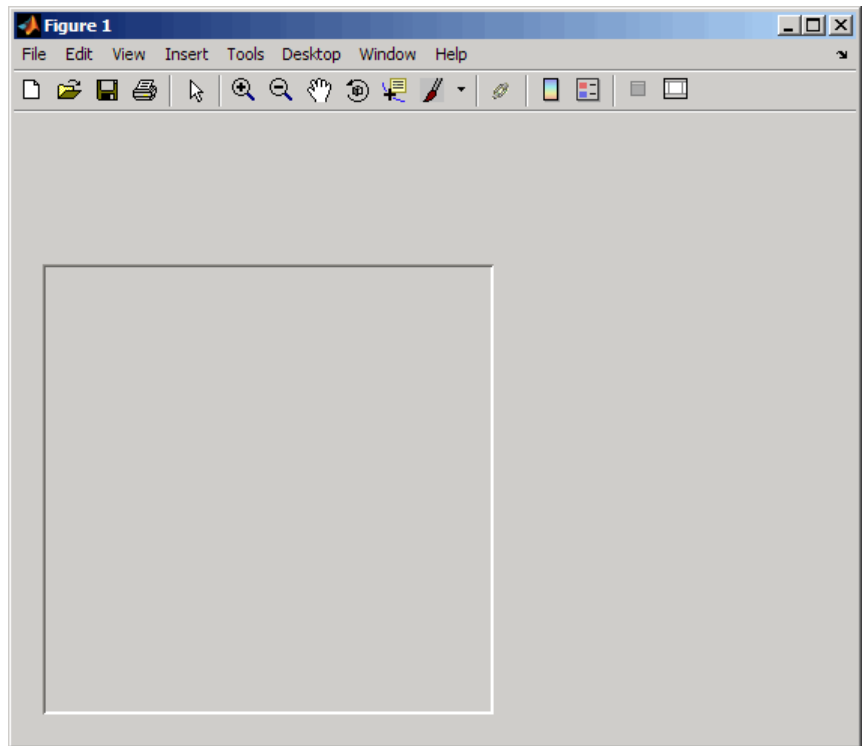
Create a table, provide magic-square data, set column widths uniformly, and specify the **uitable** **ColumnWidth** property as a cell array:

- 1 Create a table in the current figure. If no figure exists, one opens:

```
t = uitable;
```

# uitable

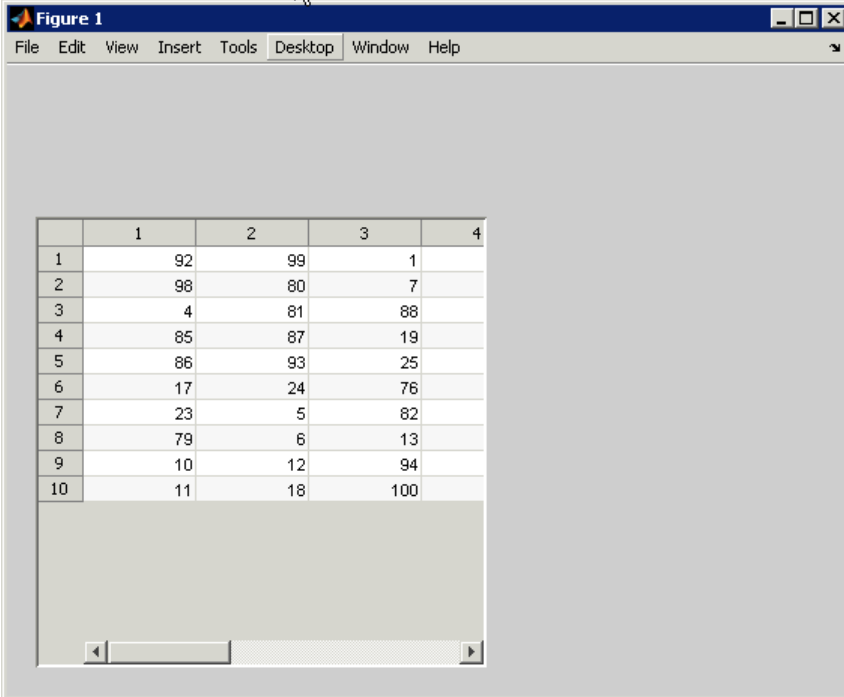
---



- 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))
```

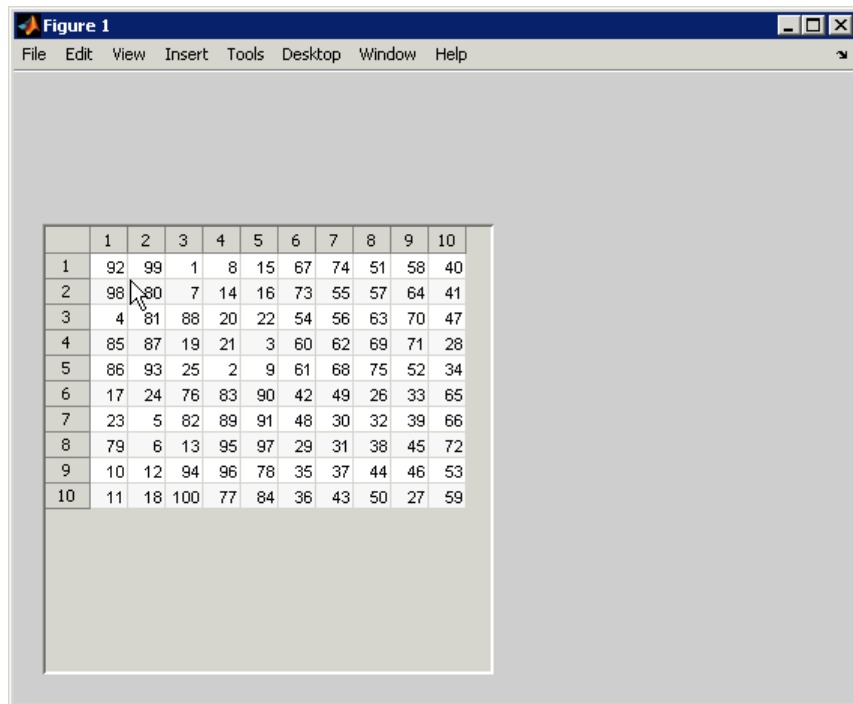




|    | 1  | 2  | 3   | 4 |
|----|----|----|-----|---|
| 1  | 92 | 99 | 1   |   |
| 2  | 98 | 80 | 7   |   |
| 3  | 4  | 81 | 88  |   |
| 4  | 85 | 87 | 19  |   |
| 5  | 86 | 93 | 25  |   |
| 6  | 17 | 24 | 76  |   |
| 7  | 23 | 5  | 82  |   |
| 8  | 79 | 6  | 13  |   |
| 9  | 10 | 12 | 94  |   |
| 10 | 11 | 18 | 100 |   |

- 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})
```



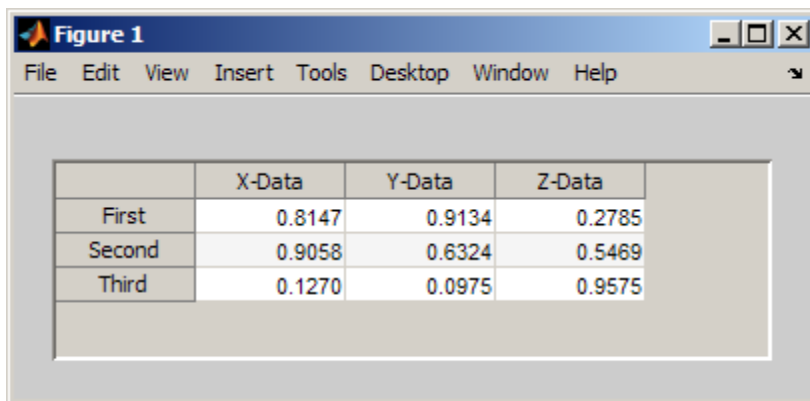
Cell arrays that specify `ColumnWidth` can contain:

- 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]);
```



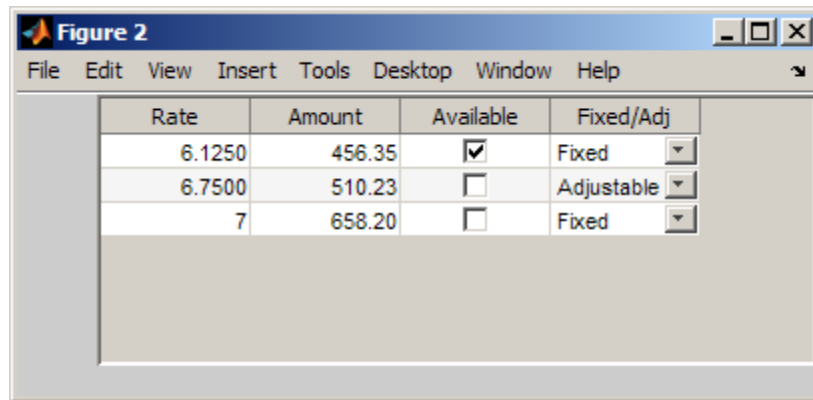
Create a table to contain a 3-by-4 array that contains numeric, logical, and string data, as follows:

- 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]);
```

# uitable

```
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', []);
```



| Rate   | Amount | Available                           | Fixed/Adj  |
|--------|--------|-------------------------------------|------------|
| 6.1250 | 456.35 | <input checked="" type="checkbox"/> | Fixed      |
| 6.7500 | 510.23 | <input type="checkbox"/>            | Adjustable |
| 7      | 658.20 | <input type="checkbox"/>            | Fixed      |

## 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. |

# Uitable Properties

| 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 editability 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                                |

# Uitable Properties

---

| 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.

# Uitable Properties

---

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, <code>[]</code> . |
| EditData                   | String                      | User-entered string.                                                                  |

# Uitable Properties

---

| <b>Event Data Structure Field</b> | <b>Type</b>                 | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NewData                           | 1-by-1 matrix or cell array | <p>Value that <code>uitable</code> wrote to <code>Data</code>. It is either the same as <code>EditData</code> or a converted value, for example, 2 where <code>EditData</code> is '2' and the cell is numeric.</p> <p>Empty if <code>uitable</code> detected an error in the user-entered data and did not write it to <code>Data</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Error                             | String                      | <p>Error that occurred when <code>uitable</code> tried to convert the <code>EditData</code> string into a value appropriate for <code>Data</code>. For example, <code>uitable</code> could not convert the <code>EditData</code> string consistent with the <code>Column Format</code> property, if any, or the data type for the changed cell.</p> <p>Empty if <code>uitable</code> wrote the value to <code>Data</code>.</p> <p>If <code>Error</code> is not empty, the <code>CellEditCallback</code> 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 <code>CellEditCallback</code> could convert it to its numerical equivalent and store it in <code>Data</code> without passing the error to the user.</p> |

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 in 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.*

# Editable Properties

---

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'      | <p>Displays a left-aligned string.</p> <p>To edit, the user types a string that replaces the existing string.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 'logical'   | <p>Displays a check box.</p> <p>To edit, the user checks or unchecks the check box. <code>uitable</code> sets the corresponding <code>Data</code> value to <code>true</code> or <code>false</code> accordingly.</p> <p>Initially, the check box is checked if the corresponding <code>Data</code> value would produce</p>                                                                                                                                                                                                                                                                                                                                                                               |
| 'numeric'   | <p>Displays a right-aligned string equivalent to the command window, for numeric data. If the cell <code>Data</code> value is boolean, then 1 or 0 is displayed. If the cell <code>Data</code> value is not numeric and not boolean, then NaN is displayed.</p> <p>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 <code>CellEditCallback</code>. The <code>uitable</code> function first attempts to convert the user-entered string to a numeric value and store it in <code>Data</code>. It then calls the <code>CellEditCallback</code>. See <code>CellEditCallback</code> for more information.</p> |

# Uitable Properties

---

| Cell Format                                                                         | Description                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1-by-n cell array of strings that define a pop-up menu, e.g., {'one' 'two' 'three'} | <p>Displays a pop-up menu.</p> <p>To edit, the user makes a selection from the pop-up menu. <code>uitable</code> sets the corresponding Data value to the selected menu item.</p> <p>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 <code>uitable</code> reference page.</p> |
| Valid string accepted by the format function, e.g., 'short' or 'bank'               | <p>Displays the Data value using the specified format. For example, for a two-column table, <code>set(htable, 'ColumnFormat', {'short', 'bank'})</code>.</p>                                                                                                                                                                                                                                                     |

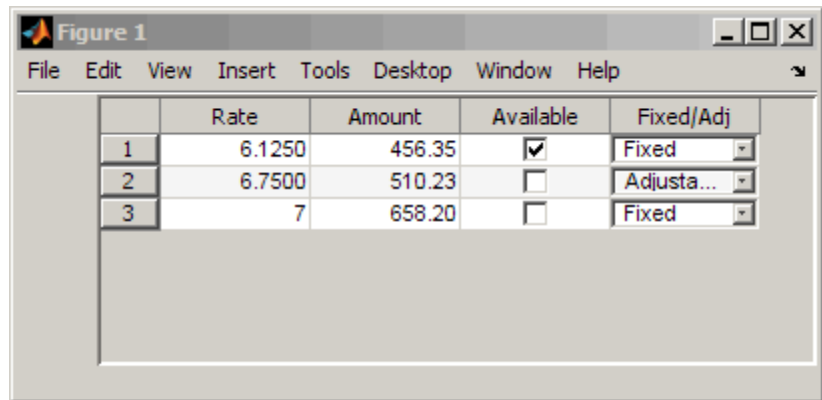
In some cases, you may need to insert an appropriate column in Data. If Data is a numerical or logical matrix, you must first convert it to a cell array using the `mat2cell` function.

## 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.

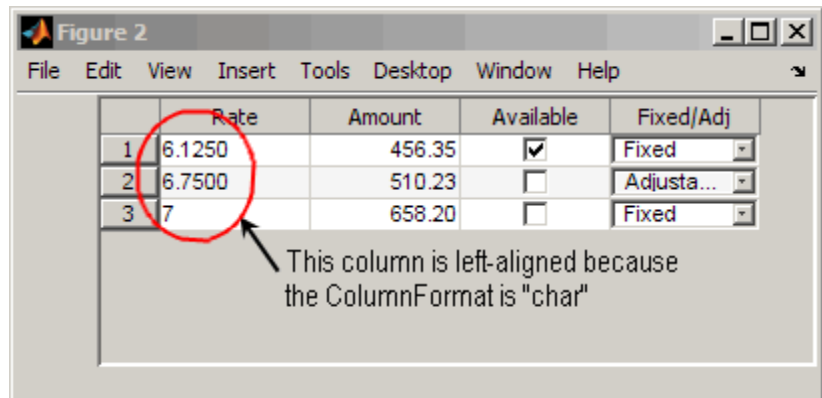


# Uitable Properties



|   | Rate   | Amount | Available                           | Fixed/Adj  |
|---|--------|--------|-------------------------------------|------------|
| 1 | 6.1250 | 456.35 | <input checked="" type="checkbox"/> | Fixed      |
| 2 | 6.7500 | 510.23 | <input type="checkbox"/>            | Adjusta... |
| 3 | 7      | 658.20 | <input type="checkbox"/>            | Fixed      |

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

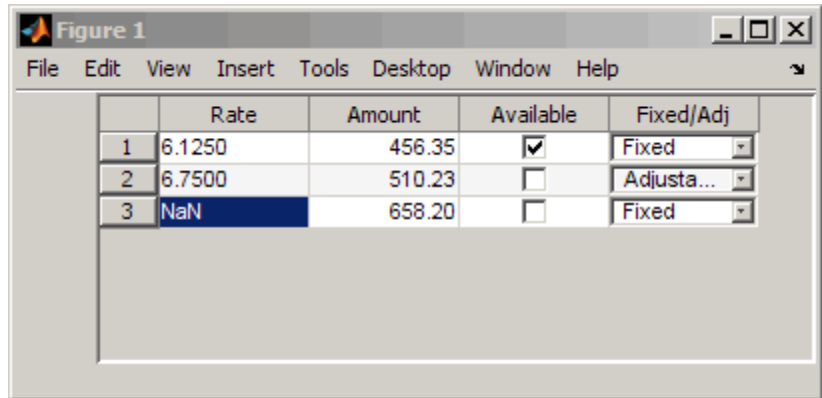


|   | Rate   | Amount | Available                           | Fixed/Adj  |
|---|--------|--------|-------------------------------------|------------|
| 1 | 6.1250 | 456.35 | <input checked="" type="checkbox"/> | Fixed      |
| 2 | 6.7500 | 510.23 | <input type="checkbox"/>            | Adjusta... |
| 3 | 7      | 658.20 | <input type="checkbox"/>            | Fixed      |

This column is left-aligned because the ColumnFormat is "char"

# Editable Properties

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**.



|   | Rate   | Amount | Available                           | Fixed/Adj  |
|---|--------|--------|-------------------------------------|------------|
| 1 | 6.1250 | 456.35 | <input checked="" type="checkbox"/> | Fixed      |
| 2 | 6.7500 | 510.23 | <input type="checkbox"/>            | Adjusta... |
| 3 | NaN    | 658.20 | <input type="checkbox"/>            | Fixed      |

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 |

## Uitable Properties

|                  |         |                                                                                                                                                  |                                                                                                                                                                                                     |                                                                                                                                                  |
|------------------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Data Type</b> | numeric | Values match. MATLAB displays numbers as is.                                                                                                     | MATLAB converts the text string entered to a double. See <code>str2double</code> for more information. If string cannot be converted, <b>NaN</b> is displayed.                                      | Does not work: warning is thrown.<br><hr/> <b>Note</b> If you have defined <code>CellEditCallback</code> , this warning will not be thrown <hr/> |
|                  | char    | MATLAB converts the entered number to a text string.                                                                                             | Values match. MATLAB displays the string as is.                                                                                                                                                     | Does not work: warning is thrown.<br><hr/> <b>Note</b> If you have defined <code>CellEditCallback</code> , this warning will not be thrown <hr/> |
|                  | logical | Does not work: warning is thrown.<br><hr/> <b>Note</b> If you have defined <code>CellEditCallback</code> , this warning will not be thrown <hr/> | If text string entered is <code>true</code> or <code>false</code> , 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.                                                                                |

# Uitable Properties

---

|  |  |  |                                                                                                 |  |
|--|--|--|-------------------------------------------------------------------------------------------------|--|
|  |  |  | <b>Note</b> If you have defined <code>CellEditCallback</code> , 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 `gcb0` 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(gcb0, ...
```

# Uitable Properties

---

```
'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).

# Uitable Properties

---

- `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')
```

# Uitable Properties

---

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 =  $\frac{1}{72}$  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

# Uitable Properties

---

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.

# Uitable Properties

---

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.

# Uitable Properties

---

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 rearrangeable 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

# Uitable Properties

---

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.

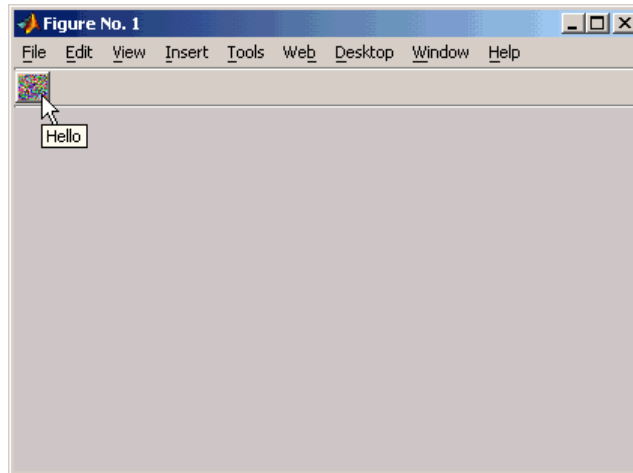
Toggle tools appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures with a `'modal'` `WindowStyle`. If the `WindowStyle` property of a figure containing a tool bar and its toggle tool children changes to `modal`, the toggle tools continue to exist

as Children of the tool bar. The toggle tools do not display until you change the `WindowState` 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. |

# Uitoggletool Properties property

| 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

# UIToggletool Properties property

---

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



# Uitoggletool Properties property

---

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.

## Uitoggletool Properties property

---

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.

# Uitoggletool Properties property

---

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

# Uitoggletool Properties property

---

*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.

# Uitoggletool Properties property

---

- 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

# Uitoggletool Properties property

---

*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.

# Uitoggletool Properties property

---

## 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)
```



# Uitoggletool Properties property

---

## Type

string (read-only)

Object class. This property identifies the kind of graphics object. For uitoggletool objects, Type is always the string 'uitoggletool'.

## UINavigationController

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.

# uitoolbar

---

**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**

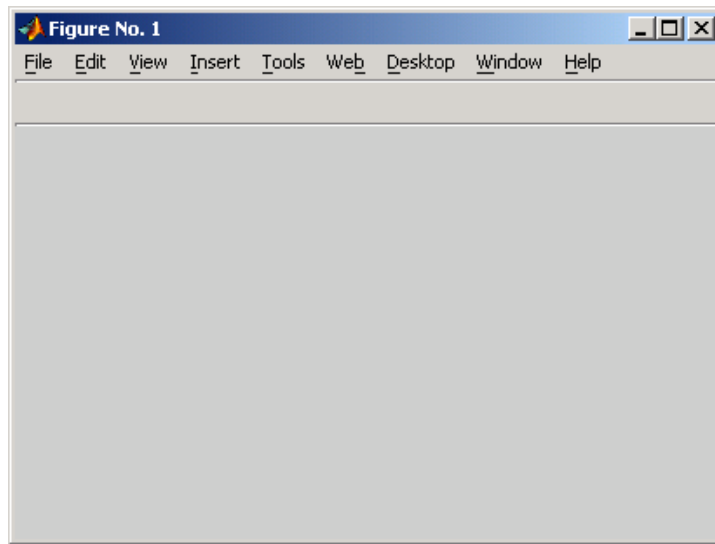
uitoolbar accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

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.

# Uitoolbar Properties

---

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 `gcb0` 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.

# Uitoolbar Properties

---

## 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`

# Uitoolbar Properties

---

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)

# Uitoolbar Properties

---

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.

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Block program execution and wait to resume                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Syntax</b>      | <pre>uiwait uiwait(h) uiwait(h,timeout)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p><code>uiwait</code> blocks execution until <code>uiresume</code> is called or the current figure is deleted. This syntax is the same as <code>uiwait(gcf)</code>.</p> <p><code>uiwait(h)</code> blocks execution until <code>uiresume</code> is called or the figure <code>h</code> is deleted.</p> <p><code>uiwait(h,timeout)</code> blocks execution until <code>uiresume</code> is called, the figure <code>h</code> is deleted, or <code>timeout</code> seconds elapse. The minimum value of <code>timeout</code> is 1. If <code>uiwait</code> receives a smaller value, it issues a warning and uses a 1 second <code>timeout</code>.</p>                                                                                                                |
| <b>Tips</b>        | <p>The <code>uiwait</code> and <code>uiresume</code> functions block and resume MATLAB and Simulink program execution. <code>uiwait</code> also blocks the execution of Simulink models. The functions <code>pause</code> (with no argument) and <code>waitfor</code> also block execution in this manner. <code>uiwait</code> is a convenient way to use the <code>waitfor</code> 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, <code>uiwait</code> can block the execution of the program file <i>and</i> restrict user interaction to the dialog only.</p> |
| <b>Examples</b>    | <p>This example creates a GUI with a <b>Continue</b> push button. The example calls <code>uiwait</code> to block MATLAB execution until <code>uiresume</code> is called. This happens when the user clicks the <b>Continue</b> push button because the push button's <code>Callback</code>, which responds to the click, calls <code>uiresume</code>.</p> <pre>f = figure; h = uicontrol('Position',[20 20 200 40],'String','Continue',...              'Callback','uiresume(gcf)'); disp('This will print immediately'); uiwait(gcf);</pre>                                                                                                                                                                                                                     |

# uiwait

---

```
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)”



|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Convert Unicode characters to numeric bytes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Syntax</b>      | <pre>bytes = unicode2native(unicodestr) bytes = unicode2native(unicodestr, encoding)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Description</b> | <p><code>bytes = unicode2native(unicodestr)</code> takes a char vector of Unicode characters, <code>unicodestr</code>, converts it to the MATLAB default character encoding scheme, and returns the bytes as a <code>uint8</code> vector, <code>bytes</code>. Output vector <code>bytes</code> has the same general array shape as the <code>unicodestr</code> input. You can save the output of <code>unicode2native</code> to a file using the <code>fwrite</code> function.</p> <p><code>bytes = unicode2native(unicodestr, encoding)</code> converts the Unicode characters to the character encoding scheme specified by the string <code>encoding</code>. <code>encoding</code> must be the empty string (<code>''</code>) or a name or alias for an encoding scheme. Some examples are <code>'UTF-8'</code>, <code>'latin1'</code>, <code>'US-ASCII'</code>, and <code>'Shift_JIS'</code>. For common names and aliases, see the Web site <a href="http://www.iana.org/assignments/character-sets">http://www.iana.org/assignments/character-sets</a>. If <code>encoding</code> is unspecified or is the empty string (<code>''</code>), the MATLAB default encoding scheme is used.</p> |
| <b>Examples</b>    | <p>This example begins with two strings containing Unicode characters. It assumes that string <code>str1</code> contains text in a Western European language and string <code>str2</code> 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 <code>unicode2native</code> to convert the two strings to the appropriate encoding schemes.</p> <pre>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);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>See Also</b>    | <code>native2unicode</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

# union

---

**Purpose** Find set union of two arrays

**Compatibility** In a future release, the behavior of union will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assessing the Impact of the Forthcoming Changes to union” on page 2-4952. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.

**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( __ , 'R2012a' )

[C,ia,ib] = union( __ , 'legacy' )

[C,ia,ib] = union(A,B,setOrder)

[C,ia,ib] = union(A,B, 'rows', setOrder)
```

**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')` returns the combined rows from A and B with no repetitions. The rows of the matrix C are in sorted order.

`[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( __ , 'R2012a' )` adopts the future behavior of the `union` function. `[C,ia,ib] = union( __ , 'legacy' )` preserves the current behavior. You can specify either flag, 'R2012a' or 'legacy', as the final input argument in any previous syntax.

`[C,ia,ib] = union(A,B,setOrder)` and `[C,ia,ib] = union(A,B,'rows',setOrder)` returns 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 then B.

## Input Arguments

### A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays of which to find the union, specified as `logical`, `char`, a cell array of strings, or any numeric class. A and B also can be any object with the class methods: `sort` (or `sortrows` for the 'rows' option), and `ne`. This includes 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`

**Complex Number Support:** Yes

### 'rows' - Rows flag

Rows flag, specified as 'rows', identifies the rows of A and B as individual elements. When you specify this flag, `union` returns the combined rows of A and B. This option does not support cell arrays.

# union

## setOrder - Order flag

'sorted' | '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, are returned in sorted order. For example:<br>C = union([5 5 3],[1 2],'sorted') returns<br>C = [1 2 3 5].                             |
| 'stable'   | The values (or rows) in C, are returned in the same order as they appear in A and B. For example:<br>C = union([5 5 3],[1 2],'stable') returns<br>C = [5 3 1 2]. |

## 'legacy' - Legacy behavior flag

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

## 'R2012a' - Future behavior flag

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior will have on your existing code.

## Output Arguments

### C - Combined values of A and B

vector | matrix

Combined values of A and B, returned as a vector or matrix. C is a vector unless you specify the 'rows' flag. If 'rows' is specified, C is a matrix containing the combined rows of A and B.

### ia - Index to A

vector

Index to A, returned a vector that indicates the values (or rows) in A that contribute to the union. If there is a repeated value (or row) appearing exclusively in A, then `ia` contains the index to the last occurrence of the value (or row).

If you call `union` with the `setOrder` or 'R2012a' argument, `ia` contains the index to the first occurrence of any repeated value (or row) in A.

### **ib - Index to B**

vector

Index to B, returned as a vector that indicates the values (or rows) in B that contribute to the union. If a value (or row) appears multiple times in B, then `ib` contains the index to the last occurrence of the value (or row). If a value (or row) appears in both A and B, then `ib` contains the index to the last occurrence in B.

If you call `union` with the `setOrder` or 'R2012a' argument, `ib` contains the index to the first occurrence of any repeated values (or rows).

## **Examples**

### **Union of Two Vectors**

```
A = [5 7 1]; B = [3 1 1];
C = union(A,B)
```

```
C =
```

```
     1     3     5     7
```

### **Find the Union of Two Vectors and Their Indices**

```
A = [5 7 1]; B = [3 1 1];
[C,ia,ib] = union(A,B)
```

```
C =
```

```
     1     3     5     7
```

# union

---

```
ia =
```

```
    1    2
```

```
ib =
```

```
    3    1
```

## Union of the Rows in Two Matrices

```
A = [2 2 2; 0 0 1];  
B = [1 2 3; 2 2 2; 2 2 2];  
[C,ia,ib] = union(A,B,'rows')
```

```
C =
```

```
    0    0    1  
    1    2    3  
    2    2    2
```

```
ia =
```

```
    2
```

```
ib =
```

```
    1  
    3
```

## 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` adopts the forthcoming behavior when you specify the `setOrder` argument. If you omit the `setOrder` argument, the output is sorted, but the size and content of `C`, `ia`, and `ib` conforms to the current behavior. Compare the sorted output here to the output from the example, “Find the Union of Two Vectors and Their Indices” on page 2-4949.

## Assessing the Impact of the Forthcoming Changes to `union`

Use the `'R2012a'` flag to assess the impact of the forthcoming behavior changes. Use the `'legacy'` flag to preserve the current behavior of your existing code.

Find the union of `A` and `B` with the current default behavior.

```
A = [5 7 1]; B = [3 1 1];  
[C1,ia1,ib1] = union(A,B)
```

```
C1 =
```

```
1 3 5 7
```

```
ia1 =
```

```
1 2
```

```
ib1 =
```

```
3 1
```

Find the union of `A` and `B`, and opt into the forthcoming behavior. In the future, this behavior will be the default.



```
A = [5 7 1]; B = [3 1 1];  
[C2,ia2,ib2] = union(A,B, 'R2012a')
```

```
C2 =
```

```
    1    3    5    7
```

```
ia2 =
```

```
    3  
    1  
    2
```

```
ib2 =
```

```
    1
```

Notice that `ia1` and `ia2` have different shapes and content. The same is also true for `ib1` and `ib2`.

Find the union of `A` and `B`, and preserve the current behavior.

```
A = [5 7 1]; B = [3 1 1];  
[C3,ia3,ib3] = union(A,B, 'legacy')
```

```
C3 =
```

```
    1    3    5    7
```

```
ia3 =
```

```
    1    2
```

```
ib3 =
```

# union

---

```
3 1
```

C3, ia3, and ib3 match C1, ia1, and ib1, respectively.

## Union of Vectors Containing NaNs

```
A = [5 NaN 1]; B = [4 NaN NaN];  
C = union(A,B)
```

```
C =
```

```
1 4 5 NaN NaN NaN
```

union treats NaN values as distinct.

## See Also

```
unique | intersect | ismember | issorted | setdiff | setxor  
| sort
```

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>       | Find unique values in array                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Compatibility</b> | In a future release, the behavior of <code>unique</code> will change. This change is introduced for adoption in R2012a. For a detailed demonstration that shows how you can preview the future behavior and preserve the current behavior of your existing code, see the example, “Assessing the Impact of the Forthcoming Changes to <code>unique</code> ” on page 2-4963. For a detailed explanation of all the forthcoming changes, see “Set Functions Changing Behavior in a Future Release”.                                             |
| <b>Syntax</b>        | <pre>C = unique(A) C = unique(A, 'rows') [C,ia,ic] = unique(A) [C,ia,ic] = unique(A, 'rows') [C,ia,ic] = unique(A,occurrence) [C,ia,ic] = unique(A, 'rows',occurrence)  [C,ia,ic] = unique( __ , 'R2012a')  [C,ia,ic] = unique( __ , 'legacy')  [C,ia,ic] = unique(A,setOrder)  [C,ia,ic] = unique(A, 'rows',setOrder)</pre>                                                                                                                                                                                                                  |
| <b>Description</b>   | <p><code>C = unique(A)</code> for the array <code>A</code>, returns the same values as in <code>A</code> but with no repetitions. The values of <code>C</code> are in sorted order.</p> <p><code>C = unique(A, 'rows')</code> for the matrix <code>A</code>, returns the unique rows of <code>A</code>. The rows of the matrix <code>C</code> are in sorted order.</p> <p><code>[C,ia,ic] = unique(A)</code> also returns index vectors <code>ia</code> and <code>ic</code>, such that <code>C = A(ia)</code> and <code>A = C(ic)</code>.</p> |

# unique

---

`[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,occurrence)` and `[C,ia,ic] = unique(A,'rows',occurrence)` specify which index is returned in `ia` when there are repeated values (or rows) in `A`. The default value is `occurrence = 'last'`, and `unique` returns the index of the last occurrence of each unique value (or row) in `A`. If `occurrence = 'first'`, `unique` returns the index of the first occurrence of each unique value (or row).

`[C,ia,ic] = unique(___, 'R2012a')` adopts the future behavior of the `unique` function. `[C,ia,ic] = unique(___, 'legacy')` preserves the current behavior. You can specify either flag, `'R2012a'` or `'legacy'`, as the final input argument in any previous syntax.

`[C,ia,ic] = unique(A,setOrder)` and `[C,ia,ic] = unique(A,'rows',setOrder)` returns `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`.

## Input Arguments

### **A - Input array**

vector | matrix | N-D array

Array in which to find unique values (or rows), specified as `logical`, `char`, a cell array of strings, or any numeric class. `A` also can be any object with the class methods: `sort` (or `sortrows` for the `'rows'` option), and `ne`. This includes heterogeneous arrays derived from the same root class.

### **Data Types**

double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char

**Complex Number Support:** Yes

**'rows' - Rows flag**

Rows flag, specified as `'rows'`, identifies the rows of matrix A as individual elements. When you specify this flag, `unique` returns the unique rows of A. This option does not support cell arrays.

**occurrence - Occurrence flag**

`'last' | 'first'`

Occurrence flag, specified as `'first'` or `'last'`, indicates whether `ia` should contain the first or last indices to repeated values found in C. The default value is `'last'` in most cases. However, if you call `unique` with either the `setOrder` or `'R2012a'` argument, the default occurrence is `'first'`.

| Occurrence Flag      | Meaning                                                                                                                                                                                                                               |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>'last'</code>  | If there are repeated values (or rows) in A, then <code>ia</code> contains the index to the last occurrence of the repeated value. For example:<br><code>[C,ia,ic] = unique([9 9 9], 'last')</code> returns<br><code>ia = 3.</code>   |
| <code>'first'</code> | If there are repeated values (or rows) in A, then <code>ia</code> contains the index to the first occurrence of the repeated value. For example:<br><code>[C,ia,ic] = unique([9 9 9], 'first')</code> returns<br><code>ia = 1.</code> |

**setOrder - Order flag**

`'sorted' | 'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in C.

# unique

---

| Order Flag | Meaning                                                                                                                                    |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| 'sorted'   | The values (or rows) in C, are returned in sorted order. For example:<br>C = unique([5 5 3 4], 'sorted') returns<br>C = [3 4 5].           |
| 'stable'   | The values (or rows) in C, are returned in the same order as in A. For example:<br>C = unique([5 5 3 4], 'stable') returns<br>C = [5 3 4]. |

## 'legacy' - Legacy behavior flag

Legacy behavior flag, when specified, preserves the current behavior. If the new behavior adversely affects your existing code, you can preserve the current behavior by specifying 'legacy' as the final argument.

## 'R2012a' - Future behavior flag

Future behavior flag, when specified, adopts the future behavior change. Specify 'R2012a' as the final argument to see what impact the new behavior has on your existing code.

## Output Arguments

### C - Unique values of A

vector | matrix

Unique values of A, returned as a vector or matrix. C is a vector, unless you specify the 'rows' flag. If 'rows' is specified, C is a matrix containing the unique rows of A.

### ia - Index to A

vector

Index to A, returned as a vector that contains indices such that C = A(ia).

### ic - Index to C

vector

Index to C, returned as a vector that contains indices such that  $A = C(ic)$ .

## Examples

### Unique Values in a Vector

```
A = [9 2 9 5];
C = unique(A)
```

```
C =
```

```
     2     5     9
```

### Find Unique Values and Their Indices

```
A = [9 2 9 5];
[C, ia, ic] = unique(A)
```

```
C =
```

```
     2     5     9
```

```
ia =
```

```
     2     4     3
```

```
ic =
```

```
     3     1     3     2
```

### Unique Rows in a Matrix

```
A = [9 2 9 5; 9 2 9 0; 9 2 9 5];
[C, ia, ic] = unique(A, 'rows')
```

```
C =
```

```
     9     2     9     0
```

# unique

---

```
          9    2    9    5

ia =

     2
     3

ic =

     2
     1
     2
```

Notice that the second row in *A* is identified as unique even though 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` adopts the forthcoming behavior when you specify the `setOrder` argument. If you omit the `setOrder` argument, the output is sorted, but the size and content of `C`, `ia`, and `ic` conforms to the current behavior. Compare the sorted output here to the output from the example, “Find Unique Values and Their Indices” on page 2-4959.

## Unique Values in a Vector with Repeated Items

Specify whether `ia` should contain the first or last index to repeated values.

Specify the first occurrence of 9 in A.

```
A = [9 2 9 5];  
[C1, ia1, ic1] = unique(A, 'first')
```

```
C1 =
```

```
     2     5     9
```

```
ia1 =
```

```
     2     4     1
```

```
ic1 =
```

```
     3     1     3     2
```

Alternatively, you can specify the last occurrence of 9 in A.

```
A = [9 2 9 5];  
[C2, ia2, ic2] = unique(A, 'last')
```

```
C2 =
```

```
     2     5     9
```

```
ia2 =
```

```
     2     4     3
```

```
ic2 =  
      3      1      3      2
```

### Assessing the Impact of the Forthcoming Changes to unique

Use the 'R2012a' flag to assess the impact of the forthcoming behavior changes. Use the 'legacy' flag to preserve the current behavior of your existing code.

Find the unique elements of A with the current default behavior.

```
A = [9 2 9 5];  
[C1, ia1, ic1] = unique(A)
```

```
C1 =  
      2      5      9
```

```
ia1 =  
      2      4      3
```

```
ic1 =  
      3      1      3      2
```

Find the unique elements of A, and opt into the forthcoming behavior. In the future, this behavior will be the default.

```
[C2, ia2, ic2] = unique(A, 'R2012a')
```

```
C2 =  
      2      5      9
```

# unique

---

```
ia2 =
```

```
2  
4  
1
```

```
ic2 =
```

```
3  
1  
3  
2
```

`ia1` and `ia2` have different shapes and content, while `ic1` and `ic2` differ only in shape.

Find the unique elements of `A`, and preserve the current behavior.

```
[C3, ia3, ic3] = unique(A, 'legacy')
```

```
C3 =
```

```
2    5    9
```

```
ia3 =
```

```
2    4    3
```

```
ic3 =
```

```
3    1    3    2
```

`C3`, `ia3`, and `ic3` match `C1`, `ia1`, and `ic1` respectively.

## Unique Values in Array Containing NaNs

```
A = [5 5 NaN NaN];  
C = unique(A)
```

```
C =  
  
     5     NaN     NaN
```

unique treats NaN values as distinct.

### See Also

```
union | intersect | ismember | issorted | setdiff | setxor  
| sort
```

**Purpose** Execute UNIX command and return result

**Syntax**

```
unix command
status = unix('command')
[status, result] = unix('command')
[status,result] = unix('command','-echo')
```

**Description** `unix` command calls upon the UNIX operating system to execute the given command. The command executes in a UNIX shell, which might not be the shell from which you launched MATLAB.

`status = unix('command')` returns completion status to the `status` variable.

`[status, result] = unix('command')` returns the standard output to the `result` variable.

`[status,result] = unix('command','-echo')` displays the results in the Command Window as it executes, and assigns the results to `result`.

This function is interchangeable with the `system` and `dos` functions. They all have the same effect.

**Examples** List all users that are currently logged in.

```
[s,w] = unix('who');
```

MATLAB returns 0 (success) in `s` and a string containing the list of users in `w`.

---

Execute a string, `why`, that is not a UNIX command.

```
[s,w] = unix('why')
s =
    1
w =
why: Command not found.
```

MATLAB returns a nonzero value in `s` to indicate failure, and returns an error message in `w`.

**Algorithms**

The MATLAB software 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

**How To**

- “Running External Commands, Scripts, and Programs”

# unloadlibrary

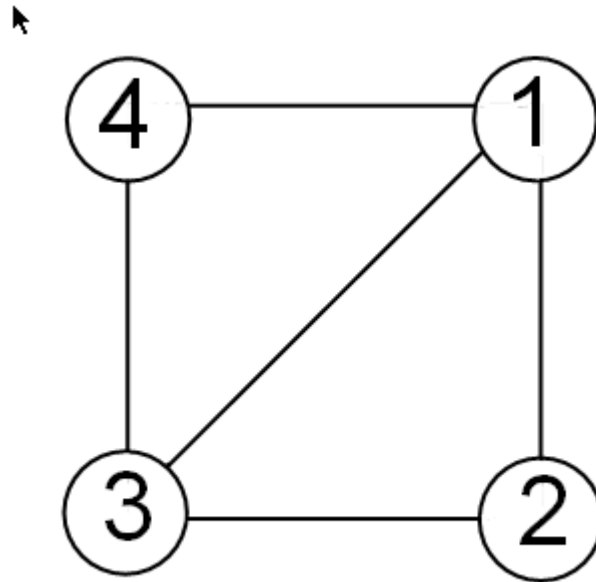
---

|                        |                                                                                                                                                                                                                                                                                                                                    |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Unload shared library from memory                                                                                                                                                                                                                                                                                                  |
| <b>Syntax</b>          | <code>unloadlibrary libname</code>                                                                                                                                                                                                                                                                                                 |
| <b>Description</b>     | <code>unloadlibrary libname</code> unloads library <code>libname</code> from memory.                                                                                                                                                                                                                                               |
| <b>Limitations</b>     | <ul style="list-style-type: none"><li>• Use with libraries that are loaded using the <code>loadlibrary</code> function.</li></ul>                                                                                                                                                                                                  |
| <b>Input Arguments</b> | <p><b>libname</b> - Name of shared library</p> <p>string</p> <p>Name of shared library, specified as a string. If you call <code>loadlibrary</code> using the <code>alias</code> option, then you must use the alias name for the <code>libname</code> argument.</p>                                                               |
| <b>Examples</b>        | <p><b>Unload Library</b></p> <p>Add path to examples folder.</p> <pre>addpath(fullfile(matlabroot,'extern','examples','shrlib'))</pre> <p>Load the library, if it is not already loaded.</p> <pre>if ~libisloaded('shrlibsample')     loadlibrary('shrlibsample') end</pre> <p>Clean up.</p> <pre>unloadlibrary shrlibsample</pre> |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• To unload a MEX-file, use the <code>clear</code> function.</li></ul>                                                                                                                                                                                                                       |
| <b>See Also</b>        | <code>loadlibrary</code>   <code>libisloaded</code>   <code>clear</code>                                                                                                                                                                                                                                                           |



---

|                         |                                                                                                                                                                                                                                                                                                    |                |                                                   |                 |                                |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|---------------------------------------------------|-----------------|--------------------------------|
| <b>Purpose</b>          | Convert edge matrix to coordinate and Laplacian matrices                                                                                                                                                                                                                                           |                |                                                   |                 |                                |
| <b>Syntax</b>           | <code>[L,XY] = unmesh(E)</code>                                                                                                                                                                                                                                                                    |                |                                                   |                 |                                |
| <b>Description</b>      | <code>[L,XY] = unmesh(E)</code> returns the Laplacian matrix <code>L</code> and mesh vertex coordinate matrix <code>XY</code> for the <code>M</code> -by-4 edge matrix <code>E</code> . Each row of the edge matrix must contain the coordinates <code>[x1 y1 x2 y2]</code> of the edge endpoints. |                |                                                   |                 |                                |
| <b>Input Arguments</b>  | <table><tr><td><code>E</code></td><td><code>M</code>-by-4 edge matrix <code>E</code>.</td></tr></table>                                                                                                                                                                                            | <code>E</code> | <code>M</code> -by-4 edge matrix <code>E</code> . |                 |                                |
| <code>E</code>          | <code>M</code> -by-4 edge matrix <code>E</code> .                                                                                                                                                                                                                                                  |                |                                                   |                 |                                |
| <b>Output Arguments</b> | <table><tr><td><code>L</code></td><td>Laplacian matrix representation of the graph.</td></tr><tr><td><code>XY</code></td><td>Mesh vertex coordinate matrix.</td></tr></table>                                                                                                                      | <code>L</code> | Laplacian matrix representation of the graph.     | <code>XY</code> | Mesh vertex coordinate matrix. |
| <code>L</code>          | Laplacian matrix representation of the graph.                                                                                                                                                                                                                                                      |                |                                                   |                 |                                |
| <code>XY</code>         | Mesh vertex coordinate matrix.                                                                                                                                                                                                                                                                     |                |                                                   |                 |                                |
| <b>Examples</b>         | 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.

# unmesh

---

XY

XY =

```
-1  -1
-1   1
 1  -1
 1   1
```

## See Also

[gplot](#) | [treepplot](#)

**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`

# unregisterallevents

---

**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'; 'Db1Click' '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](#)

# unregisterevent

---

**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'  
    'DbClick'       '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.unregister(event({'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

## unregisterevent

---

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”

# untar

---

**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 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('myfiles.tar.gz','*.m');  
untar('myfiles','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

# unwrap

---

**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  $\pi$  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,  $\pi$ .

`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  $\pi$  has the same effect as a tolerance of  $\pi$ . For a tolerance less than  $\pi$ , if a jump is greater than the tolerance but less than  $\pi$ , 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  $\pi$ , try using a finer grid in the domain.

---

## Examples

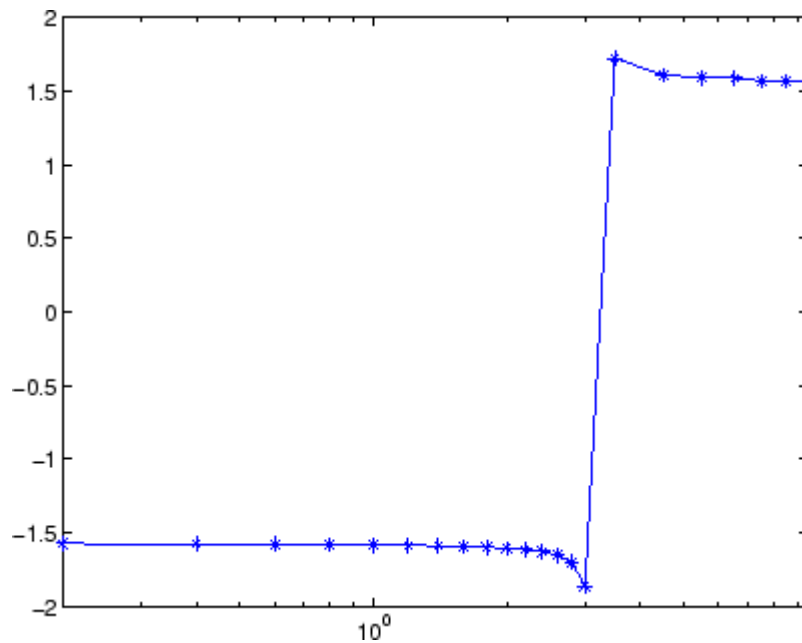
### Example 1

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
```

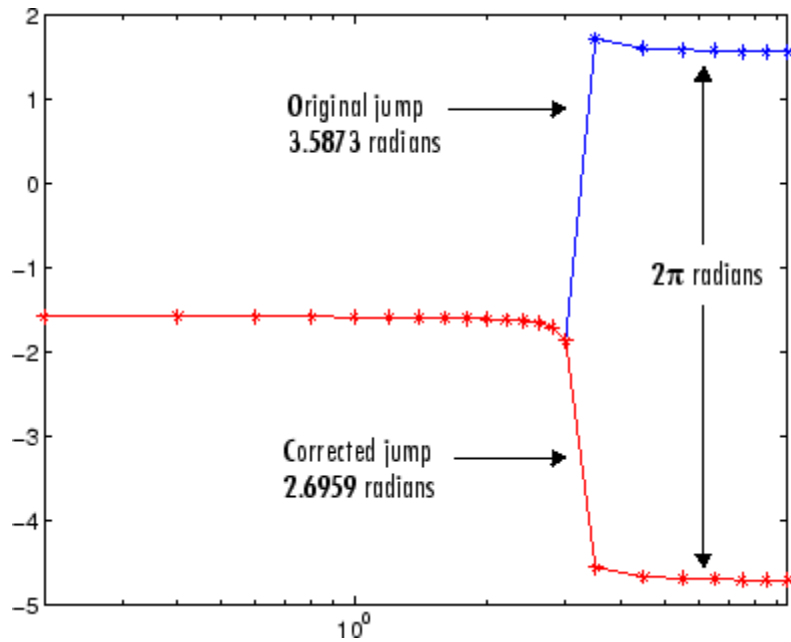
# unwrap



Using `unwrap` to correct the phase angle, the resulting jump is 2.6959, which is less than the default jump tolerance  $\pi$ . 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  $Q = \text{unwrap}(P)$  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
```

# unwrap

---

## See Also

[abs](#) | [angle](#)

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | Extract contents of zip file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>          | <pre>unzip(zipfilename) unzip(zipfilename,outputdir) filenames = unzip(zipfilename,outputdir)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b>     | <p><code>unzip(zipfilename)</code> extracts the archived contents of <code>zipfilename</code> into the current folder, preserving the files' attributes and timestamps. The <code>unzip</code> function can extract files from your local system or files from an Internet URL.</p> <p><code>unzip(zipfilename,outputdir)</code> extracts the contents of <code>zipfilename</code> into the folder <code>outputdir</code>.</p> <p><code>filenames = unzip(zipfilename,outputdir)</code> returns the names of the extracted files in the string cell array <code>filenames</code>. Specifying <code>outputdir</code> is optional.</p> |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• <code>unzip</code> does not support password-protected or encrypted zip archives.</li><li>• 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, <code>unzip</code> overwrites the existing files with the archived versions. If you do not have write permission, <code>unzip</code> issues a warning.</li><li>• Extract files that contain non-7-bit ASCII characters on a machine that has the appropriate language/encoding settings.</li></ul>                                                                    |
| <b>Input Arguments</b> | <p><b>zipfilename</b></p> <p>String that specifies the name of the zip file.</p> <p>If <code>zipfilename</code> does not include the full path, <code>unzip</code> searches for the file in the current folder and along the MATLAB path. If you do not specify the file extension, <code>unzip</code> appends <code>.zip</code>.</p> <p>If you are downloading a URL, <code>zipfilename</code> must include the protocol type (for example, <code>http://</code>). The <code>unzip</code> function downloads the URL to the temporary folder on your system, and deletes the URL on cleanup.</p>                                    |

# unzip

---

## **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, `entrynames` contains the relative path. If `outputdir` specifies an absolute path, `entrynames` 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`

**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**

# upper

---

**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, 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`.

**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.

## **Timeout - Timeout duration**

number

Timeout duration in seconds, specified as the comma-separated pair consisting of 'Timeout' and a number. 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'

## **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'

## 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 urldread.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
           '?term=urldread'];
str = urldread(fullURL);
```

urldread 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”

**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, 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`.

**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.

## **Timeout - Timeout duration**

number

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a number. 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'

### **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'

## **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”

# usejava

---

**Purpose** Determine whether Java feature is supported in MATLAB software

**Syntax** `usejava(feature)`

**Description** `usejava(feature)` returns 1 if the specified feature is supported and 0 otherwise.

The following table shows the valid feature arguments.

| Feature   | Description                                                                                      |
|-----------|--------------------------------------------------------------------------------------------------|
| 'awt'     | Java GUI components in the Abstract Window Toolkit 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

The following conditional code ensures that the AWT GUI components are available before the script attempts to display a Java Frame.

```
if usejava('awt')
    myFrame = java.awt.Frame;
else
    disp('Unable to open a Java Frame');
end
```

The next example is part of a script that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to JVM software.



```
if ~usejava('jvm')
    error([mfilename ' requires Java to run.']);
end
```

## See Also

javachk

# 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 user portion of the search path. The user portion of the search path is the first folder on the search path, above the folders supplied by MathWorks. The default folder is Documents/MATLAB on all platforms except Windows platforms released before Windows Vista™—for those Windows platforms, the default is My Documents/MATLAB. By default, the *userpath* folder is the current folder when you start up MATLAB — that is, it is the default startup folder. For information on specifying a different folder for the startup folder, see “Startup Folder for the MATLAB Program”. On Macintosh and UNIX platforms, you can automatically add additional subfolders to the top of the search path upon startup by specifying the path for the subfolders via the MATLABPATH environment variable.

`userpath('newpath')` sets the *userpath* value 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. `newpath` must be an absolute path.

`userpath('reset')` sets the *userpath* value 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.

`userpath('clear')` clears the value for *userpath*. MATLAB removes the folder previously specified by *userpath* from the search path.

## Examples

- “Viewing `userpath`” on page 2-5003
- “Setting a New Value for `userpath`” on page 2-5004

- “Clearing the Value for userpath, and Specifying a New Startup Folder on Windows Platforms” on page 2-5004
- “Removing userpath from the Search Path Resets the Startup Folder” on page 2-5006
- “Assigning userpath as the Startup Folder on a UNIX or Macintosh Platform” on page 2-5008
- “Adding Folders to the Search Path Upon Startup on a UNIX or Macintosh Platform” on page 2-5009

## Viewing userpath

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:

```
cd
```

MATLAB returns:

```
H:\My Documents\MATLAB
```

where H is the drive at which `My Documents` is located for this example. Confirm the current folder is the *userpath*:

```
userpath
```

MATLAB returns:

```
H:\My Documents\MATLAB;
```

Display the search path:

```
path
```

MATLAB returns the search path. The *userpath* portion is at the top:

```
MATLABPATH
```

```
H:\My Documents\MATLAB
```

```
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
```

## Setting a New Value for userpath

This example assumes `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`:

```
userpath('C:\Research_Project')
```

View the effect of the change on the search path:

```
path
```

MATLAB displays the search path, with the new value for `userpath` portion at the top:

```
MATLABPATH
```

```
C:\Research_Project  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops  
...
```

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`. The next time you start MATLAB, the current folder will be `C:\Research_Project` on Windows platforms.

## Clearing the Value for userpath, and Specifying a New Startup Folder on Windows Platforms

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
```

MATLAB returns:

```
H:\My Documents\MATLAB;
```

Verify that the *userpath* folder is at the top of the search path:

```
path
```

MATLAB returns:

```
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
```

MATLAB returns:

```
ans =  
    ''
```

Confirm the *userpath* folder was removed from the search path:

```
path
```

MATLAB returns

```
MATLABPATH
```

```
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops  
...
```

Clearing the *userpath* value might affect the startup folder. Unless you otherwise specify the startup folder, the startup folder on Windows platforms will be the folder from which you invoke MATLAB. If you specify a startup folder, such as described in “Changing the Startup Folder”, that startup folder will *not* be on the search path. Note that you do not have to clear *userpath* to specify a different startup folder; when you otherwise specify a startup folder, the *userpath* folder is added to the search path upon startup, but is not the startup folder.

## **Removing *userpath* from the Search Path Resets the Startup Folder**

In this example, *userpath* is set to the default value and you remove the *userpath* folder from the search path, then save the changes. This has the same effect as clearing the value for *userpath*. Confirm the default is currently set:

```
userpath
```

```
MATLAB returns:
```

```
H:\My Documents\MATLAB;
```

View the *userpath* folder at the top of the search path:

```
path
```

```
MATLAB returns:
```

```
MATLABPATH
```

```
H:\My Documents\MATLAB  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general  
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops  
...
```

Remove H:\My Documents\MATLAB from the search path and confirm the result:

```
rmpath('H:\My Documents\MATLAB')
```

path

MATLAB returns:

MATLABPATH

```
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Verify the value:

userpath

MATLAB returns:

```
H:\My Documents\MATLAB;
```

Save changes to the search path:

savepath

View the value:

userpath

MATLAB returns:

```
ans =
     ''
```

The value is now cleared. Removing the folder from the search path *and* saving the changes to the path has the same effect as clearing the value for `userpath`. At the next startup, the startup folder will *not* be `H:\My Documents\MATLAB`, and `H:\My Documents\MATLAB` will *not* be on the search path.

## Assigning userpath as the Startup Folder on a UNIX or Macintosh Platform

Assume `userpath` is set to the default value on a Macintosh platform and you start MATLAB using a bash shell, where `smith` is the home folder. Set the `MATLAB_USE_USERPATH` environment variable so that `userpath` will be used as the startup folder:

```
export MATLAB_USE_USERPATH=1
```

From that shell, start MATLAB. After MATLAB starts, verify the current folder in MATLAB:

```
pwd
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB
```

That is the folder defined for `userpath`, which you can confirm:

```
userpath
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB;
```

The *userpath* is at the top of the search path, which you can confirm:

```
path
```

MATLAB returns:

```
/Users/smith/Documents/MATLAB  
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/general  
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/ops
```

```
...
```



## Adding Folders to the Search Path Upon Startup on a UNIX or Macintosh Platform

Assume userpath is set to the default value on a UNIX platform with a csh shell, where *j* is your home folder.

To add additional folders to the search path upon startup, for example, `/home/j/Documents/MATLAB/mine` and `/home/j/Documents/MATLAB/mine/research`, run the following in a terminal:

```
setenv MATLABPATH '/home/j/Documents/MATLAB/mine': '/home/j/Documents/MATLAB/mine/research'
```

Separate multiple folders using a colon (:).

MATLAB displays

```
MATLABPATH
```

```
home/j/Documents/MATLAB
home/j/Documents/MATLAB/mine
home/j/Documents/MATLAB/mine/research
home/j/Applications/MATLAB/R2009a/toolbox/matlab/general
home/j/Applications/MATLAB/R2009a/toolbox/matlab/ops
...
```

### See Also

`addpath` | `path` | `pathtool` | `rmpath` | `savepath` | `startup`

### How To

- “Using the MATLAB Search Path”
- “Startup Folder for the MATLAB Program”

# 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**  
Any type of array.

**classes**

Cell array of strings that specify valid data types for array `A`. Each string can be the name of any MATLAB built-in or custom class, including:

|                   |                                                                                                                                                                                                                                                                                                          |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'numeric'         | Any value for which the <code>isnumeric</code> function returns true, including <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code> , <code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code> , <code>single</code> , or <code>double</code> . |
| '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' | Scalar function handle.                                                                                                                                                                                                                                                                                  |

## attributes

Cell array that contains descriptions of valid attributes for array A. All attribute names are 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:

# validateattributes

---

|                        |                                                                                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| '2d'                   | Two-dimensional array, including scalars, vectors, matrices, and empty arrays.                                                                  |
| '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',<br>[d1,...,dN] | Array with dimensions d1-by-...-by-dN. If you do not want to check 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.                                                                                                                              |
| 'nonempty'             | No dimensions equal zero.                                                                                                                       |
| 'nonsparse'            | Array that is not sparse.                                                                                                                       |

Attributes that specify valid ranges for values in A:

|         |                                            |
|---------|--------------------------------------------|
| '>', N  | All values are greater than N.             |
| '>=', N | All values are greater than or equal to N. |
| '<', N  | All values are less than N.                |
| '<=', N | All values are less than or equal to N.    |

Attributes that check types of values in A, where A is a numeric or logical array:

|          |                                                 |
|----------|-------------------------------------------------|
| 'binary' | Array of ones and zeros.                        |
| 'even'   | All elements are even integers (includes zero). |
| 'odd'    | All elements are odd integers.                  |

|               |                                             |
|---------------|---------------------------------------------|
| 'integer'     | All elements are integer-valued.            |
| 'real'        | All elements are real.                      |
| 'finite'      | All elements are finite.                    |
| 'nonnan'      | No elements equal to NaN (Not a Number).    |
| 'nonnegative' | No elements are less than zero.             |
| 'nonzero'     | No elements are equal to zero.              |
| 'positive'    | No elements are less than or equal to zero. |

## **funcName**

String that specifies the name of the function whose input you are validating. If you specify an empty string, '', the validateattribute function ignores the funcName input.

## **varName**

String that specifies the name of the input variable. If you specify an empty string, '', the validateattribute function ignores the varName input.

## **argIndex**

Positive integer that specifies the position of the input argument.

## **Examples**

Check the size of an array.

```
classes = {'numeric'};
attributes = {'size',[4,6,2]};

A = rand(3,5,2);
validateattributes(A,classes,attributes)
```

Because A is not of the expected size, validateattributes throws an error and displays this message:

# validateattributes

---

Expected input to be of size 4x6x2 when it is actually size 3x5x2.

---

Check that the values in an array are 8-bit integers within a specific range.

```
classes = {'uint8','int8'};
attributes = {'>',0,'<',70};
```

```
A = randi(100,4,6,2,'uint8');
validateattributes(A,classes,attributes)
```

If some of the values are outside the range, MATLAB displays this error message:

Expected input to be an array with all of the values < 70.

---

Check the attributes of a complex number.

```
a = complex(1,1);
validateattributes(a,{'numeric'},{'nonnegative'});
```

MATLAB displays this error message:

Expected input to be nonnegative.

Because complex numbers lack a well-defined ordering in the complex plane, `validateattributes` does not recognize them as positive or negative.

---

Check inputs to a custom function, and include information about the input name and position in generated errors.

```
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
```

When you call the function without the shape input string,

```
vol = findVolume(10,7,4)
```

MATLAB displays

```
Error using findVolume
```

```
Expected input number 1, Shape, to be one of these types:
```

```
char
```

Instead its type was double.

The function name is part of the error identifier, so

```
MException.last.identifier
```

returns

```
ans =  
    MATLAB:findVolume:invalidType
```

---

Create a custom function that checks input parameters with `inputParser`, and use `validateattributes` as the validating function for the `addRequired` and `addOptional` methods.

```
function a = findArea(shape,dim1,varargin)  
    p = inputParser;  
    charchk = {'char'};  
    numchk = {'numeric'};  
    nempty = {'nonempty'};  
  
    addRequired(p,'shape',@(x)validateattributes(x,charchk,nempty));  
    addRequired(p,'dim1',@(x)validateattributes(x,numchk,nempty));
```

# validateattributes

---

```
addOptional(p,'dim2',1,@(x)validateattributes(x,numchk,nempty));  
parse(p,shape,dim1,varargin{:});  
  
switch shape  
    case 'circle'  
        a = pi * dim1.^2;  
    case 'rectangle'  
        a = dim1 .* p.Results.dim2;  
end
```

If you call the function with a nonnumeric third input,

```
myarea = findArea('rectangle',3,'x')
```

MATLAB displays

```
Error using findArea  
Argument 'dim2' failed validation with error:  
Expected input to be one of these types:
```

```
    numeric
```

Instead its type was char.

## See Also

[validatestring](#) | [is\\*](#) | [isa](#) | [isnumeric](#) | [inputParser](#)



## 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**

# validatestring

---

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 validatestring 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.

| <b>Example – Match 'ball' with . . .</b> | <b>Return Value</b>       | <b>Type of Match</b>                                              |
|------------------------------------------|---------------------------|-------------------------------------------------------------------|
| ball, barn, bell                         | ball                      | Exact match                                                       |
| balloon, barn                            | balloon                   | Partial match (leading characters)                                |
| ballo, balloo, balloon                   | ballo<br>(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'};
```

# validatestring

---

```
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
```

**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]
```

# containers.Map.values

---

## 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  $n = \text{length}(v)$ .

**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`

# var

---

**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  $V * \text{mean}(w)/w(i)$  is an estimate of the variance of `X(i)`. In other words,  $V * \text{mean}(w)$  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
```

# varargin

---

**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”

# varargout

---

**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

# vectorize

---

**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.

`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 *toolboxfcn*:

```
n = 'toolboxfcn';
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 HotSpot(T
```

```
-----  
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  
-----  
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](#) | [hostid](#) | [license](#) | [verlessthan](#) | [version](#)

## Concepts

- “Information About your Installation”
- “Create Help Summary Files (Contents.m)”

**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:

| <b>action Argument</b> | <b>Purpose</b>                                                                                                                 |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| '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. |

# verctrl

---

| <b>action Argument</b> | <b>Purpose</b>                                                                                                                                                               |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'runsc'                | 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:

| <b>action Argument</b> | <b>Purpose</b>                                                                                                      |
|------------------------|---------------------------------------------------------------------------------------------------------------------|
| '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` | `undoccheckout` | `cmopts`

## How To

- “Source Control Interface on Microsoft Windows”

**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>

**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.');
```

# verLessThan

---

```
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



|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Version number for MATLAB and libraries                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Syntax</b>      | <pre>version version -date version -description version -release version -java v = version('-versionOption') [v d] = version</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Description</b> | <p>version returns in ans the version and release number for the MATLAB software currently running.</p> <p>version -date returns in ans the release date for the MATLAB software.</p> <p>version -description returns in ans a description of the version. Usually, the description is for special versions, such as beta versions.</p> <p>version -release returns in ans the release number for the MATLAB software currently running.</p> <p>version -java returns in ans the version of the Sun Microsystems™ JVM software that MATLAB is using.</p> <p>v = version('-versionOption'), where <i>versionOption</i> is one of the above option strings, is an alternate form of the syntax.</p> <p>[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.</p> |
| <b>Examples</b>    | <p>Display the version:</p> <pre>version</pre> <p>MATLAB returns:</p> <pre>7.9.0.2601 (R2009b)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

# version

---

---

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 Serv
```

## See Also

## How To

computer | ver | verlessthan

- “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
```

## vertcat

---

```
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”

**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`

# TriRep.vertexAttachments

---

**Purpose** Return simplices attached to specified vertices

**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, <code>TR.X</code> . The simplices associated with vertex <code>i</code> are the <code>i</code> 'th entry in the cell array. If <code>VI</code> 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 <code>i</code> are in the <code>i</code> 'th entry in the cell array <code>SI</code> . |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**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:

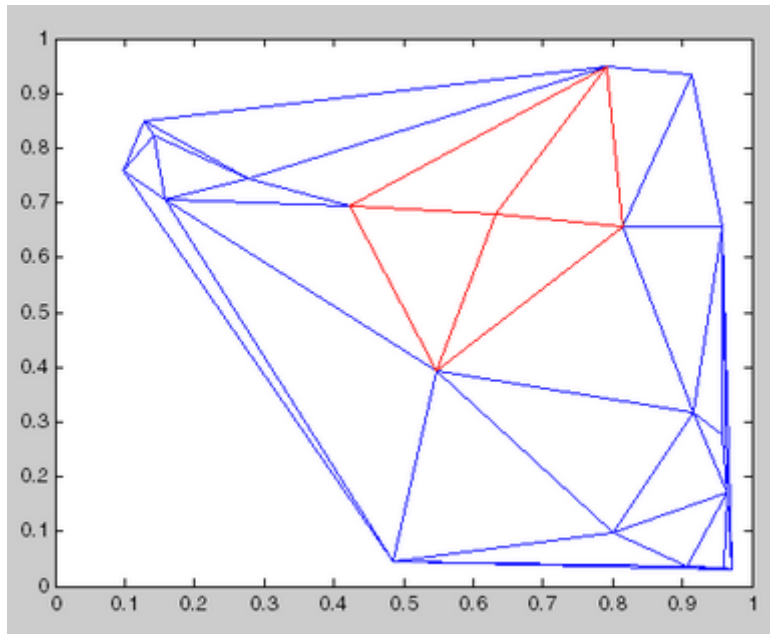
```
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```

# TriRep.vertexAttachments

---



**See Also**

`DelaunayTri`



**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),<br>Motion JPEG 2000 (.mj2)                                                                                                                                                                                                                                  |
| All Windows   | MPEG-1 (.mpg),<br>Windows Media Video (.wmv, .asf, .asx),<br>and any format supported by Microsoft DirectShow®.                                                                                                                                                         |
| Windows 7     | MPEG-4, including H.264 encoded video (.mp4, .m4v),<br>Apple QuickTime Movie (.mov),<br>and any format supported by Microsoft Media<br>Foundation.                                                                                                                      |
| Macintosh     | MPEG-1 (.mpg),<br>MPEG-4, including H.264 encoded video (.mp4, .m4v),<br>Apple QuickTime Movie (.mov),<br>and any format supported by QuickTime as listed on<br><a href="http://support.apple.com/kb/HT3775">http://support.apple.com/kb/HT3775</a> .                   |
| Linux         | Any format supported by your installed plug-ins<br>for GStreamer 0.10 or above, as listed on<br><a href="http://gstreamer.freedesktop.org/documentation/plugins.html">http://gstreamer.freedesktop.org/documentation/plugins.html</a> ,<br>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

# VideoReader

---

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:** ''

# VideoReader

---

## 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, such as 'RGB24'.

## 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 Programming Fundamentals 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”

# VideoWriter

---

## Purpose

Write video files

## Description

Use the `VideoWriter` function with the `open`, `writeVideo`, and `close` methods to create video files from figures, still images, or MATLAB movies. `VideoWriter` can create AVI and Motion JPEG 2000 files on all platforms, and MPEG-4 files on Windows 7 platforms. `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 a compressed AVI file.

`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:

|                                        |                                      |
|----------------------------------------|--------------------------------------|
| <code>.avi</code>                      | AVI file                             |
| <code>.mj2</code>                      | Motion JPEG 2000 file                |
| <code>.mp4</code> or <code>.m4v</code> | MPEG-4 file (Windows 7 systems only) |

If you do not specify the 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 an 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 (Windows 7 systems only) |
| 'Uncompressed AVI' | Uncompressed AVI file with RGB24 video                              |

**Default:** 'Motion JPEG AVI'

## Properties

### ColorChannels

Number of color channels in each output video frame. (Read-only)

AVI and MPEG-4 files have three color channels (RGB). 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.

### 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.

# VideoWriter

---

**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**

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.

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)

For AVI or MPEG-4 files, `VideoFormat` is 'RGB24'.

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.

| Image Data                       | VideoFormat     |
|----------------------------------|-----------------|
| Single-band <code>uint8</code>   | 'Mono8'         |
| Single-band <code>int8</code>    | 'Mono8 Signed'  |
| Single-band <code>uint16</code>  | 'Mono16'        |
| Single-band <code>int16</code>   | 'Mono16 Signed' |
| Three-banded <code>uint8</code>  | 'RGB24'         |
| Three-banded <code>int8</code>   | 'RGB24 Signed'  |
| Three-banded <code>uint16</code> | 'RGB48'         |
| Three-banded <code>int16</code>  | '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

|                          |                                                                      |
|--------------------------|----------------------------------------------------------------------|
| <code>close</code>       | Close file after writing video data                                  |
| <code>getProfiles</code> | List profiles and file formats supported by <code>VideoWriter</code> |
| <code>open</code>        | Open file for writing video data                                     |
| <code>writeVideo</code>  | Write video data to file                                             |

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects in the MATLAB Programming Fundamentals documentation](#).

## Examples

### Object Properties

Construct a `VideoWriter` object and view its properties.

```
writerObj = VideoWriter('newfile.avi')
```

Set the frame rate to 60 frames per second.

```
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');
```

# VideoWriter

---

```
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](#)

**Purpose**

Viewpoint specification

**Syntax**

```
view(az,e1)
view([az,e1])
view([x,y,z])
view(2)
view(3)
view(ax,...)
[az,e1] = 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,e1)` and `view([az,e1])` 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. `e1` 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`, `e1 = 90`.

`view(3)` sets the default three-dimensional view, `az = 37.5`, `e1 = 30`.

`view(ax,...)` uses axes `ax` instead of the current axes.

`[az,e1] = view` returns the current azimuth and elevation.

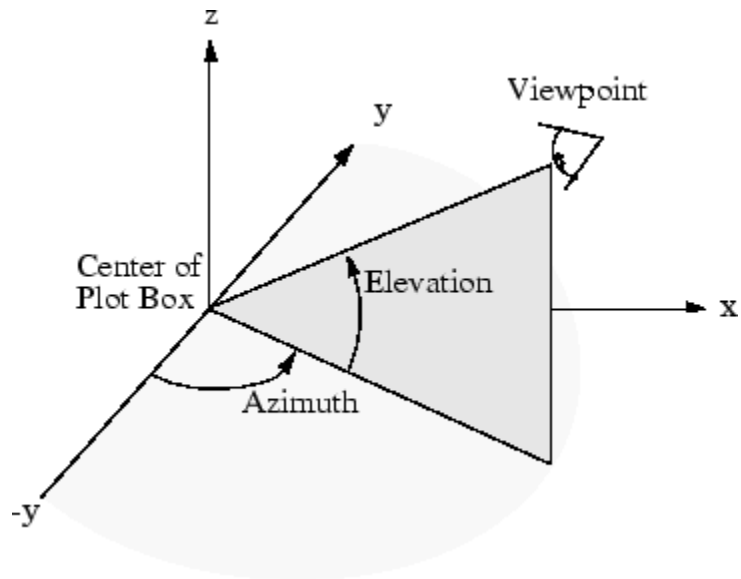
**Tips**

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.

# view

---



## 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  $180^\circ$ .

```
az = 180;  
el = 90;  
view(az, el);
```

**See Also**

[hgtransform](#) | [rotate3d](#) | [CameraPosition](#) | [CameraTarget](#) | [CameraViewAngle](#) | [Projection](#)

**How To**

- “View Overview”
-

# viewmtx

---

**Purpose** View transformation matrices

**Syntax**

```
viewmtx  
T = viewmtx(az,e1)  
T = viewmtx(az,e1,phi)  
T = viewmtx(az,e1,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,e1) returns an *orthographic* transformation matrix corresponding to azimuth az and elevation e1. az is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. e1 is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,e1)  
T = view
```

but does not change the current view.

T = viewmtx(az,e1,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,e1,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
```

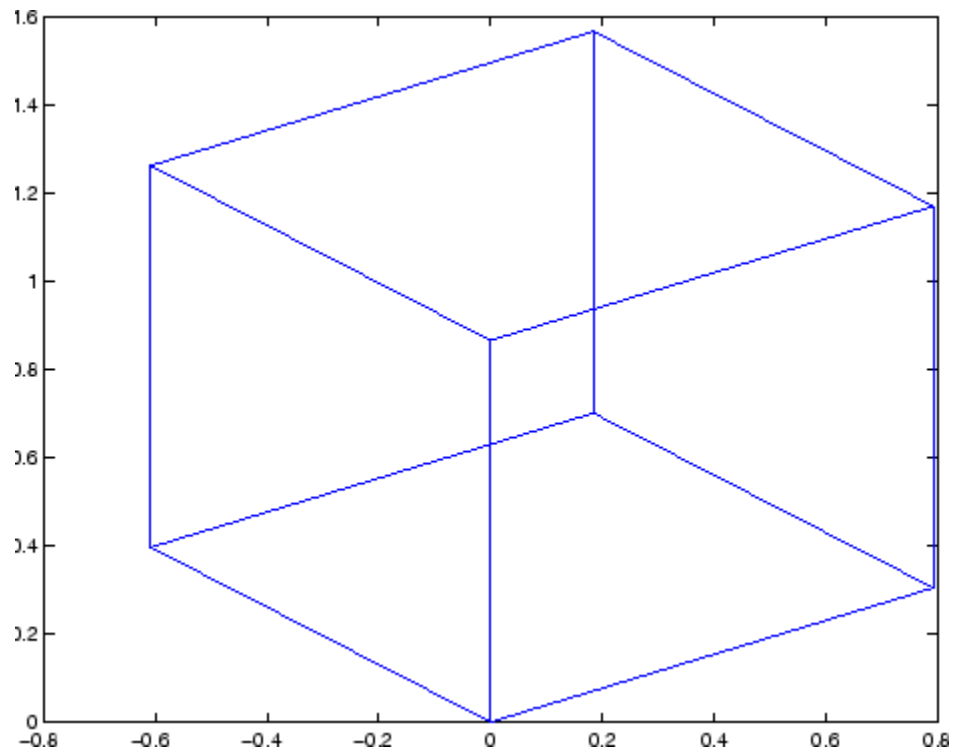
These vectors trace the edges of a unit cube:

```
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 1 0 0 1 1];
```

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)
```

# viewmtx



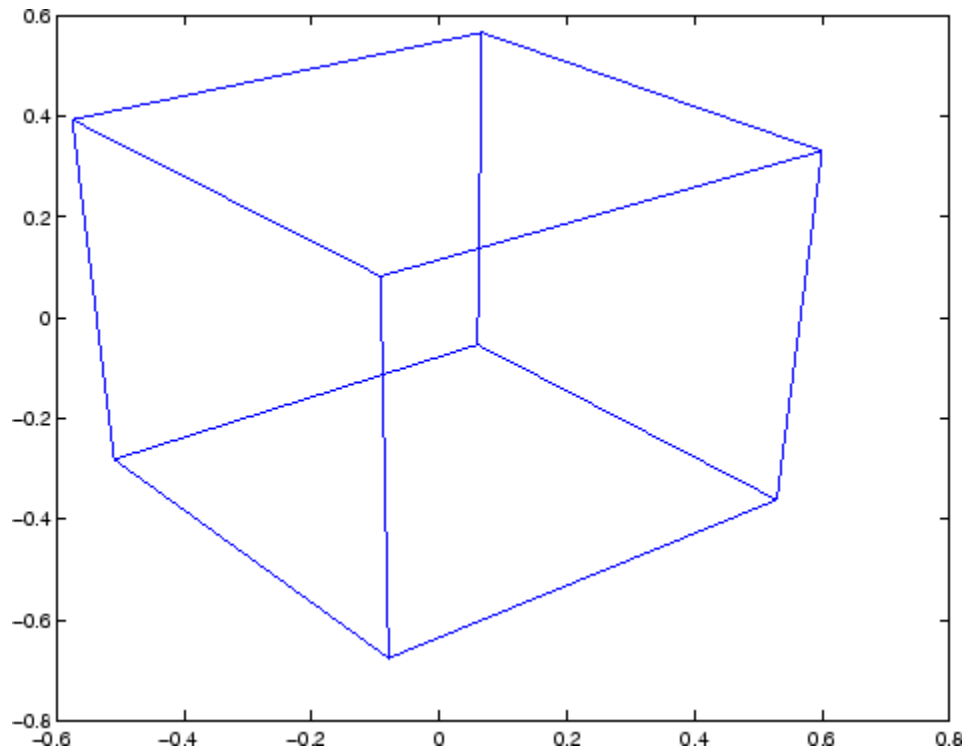
Use a perspective transformation with a 25 degree viewing angle:

```
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”

# visdiff

---

**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 (.mdl files) 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, select **Desktop > Comparison Tool**, and then select the files or folders to compare.
- If you have a file open in the Editor, select **Tools > Compare Against**. You can use the Editor options browse, **Autosave Version**, or **Compare Against Version on Disk**.

## How To

- “Comparing Files and Folders”



## 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)`.

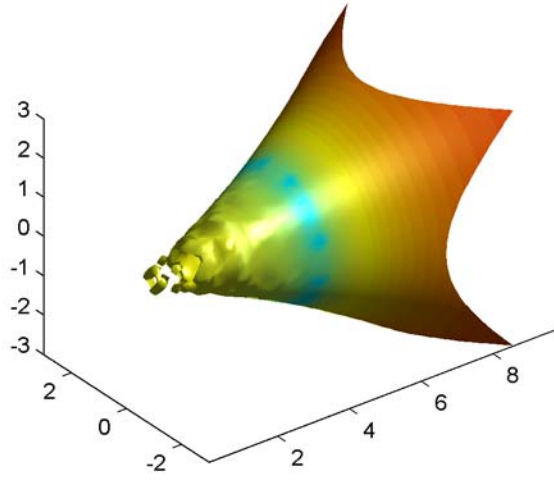
## Examples

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
```

# volumebounds

---



## 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(...,'LineStyle')
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(...,'LineStyle')` 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:

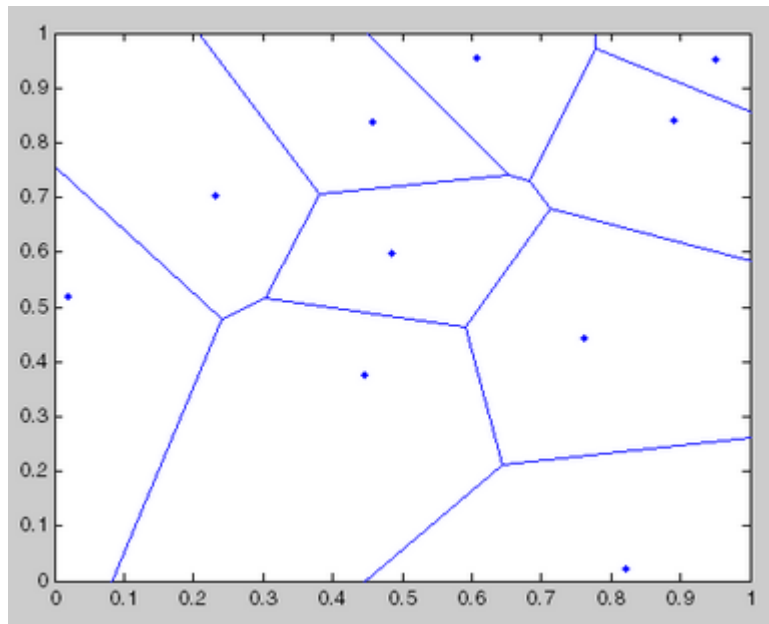
- If you provide no output argument, `voronoi` plots the diagram.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function.
- To fill 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)
```

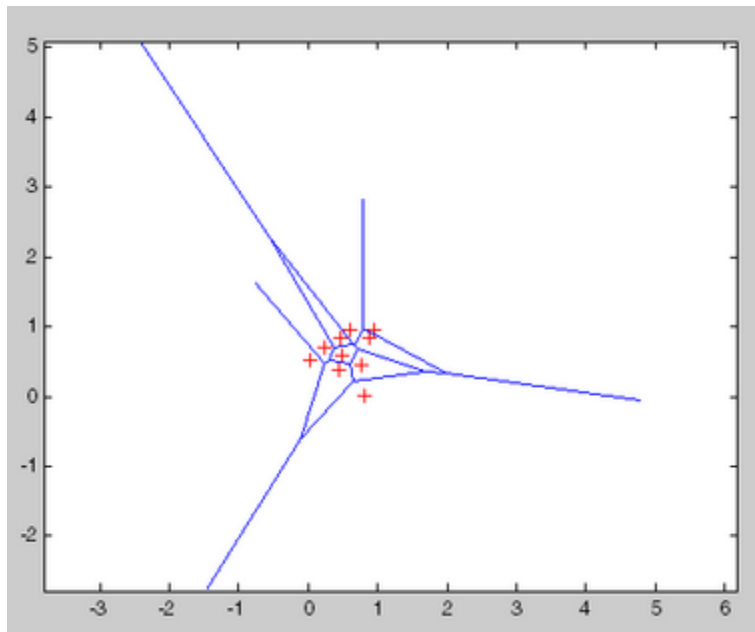


### Example 2

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
```

# voronoi



Note that you can add this code to get the figure shown in Example 1.

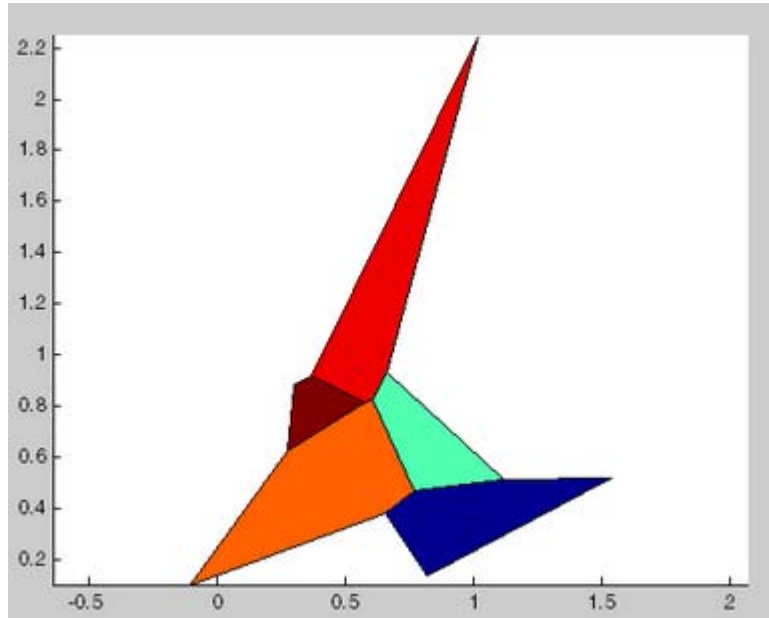
```
xlim([min(x) max(x)])  
ylim([min(y) max(y)])
```

### 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**

[DelaunayTri](#) | [convhull](#) | [delaunay](#) | [LineSpec](#) | [plot](#) | [voronoin](#)

# DelaunayTri.voronoiDiagram

---

**Purpose** Voronoi diagram

**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](#)

# voronoin

---

**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
```

# voronoin

---

```
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/>. 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**

DelaunayTri | convhull | convhulln | delaunay | delaunayn | voronoi

# wait

---

**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 user 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 `WindowState` 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 `WindowState` 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

---

`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`.

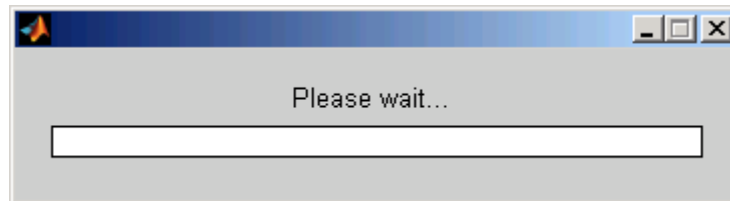
`waitbar(x,h,'updated message')` updates the message text in the waitbar figure, in addition to setting the fractional length to `x`.

## 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  $\pi$ . 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  $\pi$  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  $\pi$ .

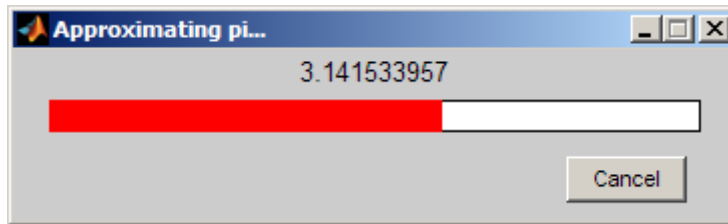
```
function [valueofpi step] = approxpi(steps)
% Converge on pi in steps iterations, displaying waitbar.
% User can click Cancel or close button to exit the loop.
% Ten thousand steps yields error of about 0.001 percent.

h = waitbar(0,'1','Name','Approximating pi...',...
           'CreateCancelBtn',...
           'setappdata(gcf,'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.
```

# waitbar

---

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');
```

---

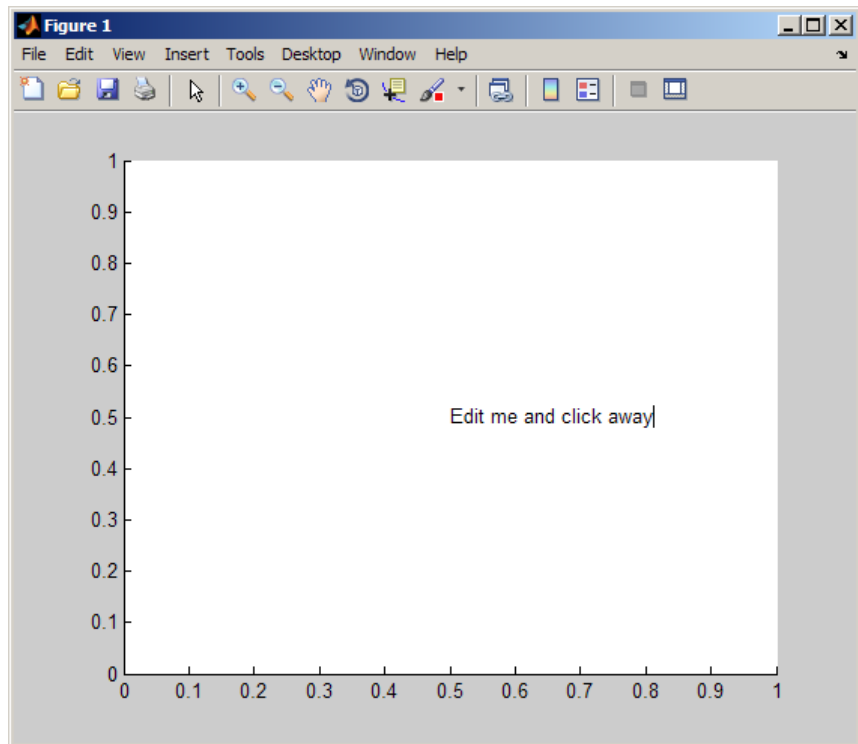
Suspend execution until name of figure changes:

```
f = figure('Name', datestr(now));
h = uicontrol('String', 'Change Name', 'Position', [20 20 100 30], ...
'Callback', 'set(gcf, '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]);
```

# waitfor

---

```
        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”

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Wait for key press or mouse-button click                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Syntax</b>      | <code>k = waitforbuttonpress</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Description</b> | <p><code>k = waitforbuttonpress</code> 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</p> <ul style="list-style-type: none"><li>• 0 if it detects a mouse button click</li><li>• 1 if it detects a key press</li></ul> <p>If a <code>WindowButtonDownFcn</code> is defined for the figure, its callback is executed before <code>waitforbuttonpress</code> returns a value.</p> <p>Only keys that generate characters cause the function to return. Pressing any of the following keys by itself does nothing: <b>Ctrl</b>, <b>Shift</b>, <b>Alt</b>, <b>Caps_lock</b>, <b>Num_lock</b>, <b>Scroll_lock</b>.</p> <p>Additional information about the event that causes execution to resume is available through the figure <code>CurrentCharacter</code>, <code>SelectionType</code>, and <code>CurrentPoint</code> properties.</p> <p>You can interrupt <code>waitforbuttonpress</code> by typing <b>Ctrl+C</b>, but an error results unless the function is called from within a <code>try/catch</code> block. You also receive an error from <code>waitforbuttonpress</code> if you close the figure by clicking the <b>X</b> close box unless you call <code>waitforbuttonpress</code> within a <code>try/catch</code> block.</p> |

**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
```

# waitforbuttonpress

---

## See Also

`dragrect` | `ginput` | `rbbox` | `waitfor`



**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 `WindowState` in the Figure Properties.

---

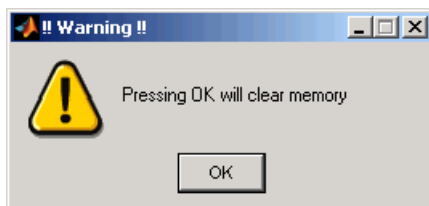
If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings 'tex' or 'none'. The default value for `Interpreter` is 'none'.

## 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](#)

# 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
```

# warning

---

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)
```

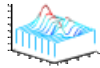
## See Also

`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  $x = 1:\text{size}(Z,2)$  and  $y = 1:\text{size}(Z,1)$ .  $Z$  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  $\text{length}(x) = n$ ,  $\text{length}(y) = m$ , and  $[m,n] = \text{size}(Z)$ .  $X$  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.

# waterfall

---

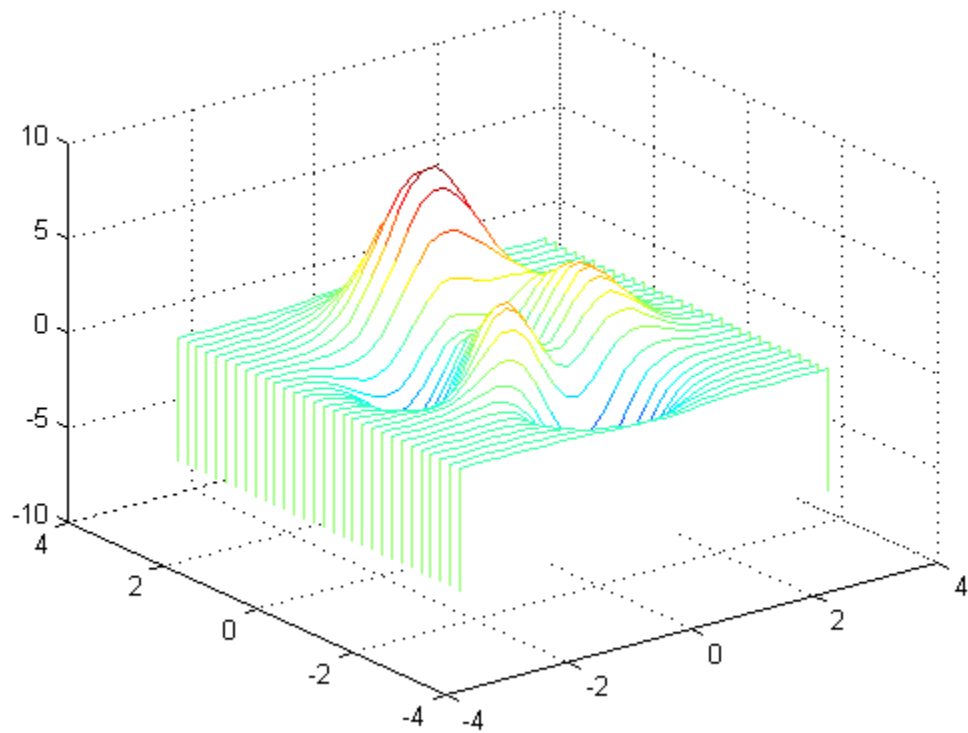
## Tips

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`

# wavinfo

---

**Purpose** Information about WAVE (.wav) sound file

**Syntax** `[m d] = wavinfo(filename)`

**Description** `[m d] = wavinfo(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

**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`

**Purpose** Read WAVE (.wav) sound file

**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.

# wavread

`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) | $0 \leq y \leq 255$            |
| 16             | int16 (signed integer)   | $-32768 \leq y \leq +32767$    |
| 24             | int32 (signed integer)   | $-2^{23} \leq y \leq 2^{23}-1$ |
| 32             | single (floating point)  | $-1.0 \leq y < +1.0$           |

### Double Formats

| Number of Bits | MATLAB Data Type | Data Range                                                                                                                                        |
|----------------|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| N<32           | double           | $-1.0 \leq y < +1.0$                                                                                                                              |
| N=32           | double           | $-1.0 \leq y \leq +1.0$<br>Note: Values in <code>y</code> 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`.

| <b>dtype</b> | <b>Bits/sample</b> | <b>y Data Range</b>         |
|--------------|--------------------|-----------------------------|
| 'double'     | 16                 | $-1.0 \leq y < +1.0$        |
| 'single'     | 16                 | $-1.0 \leq y < +1.0$        |
| 'int16'      | 16                 | $-32768 \leq y \leq +32767$ |
| 'uint8'      | 8                  | $0 \leq y \leq 255$         |

**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

**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       | $0 \leq y \leq 255$              | uint8         |
| 16     | int16       | $-32768 \leq y \leq +32767$      | int16         |
| 24     | int32       | $-2^{23} \leq y \leq 2^{23} - 1$ | int32         |

If  $y$  contains floating-point data:

| N Bits | y Data Type      | y Data Range            | Output Format |
|--------|------------------|-------------------------|---------------|
| 8      | single or double | $-1.0 \leq y < +1.0$    | uint8         |
| 16     | single or double | $-1.0 \leq y < +1.0$    | int16         |
| 24     | single or double | $-1.0 \leq y < +1.0$    | int32         |
| 32     | single or double | $-1.0 \leq y \leq +1.0$ | single        |

For floating point data where  $N < 32$ , amplitude values are clipped to the range  $-1.0 \leq y < +1.0$ .

---

**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**  
[N, S] = weekday(D)  
[N, S] = weekday(D, form)  
[N, S] = weekday(D, locale)  
[N, S] = weekday(D, form, locale)

**Description** [N, S] = weekday(D) returns the day of the week in numeric (N) and string (S) form for a given serial date number or date string D. Input argument D can represent more than one date in an array of serial date numbers or a cell array of date strings.

[N, S] = weekday(D, form) returns the day of the week in numeric (N) and string (S) form, where the content of S depends on the form argument. If form is **'long'**, then S contains the full name of the weekday (e.g., Tuesday). If form is **'short'**, then S contains an abbreviated name (e.g., Tues) from this table.

The days of the week are assigned these numbers and abbreviations.

| <b>N</b> | <b>S (short)</b> | <b>S (long)</b> |
|----------|------------------|-----------------|
| 1        | Sun              | Sunday          |
| 2        | Mon              | Monday          |
| 3        | Tue              | Tuesday         |
| 4        | Wed              | Wednesday       |
| 5        | Thu              | Thursday        |
| 6        | Fri              | Friday          |
| 7        | Sat              | Saturday        |

[N, S] = weekday(D, locale) returns the day of the week in numeric (N) and string (S) form, where the format of the output depends on the locale argument. If locale is **'local'**, then weekday uses local format for its output. If locale is **'en\_US'**, then weekday uses US English.

`[N, S] = weekday(D, form, locale)` returns the day of the week using the formats described above for `form` and `locale`.

## Examples

Either

```
[n, s] = weekday(728647)
```

or

```
[n, s] = weekday('19-Dec-1994')
```

returns `n = 2` and `s = Mon`.

## 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      auinfo      mmcompinfo  wavplay
audiodevinfo  aread      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
```

# what

---

```
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`



|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>      | Locate functions and files                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Alternatives</b> | As an alternative to the <code>which</code> function, you can use the Current Folder Browser to find files. You can find functions using the Function Browser in the Command Window or Editor.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>       | <pre>which fun which classname/fun which <b>private</b>/fun which classname/<b>private</b>/fun which fun1 <b>in</b> fun2 which fun(a,b,c,...) which file.ext which fun <b>-all</b> s = which('fun',...)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Description</b>  | <p><code>which fun</code> displays the full pathname for the argument <code>fun</code>. If <code>fun</code> is a</p> <ul style="list-style-type: none"><li>• MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then <code>which</code> displays the full pathname for the corresponding file</li><li>• Workspace variable, then <code>which</code> displays a message identifying <code>fun</code> as a variable</li><li>• Method in a loaded Java class, then <code>which</code> displays the package, class, and method name for that method</li></ul> <p>If <code>fun</code> is an overloaded function or method, then <code>which fun</code> returns only the pathname of the first function or method found.</p> <p><code>which classname/fun</code> displays the full pathname for the file that defines the <code>fun</code> method in MATLAB class, <code>classname</code>. For example, <code>which serial/fopen</code> displays the path for <code>fopen.m</code> in the MATLAB class folder, <code>@serial</code>.</p> <p><code>which <b>private</b>/fun</code> limits the search to private functions. For example, <code>which private/orthog</code> displays the path for <code>orthog.m</code> in the <code>/private</code> subfolder of <code>toolbox/matlab/elmat</code>.</p> |

# which

---

`which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the private folder of the `dfilt` class.

`which fun1 in fun2` displays the pathname to function `fun1` in the context of file `fun2`. Use this form to determine whether a local function is being called instead of a function on the path. For example, `which parseargs in area` tells you which `parseargs` function is called by `area.m`.

`which fun(a,b,c,...)` displays the path to the specified function with the given input arguments. For example, `which feval(g)`, when `g=inline('sin(x)')`, indicates that `inline/feval.m` would be invoked. `which toLowerCase(s)`, when `s=java.lang.String('my Java string')`, indicates that the `toLowerCase` method in class `java.lang.String` would be invoked.

`which file.ext` displays the full pathname of the specified file if that file is in the current working folder or on the MATLAB path. To display the path for a file that has no file extension, type “`which file.`” (the period following the filename is required). Use `exist` to check for the existence of files anywhere else.

`which fun -all` displays the paths to all items on the MATLAB path with the name `fun`. Such items include methods of instantiated classes. You may use the `-all` qualifier with any of the above formats of the `which` function.

`s = which('fun',...)` returns the results of `which` in the string `s`. For workspace variables, `s` is the string `'variable'`. You may specify an output variable in any of the above formats of the `which` function.

If `-all` is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.

## Tips

For more information about how MATLAB uses scope and precedence when calling a function, see “Function Precedence Order”.

**Examples**

The statement below indicates that `pinv` is in the `matfun` folder of MATLAB.

```
which pinv
matlabroot\toolbox\matlab\matfun\pinv.m
```

To find the `fopen` function used on MATLAB serial class objects

```
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m % serial method
```

To find the `setMonth` method used on objects of the Java `Date` class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

```
myDate = java.util.Date;
which setMonth
```

MATLAB displays:

```
setMonth is a Java method % java.util.Date method
```

When you specify an output variable, `which` returns a cell array of strings to the variable. You must use the *function* form of `which`, enclosing all arguments in parentheses and single quotes:

```
s = which('private/stradd', '-all');
whos s
  Name      Size      Bytes  Class
  s         3x1         562   cell array
Grand total is 146 elements using 562 bytes
```

**See Also**

`dir` | `doc` | `exist` | `lookfor` | `mfilename` | `path` | `type` | `what` | `who`

# 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)
```

# while

---

---

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”

---

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>     | Change axes background color                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>      | <pre>whitebg whitebg(fig) whitebg(ColorSpec) whitebg(fig, ColorSpec) whitebg(fig, ColorSpec) whitebg(fig)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b> | <p><code>whitebg</code> complements the colors in the current figure.</p> <p><code>whitebg(fig)</code> complements colors in all figures specified in the vector <code>fig</code>.</p> <p><code>whitebg(ColorSpec)</code> and <code>whitebg(fig, ColorSpec)</code> change the color of the axes, which are children of the figure, to the color specified by <code>ColorSpec</code>. Without a figure specification, <code>whitebg</code> or <code>whitebg(ColorSpec)</code> affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.</p> <p><code>whitebg(fig, ColorSpec)</code> sets the default axes background color of the figures in the vector <code>fig</code> to the color specified by <code>ColorSpec</code>. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. <code>ColorSpec</code> can be a 1-by-3 RGB color or a color string such as 'white' or 'w'.</p> <p><code>whitebg(fig)</code> 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 <code>whitebg</code> gets its name. Include the root window handle (0) in <code>fig</code> to affect the default properties for new windows or for <code>clf reset</code>.</p> |
| <b>Tips</b>        | <p><code>whitebg</code> works best in cases where all the axes in the figure have the same background color.</p> <p><code>whitebg</code> 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. <code>whitebg</code> sets the default properties on the root such that all subsequent figures use the new background color.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

# whitebg

---

## Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

## See Also

[ColorSpec](#) | [colordef](#)



---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>         | List variables in workspace                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Syntax</b>          | <pre>who who(variables) who(location) who(variables,location) c = who(variables,location)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Description</b>     | <p><code>who</code> lists in alphabetical order all variables in the currently active workspace.</p> <p><code>who(variables)</code> lists only the specified variables.</p> <p><code>who(location)</code> 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.</p> <p><code>who(variables,location)</code> lists the specified variables in the specified location. The <code>location</code> input can appear before or after <code>variables</code>.</p> <p><code>c = who(variables,location)</code> stores the names of the variables in cell array <code>c</code>. Specifying <code>variables</code> and <code>location</code> is optional.</p> |
| <b>Tips</b>            | <ul style="list-style-type: none"><li>• The <code>who</code> function displays the variable list unless you specify an output argument.</li><li>• When used within a nested function, the <code>who</code> function lists the variables in the workspaces of that function and all functions containing that function, grouped by workspace. This applies whether you call <code>who</code> from your function code or from the MATLAB debugger.</li></ul>                                                                                                                                                                                                                                                                                                              |
| <b>Input Arguments</b> | <p><b>variables</b></p> <p>Strings that specify the variables to list. Use one of these forms:</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

# who

---

*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', '- :');
```

# who

---

```
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`

|                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                                                                                                                                          |                                    |                                                          |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------|----------------------------------------------------------|
| <b>Purpose</b>                     | Names of variables in MAT-file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                            |                                                                                                                                          |                                    |                                                          |
| <b>Syntax</b>                      | <pre>varlist = who(matObj) varlist = who(matObj,variables)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                            |                                                                                                                                          |                                    |                                                          |
| <b>Description</b>                 | <p><code>varlist = who(matObj)</code> lists alphabetically all variables in the MAT-file associated with <code>matObj</code>. Optionally, returns the list in cell array <code>varlist</code>.</p> <p><code>varlist = who(matObj,variables)</code> lists the specified variables.</p>                                                                                                                                                                                                                                                       |                            |                                                                                                                                          |                                    |                                                          |
| <b>Input Arguments</b>             | <p><b>matObj</b><br/>Object created by the <code>matfile</code> function.</p> <p><b>variables</b><br/>Names of variables in the MAT-file corresponding to <code>matObj</code>. Use one of these forms:</p> <table><tr><td><code>var1,...,varN</code></td><td>Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>who(matobj,'A*')</code>.</td></tr><tr><td><code>'-regexp',expressions</code></td><td>Regular expression strings that describe variable names.</td></tr></table> | <code>var1,...,varN</code> | Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>who(matobj,'A*')</code> . | <code>'-regexp',expressions</code> | Regular expression strings that describe variable names. |
| <code>var1,...,varN</code>         | Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>who(matobj,'A*')</code> .                                                                                                                                                                                                                                                                                                                                                                                                    |                            |                                                                                                                                          |                                    |                                                          |
| <code>'-regexp',expressions</code> | Regular expression strings that describe variable names.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                            |                                                                                                                                          |                                    |                                                          |
| <b>Output Arguments</b>            | <p><b>varlist</b><br/>Cell array of strings that correspond to each variable name.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                            |                                                                                                                                          |                                    |                                                          |
| <b>Examples</b>                    | <p>Display a list of variables in the example file <code>topography.mat</code>:</p> <pre>matObj = matfile('topography.mat'); who(matObj)</pre> <p>This code returns:</p>                                                                                                                                                                                                                                                                                                                                                                    |                            |                                                                                                                                          |                                    |                                                          |

Your variables are:

topo          toplegend    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(variables,location)
```

**Description**

`whos` lists in alphabetical order all variables in the currently active workspace, including their sizes and types.

`whos(variables)` lists only the specified variables.

`whos(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.

`whos(variables,location)` lists the specified variables in the specified location. The `location` input can appear before or after `variables`.

`s = whos(variables,location)` stores information about the variables in structure `s`. Specifying `variables` and `location` is optional.

**Tips**

- The `whos` function displays the variable list unless you specify an output argument.
- `whos` returns the number of bytes each variable occupies in the workspace, not in a MAT-file. Version 7 and later MAT-files (the default starting with R2006b) are compressed, so the number of bytes in the file is typically less than the number of bytes required in the workspace.
- `whos -file filename` does not return the sizes of any MATLAB objects in file `filename`.
- When used within a nested function, `whos` lists the variables in the workspaces of that function and all functions containing that function, grouped by workspace. This applies whether you call `whos` from your function code or from the MATLAB debugger.

# whos

---

## 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, `whos('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

### s

Nested structure array that contains a scalar struct for each variable. Each scalar struct contains these fields:

|       |                                                   |
|-------|---------------------------------------------------|
| name  | Name of the variable.                             |
| size  | Dimensions of the variable array.                 |
| bytes | Number of bytes allocated for the variable array. |



|                         |                                                                                                                                                                                                                                          |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>class</code>      | Class of the variable. If the variable has no value, class is '(unassigned)'.                                                                                                                                                            |
| <code>global</code>     | True if the variable is global; otherwise, false.                                                                                                                                                                                        |
| <code>sparse</code>     | True if the variable is sparse; otherwise, false.                                                                                                                                                                                        |
| <code>complex</code>    | True if the variable is complex; otherwise, false.                                                                                                                                                                                       |
| <code>nesting</code>    | Structure with these fields: <ul style="list-style-type: none"> <li>• <code>function</code> — Name of the nested or outer function that defines the variable.</li> <li>• <code>level</code> — Nesting level of that function.</li> </ul> |
| <code>persistent</code> | True if the variable is persistent; otherwise, false.                                                                                                                                                                                    |

## Examples

Display information about variables in the current workspace whose names start with the letter a:

```
whos a*
```

---

Show variables stored in MAT-file `durer.mat`:

```
whos -file durer
```

This code returns:

| Name    | Size    | Bytes   | Class  | Attributes |
|---------|---------|---------|--------|------------|
| X       | 648x509 | 2638656 | double |            |
| caption | 2x28    | 112     | char   |            |
| map     | 128x3   | 3072    | double |            |

---

Store information about the variables from `durer.mat` in structure array `durerInfo`:

```
durerInfo = whos('-file', 'durer');
```

Display the contents of structure durerInfo:

```
for k=1:length(durerInfo)
    disp([' ' durerInfo(k).name ...
         ' ' mat2str(durerInfo(k).size) ...
         ' ' durerInfo(k).class]);
end
```

This code returns:

```
X [648 509] double
caption [2 28] char
map [128 3] double
```

---

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\>
```

---

Create a function that displays information about variables with persistent, global, sparse, and complex attributes:

```
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:

| 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 function (get\_date):

```
function whos_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};
        whos
    end

end
```

When you run whos\_demo, the whos function displays the variables by function workspace:

| Name                | Size | Bytes | Class | Attributes |
|---------------------|------|-------|-------|------------|
| ---- get_date ----- |      |       |       |            |
| d                   | 1x3  | 690   | cell  |            |
| day                 | 1x1  | 116   | cell  |            |
| mon                 | 1x1  | 118   | cell  |            |
| year                | 1x1  | 120   | cell  |            |

# whos

---

```
---- whos_demo -----  
ans          0x0          0 (unassigned)  
date_time    1x20         40 char  
pos          1x1          8 double  
str          1x3         690 cell
```

## 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` | `who` | `workspace`

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                            |                                                                                                                                            |                                   |                                                          |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|----------------------------------------------------------|
| <b>Purpose</b>                    | Names, sizes, and types of variables in MAT-file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                            |                                                                                                                                            |                                   |                                                          |
| <b>Syntax</b>                     | <pre>details = whos(matObj) details = whos(matObj,variables)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                            |                                                                                                                                            |                                   |                                                          |
| <b>Description</b>                | <p><code>details = whos(matObj)</code> returns information about all variables in the MAT-file associated with <code>matObj</code>.</p> <p><code>details = whos(matObj,variables)</code> returns information about the specified variables.</p>                                                                                                                                                                                                                                                                                                                                                                                                       |                            |                                                                                                                                            |                                   |                                                          |
| <b>Input Arguments</b>            | <p><b>matObj</b><br/>Object created by the <code>matfile</code> function.</p> <p><b>variables</b><br/>Names of variables in the MAT-file corresponding to <code>matObj</code>. Use one of these forms:</p> <table border="0" style="margin-left: 40px;"> <tr> <td style="padding-right: 20px;"><code>var1,...,varN</code></td> <td>Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>whos(matobj, 'A*')</code>.</td> </tr> <tr> <td style="padding-right: 20px;"><code>'-regex',expressions</code></td> <td>Regular expression strings that describe variable names.</td> </tr> </table> | <code>var1,...,varN</code> | Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>whos(matobj, 'A*')</code> . | <code>'-regex',expressions</code> | Regular expression strings that describe variable names. |
| <code>var1,...,varN</code>        | Comma-separated list of variable name strings. Optionally, match patterns with the '*' wildcard, such as <code>whos(matobj, 'A*')</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                            |                                                                                                                                            |                                   |                                                          |
| <code>'-regex',expressions</code> | Regular expression strings that describe variable names.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                            |                                                                                                                                            |                                   |                                                          |
| <b>Output Arguments</b>           | <p><b>details</b><br/>Structure array with these fields (identical to the structure returned by the <code>whos</code> function):</p> <table border="0" style="margin-left: 40px;"> <tr> <td style="padding-right: 40px;"><code>name</code></td> <td>Variable name</td> </tr> <tr> <td style="padding-right: 40px;"><code>size</code></td> <td>Dimensions of the variable</td> </tr> </table>                                                                                                                                                                                                                                                          | <code>name</code>          | Variable name                                                                                                                              | <code>size</code>                 | Dimensions of the variable                               |
| <code>name</code>                 | Variable name                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                            |                                                                                                                                            |                                   |                                                          |
| <code>size</code>                 | 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: <ul style="list-style-type: none"><li>• <code>function</code> — Name of the nested or outer function that defines the variable</li><li>• <code>level</code> — Nesting level</li></ul> |
| 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 with 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** wk1info has been removed.

---

**Syntax** [extens, typ] = wk1info(filename)

**Description** [extens, typ] = wk1info(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] = wk1info('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

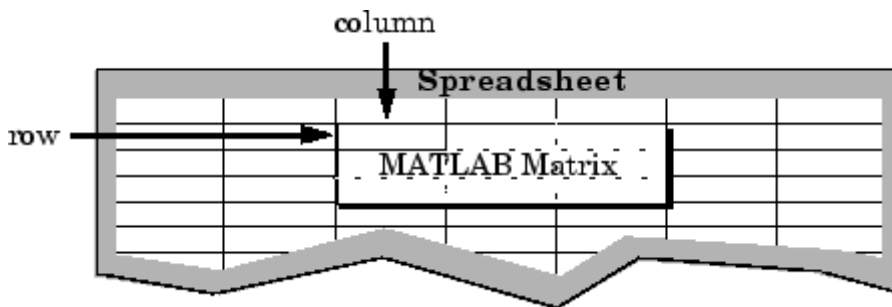
`M = wk1read(filename)` reads a Lotus1-2-3 WK1 spreadsheet file into the matrix `M`. The `filename` input is a string enclosed in single quotes.

`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 |

# wk1read

---

## **See Also**

xlsread



**Purpose** Write matrix to Lotus 1-2-3 WK1 spreadsheet file

---

**Note** wk1write has been removed.

---

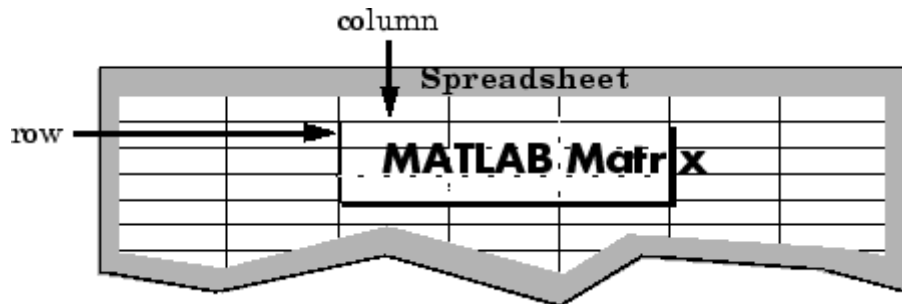
**Syntax**

```
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

---

```
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](#)

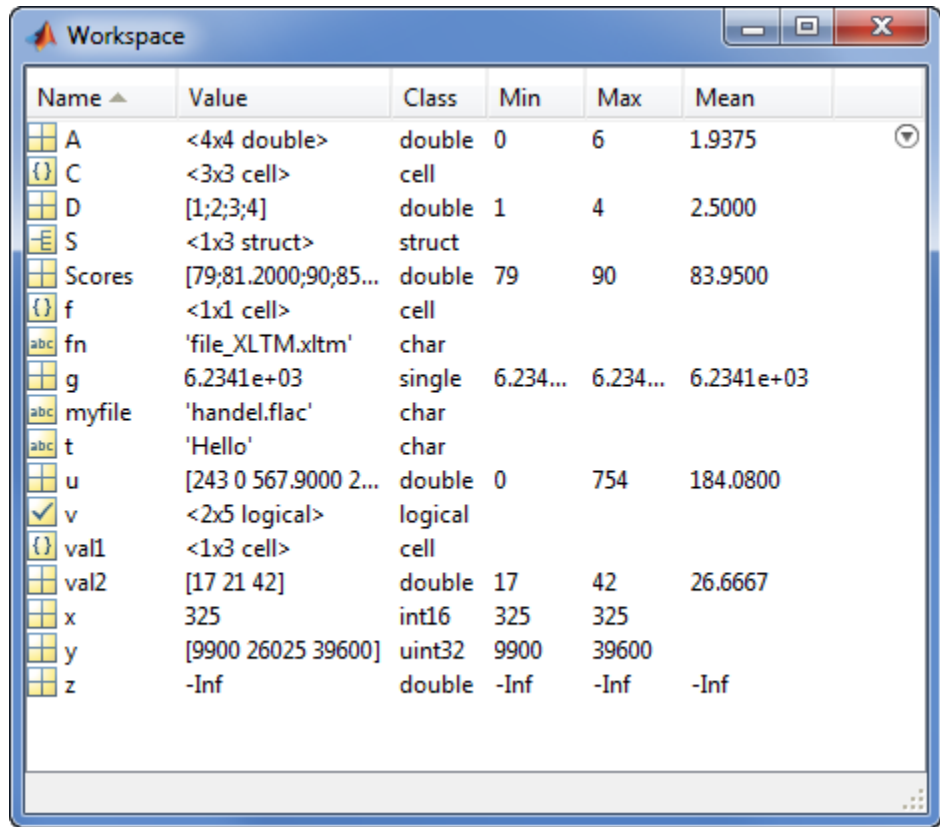
**Purpose** Open Workspace browser to manage workspace

**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.

# workspace



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`

# Tiff.writeEncodedStrip

---

## **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`

# Tiff.writeEncodedTile

---

## **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,frame)
writeVideo(writerObj,mov)
writeVideo(writerObj,img)
writeVideo(writerObj,images)
```

**Description**

`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.

`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.

**Input Arguments****writerObj**

VideoWriter object created by the VideoWriter function.

**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`, and constructs RGB image frames. The height and width must be consistent for all frames within a file.

**mov**

Array of frame structures, each of which contains fields `cdata` and `colormap`.

**img**

When creating AVI or MPEG-4 files, `img` is an array of `single`, `double`, or `uint8` values representing a grayscale or RGB color

# VideoWriter.writeVideo

---

image, which `writeVideo` writes as an RGB video frame. Data of type `single` or `double` must be in the range `[0,1]`.

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 or monochrome data, `img` is two-dimensional: height-by-width. For color data, `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”.

## images

Four-dimensional array of grayscale (height-by-width-by-1-by-frames) or RGB (height-by-width-by-3-by-frames) images.

## 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](#)

# xlabel

---

## Purpose

Label *x*-axis

## Syntax

```
xlabel('string')
xlabel(fname)
xlabel(..., 'PropertyName', PropertyValue, ...)
xlabel(axes_handle, ...)
h = xlabel(...)
```

## Description

Every axes graphics object can have one label for the *x*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`xlabel('string')` labels the *x*-axis of the current axes.

`xlabel(fname)` evaluates the function `fname`, which must return a string. Then it displays the string beside the *x*-axis.

`xlabel(..., 'PropertyName', PropertyValue, ...)` specifies property name-value pairs for the text graphics object created by `xlabel`.

`xlabel(axes_handle, ...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = xlabel(...)` returns the handle to the text object used as the label.

## Tips

Reissuing `xlabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or on the side, so the plot does not hide it.

## Examples

This example shows how to give a string as input to `xlabel`.

```
xlabel('Population');
```

---

Passing a numeric input as input to the function.

```
xlabel(123);
```

---

---

Passing a handle as input to `xlabel`.

```
h = figure;  
xlabel(h);
```

---

This example displays the label in multiple lines.

```
xlabel({2010;'Population';'in Years'});
```


---

This example shows you how to use name-value pairs in `xlabel`.

```
xlabel('Population','fontsize',12,'fontweight','b','color','r');
```

When resizing the figure window, you may want to display graph to the scale or be able to see the text in title and labels completely. For more information about this, see “Properties Controlling Axes Size”.

## Alternatives

To control the presence and appearance of axis labels on a graph, use the Property Editor plotting tool . For details, see The Property Editor.

## See Also

`strings` | `ylabel` | `zlabel` | `text` | `title`

## How To

- “Adding Axis Labels to Graphs”

# ylabel

---

## Purpose

Label *y*-axis

## Syntax

```
ylabel('string')
ylabel(fname)
ylabel(..., 'PropertyName', PropertyValue, ...)
ylabel(axes_handle, ...)
h = ylabel(...)
```

## Description

Every axes graphics object can have one label for the *y*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`ylabel('string')` labels the *y*-axis of the current axes.

`ylabel(fname)` evaluates the function `fname`, which must return a string. Then it displays the string beside the *y*-axis.

`ylabel(..., 'PropertyName', PropertyValue, ...)` specifies property name-value pairs for the text graphics object created by `ylabel`.

`ylabel(axes_handle, ...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ylabel(...)` returns the handle to the text object used as the label.

## Tips

Reissuing `ylabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or on the side, so that the plot does not hide it.

## Examples

This example shows how to give a string as input to `ylabel`.

```
ylabel('Population');
```

---

Passing a numeric input as input to the function.

```
ylabel(123);
```

---



---

Passing a handle as input to ylabel.

```
h = figure;  
ylabel(h);
```

---

This example displays the label in multiple lines.

```
ylabel({2010;'Population';'in Years'});
```


---

This example shows you how to use name-value pairs in ylabel.

```
ylabel('Population','fontsize',12,'fontweight','b','color','r');
```

When resizing the figure window, you may want to display graph to the scale or be able to see the text in title and labels completely. For more information about this, see “Properties Controlling Axes Size”.

## Alternatives

To control the presence and appearance of axis labels on a graph, use the Property Editor plotting tool . For details, see The Property Editor.

## See Also

strings | xlabel | ylabel | text | title

## How To

- “Adding Axis Labels to Graphs”

# zlabel

---

## Purpose

Label *z*-axis

## Syntax

```
zlabel('string')  
zlabel(fname)  
zlabel(..., 'PropertyName', PropertyValue, ...)  
zlabel(axes_handle, ...)  
h = zlabel(...)
```

## Description

Every axes graphics object can have one label for the *z*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`zlabel('string')` labels the *z*-axis of the current axes.

`zlabel(fname)` evaluates the function `fname`, which must return a string. Then display the string beside the *z*-axis.

`zlabel(..., 'PropertyName', PropertyValue, ...)` specifies property name-value pairs for the text graphics object created by `zlabel`.

`zlabel(axes_handle, ...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = zlabel(...)` returns the handle to the text object used as the label.

## Tips

Reissuing `zlabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or on the side, so the plot does not hide it.

## Examples

This example shows how to give a string as input to `zlabel`.

```
zlabel('Population');
```

---

Passing a numeric input as input to the function.

```
zlabel(123);
```

---

---

Passing a handle as input to `zlabel`.

```
h = figure;  
zlabel(h);
```

---

This example displays the label in multiple lines.

```
zlabel({'2010; 'Population'; 'in Years'});
```


---

This example shows you how to use name-value pairs in `zlabel`.

```
zlabel('Population', 'fontsize', 12, 'fontweight', 'b', 'color', 'r');
```

When resizing the figure window, you may want to display graph to the scale or be able to see the text in title and labels completely. For more information, see “Properties Controlling Axes Size”.

## Alternatives

To control the presence and appearance of axis labels on a graph, use the Property Editor plotting tool . For details, see The Property Editor.

## See Also

`strings` | `xlabel` | `ylabel` | `text` | `title`

## How To

- “Adding Axis Labels to Graphs”

**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.

---

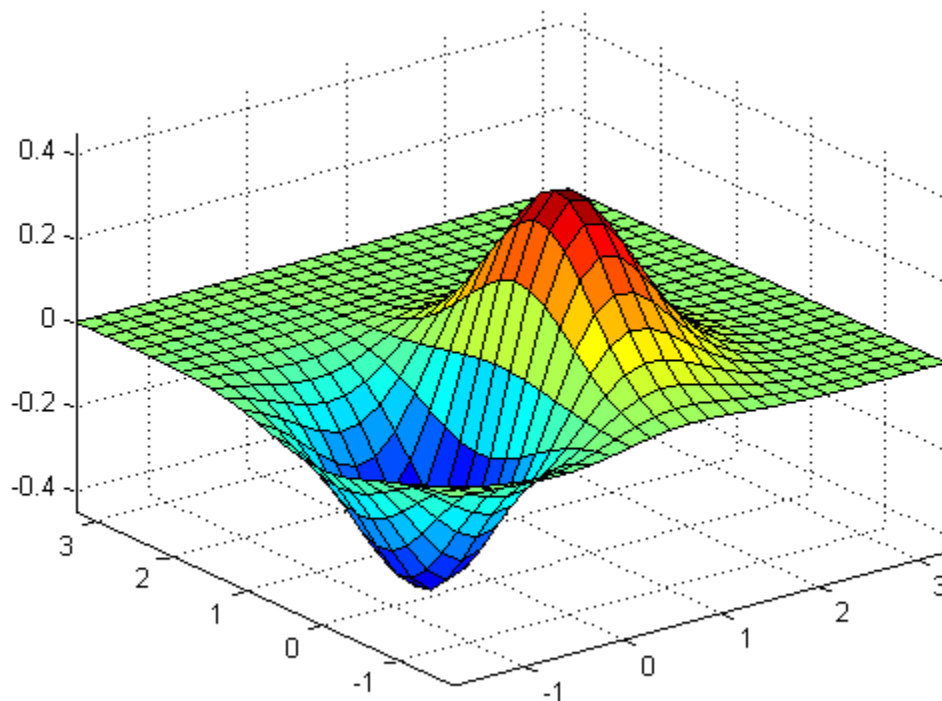
**Note** High-level plotting functions like `plot` and `surf` reset both the modes and the limits.

---


## 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

**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.

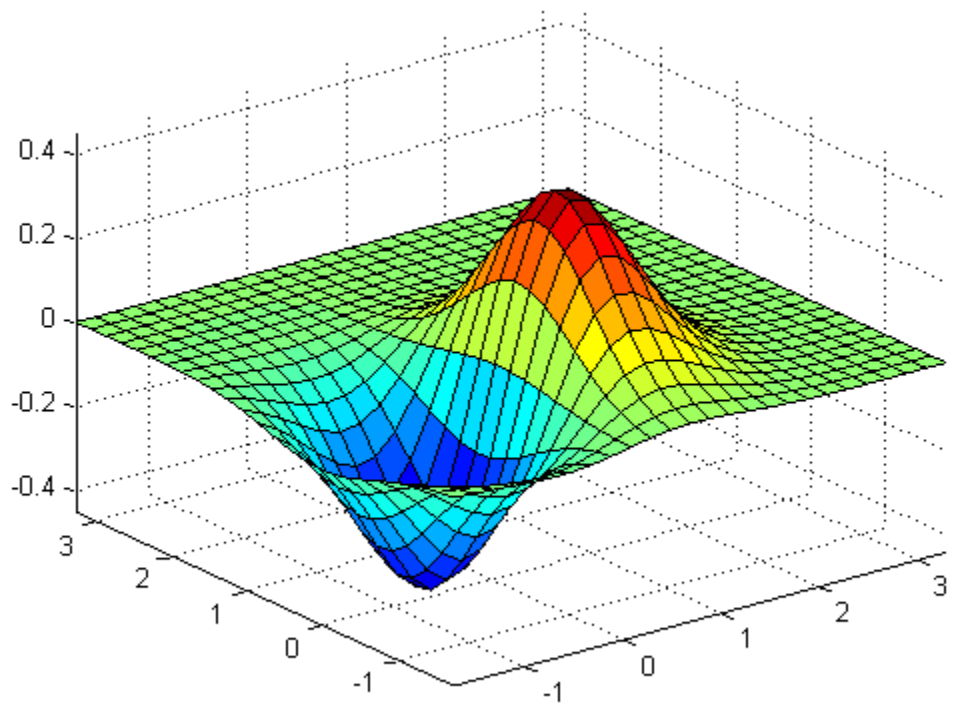
---




**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.

---

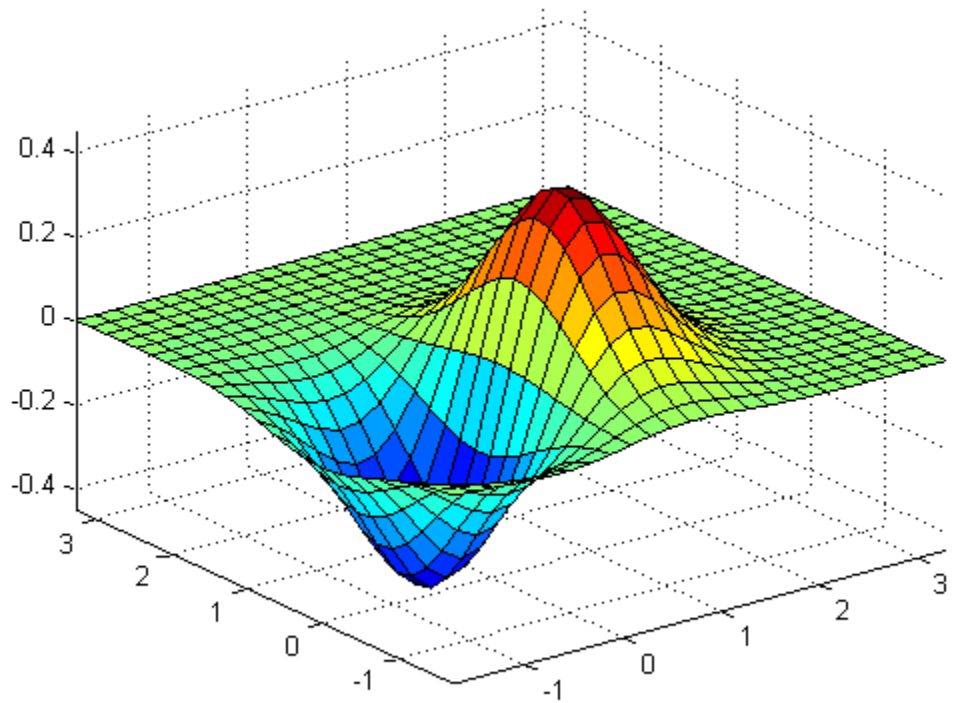
**Note** High-level plotting functions like `plot` and `surf` reset both the modes and the limits.

---


## 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:

|                                                            |                                                            |
|------------------------------------------------------------|------------------------------------------------------------|
| <code>'xlOpenXMLWorkbook'</code>                           | Spreadsheet in XLSX format (Excel 2007 or later).          |
| <code>'xlWorkbookNormal'</code> or <code>'xlExcel8'</code> | Spreadsheet in XLS format (compatible with Excel 97-2003). |
| <code>'xlCSV'</code>                                       | File in comma-separated value (CSV) format.                |
| <code>'xlHtml'</code> or <code>'xlWebArchive'</code>       | 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.

`[num,txt,row] = xlsread( ___ )` additionally returns the text fields in cell array `txt`, and the unprocessed data (numbers and text) in cell array `row` 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 `row`.

`___ = 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,row,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 `row`, 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'`

### **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).

Not supported for XLS files in `basic` mode.

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).

## **'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 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 two-dimensional array of `double` values. The array does not contain any information from header lines, or from outer rows or columns that contain nonnumeric data. `xlsread` converts text in inner rows and columns to `NaN`.

### **txt - Text data**

cell array

Text data, returned as a cell array. `xlsread` converts numeric values to an empty string, `''`.

Except for XLS files in basic mode, undefined values (such as `'#N/A'`) appear in the `txt` output as `'#N/A'`.

### **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
```

# xlsread

---

```
    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
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,row,idx] = xlsread('myExample.xlsx','MyData', '', '', @setMinMax);
disp(idx)
```

```
7      9      11      15      19      20      23      24      30
```

## Algorithms

- If you do not specify `x1Range`, then `xlsread` ignores leading blank rows and columns in the worksheet that precede your data.
- If you specify `x1Range`, then leading blank rows and columns in the worksheet that precede your data are returned in `row`, but not in `num` or `txt`.
- When the specified `x1Range` 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.

# xlswrite

---

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 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). `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](#).

# xlswrite

---

- 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`



|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Read XML document and return Document Object Model node                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Syntax</b>           | <code>DOMnode = xmlread(filename)</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b>      | <code>DOMnode = xmlread(filename)</code> reads the specified XML file and returns a Document Object Model node representing the document.                                                                                                                                                                                                                                                                                                                                                       |
| <b>Tips</b>             | <p>The display for a properly parsed document is <code>[#document: null]</code>. For example,</p> <pre>xDoc = xmlread('info.xml')</pre> <p>returns</p> <pre>xDoc =<br/>[#document: null]</pre>                                                                                                                                                                                                                                                                                                  |
| <b>Input Arguments</b>  | <p><b>filename</b></p> <p>String enclosed in single quotation marks that specifies the name of the local file or URL.</p>                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Output Arguments</b> | <p><b>DOMnode</b></p> <p>Document Object Model node, as defined by the World Wide Web consortium. For more information, see “What Is an XML Document Object Model (DOM)?”.</p>                                                                                                                                                                                                                                                                                                                  |
| <b>Examples</b>         | <p>The root element in an XML file sometimes includes an <code>xsi:noNamespaceSchemaLocation</code> attribute. The value of this attribute is the name of the preferred schema file. Call the <code>getAttribute</code> method to get this value:</p> <pre>xDoc = xmlread(fullfile(matlabroot,'toolbox',...<br/>                        'matlab','general','info.xml'));<br/><br/>xRoot = xDoc.getDocumentElement;<br/>schema = char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))</pre> |

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
```

# xmlread

---

```
        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 \(methods and properties for nodes\)](#)

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Purpose</b>          | Write XML Document Object Model node                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Syntax</b>           | <pre>xmlwrite(filename,DOMnode) str = xmlwrite(DOMnode)</pre>                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b>      | <p>xmlwrite(filename,DOMnode) writes the Document Object Model (DOM) node DOMnode to the file filename.</p> <p>str = xmlwrite(DOMnode) serializes the DOM node to a string.</p>                                                                                                                                                                                                                                                |
| <b>Input Arguments</b>  | <p><b>filename</b></p> <p>String enclosed in single quotation marks that specifies the name of a local file or a URL.</p> <p><b>DOMnode</b></p> <p>Document Object Model node, as defined by the World Wide Web consortium. For more information, see “What Is an XML Document Object Model (DOM)?”</p>                                                                                                                        |
| <b>Output Arguments</b> | <p><b>str</b></p> <p>String that contains the serialized DOM node as it appears in an XML file.</p>                                                                                                                                                                                                                                                                                                                            |
| <b>Examples</b>         | <p>Create and view an XML document:</p> <pre>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</pre> |

# xmlwrite

---

```
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.

| <b>A</b> | <b>B</b> | <b>C</b> |
|----------|----------|----------|
| 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

# xslt

---

**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**

```
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(m,n,p,...)
B = zeros([m n p ...])
B = zeros(size(A))
Y = zeros
zeros(m, n,...,classname)
zeros([m,n,...],classname)
```

**Description** `B = zeros(n)` returns an  $n$ -by- $n$  matrix of zeros. An error message appears if  $n$  is not a scalar.

`B = zeros(m,n)` or `B = zeros([m n])` returns an  $m$ -by- $n$  matrix of zeros.

`B = zeros(m,n,p,...)` or `B = zeros([m n p ...])` returns an  $m$ -by- $n$ -by- $p$ -by-... array of zeros.

---

**Note** The size inputs  $m$ ,  $n$ ,  $p$ , ... should be nonnegative integers. Negative integers are treated as 0. If any trailing dimensions are 0, output  $B$  does not include those dimensions.

---

`B = zeros(size(A))` returns an array the same size as  $A$  consisting of all zeros.

`Y = zeros` with no arguments is the scalar 0.

`zeros(m, n,...,classname)` or `zeros([m,n,...],classname)` is an  $m$ -by- $n$ -by-... array of zeros of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

**Examples**

```
x = zeros(2,3,'int8');
```

**Tips**

The MATLAB language does not have a dimension statement; MATLAB automatically allocates storage for matrices. Nevertheless, for large matrices, MATLAB programs may execute faster if the `zeros` function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time. For example

```
x = zeros(1,n);  
for i = 1:n, x(i) = i; end
```

**See Also**

`eye` | `ones` | `rand` | `randn` | `complex` | `false`

# zip

---

**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

---

```
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`

**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  $1/\text{factor}$ .

`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 [%f %f].',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');
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`

## Symbols and Numerics

& 2-59 2-66  
 ' 2-48  
 \* 2-48  
 + 2-48  
 - 2-48  
 / 2-48  
 : 2-73  
 < 2-57  
 > 2-57  
 @ 2-1783  
 \ 2-48  
 ^ 2-48  
 | 2-59 2-66  
 ~ 2-59 2-66  
 && 2-66  
 == 2-57  
 ]) 2-71  
 || 2-66  
 ~= 2-57  
 1-norm 2-3720  
 2-norm (estimate of) 2-3143

## A

abs 2-76  
 absolute accuracy  
   BVP 2-570  
   DDE 2-1219  
   ODE 2-3286  
 absolute value 2-76  
 Accelerator  
   Uimenu property 2-4802  
 accumarray 2-77  
 accuracy  
   of linear equation solution 2-982  
   of matrix inversion 2-982  
 acos 2-84  
 acosh 2-88  
 acot 2-89

acoth 2-93  
 acsc 2-94  
 acsch 2-97  
 activelegend 2-3495  
 actxcontrol 2-98  
 Adams-Bashforth-Moulton ODE solver 2-3191  
   2-3206 2-3221 2-3236 2-3251 2-3266 2-3281  
 addCause, MException method 2-126  
 addevent 2-129  
 addframe  
   AVI files 2-149  
 addition (arithmetic operator) 2-48  
 addlistener 2-151  
 addpath 2-160  
 addpref function 2-162  
 addprop dynamicprops method 2-164  
 addressing selected array elements 2-73  
 addsampletocollection 2-169  
 addtdate 2-171  
 addts 2-173  
 adjacency graph 2-1322  
 airy 2-175  
 Airy functions  
   relationship to modified Bessel  
     functions 2-175  
 align function 2-177  
 aligning scattered data  
   multi-dimensional 2-2993  
 ALim, Axes property 2-327  
 all 2-183  
 allchild function 2-185  
 allocation of storage (automatic) 2-5219  
 AlphaData  
   image property 2-2183  
   surface property 2-4434  
   surfaceplot property 2-4458  
 AlphaDataMapping  
   image property 2-2184  
   patch property 2-3385  
   surface property 2-4435

- surfaceplot property 2-4458
- AmbientLightColor, Axes property 2-327
- AmbientStrength
  - Patch property 2-3386
  - Surface property 2-4435
  - surfaceplot property 2-4459
- amd 2-193
- analytical partial derivatives (BVP) 2-572
- and 2-197
- and (function equivalent for &) 2-63
- angle 2-199
- annotating graphs
  - in plot edit mode 2-3496
- Annotation
  - areaserie property 2-255
  - contourgroup property 2-1010
  - errorbarseries property 2-1415
  - hggroup property 2-2077
  - hgtransform property 2-2107
  - image property 2-2184
  - line property 2-421 2-2597
  - lineseries property 2-2615
  - Patch property 2-3386
  - quivergroup property 2-3661
  - rectangle property 2-3837
  - scattergroup property 2-4023
  - stairsereis property 2-4237
  - stemseries property 2-4269
  - Surface property 2-4436
  - surfaceplot property 2-4459
  - text property 2-4542
- annotation function 2-200
- ans 2-245
- anti-diagonal 2-2007
- any 2-246
- arccosecant 2-94
- arccosine 2-84
- arccotangent 2-89
- arcsecant 2-278
- arctangent 2-291
- four-quadrant 2-292
- arguments
  - checking number of inputs 2-2979
  - checking number of outputs 2-2988
  - passing variable numbers of 2-5028
- arithmetic operations, matrix and array
  - distinguished 2-48
- arithmetic operators
  - reference 2-48
- array
  - addressing selected elements of 2-73
  - dimension
    - rearrange 2-1692
  - displaying 2-1298
  - flip dimension of 2-1692
  - left division (arithmetic operator) 2-50
  - maximum elements of 2-2785
  - mean elements of 2-2790
  - median elements of 2-2791
  - minimum elements of 2-2892
  - multiplication (arithmetic operator) 2-49
  - of all ones 2-3306
  - of all zeros 2-5218
  - power (arithmetic operator) 2-50
  - product of elements 2-3569
  - rearrange
    - dimension 2-1692
  - removing first n singleton dimensions
    - of 2-4121
  - removing singleton dimensions of 2-4224
  - reshaping 2-3922
  - reverse dimension of 2-1692
  - right division (arithmetic operator) 2-49
  - shift circularly 2-884
  - shifting dimensions of 2-4121
  - size of 2-4134
  - sorting elements of 2-4156
  - structure 2-1889 2-3953 2-4095
  - sum of elements 2-4413
  - swapping dimensions of 2-2377 2-3464



- transpose (arithmetic operator) 2-50
  - arrays
    - detecting empty 2-2393
  - arrays, structure
    - field names of 2-1557
  - arrowhead matrix 2-964
  - ASCII
    - delimited files
      - writing 2-1317
  - ASCII data
    - converting sparse matrix after loading
      - from 2-4173
    - reading 2-1313
    - reading from disk 2-2672
    - saving to disk 2-3997
  - asec 2-278
  - asech 2-281
  - asinh 2-286
  - aspect ratio of axes 2-1116 2-3421
  - assert 2-287
  - assignin 2-289
  - atan 2-291
  - atan2 2-292
  - atanh 2-298
  - .au files
    - reading 2-311
    - writing 2-313
  - audio
    - saving in AVI format 2-314
    - signal conversion 2-2590 2-2963
  - audiodevinfo 2-299
  - audioplayer 2-301
  - audiorecorder 2-305
  - aufinfo 2-310
  - auread 2-311
  - AutoScale
    - quivergroup property 2-3662
  - AutoScaleFactor
    - quivergroup property 2-3662
  - autoselection of OpenGL 2-1597
  - auwrite 2-313
  - average of array elements 2-2790
  - average,running 2-1650
  - avi 2-314
  - avifile 2-314
  - aviinfo 2-318
  - aviread 2-320
  - axes 2-321
    - editing 2-3496
    - setting and querying data aspect ratio 2-1116
    - setting and querying limits 2-5180 2-5184 2-5188
    - setting and querying plot box aspect ratio 2-3421
  - Axes
    - creating 2-321
    - defining default properties 2-322
    - fixed-width font 2-343
    - property descriptions 2-326
  - axis 2-365
  - axis crossing. *See* zero of a function
  - azimuth (spherical coordinates) 2-4189
  - azimuth of viewpoint 2-5061
- ## B
- BackFaceLighting
    - Surface property 2-4437
    - surfaceplot property 2-4461
  - BackFaceLightingpatch property 2-3388
  - BackgroundColor
    - annotation textbox property 2-235
    - Text property 2-4543
    - Uitable property 2-4888
  - BackgroundColor
    - Uicontrol property 2-4748
  - badly conditioned 2-3720
  - balance 2-371
  - BarLayout
    - barseries property 2-422

- BarWidth
  - barseries property 2-422
- base to decimal conversion 2-442
- base two operations
  - conversion from decimal to binary 2-1233
  - logarithm 2-2691
  - next power of two 2-3138
- base2dec 2-442
- BaseLine
  - barseries property 2-422
  - stem property 2-4270
- BaseValue
  - areaseries property 2-256
  - barseries property 2-423
  - stem property 2-4270
- beep 2-443
- BeingDeleted
  - areaseries property 2-256
  - barseries property 2-424
  - contour property 2-1011
  - errorbar property 2-1416
  - group property 2-1562 2-2185 2-4545
  - hggroup property 2-2078
  - hgtransform property 2-2108
  - light property 2-2579
  - line property 2-2598
  - lineseries property 2-2616
  - quivergroup property 2-3662
  - rectangle property 2-3838
  - scatter property 2-4024
  - stairservices property 2-4238
  - stem property 2-4270
  - surface property 2-4437
  - surfaceplot property 2-4461
  - transform property 2-3388
  - Uipushtool property 2-4850
  - Uitable property 2-4889
  - Uitoggletool property 2-4919
  - Uitoolbar property 2-4933
- bench 2-445
- benchmark 2-445
- Bessel functions
  - first kind 2-454
  - modified, first kind 2-451
  - modified, second kind 2-458
  - second kind 2-462
- Bessel functions, modified
  - relationship to Airy functions 2-175
- Bessel's equation
  - (defined) 2-454
  - modified (defined) 2-451
- besseli 2-451
- besselj 2-454
- besselk 2-458
- bessely 2-462
- beta 2-466
- beta function
  - (defined) 2-466
  - incomplete (defined) 2-468
  - natural logarithm 2-471
- betainc 2-468
- betaln 2-471
- bicg 2-472
- bicgstab 2-480
- bicgstabl 2-487
- BiConjugate Gradients method 2-472
- BiConjugate Gradients Stabilized method 2-480 2-487
- bin2dec 2-493
- binary data
  - reading from disk 2-2672
  - saving to disk 2-3997
- binary to decimal conversion 2-493
- bisection search 2-1808
- bit depth
  - querying 2-2210
- bitmaps
  - writing 2-2242
- bitmax 2-504
- blanks 2-521

- removing trailing 2-1230
- blkdiag 2-522
- BMP files
  - writing 2-2242
- bold font
  - TeX characters 2-4569
- boundary value problems 2-577
- box 2-523
- Box, Axes property 2-328
- braces, curly (special characters) 2-69
- brackets (special characters) 2-69
- break 2-524
- breakpoints
  - listing 2-1182
  - removing 2-1169
  - resuming execution from 2-1172
- brush 2-529
- bsxfun 2-540
- bubble plot (scatter function) 2-4016
- Buckminster Fuller 2-4516
- BusyAction
  - areaserie property 2-256
  - Axes property 2-328
  - barseries property 2-424
  - contour property 2-1011
  - errorbar property 2-1417
  - Figure property 2-1563
  - hggroup property 2-2079
  - hgtransform property 2-2109
  - Image property 2-2186
  - Light property 2-2579
  - line property 2-2599
  - Line property 2-2616
  - patch property 2-3388
  - quivergroup property 2-3663
  - rectangle property 2-3839
  - Root property 2-3960
  - scatter property 2-4024
  - stairsereis property 2-4238
  - stem property 2-4271
  - Surface property 2-4437
  - surfaceplot property 2-4461
  - Text property 2-4545
  - Uicontextmenu property 2-4727
  - Uicontrol property 2-4749
  - Uimenu property 2-4804
  - Uipushtool property 2-4851
  - Uitable property 2-4889
  - Uitoggletool property 2-4920
  - Uitoolbar property 2-4933
- ButtonDownFcn
  - area series property 2-257
  - Axes property 2-329
  - barseries property 2-425
  - contour property 2-1012
  - errorbar property 2-1417
  - Figure property 2-1563
  - hggroup property 2-2079
  - hgtransform property 2-2109
  - Image property 2-2186
  - Light property 2-2580
  - Line property 2-2599
  - lineserie property 2-2617
  - patch property 2-3389
  - quivergroup property 2-3663
  - rectangle property 2-3839
  - Root property 2-3960
  - scatter property 2-4025
  - stairsereis property 2-4239
  - stem property 2-4271
  - Surface property 2-4438
  - surfaceplot property 2-4462
  - Text property 2-4546
  - Uicontrol property 2-4749
  - Uitable property 2-4890
- BVP solver properties
  - analytical partial derivatives 2-572
  - error tolerance 2-570
  - Jacobian matrix 2-572
  - mesh 2-574

- singular BVPs 2-573
  - solution statistics 2-575
  - vectorization 2-571
- bvp4c 2-549
- bvp5c 2-560
- bvpget 2-565
- bvpinit 2-566
- bvpset 2-569
- bvpxtend 2-577
- ## C
- calendar 2-578
- call history 2-3576
- Callback
- Uicontextmenu property 2-4728
  - Uicontrol property 2-4750
  - Uimenu property 2-4804
- CallbackObject, Root property 2-3960
- camdolly 2-584
- camera
- dolly position 2-584
  - moving camera and target positions 2-584
  - positioning to view objects 2-590
  - rotating around camera target 2-592 2-594
  - rotating around viewing axis 2-600
  - setting and querying position 2-596
  - setting and querying projection type 2-598
  - setting and querying target 2-601
  - setting and querying up vector 2-603
  - setting and querying view angle 2-605
- CameraPosition, Axes property 2-330
- CameraPositionMode, Axes property 2-331
- CameraTarget, Axes property 2-331
- CameraTargetMode, Axes property 2-331
- CameraUpVector, Axes property 2-331
- CameraUpVectorMode, Axes property 2-332
- CameraViewAngle, Axes property 2-332
- CameraViewAngleMode, Axes property 2-332
- camlookat 2-590
- camorbit 2-592
- campan 2-594
- campos 2-596
- camproj 2-598
- camroll 2-600
- camtarget 2-601
- camup 2-603
- camva 2-605
- camzoom 2-607
- cart2pol 2-611
- cart2sph 2-613
- Cartesian coordinates 2-611 2-613 2-3509 2-4189
- case
- lower to upper 2-4990
  - upper to lower 2-2703
- cast 2-617
- cat 2-618
- caxis 2-622
- Cayley-Hamilton theorem 2-3529
- cd 2-628
- CData
- Image property 2-2187
  - scatter property 2-4026
  - Surface property 2-4439
  - surfaceplot property 2-4463
  - Uicontrol property 2-4751
  - Uipushtool property 2-4851
  - Uitoggletool property 2-4920
- CDataMapping
- Image property 2-2189
  - patch property 2-3391
  - Surface property 2-4440
  - surfaceplot property 2-4463
- CDataMode
- surfaceplot property 2-4464
- CDatapatch property 2-3389
- CDataSource
- scatter property 2-4026
  - surfaceplot property 2-4464
- cdf2rdf 2-635

- cdfepoch 2-637
- cdfinfo 2-639
- cdflib
  - summary of capabilities 2-643
- cdfread 2-818
- cdfwrite 2-822
- ceil 2-825
- cell array
  - structure of, displaying 2-846
- cell2struct 2-831
- celldisp 2-840
- CellEditCallback
  - Uitable property 2-4891
- cellplot 2-846
- CellSelectionCallback
  - Uitable property 2-4893
- cgs 2-849
- char 2-856
- characters
  - conversion, in serial format specification string 2-1741
- check boxes 2-4735
- Checked, Uimenu property 2-4805
- checkerboard pattern (example) 2-3914
- checkin 2-863
  - examples 2-864
  - options 2-863
- checkout 2-865
  - examples 2-866
  - options 2-865
- child functions 2-3571
- Children
  - areaseries property 2-258
  - Axes property 2-334
  - barseries property 2-425
  - contour property 2-1013
  - errorbar property 2-1418
  - Figure property 2-1564
  - hggroup property 2-2080
  - hgtransform property 2-2110
  - Image property 2-2190
  - Light property 2-2580
  - Line property 2-2600
  - lineseries property 2-2617
  - patch property 2-3392
  - quivergroup property 2-3664
  - rectangle property 2-3840
  - Root property 2-3960
  - scatter property 2-4026
  - stairs series property 2-4240
  - stem property 2-4272
  - Surface property 2-4440
  - surfaceplot property 2-4465
  - Text property 2-4547
  - Uicontextmenu property 2-4728
  - Uicontrol property 2-4751
  - Uimenu property 2-4806
  - Uitable property 2-4893
  - Uitoolbar property 2-4934
- chol 2-868
- Cholesky factorization 2-868
  - lower triangular factor 2-3364
  - preordering for 2-964
- cholinc 2-873
- cholupdate 2-881
- circle
  - rectangle function 2-3832
- circshift 2-884
- cla 2-888
- clabel 2-889
- classes
  - field names 2-1557
  - loaded 2-2274
- clc 2-900 2-909 2-4120
- clear
  - serial port I/O 2-908
- clearing
  - Command Window 2-900
  - Java import list 2-903
- clf 2-909

- ClickedCallback
  - Uipushtool property 2-4851
  - Uitoggletool property 2-4921
- CLim, Axes property 2-334
- CLimMode, Axes property 2-335
- clipboard 2-910
- Clipping
  - areaserie property 2-258
  - Axes property 2-335
  - barseries property 2-426
  - contour property 2-1013
  - errrobar property 2-1418
  - Figure property 2-1565
  - hggroup property 2-2080
  - hgtransform property 2-2110
  - Image property 2-2190
  - Light property 2-2580
  - Line property 2-2601
  - lineseries property 2-2618
  - quivergroup property 2-3664
  - rectangle property 2-3840
  - Root property 2-3961
  - scatter property 2-4027
  - stairs series property 2-4240
  - stem property 2-4272
  - Surface property 2-4441
  - surfaceplot property 2-4465
  - Text property 2-4547
  - Uicontrol property 2-4751
  - Uitable property 2-4893
- Clippingpatch property 2-3392
- close 2-912
  - AVI files 2-915
- CloseRequestFcn, Figure property 2-1565
- closest point search 2-1342
- closest triangle search 2-4692
- closing
  - MATLAB 2-3652
- cmapeditor 2-944
- cmpermute 2-922
- cmunique 2-923
- Code Analyzer
  - function for entire folder 2-2922
  - HTML report 2-2922
- colamd 2-926
- colon operator 2-73
- color
  - quantization performed by rgb2ind 2-3939
- Color
  - annotation arrow property 2-206
  - annotation doublearrow property 2-210
  - annotation line property 2-218
  - annotation textbox property 2-235
  - Axes property 2-335
  - errorbar property 2-1418
  - Figure property 2-1568
  - Light property 2-2580
  - Line property 2-2601
  - lineseries property 2-2618
  - quivergroup property 2-3664
  - stairs series property 2-4240
  - stem property 2-4273
  - Text property 2-4547
  - textarrow property 2-224
- color approximation
  - performed by rgb2ind 2-3939
- color of fonts, see also FontColor property 2-4569
- colorbar 2-930
- colormap 2-938
  - editor 2-944
- Colormap, Figure property 2-1568
- colormaps
  - converting from RGB to HSV 2-3937
  - plotting RGB components 2-3943
  - rearranging colors in 2-922
  - removing duplicate entries in 2-923
- ColorOrder, Axes property 2-336
- ColorSpec 2-962
- colperm 2-964
- ColumnEditable

- Uitable property 2-4893
- ColumnFormat
  - Uitable property 2-4894
- ColumnName
  - Uitable property 2-4900
- ColumnWidth
  - Uitable property 2-4900
- COM
  - object methods
    - actxcontrol 2-98
    - delete 2-1260
    - events 2-1454
    - get 2-1865
    - inspect 2-2291
    - load 2-2677
    - move 2-2940
    - propedit 2-3580
    - save 2-4004
    - set 2-4071
  - server methods
    - Execute 2-1456
    - Feval 2-1527
- combinations of n elements 2-2992
- combs 2-2992
- comet 2-967
- comet3 2-969
- comma (special characters) 2-70
- command syntax 2-4519
- Command Window
  - clearing 2-900
  - cursor position 2-2133
  - get width 2-971
- commandhistory 2-970
- commands
  - help for 2-2043
  - system 2-4522
  - UNIX 2-4966
- commandwindow 2-971
- comments
  - block of 2-71
- companion 2-972
- companion matrix 2-972
- compass 2-973
- CompilerConfiguration 2-2877
- CompilerConfigurationDetails 2-2877
- complementary error function
  - (defined) 2-1403
- complete elliptic integral
  - (defined) 2-1368
  - modulus of 2-1366 2-1368
- complex 2-975 2-2174
  - exponential (defined) 2-1464
  - logarithm 2-2688 to 2-2689
  - numbers 2-2143
  - numbers, sorting 2-4156 2-4160
  - phase angle 2-199
  - See also* imaginary
- complex conjugate 2-991
  - sorting pairs of 2-1075
- complex data
  - creating 2-975
- complex numbers, magnitude 2-76
- complex Schur form 2-4042
- compression
  - lossy 2-2247
- concatenation
  - of arrays 2-618
- cond 2-982
- condeig 2-983
- condest 2-984
- condition number of matrix 2-982 2-3720
  - improving 2-371
- coneplot 2-986
- conj 2-991
- conjugate, complex 2-991
  - sorting pairs of 2-1075
- context menu 2-4723
- continuation (. . . , special characters) 2-70
- continue 2-992
- continued fraction expansion 2-3715

- contour
  - and mesh plot 2-1484
  - filled plot 2-1476
  - functions 2-1472
  - of mathematical expression 2-1473
  - with surface plot 2-1507
- contour3 2-1002
- contourc 2-1005
- contourf 2-1007
- ContourMatrix
  - contour property 2-1013
- contours
  - in slice planes 2-1032
- contourslice 2-1032
- contrast 2-1035
- conv 2-1036
- conv2 2-1038
- conversion
  - base to decimal 2-442
  - binary to decimal 2-493
  - Cartesian to cylindrical 2-611
  - Cartesian to polar 2-611
  - complex diagonal to real block diagonal 2-635
  - cylindrical to Cartesian 2-3509
  - decimal number to base 2-1227 2-1232
  - decimal to binary 2-1233
  - decimal to hexadecimal 2-1234
  - full to sparse 2-4170
  - hexadecimal to decimal 2-2071
  - integer to string 2-2304
  - lowercase to uppercase 2-4990
  - matrix to string 2-2751
  - numeric array to logical array 2-2692
  - numeric array to string 2-3160
  - partial fraction expansion to
    - pole-residue 2-3924
  - polar to Cartesian 2-3509
  - pole-residue to partial fraction expansion 2-3924
  - real to complex Schur form 2-3993
  - spherical to Cartesian 2-4189
  - string matrix to cell array 2-848
  - string to numeric array 2-4297
  - uppercase to lowercase 2-2703
  - vector to character string 2-856
- conversion characters in serial format specification string 2-1741
- convex hulls
  - multidimensional visualization 2-1045
  - two-dimensional visualization 2-1043
- convhull 2-1043
- convhulln 2-1045
- convn 2-1047
- convolution 2-1036
  - inverse. *See* deconvolution
  - two-dimensional 2-1038
- coordinate system and viewpoint 2-5061
- coordinates
  - Cartesian 2-611 2-613 2-3509 2-4189
  - cylindrical 2-611 2-613 2-3509
  - polar 2-611 2-613 2-3509
  - spherical 2-4189
- coordinates. 2-611
  - See also* conversion
- copyfile 2-1057
- copying
  - files and folders 2-1057
- copyobj 2-1060
- corrcoef 2-1062
- cosecant
  - hyperbolic 2-1090
  - inverse 2-94
  - inverse hyperbolic 2-97
- cosh 2-1068
- cosine
  - hyperbolic 2-1068
  - inverse 2-84
  - inverse hyperbolic 2-88
- cot 2-1069
- cotangent 2-1069



- hyperbolic 2-1072
- inverse 2-89
- inverse hyperbolic 2-93
- coth 2-1072
- cov 2-1073
- cplxpair 2-1075
- cputime 2-1076
- create, RandStream method 2-1077
- CreateFcn
  - areaseries property 2-258
  - Axes property 2-336
  - barseries property 2-426
  - contour property 2-1014
  - errorbar property 2-1419
  - Figure property 2-1568
  - group property 2-2110
  - hggroup property 2-2080
  - Image property 2-2190
  - Light property 2-2581
  - Line property 2-2601
  - lineseries property 2-2618
  - patch property 2-3392
  - quivergroup property 2-3665
  - rectangle property 2-3841
  - Root property 2-3961
  - scatter property 2-4027
  - stairs series property 2-4241
  - stemseries property 2-4273
  - Surface property 2-4441
  - surfaceplot property 2-4465
  - Text property 2-4547
  - Uicontextmenu property 2-4728
  - Uicontrol property 2-4752
  - Uimenu property 2-4806
  - Uipushtool property 2-4852
  - Uitable property 2-4901
  - Uitoggletool property 2-4921
  - Uitoolbar property 2-4934
- cross 2-1086
- cross product 2-1086
- csc 2-1087
- csch 2-1090
- csvread 2-1091
- csvwrite 2-1095
- ctranspose (function equivalent for \q) 2-54
- cubic interpolation 2-2345 2-2349 2-2353 2-3432
  - piecewise Hermite 2-2337
- cubic spline interpolation
  - one-dimensional 2-2337 2-2345 2-2349 2-2353
- cumprod 2-1097
- cumsum 2-1099
- cumtrapz 2-1101
- cumulative
  - product 2-1097
  - sum 2-1099
- curl 2-1103
- curly braces (special characters) 2-69
- current folder 2-628
  - changing 2-628
  - See also* search path
- CurrentAxes 2-1569
- CurrentAxes, Figure property 2-1569
- CurrentCharacter, Figure property 2-1570
- CurrentFigure, Root property 2-3961
- CurrentObject, Figure property 2-1570
- CurrentPoint
  - Axes property 2-337
  - Figure property 2-1571
- cursor images
  - reading 2-2228
- cursor position 2-2133
- Curvature, rectangle property 2-3842
- curve fitting (polynomial) 2-3521
- customverctrl 2-1107
- Cuthill-McKee ordering, reverse 2-4506 2-4516
- cylinder 2-1108
- cylindrical coordinates 2-611 2-613 2-3509

**D**

- daqread 2-1111
- daspect 2-1116
- data
  - ASCII
    - reading from disk 2-2672
    - ASCII, saving to disk 2-3997
    - binary, saving to disk 2-3997
    - computing 2-D stream lines 2-4311
    - computing 3-D stream lines 2-4313
    - formatted
      - reading from files 2-1761
    - isosurface from volume data 2-2453
    - reading binary from disk 2-2672
    - reading from files 2-4575
    - reducing number of elements in 2-3858
    - smoothing 3-D 2-4151
  - Data
    - Uitable property 2-4902
  - data aspect ratio of axes 2-1116
  - data brushing
    - different plot types 2-530
    - gestures for 2-535
    - restrictions on 2-532
  - data types
    - complex 2-975
  - data, aligning scattered
    - multi-dimensional 2-2993
  - data, ASCII
    - converting sparse matrix after loading
      - from 2-4173
  - DataAspectRatio, Axes property 2-339
  - DataAspectRatioMode, Axes property 2-341
  - datatipinfo 2-1131
  - date 2-1132
  - date and time functions 2-1392
  - dbclear 2-1169
  - dbcont 2-1172
  - dbdown 2-1173
  - dblquad 2-1174
  - dbmex 2-1176
  - dbquit 2-1177
  - dbstack 2-1179
  - dbstatus 2-1182
  - dbstep 2-1184
  - dbtype 2-1195
  - dbup 2-1196
  - DDE solver properties
    - error tolerance 2-1218
    - event location 2-1223
    - solver output 2-1219
    - step size 2-1222
  - dde23 2-1197
  - ddeget 2-1201
  - ddephas2 output function 2-1221
  - ddephas3 output function 2-1221
  - ddeplot output function 2-1221
  - ddeprint output function 2-1221
  - ddesd 2-1212
  - ddeset 2-1217
  - deal 2-1227
  - deblank 2-1230
  - debugging
    - changing workspace context 2-1173
    - changing workspace to calling file 2-1196
    - displaying function call stack 2-1179
    - files 2-3571
    - function 2-2514
    - MEX-files on UNIX 2-1176
    - removing breakpoints 2-1169
    - resuming execution from breakpoint 2-1184
    - stepping through lines 2-1184
  - dec2base 2-1227 2-1232
  - dec2bin 2-1233
  - dec2hex 2-1234
  - decic function 2-1236
  - decimal number to base conversion 2-1227
    - 2-1232
  - decimal point (.)
    - (special characters) 2-70

- to distinguish matrix and array operations 2-48
- decomposition
  - Dulmage-Mendelsohn 2-1321
  - "economy-size" 2-4495
  - Schur 2-4042
  - singular value 2-3714 2-4495
- deconv 2-1238
- deconvolution 2-1238
- definite integral 2-3626
- del operator 2-1239
- del2 2-1239
- Delaunay tessellation
  - multidimensional visualization 2-1255
- delaunayn 2-1255
- delete 2-1258 2-1260
  - serial port I/O 2-1264
  - timer object 2-1265
- delete handle method 2-1263
- DeleteFcn
  - areaseries property 2-259
  - Axes property 2-342
  - barseries property 2-427
  - contour property 2-1015
  - errorbar property 2-1419
  - Figure property 2-1572
  - hggroup property 2-2081
  - hgtransform property 2-2111
  - Image property 2-2191
  - Light property 2-2582
  - lineseries property 2-2619
  - quivergroup property 2-3665
  - Root property 2-3962
  - scatter property 2-4028
  - stairs property 2-4241
  - stem property 2-4274
  - Surface property 2-4441
  - surfaceplot property 2-4466
  - Text property 2-4548
  - Uicontextmenu property 2-4730 2-4753
  - Uimenu property 2-4807
  - Uipushtool property 2-4853
  - Uitable property 2-4903
  - Uitoggletool property 2-4922
  - Uitoolbar property 2-4936
- DeleteFcn, line property 2-2602
- DeleteFcn, rectangle property 2-3842
- DeleteFcnpatch property 2-3393
- deleting
  - files 2-1258
- delevent 2-1268
- delimiters in ASCII files 2-1313 2-1317
- delsamplefromcollection 2-1269
- density
  - of sparse matrix 2-3139
- depdir 2-1272
- dependence, linear 2-4405
- dependent functions 2-3571
- depfun 2-1273
- derivative
  - approximate 2-1288
  - polynomial 2-3518
- desktop
  - starting without 2-2772
- det 2-1277
- detecting
  - alphabetic characters 2-2425
  - empty arrays 2-2393
  - global variables 2-2411
  - logical arrays 2-2426
  - positive, negative, and zero array elements 2-4128
  - sparse matrix 2-2473
- determinant of a matrix 2-1277
- detrend 2-1278
- deval 2-1280
- diag 2-1282
- diagonal 2-1282
  - anti- 2-2007
  - k-th (illustration) 2-4661

- main 2-1282
- sparse 2-4175
- dialog 2-1284
- dialog box
  - error 2-1434
  - help 2-2048
  - input 2-2279
  - list 2-2667
  - message 2-2957
  - print 2-3559
  - question 2-3648
  - warning 2-5097
- diary 2-1286
- Diary, Root property 2-3962
- DiaryFile, Root property 2-3962
- diff 2-1288
- differences
  - between adjacent array elements 2-1288
- differential equation solvers
  - ODE boundary value problems 2-549 2-560
    - adjusting parameters 2-569
    - extracting properties 2-565
    - extracting properties of 2-1439 to 2-1440 2-4658 to 2-4659
    - forming initial guess 2-566
  - ODE initial value problems
    - adjusting parameters of 2-3284
    - extracting properties of 2-3283
  - parabolic-elliptic PDE problems 2-3444
- diffuse 2-1290
- DiffuseStrength
  - Surface property 2-4442
  - surfaceplot property 2-4466
- DiffuseStrengthpatch property 2-3393
- digamma function 2-3585
- dimension statement (lack of in MATLAB) 2-5219
- dimensions
  - size of 2-4134
- Diophantine equations 2-1846
- dir 2-1291
- direct term of a partial fraction expansion 2-3924
- directories
  - copying 2-1057
- directory, changing 2-628
- discontinuities, eliminating (in arrays of phase angles) 2-4985
- discontinuities, plotting functions with 2-1501
- discontinuous problems 2-1707
- disp 2-1298
  - memmapfile object 2-1301
  - serial port I/O 2-1304
  - timer object 2-1305
- disp, MException method 2-1302
- displaying output in Command Window 2-2938
- DisplayName
  - areaseries property 2-259
  - barseries property 2-427
  - contourgroup property 2-1015
  - errorbarseries property 2-1419
  - hggroup property 2-2081
  - hgtransform property 2-2112
  - image property 2-2191
  - Line property 2-2603
  - lineseries property 2-2619
  - Patch property 2-3393
  - quivergroup property 2-3666
  - rectangle property 2-3843
  - scattergroup property 2-4028
  - stairs series property 2-4242
  - stemseries property 2-4274
  - surface property 2-4443
  - surfaceplot property 2-4467
  - text property 2-4549
- distribution
  - Gaussian 2-1402
- dither 2-1308
- division
  - array, left (arithmetic operator) 2-50
  - array, right (arithmetic operator) 2-49

- by zero 2-2267
- matrix, left (arithmetic operator) 2-49
- matrix, right (arithmetic operator) 2-49
- of polynomials 2-1238
- divisor
  - greatest common 2-1846
- dlmread 2-1313
- dlmwrite 2-1317
- dmperm 2-1321
- Dockable, Figure property 2-1573
- dolly camera 2-584
- dot 2-1335
- dot product 2-1086 2-1335
- dot-parentheses (special characters) 2-70
- double 2-1336
- double click, detecting 2-1600
- double integral
  - numerical evaluation 2-1174
- DoubleBuffer, Figure property 2-1573
- dragrect 2-1337
- drawing shapes
  - circles and rectangles 2-3832
- DrawMode, Axes property 2-342
- drawnow 2-1339
- dsearchn 2-1342
- Dulmage-Mendelsohn decomposition 2-1321
- dynamic fields 2-70
- dynamicprops class 2-1343
- dynamicprops.addprop 2-164

**E**

- echo 2-1344
- Echo, Root property 2-3962
- echoing
  - functions 2-1344
- edge finding, Sobel technique 2-1040
- EdgeAlpha
  - patch property 2-3394
  - surface property 2-4443

- surfaceplot property 2-4467
- EdgeColor
  - annotation ellipse property 2-215
  - annotation rectangle property 2-221
  - annotation textbox property 2-235
  - areaserie property 2-260
  - barserie property 2-428
  - patch property 2-3395
  - Surface property 2-4444
  - surfaceplot property 2-4468
  - Text property 2-4550
- EdgeColor, rectangle property 2-3844
- EdgeLighting
  - patch property 2-3395
  - Surface property 2-4445
  - surfaceplot property 2-4469
- editable text 2-4736
- Editing
  - Text property 2-4551
- eig 2-1353
- eigensystem
  - transforming 2-635
- eigenvalue
  - accuracy of 2-1353
  - complex 2-635
  - matrix logarithm and 2-2697
  - modern approach to computation of 2-3514
  - of companion matrix 2-972
  - problem 2-1354 2-3519
  - problem, generalized 2-1354 2-3519
  - problem, polynomial 2-3519
  - repeated 2-1355
  - Wilkinson test matrix and 2-5152
- eigenvalues
  - effect of roundoff error 2-371
  - improving accuracy 2-371
- eigenvector
  - left 2-1354
  - matrix, generalized 2-3683
  - right 2-1354

- eigs 2-1356
- elevation (spherical coordinates) 2-4189
- elevation of viewpoint 2-5061
- ellipj 2-1366
- ellipke 2-1368
- ellipsoid 2-1370
- elliptic functions, Jacobian
  - (defined) 2-1366
- elliptic integral
  - complete (defined) 2-1368
  - modulus of 2-1366 2-1368
- else 2-1372
- elseif 2-1373
- Enable
  - Uicontrol property 2-4753
  - Uimenu property 2-4808
  - Uipushtool property 2-4854
  - Uitable property 2-4903
  - Uitogglehtool property 2-4923
- end 2-1383
- end caps for isosurfaces 2-2443
- end of line, indicating 2-71
- eomday 2-1392
- eq 2-1399
- eq, MException method 2-1401
- equal arrays
  - detecting 2-2395 2-2403
- equal sign (special characters) 2-69
- equations, linear
  - accuracy of solution 2-982
- EraseMode
  - areaseries property 2-260
  - barseries property 2-428
  - contour property 2-1016
  - errorbar property 2-1420
  - hggroup property 2-2082
  - hgtransform property 2-2112
  - Image property 2-2192
  - Line property 2-2604
  - lineseries property 2-2620
  - quivergroup property 2-3667
  - rectangle property 2-3844
  - scatter property 2-4029
  - stairs series property 2-4242
  - stem property 2-4275
  - Surface property 2-4445
  - surfaceplot property 2-4469
  - Text property 2-4551
- EraseModepatch property 2-3396
- error 2-1407
  - roundoff. *See* roundoff error
- error function
  - complementary 2-1402
  - (defined) 2-1402
  - scaled complementary 2-1402
- error message
  - displaying 2-1407
  - Index into matrix is negative or zero 2-2693
  - retrieving last generated 2-2521 2-2529
- error messages
  - Out of memory 2-3343
- error tolerance
  - BVP problems 2-570
  - DDE problems 2-1218
  - ODE problems 2-3285
- errorbars, confidence interval 2-1412
- errordlg 2-1434
- ErrorMessage, Root property 2-3962
- errors
  - MException class 2-1401
    - addCause 2-126
    - constructor 2-2883
    - disp 2-1302
    - eq 2-1401
    - getReport 2-1908
    - isequal 2-2402
    - last 2-2518
    - ne 2-3000
    - rethrow 2-3931
    - throw 2-4612

- throwAsCaller 2-4616
  - ErrorType, Root property 2-3963
  - etree 2-1439
  - etreeplot 2-1440
  - evalc 2-1443
  - evalin 2-1444
  - event location (DDE) 2-1223
  - event location (ODE) 2-3292
  - event.EventData 2-1446
  - event.listener 2-1447
  - event.PropertyEvent 2-1449
  - event.propListener 2-1450
  - events 2-1454
  - examples
    - calculating isosurface normals 2-2450
    - contouring mathematical expressions 2-1473
    - isosurface end caps 2-2443
    - isosurfaces 2-2454
    - mesh plot of mathematical function 2-1482
    - mesh/contour plot 2-1486
    - plotting filled contours 2-1477
    - plotting function of two variables 2-1491
      - 2-1494
    - plotting parametric curves 2-1494
    - polar plot of function 2-1497
    - reducing number of patch faces 2-3855
    - reducing volume data 2-3858
    - subsampling volume data 2-4410
    - surface plot of mathematical function 2-1501
    - surface/contour plot 2-1509
  - Excel spreadsheets
    - loading 2-5194
  - exclamation point (special characters) 2-71
  - Execute 2-1456
  - executing statements repeatedly 2-1717
  - executing statements repeatedly in
    - parallel 2-3359
  - execution
    - improving speed of by setting aside
      - storage 2-5219
    - pausing function 2-3419
    - resuming from breakpoint 2-1172
    - time for files 2-3571
  - exifread 2-1458
  - exist 2-1459
  - exit 2-1463
  - expint 2-1465
  - expm 2-1466
  - expm1 2-1468
  - exponential 2-1464
    - complex (defined) 2-1464
    - integral 2-1465
    - matrix 2-1466
  - exponentiation
    - array (arithmetic operator) 2-50
    - matrix (arithmetic operator) 2-50
  - export2wsdlg 2-1469
  - extension, filename
    - .mat 2-3997
  - Extent
    - Text property 2-4553
    - Uicontrol property 2-4755
    - Uitable property 2-4904
  - ezcontour 2-1472
  - ezcontourf 2-1476
  - ezmesh 2-1480
  - ezmeshc 2-1484
  - ezplot 2-1488
  - ezplot3 2-1493
  - ezpolar 2-1496
  - ezsurf 2-1499
  - ezsurf 2-1507
- F**
- FaceAlpha
    - annotation textbox property 2-236
  - FaceAlphapatch property 2-3397
  - FaceAlphasurface property 2-4446
  - FaceAlphasurfaceplot property 2-4471

- FaceColor
  - annotation ellipse property 2-215
  - annotation rectangle property 2-221
  - areaserie property 2-262
  - barserie property 2-429
  - Surface property 2-4447
  - surfaceplot property 2-4471
- FaceColor, rectangle property 2-3845
- FaceColorpatch property 2-3398
- FaceLighting
  - Surface property 2-4447
  - surfaceplot property 2-4472
- FaceLightingpatch property 2-3398
- faces, reducing number in patches 2-3854
- Faces,patch property 2-3399
- FaceVertexAlphaData, patch property 2-3400
- FaceVertexCData,patch property 2-3400
- factor 2-1514
- factorial 2-1515
- factorization
  - LU 2-2721
  - QZ 2-3520 2-3683
- factorization, Cholesky 2-868
  - preordering for 2-964
- factors, prime 2-1514
- false 2-1516
- fclose
  - serial port I/O 2-1518
- feather 2-1519
- feval 2-1525
- Feval 2-1527
- fft 2-1532
- FFT. *See* Fourier transform
- fft2 2-1537
- fftn 2-1539
- fftshift 2-1541
- fftw 2-1544
- FFTW 2-1535
- fgetl
  - serial port I/O 2-1550
- fgets
  - serial port I/O 2-1554
- field names of a structure, obtaining 2-1557
- fieldnames 2-1557
- fields, of structures
  - dynamic 2-70
- figure 2-1559
- Figure
  - creating 2-1559
  - defining default properties 2-1561
  - properties 2-1562
  - redrawing 2-3860
- figure windows
  - moving in front of MATLAB® desktop 2-4120
- figure windows, displaying 2-1660
- figurepalette 2-1620
- figures
  - annotating 2-3496
  - saving 2-4008
- Figures
  - updating from file 2-1339
- file
  - extension, getting 2-1637
  - modification date 2-1291
- file formats
  - getting list of supported formats 2-2212
  - reading 2-1111 2-2226
  - writing 2-2240
- file size
  - querying 2-2210
- filebrowser 2-1634
- filemarker 2-1635
- filename
  - parts 2-1637
  - temporary 2-4531
- filename extension
  - .mat 2-3997
- fileparts 2-1637
- files
  - ASCII delimited



- reading 2-1313
- writing 2-1317
- checking existence of 2-1459
- contents, listing 2-4694
- copying 2-1057
- copying with copyfile 2-1057
- debugging with profile 2-3571
- deleting 2-1258
- Excel spreadsheets
  - loading 2-5194
- fig 2-4008
- figure, saving 2-4008
- line numbers, listing 2-1195
- listing 2-1291
  - in folder 2-5124
- listing contents of 2-4694
- locating 2-5129
- mdl 2-4008
- model, saving 2-4008
- opening in Windows applications 2-5153
- optimizing 2-3571
- path, getting 2-1637
- pathname for 2-5129
- reading
  - data from 2-4575
  - formatted 2-1761
- reading data from 2-1111
- reading image data from 2-2226
- size, determining 2-1293
- sound
  - reading 2-311 2-5109
  - writing 2-313 to 2-314 2-5114
- startup 2-2764
- .wav
  - reading 2-5109
  - writing 2-5114
- WK1
  - loading 2-5158
  - writing to 2-5161
- writing image data to 2-2240
- filesep 2-1641
- fill 2-1643
- Fill
  - contour property 2-1017
- fill3 2-1646
- filter 2-1649
  - digital 2-1649
  - finite impulse response (FIR) 2-1649
  - infinite impulse response (IIR) 2-1649
  - two-dimensional 2-1038
- filter2 2-1652
- find 2-1654
- findall function 2-1659
- findfigs 2-1660
- finding 2-1654
  - sign of array elements 2-4128
  - zero of a function 2-1804
  - See also* detecting
- findobj 2-1661
- findobj handle method 2-1665
- findprop handle method 2-1668
- findstr 2-1670
- finish 2-1671
- finish.m 2-3652
- FIR filter 2-1649
- FitBoxToText, annotation textbox
  - property 2-236
- FitHeightToText
  - annotation textbox property 2-236
- fitsinfo 2-1675
- fix 2-1691
- fixed-width font
  - axes 2-343
  - text 2-4554
  - uicontrols 2-4756
  - uitables 2-4905
- FixedColors, Figure property 2-1574
- FixedWidthFontName, Root property 2-3963
- flints 2-2963
- flip

- array dimension 2-1692
- flip array
  - along dimension 2-1692
- flip matrix
  - on horizontal axis 2-1694
  - on vertical axis 2-1693
- flipdim 2-1692
- flipplr 2-1693
- flipud 2-1694
- floating-point
  - integer, maximum 2-504
- floating-point arithmetic, IEEE
  - smallest positive number 2-3825
- floor 2-1696
- flow control
  - break 2-524
  - end 2-1383
  - error 2-1409
  - for 2-1717
  - keyboard 2-2514
  - parfor 2-3359
  - return 2-3935
- fminbnd 2-1698
- fminsearch 2-1703
- folder
  - listing MATLAB files in 2-5124
  - root 2-2766
  - temporary
    - system 2-4530
- folders
  - adding to search path 2-160
  - checking existence of 2-1459
  - copying 2-1057
  - creating 2-2903
  - listing contents of 2-1291
  - removing from search path 2-3954
- font
  - fixed-width, axes 2-343
  - fixed-width, text 2-4554
  - fixed-width, uicontrols 2-4756
  - fixed-width, uitables 2-4905
- FontAngle
  - annotation textbox property 2-238
  - Axes property 2-343
  - Text property 2-225 2-4553
  - Uicontrol property 2-4755
  - Uitable property 2-4905
- FontName
  - annotation textbox property 2-238
  - Axes property 2-343
  - Text property 2-4553
  - textarrow property 2-225
  - Uicontrol property 2-4756
  - Uitable property 2-4905
- fonts
  - bold 2-225 2-239 2-4554
  - specifying size 2-4554
  - TeX characters
    - bold 2-4569
    - italics 2-4569
    - specifying family 2-4569
    - specifying size 2-4569
  - units 2-225 2-239 2-4555
- FontSize
  - annotation textbox property 2-239
  - Axes property 2-344
  - Text property 2-4554
  - textarrow property 2-225
  - Uicontrol property 2-4757
  - Uitable property 2-4906
- FontUnits
  - Axes property 2-344
  - Text property 2-4555
  - Uicontrol property 2-4757
  - Uitable property 2-4906
- FontWeight
  - annotation textbox property 2-239
  - Axes property 2-344
  - Text property 2-4554
  - textarrow property 2-225

- Uicontrol property 2-4757
  - Uitable property 2-4906
- fopen
  - serial port I/O 2-1715
- for 2-1717
- ForegroundColor
  - Uicontrol property 2-4757
  - Uimenu property 2-4808
  - Uitable property 2-4907
- Format 2-3963
- FormatSpacing, Root property 2-3964
- formatted data
  - reading from file 2-1761
- Fourier transform
  - algorithm, optimal performance of 2-1535
    - 2-2160 2-2162 2-3138
  - as method of interpolation 2-2351
  - discrete, n-dimensional 2-1539
  - discrete, one-dimensional 2-1532
  - discrete, two-dimensional 2-1537
  - fast 2-1532
  - inverse, n-dimensional 2-2164
  - inverse, one-dimensional 2-2160
  - inverse, two-dimensional 2-2162
  - shifting the zero-frequency component
    - of 2-1542
- fplot 2-1728 2-1744
- fprintf
  - serial port I/O 2-1741
- fraction, continued 2-3715
- fragmented memory 2-3343
- frame2im 2-1744
- frames 2-4736
- fread
  - serial port I/O 2-1753
- freqspace 2-1759
- frequency response
  - desired response matrix
    - frequency spacing 2-1759
- frequency vector 2-2700

- fromName meta.class method 2-2834
  - fromName meta.package method 2-2857
- fscanf
  - serial port I/O 2-1766
- full 2-1775
- func2str 2-1778
- function
  - echoing commands 2-1344
- function handle 2-1783
- function handles
  - overview of 2-1783
- function syntax 2-4519
- functions 2-1787
  - call history 2-3576
  - call stack for 2-1179
  - checking existence of 2-1459
  - debugging 2-2514
  - finding using keywords 2-2701
  - help for 2-2043
  - in memory 2-2274
  - locating 2-5129
  - locking (preventing clearing) 2-2924
  - pathname for 2-5129
  - pausing execution of 2-3419
  - that work down the first non-singleton
    - dimension 2-4121
  - unlocking (allowing clearing) 2-2974
- funm 2-1791
- fwrite
  - serial port I/O 2-1800
- fzero 2-1804

## G

- gallery 2-1810
- gamma function
  - (defined) 2-1837
  - incomplete 2-1837
  - logarithm of 2-1837
  - logarithmic derivative 2-3585

- Gauss-Kronrod quadrature 2-3640
- Gaussian distribution function 2-1402
- Gaussian elimination
  - (as algorithm for solving linear equations) 2-2371
  - Gauss Jordan elimination with partial pivoting 2-3991
  - LU factorization 2-2721
- gca 2-1843
- gcbf function 2-1844
- gcbo function 2-1845
- gcd 2-1846
- gcf 2-1848
- gco 2-1849
- ge 2-1850
- generalized eigenvalue problem 2-1354 2-3519
- genpath 2-1852
- genvarname 2-1854
- geodesic dome 2-4516
- get 2-1858
  - memmapfile object 2-1868
  - serial port I/O 2-1874
  - timer object 2-1876
- get (tscollection) 2-1878
- get hgsetget class method 2-1867
- get, RandStream method 2-1873
- getabstime (tscollection) 2-1879
- getAllPackages meta.package method 2-2858
- getappdata function 2-1881
- getCompilerConfigurations 2-2877
- getDefaultStream, RandStream method 2-1886 2-3712
- getdisp hgsetget class method 2-1887
- getenv 2-1888
- getframe 2-1894
  - image resolution and 2-1895
- getpixelposition 2-1901
- getpref function 2-1903
- getReport, MException method 2-1908
- getsamplingsingtime (tscollection) 2-1911
- gettimeseriesnames 2-1914
- gettsafteratevent 2-1915
- gettsafterevent 2-1916
- gettsatevent 2-1917
- gettsbeforeatevent 2-1918
- gettsbeforeevent 2-1919
- gettsbetweenevents 2-1920
- GIF files
  - writing 2-2242
- ginput function 2-1926
- global 2-1929
- global variable
  - defining 2-1929
- gmres 2-1933
- golden section search 2-1701
- Goup
  - defining default properties 2-2106
- gplot 2-1944
- grabcode function 2-1946
- gradient 2-1948
- gradient, numerical 2-1948
- graph
  - adjacency 2-1322
  - graph theory 2-4969
- graphics objects
  - Axes 2-321
  - Figure 2-1559
  - getting properties 2-1858
  - Image 2-2175
  - Light 2-2577
  - Line 2-2591
  - Patch 2-3365
  - resetting properties 2-3919
  - Root 2-3959
  - setting properties 2-4063
  - Surface 2-4429
  - Text 2-4537
  - uicontextmenu 2-4723
  - Uicontrol 2-4735
  - Uimenu 2-4797

- graphics objects, deleting 2-1258
- graphs
  - editing 2-3496
- graymon 2-1951
- greatest common divisor 2-1846
- Greek letters and mathematical symbols 2-229
  - 2-241 2-4567
- grid 2-1952
- grid arrays
  - for volumetric plots 2-2826
  - multi-dimensional 2-2993
- griddatan 2-1963
- GridLineStyle, Axes property 2-345
- group
  - hggroup function 2-2074
- gsvd 2-1969
- gt 2-1975
- gtext 2-1977
- guidata function 2-1978
- GUIDE
  - object methods
    - inspect 2-2291
- guihandles function 2-1983
- GUIs, printing 2-3553
- gunzip 2-1984
- gzip 2-1986
  
- H**
- hadamard 2-2003
- Hadamard matrix 2-2003
  - subspaces of 2-4405
- handle class 2-2004
- handle graphics
  - hgtransform 2-2095
- handle graphicshggroup 2-2074
- handle relational operators 2-3906
- handle.addlistener 2-151
- handle.delete 2-1263
- handle.findobj 2-1665
- handle.findprop 2-1668
- handle.isvalid 2-2483
- handle.notify 2-3146
- HandleVisibility
  - areaseries property 2-262
  - Axes property 2-345
  - barseries property 2-430
  - contour property 2-1017
  - errorbar property 2-1421
  - Figure property 2-1575
  - hggroup property 2-2084
  - hgtransform property 2-2114
  - Image property 2-2193
  - Light property 2-2583
  - Line property 2-2605
  - lineseries property 2-2621
  - patch property 2-3402
  - quivergroup property 2-3668
  - rectangle property 2-3845
  - Root property 2-3964
  - stairs series property 2-4244
  - stem property 2-4276
  - Surface property 2-4448
  - surfaceplot property 2-4472
  - Text property 2-4555
  - Uicontextmenu property 2-4730
  - Uicontrol property 2-4758
  - Uimenu property 2-4808
  - Uipushtool property 2-4854
  - Uitable property 2-4907
  - Uitoggletool property 2-4923
  - Uitoolbar property 2-4936
- hankel 2-2007
- Hankel matrix 2-2007
- HDF
  - appending to when saving
    - (WriteMode) 2-2246
  - compression 2-2246
  - setting JPEG quality when writing 2-2246
- HDF files

- writing images 2-2242
- HDF4
  - summary of capabilities 2-2008
- HDF5
  - high-level access 2-2011
  - summary of capabilities 2-2011
- HDF5 class
  - low-level access 2-2012
- hdf5info 2-2015
- hdf5read 2-2017
- hdf5write 2-2019
- hdfinfo 2-2023
- hdfread 2-2031
- hdftool 2-2042
- Head1Length
  - annotation doublearrow property 2-210
- Head1Style
  - annotation doublearrow property 2-211
- Head1Width
  - annotation doublearrow property 2-212
- Head2Length
  - annotation doublearrow property 2-210
- Head2Style
  - annotation doublearrow property 2-212
- Head2Width
  - annotation doublearrow property 2-212
- HeadLength
  - annotation arrow property 2-206
  - textarrow property 2-226
- HeadStyle
  - annotation arrow property 2-206
  - textarrow property 2-226
- HeadWidth
  - annotation arrow property 2-207
  - textarrow property 2-227
- Height
  - annotation ellipse property 2-215
- help 2-2043
  - keyword search in functions 2-2701
  - online 2-2043
- helpbrowser 2-2046
- helpdesk 2-2047
- helpdlg 2-2048
- helpwin 2-2050
- Hermite transformations, elementary 2-1846
- hess 2-2051
- Hessenberg form of a matrix 2-2051
- hex2dec 2-2071
- hex2num 2-2072
- hgsetget class 2-2094
- hgsetget.get 2-1867
- hgsetget.getdisp 2-1887
- hgsetget.set 2-4072
- hidden 2-2120
- Hierarchical Data Format (HDF) files
  - writing images 2-2242
- hilb 2-2123
- Hilbert matrix 2-2123
  - inverse 2-2373
- hist 2-2124
- hisc 2-2128
- HitTest
  - areaseries property 2-263
  - Axes property 2-346
  - barseries property 2-431
  - contour property 2-1019
  - errorbar property 2-1423
  - Figure property 2-1576
  - hggroup property 2-2085
  - hgtransform property 2-2115
  - Image property 2-2195
  - Light property 2-2584
  - Line property 2-2606
  - lineseries property 2-2623
  - Patch property 2-3403
  - quivergroup property 2-3669
  - rectangle property 2-3846
  - Root property 2-3964
  - scatter property 2-4032
  - stairs series property 2-4245

- stem property 2-4278
  - Surface property 2-4449
  - surfaceplot property 2-4474
  - Text property 2-4556
  - Uicontrol property 2-4759
  - Uipushtool property 2-4855
  - Uitable property 2-4908
  - Uitoggletool property 2-4924
  - Uitoolbarl property 2-4937
  - HitTestArea
    - areaserie property 2-264
    - barseries property 2-431
    - contour property 2-1019
    - errorbar property 2-1423
    - quivergroup property 2-3670
    - scatter property 2-4032
    - stairs property 2-4245
    - stem property 2-4278
  - hold 2-2131
  - home 2-2133
  - HorizontalAlignment
    - Text property 2-4557
    - textbox property 2-227 2-239
    - Uicontrol property 2-4759
  - horzcat 2-2134
  - horzcat (function equivalent for [,]) 2-71
  - horzcat (tscollection) 2-2136
  - hostid 2-2137
  - hsv2rgb 2-2139
  - hyperbolic
    - cosecant 2-1090
    - cosecant, inverse 2-97
    - cosine 2-1068
    - cosine, inverse 2-88
    - cotangent 2-1072
    - cotangent, inverse 2-93
    - secant 2-4049
    - secant, inverse 2-281
    - sine 2-4133
    - sine, inverse 2-286
    - tangent 2-4527
    - tangent, inverse 2-298
  - hyperlink
    - displaying in Command Window 2-1298
  - hyperplanes, angle between 2-4405
  - hypot 2-2140
- I**
- i 2-2143
  - icon images
    - reading 2-2228
  - identity matrix
    - sparse 2-4186
  - idivide 2-2155
  - IEEE floating-point arithmetic
    - smallest positive number 2-3825
  - ifft 2-2160
  - ifft2 2-2162
  - ifftn 2-2164
  - ifftshift 2-2166
  - IIR filter 2-1649
  - ilu 2-2167
  - im2java 2-2172
  - imag 2-2174
  - image 2-2175
  - Image
    - creating 2-2175
    - properties 2-2183
  - image types
    - querying 2-2210
  - images
    - file formats 2-2226 2-2240
    - reading data from files 2-2226
    - returning information about 2-2209
    - writing to files 2-2240
  - Images
    - converting MATLAB image to Java
      - Image 2-2172
  - imagesc 2-2200

- imaginary 2-2174
  - part of complex number 2-2174
  - unit (`sqrt(\xd0 1)`) 2-2143 2-2488
  - See also* complex
- imapprox 2-2205
- imfinfo
  - returning file information 2-2209
- imformats 2-2212
- import 2-2215
- importing
  - Java class and package names 2-2215
- imread 2-2226
- imwrite 2-2240
- incomplete beta function
  - (defined) 2-468
- incomplete gamma function
  - (defined) 2-1838
- ind2sub 2-2263
- Index into matrix is negative or zero (error message) 2-2693
- indexed images
  - converting from RGB 2-3938
- indexing
  - logical 2-2692
- indices, array
  - of sorted elements 2-4157
- Inf 2-2267
- infinity 2-2267
- info 2-2270
- information
  - returning file information 2-2209
- inline 2-2271
- inmem 2-2274
- inpolygon 2-2276
- input 2-2278
  - checking number of arguments 2-2979
  - name of array passed as 2-2283
  - prompting users for 2-2278
- inputdlg 2-2279
- inputname 2-2283
- inspect 2-2291
- installation, root folder 2-2766
- instance properties 2-164
- instrcallback 2-2299
- instrfind 2-2300
- instrfindall 2-2302
  - example of 2-2303
- int2str 2-2304
- integer
  - floating-point, maximum 2-504
- IntegerHandle
  - Figure property 2-1576
- integration
  - polynomial 2-3525
  - quadrature 2-3626 2-3636
- interp1 2-2336
- interp1q 2-2342
- interp2 2-2344
- interp3 2-2348
- interpft 2-2351
- interpfn 2-2352
- interpolated shading and printing 2-3554
- interpolation
  - cubic method 2-2337 2-2345 2-2349 2-2353
  - cubic spline method 2-2337 2-2345 2-2349 2-2353
  - FFT method 2-2351
  - linear method 2-2337 2-2345 2-2349 2-2353
  - multidimensional 2-2352
  - nearest neighbor method 2-2337 2-2345 2-2349 2-2353
  - one-dimensional 2-2336
  - three-dimensional 2-2348
  - two-dimensional 2-2344
- Interpreter
  - Text property 2-4558
  - textarrow property 2-227
  - textbox property 2-239
- interpstreamspeed 2-2356
- Interruptible



- areaseries property 2-264
- Axes property 2-346
- barseries property 2-432
- contour property 2-1019
- errorbar property 2-1423
- Figure property 2-1577
- hggroup property 2-2085
- hgtransform property 2-2115
- Image property 2-2195
- Light property 2-2584
- Line property 2-2606
- lineseries property 2-2623
- patch property 2-3403
- quivergroup property 2-3670
- rectangle property 2-3847
- Root property 2-3964
- scatter property 2-4033
- stairs series property 2-4246
- stem property 2-4278
- Surface property 2-4449 2-4474
- Text property 2-4559
- Uicontextmenu property 2-4731
- Uicontrol property 2-4760
- Uimenu property 2-4809
- Uipushtool property 2-4855
- Uitable property 2-4908
- Uitoggletool property 2-4924
- Uitoolbar property 2-4937
- intersect 2-2360
- intmax 2-2369
- intmin 2-2370
- inv 2-2371
- inverse
  - cosecant 2-94
  - cosine 2-84
  - cotangent 2-89
  - Fourier transform 2-2160 2-2162 2-2164
  - Hilbert matrix 2-2373
  - hyperbolic cosecant 2-97
  - hyperbolic cosine 2-88
  - hyperbolic cotangent 2-93
  - hyperbolic secant 2-281
  - hyperbolic sine 2-286
  - hyperbolic tangent 2-298
  - of a matrix 2-2371
  - secant 2-278
  - tangent 2-291
  - tangent, four-quadrant 2-292
- inversion, matrix
  - accuracy of 2-982
- InvertHardCopy, Figure property 2-1577
- invhilb 2-2373
- involutary matrix 2-3364
- ipermute 2-2377
- is\* 2-2378
- isa 2-2381
- isappdata function 2-2384
- iscell 2-2385
- iscellstr 2-2386
- ischar 2-2387
- isdir 2-2390
- isempty 2-2393
- isempty (tscollection) 2-2394
- isequal 2-2395
- isequal, MException method 2-2402
- isequalwithequalnans 2-2403
- isfield 2-2407
- isfinite 2-2409
- isfloat 2-2410
- isglobal 2-2411
- ishandle 2-2413
- ishghandle 2-2414
- isinf 2-2416
- isinteger 2-2417
- isjava 2-2420
- iskeyword 2-2423
- isletter 2-2425
- islogical 2-2426
- ismac 2-2427
- ismember 2-2429

- isnan 2-2438
- isnumeric 2-2439
- isocap 2-2443
- isonormals 2-2450
- isosurface 2-2453
  - calculate data from volume 2-2453
  - end caps 2-2443
  - vertex normals 2-2450
- ispc 2-2458
- ispref function 2-2459
- isprime 2-2460
- isreal 2-2464
- isscalar 2-2468
- issorted 2-2469
- isspace 2-2472 2-2475
- issparse 2-2473
- isstr 2-2474
- isstruct 2-2479
- isstudent 2-2480
- isunix 2-2482
- isvalid 2-2484
  - timer object 2-2485
- isvalid handle method 2-2483
- isvarname 2-2486
- isvector 2-2487
- italics font
  - TeX characters 2-4569

**J**

- j 2-2488
- Jacobi rotations 2-4207
- Jacobian elliptic functions
  - (defined) 2-1366
- Jacobian matrix (BVP) 2-572
- Jacobian matrix (ODE) 2-3293
  - generating sparse numerically 2-3294 2-3296
  - specifying 2-3294 2-3296
  - vectorizing ODE function 2-3294 to 2-3296

- Java
  - class names 2-903 2-2215
  - object methods
    - inspect 2-2291
  - objects 2-2420
- Java Image class
  - creating instance of 2-2172
- Java import list
  - adding to 2-2215
  - clearing 2-903
- Java version used by MATLAB 2-5041
- java\_method 2-2503
- java\_object 2-2507
- javaMethod 2-2503
- javaObject 2-2507
- joining arrays. *See* concatenation
- Joint Photographic Experts Group (JPEG)
  - writing 2-2242
- JPEG
  - setting Bitdepth 2-2246
  - specifying mode 2-2247
- JPEG 2000
  - setting tile size 2-2248
- JPEG 2000 comment
  - setting when writing a JPEG 2000 image 2-2247
  - specifying 2-2247
- JPEG comment
  - setting when writing a JPEG image 2-2246
- JPEG files
  - parameters that can be set when writing 2-2246
  - writing 2-2242
- JPEG quality
  - setting when writing a JPEG image 2-2247 to 2-2248 2-2252
  - setting when writing an HDF image 2-2246
- JPEG2000 files
  - parameters that can be set when writing 2-2247

jvm  
 version used by MATLAB 2-5041

## K

K>> prompt  
 keyboard function 2-2514  
 keep  
 some variables when clearing 2-906  
 keyboard 2-2514  
 keyboard mode 2-2514  
 terminating 2-3935  
 KeyPressFcn  
 Uicontrol property 2-4762  
 Uitable property 2-4910  
 KeyPressFcn, Figure property 2-1578  
 KeyReleaseFcn, Figure property 2-1581  
 keyword search in functions 2-2701  
 keywords  
 iskeyword function 2-2423  
 kron 2-2516  
 Kronecker tensor product 2-2516  
 Krylov subspaces 2-4606

## L

Label, Uimenu property 2-4810  
 labeling  
 axes 2-5174 2-5176 2-5178  
 matrix columns 2-1298  
 plots (with numeric values) 2-3160  
 LabelSpacing  
 contour property 2-1021  
 Laplacian 2-1239  
 Laplacian matrix 2-4969  
 largest array elements 2-2785  
 last, MException method 2-2518  
 lasterr 2-2521  
 lasterror 2-2524  
 lastwarn 2-2529

LaTeX, see TeX 2-229 2-241 2-4567  
 Layer, Axes property 2-348  
 Layout Editor  
 starting 2-1982  
 lcm 2-2531  
 LData  
 errorbar property 2-1425  
 LDataSource  
 errorbar property 2-1425  
 ldivide (function equivalent for ./) 2-53  
 le 2-2540  
 least common multiple 2-2531  
 least squares  
 polynomial curve fitting 2-3521  
 problem, overdetermined 2-3473  
 legend 2-2542  
 properties 2-2547  
 setting text properties 2-2547  
 legendre 2-2555  
 Legendre functions  
 (defined) 2-2555  
 Schmidt semi-normalized 2-2555  
 length  
 serial port I/O 2-2562  
 length (tscollection) 2-2563  
 LevelList  
 contour property 2-1021  
 LevelListMode  
 contour property 2-1021  
 LevelStep  
 contour property 2-1022  
 LevelStepMode  
 contour property 2-1022  
 license 2-2573  
 light 2-2577  
 Light  
 creating 2-2577  
 defining default properties 2-2182 2-2578  
 properties 2-2579  
 Light object

- positioning in spherical coordinates 2-2588
  - lightangle 2-2588
  - lighting 2-2589
  - limits of axes, setting and querying 2-5180
    - 2-5184 2-5188
  - line 2-2591
    - editing 2-3496
  - Line
    - creating 2-2591
    - defining default properties 2-2595
    - properties 2-2597 2-2615
  - line numbers in files 2-1195
  - linear audio signal 2-2590 2-2963
  - linear dependence (of data) 2-4405
  - linear equation systems
    - accuracy of solution 2-982
  - linear equation systems, methods for solving
    - matrix inversion (inaccuracy of) 2-2371
  - linear interpolation 2-2337 2-2345 2-2349 2-2353
  - linear regression 2-3521
  - linearly spaced vectors, creating 2-2664
  - LineColor
    - contour property 2-1022
  - lines
    - computing 2-D stream 2-4311
    - computing 3-D stream 2-4313
    - drawing stream lines 2-4315
  - LineStyle 2-2633
  - LineStyle
    - annotation arrow property 2-208
    - annotation doublearrow property 2-212
    - annotation ellipse property 2-215
    - annotation line property 2-218
    - annotation rectangle property 2-222
    - annotation textbox property 2-240
    - areaserie property 2-265
    - barseries property 2-433
    - contour property 2-1023
    - errorbar property 2-1425
    - Line property 2-2608
  - lineseries property 2-2624
  - patch property 2-3404
  - quivergroup property 2-3671
  - rectangle property 2-3848
  - stairsereis property 2-4247
  - stem property 2-4280
  - surface object 2-4451
  - surfaceplot object 2-4475
  - text object 2-4561
  - textarrow property 2-228
- LineStyleOrder
    - Axes property 2-348
  - LineWidth
    - annotation arrow property 2-208
    - annotation doublearrow property 2-213
    - annotation ellipse property 2-216
    - annotation line property 2-219
    - annotation rectangle property 2-222
    - annotation textbox property 2-240
    - areaserie property 2-266
    - Axes property 2-349
    - barseries property 2-434
    - contour property 2-1023
    - errorbar property 2-1426
    - Line property 2-2608
    - lineseries property 2-2625
    - Patch property 2-3405
    - quivergroup property 2-3672
    - rectangle property 2-3848
    - scatter property 2-4034
    - stairsereis property 2-4248
    - stem property 2-4280
    - Surface property 2-4451
    - surfaceplot property 2-4476
    - text object 2-4562
    - textarrow property 2-228
  - linkaxes 2-2638
  - linkdata 2-2647
  - linkprop 2-2655
  - linsolve 2-2661

- linspace 2-2664
  - list boxes 2-4736
    - defining items 2-4769
  - list, RandStream method 2-2665
  - ListboxTop, Uicontrol property 2-4763
  - listdlg 2-2667
  - listfonts 2-2670
  - load 2-2672 2-2677
    - serial port I/O 2-2679
  - Lobatto IIIa ODE solver 2-558 2-564
  - local variables 2-1929
  - locking functions 2-2924
  - log 2-2688
    - saving session to file 2-1286
  - log10 [log010] 2-2689
  - log1p 2-2690
  - log2 2-2691
  - logarithm
    - base ten 2-2689
    - base two 2-2691
    - complex 2-2688 to 2-2689
    - natural 2-2688
    - of beta function (natural) 2-471
    - of gamma function (natural) 2-1842
    - of real numbers 2-3823
    - plotting 2-2694
  - logarithmic derivative
    - gamma function 2-3585
  - logarithmically spaced vectors, creating 2-2700
  - logical 2-2692
  - logical array
    - converting numeric array to 2-2692
    - detecting 2-2426
  - logical indexing 2-2692
  - logical operations
    - XOR 2-5215
  - logical operators 2-59 2-66
  - logical tests
    - all 2-183
    - any 2-246
  - logical XOR 2-5215
  - loglog 2-2694
  - logm 2-2697
  - logspace 2-2700
  - lookfor 2-2701
  - lossy compression
    - writing JPEG 2000 files with 2-2247
    - writing JPEG files with 2-2247
  - Lotus WK1 files
    - loading 2-5158
    - writing 2-5161
  - lower 2-2703
  - lower triangular matrix 2-4661
  - lowercase to uppercase 2-4990
  - lscov 2-2706
  - lsqnonneg 2-2711
  - lsqr 2-2714
  - lt 2-2719
  - lu 2-2721
  - LU factorization 2-2721
    - storage requirements of (sparse) 2-3171
  - luinc 2-2728
- ## M
- .m files
    - checking existence of 2-1459
  - M-files
    - deleting 2-1258
  - magic 2-2735
  - magic squares 2-2735
  - Margin
    - annotation textbox property 2-240
    - text object 2-4564
  - Marker
    - errorbar property 2-1426
    - Line property 2-2609
    - lineseries property 2-2625
    - Patch property 2-3405
    - quivergroup property 2-3672

- scatter property 2-4034
- stairs series property 2-4248
- stem property 2-4281
- Surface property 2-4451
- surfaceplot property 2-4476
- MarkerEdgeColor
  - errorbar property 2-1427
  - Line property 2-2609
  - lineseries property 2-2626
  - Patch property 2-3406
  - quivergroup property 2-3673
  - scatter property 2-4035
  - stairs series property 2-4249
  - stem property 2-4281
  - Surface property 2-4452
  - surfaceplot property 2-4477
- MarkerFaceColor
  - errorbar property 2-1427
  - Line property 2-2610
  - lineseries property 2-2626
  - Patch property 2-3406
  - quivergroup property 2-3673
  - scatter property 2-4035
  - stairs series property 2-4249
  - stem property 2-4282
  - Surface property 2-4453
  - surfaceplot property 2-4477
- MarkerSize
  - errorbar property 2-1428
  - Line property 2-2610
  - lineseries property 2-2627
  - Patch property 2-3407
  - quivergroup property 2-3674
  - stairs series property 2-4250
  - stem property 2-4282
  - Surface property 2-4453
  - surfaceplot property 2-4478
- mass matrix (ODE) 2-3297
  - initial slope 2-3298 to 2-3299
  - singular 2-3298
  - sparsity pattern 2-3298
  - specifying 2-3298
  - state dependence 2-3298
- MAT-file 2-3997
  - converting sparse matrix after loading from 2-4173
- MAT-files
  - listing for folder 2-5124
- mat2str 2-2751
- material 2-2753
- MATLAB
  - installation folder 2-2766
  - quitting 2-3652
  - startup 2-2764
  - version number, comparing 2-5039
  - version number, displaying 2-5031
- matlab (UNIX command) 2-2768
- matlab (Windows command) 2-2779
- MATLAB files
  - listing names of in a folder 2-5124
- matlab function for UNIX 2-2768
- matlab function for Windows 2-2779
- MATLAB startup file 2-4258
- MATLAB® desktop
  - moving figure windows in front of 2-4120
- matlab.mat 2-3997
- matlabrc 2-2764
- matlabroot 2-2766
- \$matlabroot 2-2766
- matrices
  - preallocation 2-5219
- matrix 2-48
  - addressing selected rows and columns of 2-73
  - arrowhead 2-964
  - columns
    - rearrange 2-1693
  - companion 2-972
  - condition number of 2-982 2-3720
  - condition number, improving 2-371



- Mersenne twister 2-3702 2-3709
- mesh plot
  - tetrahedron 2-4532
- mesh size (BVP) 2-574
- meshc 2-2810 2-2816 2-2821
- meshgrid 2-2826
- MeshStyle, Surface property 2-4453
- MeshStyle, surfaceplot property 2-4478
- meshz 2-2810 2-2816 2-2821
- message
  - error See error message 2-5100
  - warning See warning message 2-5100
- meta.class 2-2829
- meta.DynamicProperty 2-2835
- meta.EnumeratedValue class 2-2840
- meta.event 2-2842
- meta.method 2-2848
- meta.package class 2-2851
- meta.property 2-2859
- methods
  - for timeseries object 2-3742
  - locating 2-5129
- mex 2-2869
- mex build script
  - switches 2-2870
    - arch 2-2870
    - c 2-2870
    - compatibleArrayDims 2-2871
    - cxx 2-2871
    - Dname 2-2871
    - Dname=value 2-2871
    - f optionsfile 2-2871
    - fortran 2-2872
    - g 2-2872
    - h[elp] 2-2872
    - inline 2-2872
    - Ipathname 2-2872
    - largeArrayDims 2-2873
    - Lfolder 2-2872
    - lname 2-2872
    - n 2-2873
    - name=value 2-2874
    - O 2-2873
    - outdir dirname 2-2873
    - output resultname 2-2873
    - @rsp\_file 2-2870
    - setup 2-2873
    - Uname 2-2874
    - v 2-2874
- mex.CompilerConfiguration 2-2877
- mex.CompilerConfigurationDetails 2-2877
- MEX-files
  - debugging on UNIX 2-1176
  - listing for folder 2-5124
- mex.getCompilerConfigurations 2-2877
- MException
  - constructor 2-1401 2-2883
  - methods
    - addCause 2-126
    - disp 2-1302
    - eq 2-1401
    - getReport 2-1908
    - isequal 2-2402
    - last 2-2518
    - ne 2-3000
    - rethrow 2-3931
    - throw 2-4612
    - throwAsCaller 2-4616
- mexext 2-2889
- mfilename 2-2890
- Microsoft Excel files
  - loading 2-5194
- min 2-2892
- Min, Uicontrol property 2-4764
- MinColormap, Figure property 2-1584
- MinorGridLineStyle, Axes property 2-349
- minres 2-2896
- minus (function equivalent for -) 2-53
- mislocked 2-2902
- mkdir 2-2903



mkpp 2-2906  
 mldivide (function equivalent for \) 2-53  
 mlintrpt 2-2922  
     suppressing messages 2-2923  
 mlock 2-2924  
 mmfileinfo 2-2925  
 mod 2-2933  
 modal matrix 2-1353  
 mode 2-2935  
 mode objects  
     pan, using 2-3348  
     rotate3d, using 2-3984  
     zoom, using 2-5224  
 models  
     saving 2-4008  
 modification date  
     of a file 2-1291  
 modified Bessel functions  
     relationship to Airy functions 2-175  
 modulo arithmetic 2-2933  
 MonitorPositions  
     Root property 2-3965  
 Moore-Penrose pseudoinverse 2-3473  
 more 2-2938 2-2963  
 move 2-2940  
 movefile 2-2942  
 movegui function 2-2945  
 movie 2-2948  
 movie2avi 2-2952  
 movies  
     exporting in AVI format 2-314  
 mpower (function equivalent for ^) 2-53  
 mrdivide (function equivalent for /) 2-53  
 msgbox 2-2957  
 mtimes 2-2960  
 mtimes (function equivalent for \*) 2-53  
 mu-law encoded audio signals 2-2590 2-2963  
 multibandread 2-2964  
 multibandwrite 2-2969  
 multidimensional arrays

concatenating 2-618  
 interpolation of 2-2352  
 number of dimensions of 2-2995  
 rearranging dimensions of 2-2377 2-3464  
 removing singleton dimensions of 2-4224  
 reshaping 2-3922  
 size of 2-4134  
 sorting elements of 2-4156  
 multiple  
     least common 2-2531  
 multiplication  
     array (arithmetic operator) 2-49  
     matrix (defined) 2-49  
     of polynomials 2-1036  
 multistep ODE solver 2-3191 2-3206 2-3221  
     2-3236 2-3251 2-3266 2-3281  
 munlock 2-2974

## N

Name, Figure property 2-1585  
 namelengthmax 2-2976  
 NaN 2-2977  
 NaN (Not-a-Number) 2-2977  
     returned by rem 2-3908  
 nargchk 2-2979  
 narginchk 2-2983  
 nargoutchk 2-2988  
 native2unicode 2-2990  
 ndgrid 2-2993  
 ndims 2-2995  
 ne 2-2996  
 ne, MException method 2-3000  
 nearest neighbor interpolation 2-2337 2-2345  
     2-2349 2-2353  
 NET  
     summary of functions 2-3003  
 .NET  
     summary of functions 2-3003  
 netcdf

- summary of capabilities 2-3114
- netcdf.abort
  - revert recent netCDF file definitions 2-3055
- netcdf.close
  - close netCDF file 2-3057
- netcdf.copyAtt
  - copy attribute to new location 2-3058
- netcdf.create
  - create netCDF file 2-3060
- netcdf.defDim
  - create dimension in netCDF file 2-3062
- netcdf.defVar
  - define variable in netCDF dataset 2-3064
- netcdf.delAtt
  - delete netCDF attribute 2-3073
- netcdf.endDef
  - takes a netCDF file out of define mode 2-3075
- netcdf.getAtt
  - return data from netCDF attribute 2-3077
- netcdf.getConstant
  - get numeric value of netCDF constant 2-3080
- netcdf.getConstantNames
  - get list of netCDF constants 2-3081
- netcdf.getVar
  - return data from netCDF variable 2-3082
- netcdf.inq
  - return information about netCDF file 2-3085
- netcdf.inqAtt
  - return information about a netCDF attribute 2-3106
- netcdf.inqAttID
  - return identifier of netCDF attribute 2-3108
- netcdf.inqAttName
  - return name of netCDF attribute 2-3109
- netcdf.inqDim
  - return information about netCDF dimension 2-3111
- netcdf.inqDimID
  - return dimension ID for netCDF file 2-3112
- netcdf.inqLibVers
  - return version of netCDF library 2-3113
- netcdf.inqVarID
  - return netCDF variable identifier 2-3116
- netcdf.open
  - open an existing netCDF file 2-3117
- netcdf.putAtt
  - write a netCDF attribute 2-3119
- netcdf.putVar
  - write data to netCDF variable 2-3121
- netcdf.reDef
  - put netCDF file into define mode 2-3123
- netcdf.renameAtt
  - netCDF function to change the name of an attribute 2-3124
- netcdf.renameDim
  - netCDF function to change the name of a dimension 2-3126
- netcdf.renameVar
  - change the name of a netCDF variable 2-3127
- netcdf.setDefaultFormat
  - change the default netCDF file format 2-3131
- netcdf.setFill
  - set netCDF fill behavior 2-3132
- netcdf.sync
  - synchronize netCDF dataset to disk 2-3133
- newplot 2-3134
- NextPlot
  - Axes property 2-349
  - Figure property 2-1585
- nextpow2 2-3138
- nnz 2-3139
- no derivative method 2-1707
- nodesktop startup option 2-2772
- nonzero entries
  - specifying maximum number of in sparse matrix 2-4170
- nonzero entries (in sparse matrix)
  - allocated storage for 2-3171
  - number of 2-3139

- replacing with ones 2-4201
  - vector of 2-3141
  - nonzeros 2-3141
  - norm 2-3142
    - 1-norm 2-3720
    - 2-norm (estimate of) 2-3143
    - matrix 2-3142
    - pseudoinverse and 2-3473 2-3475
    - vector 2-3142
  - normal vectors, computing for volumes 2-2450
  - NormalMode
    - Patch property 2-3407
    - Surface property 2-4454
    - surfaceplot property 2-4478
  - normest 2-3143
  - not 2-3144
  - not (function equivalent for ~) 2-63
  - notebook 2-3145
  - notify 2-3146
  - now 2-3147
  - nthroot 2-3148
  - null 2-3149
  - null space 2-3149
  - num2hex 2-3159
  - number
    - of array dimensions 2-2995
  - numbers
    - imaginary 2-2174
    - NaN 2-2977
    - plus infinity 2-2267
    - prime 2-3537
    - real 2-3822
    - smallest positive 2-3825
  - NumberTitle, Figure property 2-1586
  - numel 2-3169
  - numerical differentiation formula ODE
    - solvers 2-3191 2-3206 2-3221 2-3236 2-3251 2-3266 2-3281
  - numerical evaluation
    - double integral 2-1174
    - triple integral 2-4664
  - nzmax 2-3171
- O**
- objects
    - Java 2-2420
  - ODE solver properties
    - error tolerance 2-3285
    - event location 2-3292
    - Jacobian matrix 2-3293
    - mass matrix 2-3297
    - ode15s 2-3299
    - solver output 2-3287
    - step size 2-3290
  - ODE solvers
    - backward differentiation formulas 2-3299
    - numerical differentiation formulas 2-3299
    - obtaining solutions at specific times 2-3178 2-3193 2-3208 2-3223 2-3238 2-3253 2-3268
    - variable order solver 2-3299
  - ode15i function 2-3172
  - odeget 2-3283
  - odephas2 output function 2-3289
  - odephas3 output function 2-3289
  - odeplot output function 2-3289
  - odeprint output function 2-3289
  - odeset 2-3284
  - odextend 2-3301
  - off-screen figures, displaying 2-1660
  - OffCallback
    - Uitoggletool property 2-4926
  - OnCallback
    - Uitoggletool property 2-4927
  - one-step ODE solver 2-3190
  - ones 2-3306
  - online help 2-2043
  - openfig 2-3311
  - OpenGL 2-1593

- autoselection criteria 2-1597
  - opening
    - files in Windows applications 2-5153
  - operating system command 2-4522
  - operating system command, issuing 2-71
  - operators
    - arithmetic 2-48
    - logical 2-59 2-66
    - overloading arithmetic 2-54
    - overloading relational 2-57
    - relational 2-57 2-2692
    - symbols 2-2043
  - optimget 2-3321
  - optimization parameters structure 2-3321 to 2-3322
  - optimizing file execution 2-3571
  - optimset 2-3322
  - or 2-3326
  - or (function equivalent for |) 2-63
  - ordeig 2-3328
  - orderfields 2-3331
  - ordering
    - reverse Cuthill-McKee 2-4506 2-4516
  - ordqz 2-3334
  - ordschur 2-3336
  - orient 2-3338
  - orth 2-3340
  - orthographic projection, setting and querying 2-598
  - Out of memory (error message) 2-3343
  - OuterPosition
    - Axes property 2-350
    - Figure property 2-1586
  - output
    - checking number of arguments 2-2988
    - in Command Window 2-2938
  - output points (ODE)
    - increasing number of 2-3287
  - output properties (DDE) 2-1219
  - output properties (ODE) 2-3287
    - increasing number of output points 2-3287
  - overflow 2-2267
  - overloading
    - arithmetic operators 2-54
    - relational operators 2-57
    - special characters 2-72
- P**
- P-files
    - checking existence of 2-1459
  - pack 2-3343
  - padecoeff 2-3345
  - pagesetupdlg 2-3346
  - paging in the Command Window 2-2938
  - pan mode objects 2-3348
  - PaperOrientation, Figure property 2-1587
  - PaperPosition, Figure property 2-1587
  - PaperPositionMode, Figure property 2-1588
  - PaperSize, Figure property 2-1588
  - PaperType, Figure property 2-1588
  - PaperUnits, Figure property 2-1590
  - parametric curve, plotting 2-1493
  - Parent
    - areaseries property 2-266
    - Axes property 2-351
    - barseries property 2-434
    - contour property 2-1023
    - errorbar property 2-1428
    - Figure property 2-1590
    - hggroup property 2-2086
    - hgtransform property 2-2117
    - Image property 2-2196
    - Light property 2-2585
    - Line property 2-2610
    - lineseries property 2-2627
    - Patch property 2-3407
    - quivergroup property 2-3674
    - rectangle property 2-3849
    - Root property 2-3968

- scatter property 2-4036
- stairs series property 2-4250
- stem property 2-4282
- Surface property 2-4454
- surfaceplot property 2-4479
- Text property 2-4566
- Uicontextmenu property 2-4733
- Uicontrol property 2-4765
- Uimenu property 2-4811
- Uipushtool property 2-4857
- Uitable property 2-4911
- Uitoggletool property 2-4927
- Uitoolbar property 2-4939
- parentheses (special characters) 2-69
- parfor 2-3359
- partial fraction expansion 2-3924
- pascal 2-3364
- Pascal matrix 2-3364 2-3528
- patch 2-3365
- Patch
  - converting a surface to 2-4427
  - creating 2-3365
  - properties 2-3385
  - reducing number of faces 2-3854
  - reducing size of face 2-4124
- path 2-3412
- path2rc 2-3415
- pathnames
  - of functions or files 2-5129
- pathsep 2-3416
- pathtool 2-3417
- pause 2-3419
- pauses, removing 2-1169
- pausing function execution 2-3419
- pbaspect 2-3421
- PBM
  - parameters that can be set when writing 2-2248
- PBM files
  - writing 2-2243
- pcg 2-3427
- pchip 2-3432
- pcolor 2-3439
- PCX files
  - writing 2-2243
- PDE. *See* Partial Differential Equations
- pdepe 2-3444
- pdeval 2-3457
- percent sign (special characters) 2-71
- percent-brace (special characters) 2-71
- perfect matching 2-1321
- performance 2-445
- period (.), to distinguish matrix and array operations 2-48
- period (special characters) 2-70
- perl 2-3461
- perl function 2-3461
- Perl scripts in MATLAB 2-3461
- perms 2-3463
- permutation
  - matrix 2-2721
  - of array dimensions 2-3464
  - random 2-3699
- permutations of n elements 2-3463
- permute 2-3464
- persistent 2-3465
- persistent variable 2-3465
- perspective projection, setting and querying 2-598
- PGM
  - parameters that can be set when writing 2-2248
- PGM files
  - writing 2-2243
- phase angle, complex 2-199
- phase, complex
  - correcting angles 2-4982
- pie 2-3469
- pie3 2-3471
- pinv 2-3473

- planerot 2-3476
- plot
  - editing 2-3496
- plot box aspect ratio of axes 2-3421
- plot editing mode
  - overview 2-3497
- Plot Editor
  - interface 2-3497
- plot, volumetric
  - generating grid arrays for 2-2826
  - slice plot 2-4145
- PlotBoxAspectRatio, Axes property 2-352
- PlotBoxAspectRatioMode, Axes property 2-352
- plottedit 2-3496
- plotting
  - 3-D plot 2-3492
  - contours (a 2-1472
  - contours (ez function) 2-1472
  - ez-function mesh plot 2-1480
  - feather plots 2-1519
  - filled contours 2-1476
  - function plots 2-1728
  - functions with discontinuities 2-1501
  - histogram plots 2-2124
  - in polar coordinates 2-1496
  - isosurfaces 2-2453
  - loglog plot 2-2694
  - mathematical function 2-1488
  - mesh contour plot 2-1484
  - mesh plot 2-2810 2-2816 2-2821
  - parametric curve 2-1493
  - plot with two y-axes 2-3503
  - ribbon plot 2-3945
  - rose plot 2-3975
  - scatter plot 2-3499
  - scatter plot, 3-D 2-4020
  - semilogarithmic plot 2-4051 2-4054
  - stem plot, 3-D 2-4266
  - surface plot 2-4419 2-4423
  - surfaces 2-1499
  - velocity vectors 2-986
  - volumetric slice plot 2-4145
  - . See visualizing
- plus (function equivalent for +) 2-53
- PNG
  - writing options for 2-2249
    - alpha 2-2249
    - background color 2-2249
    - chromaticities 2-2250
    - gamma 2-2250
    - interlace type 2-2250
    - resolution 2-2251
    - significant bits 2-2250
    - transparency 2-2251
- PNG files
  - writing 2-2243
- PNM files
  - writing 2-2243
- Pointer, Figure property 2-1590
- PointerLocation, Root property 2-3968
- PointerShapeCData, Figure property 2-1591
- PointerShapeHotSpot, Figure property 2-1591
- PointerWindow, Root property 2-3969
- pol2cart 2-3509
- polar 2-3511
- polar coordinates 2-3509
  - computing the angle 2-199
  - converting from Cartesian 2-611
  - converting to cylindrical or Cartesian 2-3509
  - plotting in 2-1496
- poles of transfer function 2-3924
- poly 2-3513
- polyarea 2-3516
- polyder 2-3518
- polyeig 2-3519
- polyfit 2-3521
- polygamma function 2-3585
- polygon
  - area of 2-3516
  - creating with patch 2-3365

- detecting points inside 2-2276
- polyint 2-3525
- polynomial
  - analytic integration 2-3525
  - characteristic 2-3513 to 2-3514 2-3973
  - coefficients (transfer function) 2-3924
  - curve fitting with 2-3521
  - derivative of 2-3518
  - division 2-1238
  - eigenvalue problem 2-3519
  - evaluation 2-3526
  - evaluation (matrix sense) 2-3528
  - make piecewise 2-2906
  - multiplication 2-1036
- polyval 2-3526
- polyvalm 2-3528
- poorly conditioned
  - matrix 2-2123
- poorly conditioned eigenvalues 2-371
- pop-up menus 2-4737
  - defining choices 2-4769
- Portable Anymap files
  - writing 2-2243
- Portable Bitmap (PBM) files
  - writing 2-2243
- Portable Graymap files
  - writing 2-2243
- Portable Network Graphics files
  - writing 2-2243
- Portable pixmap format
  - writing 2-2243
- Position
  - annotation ellipse property 2-216
  - annotation line property 2-219
  - annotation rectangle property 2-222
  - arrow property 2-208
  - Axes property 2-352
  - doubletarrow property 2-213
  - Figure property 2-1591
  - Light property 2-2585
  - Text property 2-4566
  - textarrow property 2-228
  - textbox property 2-240
  - Uicontextmenu property 2-4733
  - Uicontrol property 2-4765
  - Uimenu property 2-4811
  - Uitable property 2-4911
- position of camera
  - dollyng 2-584
- position of camera, setting and querying 2-596
- Position, rectangle property 2-3849
- PostScript
  - default printer 2-3544
  - levels 1 and 2 2-3544
  - printing interpolated shading 2-3554
- pow2 2-3530
- power 2-3531
  - matrix. *See* matrix exponential
  - of real numbers 2-3826
  - of two, next 2-3138
- power (function equivalent for .^) 2-53
- PPM
  - parameters that can be set when writing 2-2248
- PPM files
  - writing 2-2243
- ppval 2-3532
- preallocation
  - matrix 2-5219
- prefdir 2-3534
- preferences 2-3536
  - opening the dialog box 2-3536
- present working directory 2-3607
- prime factors 2-1514
  - dependence of Fourier transform on 2-1535 2-1537 2-1539
- prime numbers 2-3537
- primes 2-3537
- printdlg 2-3559
- printdlg function 2-3559

- printer
  - default for linux and unix 2-3544
- printer drivers
  - GhostScript drivers 2-3540
  - interploated shading 2-3554
  - MATLAB printer drivers 2-3540
- printing
  - GUIs 2-3553
  - interpolated shading 2-3554
  - on MS-Windows 2-3553
  - with a variable file name 2-3555
  - with nodisplay 2-3547
  - with noFigureWindows 2-3547
- printing figures
  - preview 2-3560
- printing tips 2-3552
- printing, suppressing 2-70
- printpreview 2-3560
- prod 2-3569
- product
  - cumulative 2-1097
  - Kronecker tensor 2-2516
  - of array elements 2-3569
  - of vectors (cross) 2-1086
  - scalar (dot) 2-1086
- profile 2-3571
- profsave 2-3578
- program execution
  - resuming after suspending 2-4869
  - suspending from GUI 2-4941
- projection type, setting and querying 2-598
- ProjectionType, Axes property 2-353
- prompting users for input 2-2278
- prompting users to choose an item 2-2808
- propedit 2-3579 to 2-3580
- proppanel 2-3583
- pseudoinverse 2-3473
- psi 2-3585
- push buttons 2-4737
- pwd 2-3607

**Q**

- qmr 2-3608
- QR decomposition
  - deleting column from 2-3619
- qrdelete 2-3619
- qrinsert 2-3621
- qrupdate 2-3623
- quad 2-3626
- quadgk 2-3636
- quadl 2-3642
- quadrature 2-3626 2-3636
- quadv 2-3645
- quantization
  - performed by rgb2ind 2-3939
- questdlg 2-3648
- questdlg function 2-3648
- quit 2-3652
- quitting MATLAB 2-3652
- quiver 2-3655
- quiver3 2-3658
- qz 2-3683
- QZ factorization 2-3520 2-3683

**R**

- radio buttons 2-4737
- rand, RandStream method 2-3688
- randi, RandStream method 2-3693
- randn, RandStream method 2-3698
- random
  - permutation 2-3699
  - sparse matrix 2-4205 to 2-4206
  - symmetric sparse matrix 2-4207
- random number generators 2-2665 2-3688 2-3693 2-3698 2-3702 2-3709
- randperm 2-3699
- randStream
  - constructor 2-3709
- RandStream 2-3702 2-3709
  - constructor 2-3702



- methods
  - create 2-1077
  - get 2-1873
  - getDefaultStream 2-1886 2-3712
  - list 2-2665
  - rand 2-3688
  - randi 2-3693
  - randn 2-3698
  - setDefaultStream 2-4084
- range space 2-3340
- rank 2-3714
- rank of a matrix 2-3714
- RAS files
  - parameters that can be set when writing 2-2252
  - writing 2-2244
- RAS image format
  - specifying color order 2-2252
  - writing alpha data 2-2252
- Raster image files
  - writing 2-2244
- rational fraction approximation 2-3715
- rbbox 2-3718 2-3860
- rcond 2-3720
- rdivide (function equivalent for ./) 2-53
- readasync 2-3728
- reading
  - data from files 2-4575
  - formatted data from file 2-1761
- readme files, displaying 2-2390
- real 2-3822
- real numbers 2-3822
- realloc 2-3823
- realmax 2-3824
- realmin 2-3825
- realpow 2-3826
- realsqrt 2-3827
- rearrange array
  - flip along dimension 2-1692
  - reverse along dimension 2-1692
- rearrange matrix
  - flip left-right 2-1693
  - flip up-down 2-1694
  - reverse column order 2-1693
  - reverse row order 2-1694
- RearrangeableColumns
  - Uitable property 2-4912
- rearranging arrays
  - converting to vector 2-74
  - removing first n singleton dimensions 2-4121
  - removing singleton dimensions 2-4224
  - reshaping 2-3922
  - shifting dimensions 2-4121
  - swapping dimensions 2-2377 2-3464
- rearranging matrices
  - converting to vector 2-74
  - rotating 90\xfb 2-3979
  - transposing 2-70
- record 2-3828
- rectangle
  - properties 2-3837
  - rectangle function 2-3832
- rectint 2-3851
- RecursionLimit
  - Root property 2-3969
- recycle 2-3852
- reduced row echelon form 2-3991
- reducepatch 2-3854
- reducevolume 2-3858
- refresh 2-3860
- regexprep 2-3893
- regexprtranslate 2-3897
- regression
  - linear 2-3521
- regularly spaced vectors, creating 2-73 2-2664
- rehash 2-3902
- relational operators 2-57 2-2692
- relational operators for handle objects 2-3906
- relative accuracy
  - BVP 2-570

- DDE 2-1219
  - norm of DDE solution 2-1219
  - norm of ODE solution 2-3286
  - ODE 2-3286
- rem 2-3908
- removets 2-3911
- renaming
  - using copyfile 2-1057
- renderer
  - OpenGL 2-1593
  - painters 2-1593
  - zbuffer 2-1593
- Renderer, Figure property 2-1593
- RendererMode, Figure property 2-1597
- repeatedly executing statements 2-1717
- repeatedly executing statements in
  - parallel 2-3359
- replicating a matrix 2-3914
- repmat 2-3914
- resample (tscollection) 2-3916
- reset 2-3919
- reshape 2-3922
- residue 2-3924
- residues of transfer function 2-3924
- Resize, Figure property 2-1598
- ResizeFcn, Figure property 2-1598
- restoredefaultpath 2-3928
- rethrow 2-3929
- rethrow, MException method 2-3931
- return 2-3935
- reverse
  - array along dimension 2-1692
  - array dimension 2-1692
  - matrix column order 2-1693
  - matrix row order 2-1694
- reverse Cuthill-McKee ordering 2-4506 2-4516
- RGB images
  - converting to indexed 2-3938
- RGB, converting to HSV 2-3937
- rgb2hsv 2-3937
- rgb2ind 2-3938
- rgbplot 2-3943
- ribbon 2-3945
- right-click and context menus 2-4723
- rmapdata function 2-3947
- rmfield 2-3953
- rmpath 2-3954
- rmpref function 2-3955
- rolling camera 2-600
- root folder 2-2766
- Root graphics object 2-3959
- root object 2-3959
- root, see rootobject 2-3959
- roots 2-3973
- roots of a polynomial 2-3513 to 2-3514 2-3973
- rose 2-3975
- Rosenbrock
  - banana function 2-1705
  - ODE solver 2-3191 2-3206 2-3221 2-3236 2-3251 2-3266 2-3281
- rosser 2-3978
- rot90 2-3979
- rotate 2-3980
- rotate3d 2-3984
- rotate3d mode objects 2-3984
- rotating camera 2-592
- rotating camera target 2-594
- Rotation, Text property 2-4566
- rotations
  - Jacobi 2-4207
- round 2-3990
  - to nearest integer 2-3990
  - towards infinity 2-825
  - towards minus infinity 2-1696
  - towards zero 2-1691
- roundoff error
  - characteristic polynomial and 2-3514
  - effect on eigenvalues 2-371
  - evaluating matrix functions 2-1794
  - in inverse Hilbert matrix 2-2373

- partial fraction expansion and 2-3925
- polynomial roots and 2-3973
- sparse matrix conversion and 2-4174
- RowName
  - Uitable property 2-4912
- RowStripping
  - Uitable property 2-4913
- rref 2-3991
- rrefmovie 2-3991
- rsf2csf 2-3993
- rubberband box 2-3718
- Runge-Kutta ODE solvers 2-3190 2-3205 2-3220
  - 2-3235 2-3250 2-3265 2-3280
- running average 2-1650

## S

- save 2-3997 2-4004
  - serial port I/O 2-4006
- saveas 2-4008
- savepath 2-4014
- saving
  - ASCII data 2-3997
  - session to a file 2-1286
  - workspace variables 2-3997
- scalar product (of vectors) 2-1086
- scatter 2-4016
- scatter3 2-4020
- scattered data, aligning
  - multi-dimensional 2-2993
- scattergroup
  - properties 2-4023
- Schmidt semi-normalized Legendre
  - functions 2-2555
- schur 2-4042
- Schur decomposition 2-4042
- Schur form of matrix 2-3993 2-4042
- ScreenDepth, Root property 2-3969
- ScreenPixelsPerInch, Root property 2-3970
- ScreenSize, Root property 2-3970
- script 2-4044
- search path
  - adding folders to 2-160
  - MATLAB 2-3412
  - modifying 2-3417
  - removing folders from 2-3954
  - toolbox folder 2-4652
  - user folder 2-5002
  - viewing 2-3417
- search, string 2-1670
- sec 2-4045
- secant 2-4045
  - hyperbolic 2-4049
  - inverse 2-278
  - inverse hyperbolic 2-281
- sech 2-4049
- Selected
  - areaseries property 2-266
  - Axes property 2-353
  - barseries property 2-434
  - contour property 2-1023
  - errorbar property 2-1428
  - Figure property 2-1600
  - hggroup property 2-2087
  - hgtransform property 2-2117
  - Image property 2-2196
  - Light property 2-2586
  - Line property 2-2611
  - lineseries property 2-2627
  - Patch property 2-3408
  - quivergroup property 2-3674
  - rectangle property 2-3849
  - Root property 2-3971
  - scatter property 2-4036
  - stairs property 2-4250
  - stem property 2-4283
  - Surface property 2-4454
  - surfaceplot property 2-4479
  - Text property 2-4566
  - Uicontrol property 2-4766

- Uitable property 2-4913
- selecting areas 2-3718
- SelectionHighlight
  - areaserie property 2-267
  - Axes property 2-354
  - barseries property 2-434
  - contour property 2-1024
  - errorbar property 2-1429
  - Figure property 2-1600
  - hggroup property 2-2087
  - hgtransform property 2-2117
  - Image property 2-2197
  - Light property 2-2586
  - Line property 2-2611
  - lineseries property 2-2628
  - Patch property 2-3408
  - quivergroup property 2-3675
  - rectangle property 2-3849
  - scatter property 2-4036
  - stairs series property 2-4250
  - stem property 2-4283
  - Surface property 2-4454
  - surfaceplot property 2-4479
  - Text property 2-4566
  - Uicontrol property 2-4766
  - Uitable property 2-4913
- SelectionType, Figure property 2-1600
- selectmoveresize 2-4050
- semicolon (special characters) 2-70
- sendmail 2-4057
- Separator
  - Uipushtool property 2-4857
  - Uitoggletool property 2-4927
- Separator, Uimenu property 2-4812
- serial 2-4060
- serialbreak 2-4062
- server variable 2-1527
- session
  - saving 2-1286
- set 2-4063
  - serial port I/O 2-4076
  - timer object 2-4078
- set (tscollection) 2-4081
- set hgsetget class method 2-4072
- setabstime (tscollection) 2-4082
- setappdata 2-4083
- setDefaultStream, RandStream method 2-4084
- setdiff 2-4085
- setdisp hgsetget class method 2-4093
- setenv 2-4094
- setpixelposition 2-4097
- setpref function 2-4100
- setstr 2-4102
- settimeseriesnames 2-4106
- setxor 2-4107
- shading 2-4116
- shading colors in surface plots 2-4116
- shell script 2-4522 2-4966
- shiftdim 2-4121
- shifting array
  - circular 2-884
- ShowArrowHead
  - quivergroup property 2-3675
- ShowBaseLine
  - barseries property 2-434
- ShowHiddenHandles, Root property 2-3971
- showplottool 2-4122
- ShowText
  - contour property 2-1024
- shrinkfaces 2-4124
- shutdown 2-3652
- sign 2-4128
- signum function 2-4128
- simplex search 2-1707
- Simpson's rule, adaptive recursive 2-3628
- Simulink
  - version number, comparing 2-5039
  - version number, displaying 2-5031
- sine
  - hyperbolic 2-4133

- inverse hyperbolic 2-286
- single 2-4132
- single quote (special characters) 2-70
- singular value
  - decomposition 2-3714 2-4495
  - rank and 2-3714
- sinh 2-4133
- size
  - array dimensions 2-4134
  - serial port I/O 2-4142
- size (tscollection) 2-4144
- size of array dimensions 2-4134
- size of fonts, see also `FontSize` property 2-4569
- size vector 2-3922
- `SizeData`
  - scatter property 2-4036
- `SizeDataSource`
  - scatter property 2-4037
- slice 2-4145
- slice planes, contouring 2-1032
- sliders 2-4737
- `SliderStep`, `Uicontrol` property 2-4766
- smallest array elements 2-2892
- `smooth3` 2-4151
- smoothing 3-D data 2-4151
- soccer ball (example) 2-4516
- solution statistics (BVP) 2-575
- sort 2-4156
- sorting
  - array elements 2-4156
  - complex conjugate pairs 2-1075
  - matrix rows 2-4160
- `sortrows` 2-4160
- sound
  - files
    - reading 2-311 2-5109
    - writing 2-313 2-5114
  - playing 2-301
  - recording 2-305
  - resampling 2-301
  - sampling 2-305
- source control on UNIX platforms
  - checking out files
    - function 2-865
- source control systems
  - checking in files 2-863
  - undo checkout 2-4944
- `spalloc` 2-4169
- `sparse` 2-4170
- sparse matrix
  - allocating space for 2-4169
  - applying function only to nonzero elements
    - of 2-4187
  - density of 2-3139
  - detecting 2-2473
  - diagonal 2-4175
  - finding indices of nonzero elements of 2-1654
  - identity 2-4186
  - number of nonzero elements in 2-3139
  - permuting columns of 2-964
  - random 2-4205 to 2-4206
  - random symmetric 2-4207
  - replacing nonzero elements of with
    - ones 2-4201
  - results of mixed operations on 2-4171
  - specifying maximum number of nonzero elements 2-4170
  - vector of nonzero elements 2-3141
  - visualizing sparsity pattern of 2-4218
- sparse storage
  - criterion for using 2-1775
- `spaugment` 2-4172
- `spconvert` 2-4173
- `spdiags` 2-4175
- special characters
  - descriptions 2-2043
  - overloading 2-72
- `specular` 2-4185
- `SpecularColorReflectance`
  - Patch property 2-3408

- Surface property 2-4455
- surfaceplot property 2-4480
- SpecularExponent
  - Patch property 2-3408
  - Surface property 2-4455
  - surfaceplot property 2-4480
- SpecularStrength
  - Patch property 2-3408
  - Surface property 2-4455
  - surfaceplot property 2-4480
- speye 2-4186
- spfun 2-4187
- sph2cart 2-4189
- sphere 2-4190
- spherical coordinates
  - defining a Light position in 2-2588
- spherical coordinates 2-4189
- spinmap 2-4193
- spline 2-4194
- spline interpolation (cubic)
  - one-dimensional 2-2337 2-2345 2-2349 2-2353
- Spline Toolbox 2-2341
- spones 2-4201
- spparms 2-4202
- sprand 2-4205
- sprandn 2-4206
- sprandsym 2-4207
- sprank 2-4209
- spreadsheets
  - loading WK1 files 2-5158
  - loading XLS files 2-5194
  - reading into a matrix 2-1313
  - writing from matrix 2-5161
  - writing matrices into 2-1317
- sqrt 2-4220
- sqrtm 2-4221
- square root
  - of a matrix 2-4221
  - of array elements 2-4220
  - of real numbers 2-3827
- squeeze 2-4224
- stack, displaying 2-1179
- standard deviation 2-4259
- start
  - timer object 2-4255
- startat
  - timer object 2-4256
- startup 2-4258
  - folder and path 2-5002
- startup file 2-4258
- startup files 2-2764
- State
  - Uitoggletool property 2-4928
- static text 2-4738
- std 2-4259
- stem 2-4261
- stem3 2-4266
- step size (DDE)
  - initial step size 2-1223
  - upper bound 2-1223
- step size (ODE) 2-1222 2-3290
  - initial step size 2-3291
  - upper bound 2-3291
- stop
  - timer object 2-4289
- stopasync 2-4290
- stopwatch timer 2-4619
- storage
  - allocated for nonzero entries (sparse) 2-3171
  - sparse 2-4170
- storage allocation 2-5219
- str2cell 2-848
- str2double 2-4291
- str2func 2-4292
- str2mat 2-4296
- str2num 2-4297
- strcat 2-4301
- stream lines
  - computing 2-D 2-4311

- computing 3-D 2-4313
  - drawing 2-4315
- stream2 2-4311
- stream3 2-4313
- stretch-to-fill 2-322
- strfind 2-4343
- string
  - converting from vector to 2-856
  - converting matrix into 2-2751 2-3160
  - converting to lowercase 2-2703
  - converting to numeric array 2-4297
  - converting to uppercase 2-4990
  - dictionary sort of 2-4160
  - finding first token in 2-4370
  - searching and replacing 2-4367
  - searching for 2-1670
- String
  - Text property 2-4567
  - textarrow property 2-228
  - textbox property 2-241
  - Uicontrol property 2-4768
- string matrix to cell array conversion 2-848
- strings 2-4345
- strjust 2-4347
- strmatch 2-4348
- stread 2-4358
- strep 2-4367
- strtok 2-4370
- strtrim 2-4374
- struct2cell 2-4381
- structure array
  - getting contents of field of 2-1889
  - remove field from 2-3953
  - setting contents of a field of 2-4095
- structure arrays
  - field names of 2-1557
- structures
  - dynamic fields 2-70
- strvcat 2-4386
- Style
  - Light property 2-2586
  - Uicontrol property 2-4770
- sub2ind 2-4388
- subplot 2-4391
- subplots
  - assymetrical 2-4396
  - suppressing ticks in 2-4398
- subscripts
  - in axis title 2-4648
  - in text strings 2-4571
- subspace 2-4405
- subsref (function equivalent for  $A(i, j, k, \dots)$ ) 2-71
- subtraction (arithmetic operator) 2-48
- subvolume 2-4410
- sum 2-4413
  - cumulative 2-1099
  - of array elements 2-4413
- superscripts
  - in axis title 2-4648
  - in text strings 2-4571
- support 2-4418
- surf2patch 2-4427
- surface 2-4429
- Surface
  - and contour plotter 2-1507
  - converting to a patch 2-4427
  - creating 2-4429
  - defining default properties 2-3835 2-4433
  - plotting mathematical functions 2-1499
  - properties 2-4434 2-4458
- surface normals, computing for volumes 2-2450
- surf1 2-4486
- svd 2-4495
- svds 2-4497
- swapbytes 2-4500
- symamd 2-4505
- symsfact 2-4509
- symbols
  - operators 2-2043

- symbols in text 2-229 2-241 2-4567
- symmlq 2-4511
- symrcm 2-4516
- syntax, command 2-4519
- syntax, function 2-4519
- system
  - UNC pathname error 2-4522
- system 2-4522
- system folder
  - temporary 2-4530

## T

table lookup. *See* interpolation

### Tag

- areaserie property 2-267
- Axes property 2-354
- barseries property 2-435
- contour property 2-1024
- errorbar property 2-1429
- Figure property 2-1601
- hggroup property 2-2087
- hgtransform property 2-2117
- Image property 2-2197
- Light property 2-2586
- Line property 2-2611
- lineseries property 2-2628
- Patch property 2-3409
- quivergroup property 2-3675
- rectangle property 2-3849
- Root property 2-3971
- scatter property 2-4037
- stairseries property 2-4251
- stem property 2-4283
- Surface property 2-4455
- surfaceplot property 2-4480
- Text property 2-4572
- Uicontextmenu property 2-4733
- Uicontrol property 2-4770
- Uimenu property 2-4812

- Uipushtool property 2-4857
- Uitable property 2-4913
- Uitoggletool property 2-4928
- Uitoolbar property 2-4939
- Tagged Image File Format (TIFF)
  - writing 2-2244
- tan 2-4524
- tangent 2-4524
  - four-quadrant, inverse 2-292
  - hyperbolic 2-4527
  - inverse 2-291
  - inverse hyperbolic 2-298
- tanh 2-4527
- tar 2-4528
- target, of camera 2-601
- tempdir 2-4530
- tempname 2-4531
- temporary
  - files 2-4531
  - system folder 2-4530
- tensor, Kronecker product 2-2516
- terminating MATLAB 2-3652
- test matrices 2-1810
- test, logical. *See* logical tests *and* detecting
- tetrahedron
  - mesh plot 2-4532
- tetramesh 2-4532
- TeX commands in text 2-229 2-241 2-4567
- text 2-4537
  - editing 2-3496
  - subscripts 2-4571
  - superscripts 2-4571
- Text
  - creating 2-4537
  - defining default properties 2-4541
  - fixed-width font 2-4554
  - properties 2-4542
- TextBackgroundColor
  - textarrow property 2-231
- TextColor



- textarrow property 2-231
- TextEdgeColor
  - textarrow property 2-232
- TextLineWidth
  - textarrow property 2-232
- TextList
  - contour property 2-1025
- TextListMode
  - contour property 2-1025
- TextMargin
  - textarrow property 2-232
- textread 2-4575
- TextRotation, textarrow property 2-232
- TextStep
  - contour property 2-1026
- TextStepMode
  - contour property 2-1026
- textwrap 2-4603
- tfqmr 2-4606
- throw, MException method 2-4612
- throwAsCaller, MException method 2-4616
- TickDir, Axes property 2-354
- TickDirMode, Axes property 2-355
- TickLength, Axes property 2-355
- TIFF
  - compression 2-2253
  - encoding 2-2248
  - ImageDescription field 2-2253
  - maxvalue 2-2248
  - parameters that can be set when
    - writing 2-2252
  - resolution 2-2253
  - writemode 2-2253
  - writing 2-2244
- TIFF image format
  - specifying color space 2-2252
- tiling (copies of a matrix) 2-3914
- time
  - CPU 2-1076
  - elapsed (stopwatch timer) 2-4619
- time and date functions 2-1392
- timer
  - properties 2-4635
  - timer object 2-4635
- timerfind
  - timer object 2-4642
- timerfindall
  - timer object 2-4644
- times (function equivalent for .\*) 2-53
- timeseries
  - getdatasamples method 2-3766
- timeseries object
  - methods 2-3742
- timestamp 2-1291
- title 2-4647
  - with superscript 2-4648
- Title, Axes property 2-355
- todatenum 2-4650
- toeplitz 2-4651
- Toeplitz matrix 2-4651
- toggle buttons 2-4738
- token 2-4370
  - See also* string
- Toolbar
  - Figure property 2-1602
- Toolbox
  - Spline 2-2341
- toolbox folder, path 2-4652
- toolboxdir 2-4652
- TooltipString
  - Uicontrol property 2-4770
  - Uipushtool property 2-4858
  - Uitable property 2-4914
  - Uitoggletool property 2-4928
- trace 2-4654
- trace of a matrix 2-1282 2-4654
- trailing blanks
  - removing 2-1230
- transform
  - hgtransform function 2-2095

- transform, Fourier
    - discrete, n-dimensional 2-1539
    - discrete, one-dimensional 2-1532
    - discrete, two-dimensional 2-1537
    - inverse, n-dimensional 2-2164
    - inverse, one-dimensional 2-2160
    - inverse, two-dimensional 2-2162
    - shifting the zero-frequency component
      - of 2-1542
  - transformation
    - See also* conversion 2-635
  - transformations
    - elementary Hermite 2-1846
  - transpose
    - array (arithmetic operator) 2-50
    - matrix (arithmetic operator) 2-50
  - transpose (function equivalent for `.\q`) 2-54
  - trapz 2-4656
  - treelayout 2-4658
  - treepplot 2-4659
  - triangulation
    - 2-D plot 2-4666
  - tril 2-4661
  - trimesh 2-4662
  - triple integral
    - numerical evaluation 2-4664
  - triplequad 2-4664
  - tripplot 2-4666
  - trisurf 2-4680
  - triu 2-4682
  - true 2-4683
  - truth tables (for logical operations) 2-59
  - tscollection 2-4687
  - tsdata.event 2-4690
  - tsearchn 2-4692
  - tstool 2-4693
  - type 2-4694
  - Type
    - areaseries property 2-267
    - Axes property 2-356
    - barseries property 2-435
    - contour property 2-1026
    - errorbar property 2-1429
    - Figure property 2-1602
    - hggroup property 2-2088
    - hgtransform property 2-2118
    - Image property 2-2197
    - Light property 2-2587
    - Line property 2-2611
    - lineseries property 2-2628
    - Patch property 2-3409
    - quivergroup property 2-3676
    - rectangle property 2-3850
    - Root property 2-3971
    - scatter property 2-4038
    - stairsproperty 2-4251
    - stem property 2-4284
    - Surface property 2-4455
    - surfaceplot property 2-4481
    - Text property 2-4572
    - Uicontextmenu property 2-4734
    - Uicontrol property 2-4771
    - Uimenu property 2-4812
    - Uipushtool property 2-4858
    - Uitable property 2-4914
    - Uitoggletool property 2-4929
    - Uitoolbar property 2-4939
  - typecast 2-4695
- ## U
- UData
    - errorbar property 2-1430
    - quivergroup property 2-3677
  - UDataSource
    - errorbar property 2-1430
    - quivergroup property 2-3677
  - Uibuttongroup
    - defining default properties 2-4704
  - uibuttongroup function 2-4699

- Uibuttongroup Properties 2-4704
- uicontextmenu 2-4723
- UiContextMenu
  - Uicontrol property 2-4771
  - Uipushtool property 2-4858
  - Uitoggletool property 2-4929
  - Uitoolbar property 2-4940
- UIContextMenu
  - areaserie property 2-268
  - Axes property 2-356
  - barseries property 2-435
  - contour property 2-1026
  - errorbar property 2-1430
  - Figure property 2-1603
  - hggroup property 2-2088
  - hgtransform property 2-2118
  - Image property 2-2198
  - Light property 2-2587
  - Line property 2-2612
  - lineseries property 2-2628
  - Patch property 2-3410
  - quivergroup property 2-3676
  - rectangle property 2-3850
  - scatter property 2-4038
  - stairs series property 2-4252
  - stem property 2-4284
  - Surface property 2-4456
  - surfaceplot property 2-4481
  - Text property 2-4572
  - Uitable property 2-4914
- Uicontextmenu Properties 2-4726
- uicontrol 2-4735
- Uicontrol
  - defining default properties 2-4746
  - fixed-width font 2-4756
  - types of 2-4735
- Uicontrol Properties 2-4746
- uicontrols
  - printing 2-3553
- uigetdir 2-4774
- uigetfile 2-4778
- uigetpref function 2-4789
- uiimport 2-4796
- uimenu 2-4797
- Uimenu
  - creating 2-4797
  - defining default properties 2-4801
  - Properties 2-4801
- Uimenu Properties 2-4801
- Uipanel
  - defining default properties 2-4828
- uipanel function 2-4825
- Uipanel Properties 2-4828
- uipushtool 2-4846
- Uipushtool
  - defining default properties 2-4849
- Uipushtool Properties 2-4849
- uiputfile 2-4860
- uiresume 2-4869
- uisetcolor function 2-4873
- uisetfont 2-4874
- uisetpref function 2-4876
- uistack 2-4877
- Uitable
  - defining default properties 2-4886
  - fixed-width font 2-4905
- uitable function 2-4878
- Uitable Properties 2-4886
- uitoggletool 2-4916
- Uitoggletool
  - defining default properties 2-4918
- Uitoggletool Properties 2-4918
- uitoolbar 2-4930
- Uitoolbar
  - defining default properties 2-4932
- Uitoolbar Properties 2-4932
- uiwait 2-4941
- uminus (function equivalent for unary \xd0 ) 2-53
- UNC pathname error and system 2-4522

- unconstrained minimization 2-1703
  - undefined numerical results 2-2977
  - undocheckout 2-4944
  - unicode2native 2-4945
  - unimodular matrix 2-1846
  - union 2-4946
  - unique 2-4955
  - Units
    - annotation ellipse property 2-216
    - annotation line property 2-219
    - annotation rectangle property 2-223
    - arrow property 2-208
    - Axes property 2-356
    - doublearrow property 2-213
    - Figure property 2-1603
    - Root property 2-3972
    - Text property 2-4572
    - textarrow property 2-232
    - textbox property 2-244
    - Uicontrol property 2-4771
    - Uitable property 2-4914
  - unix 2-4966
  - unlocking functions 2-2974
  - unmkpp 2-4973
  - untar 2-4980
  - unwrap 2-4982
  - unzip 2-4987
  - up vector, of camera 2-603
  - updating figure during file execution 2-1339
  - uplus (function equivalent for unary +) 2-53
  - upper 2-4990
  - upper triangular matrix 2-4682
  - uppercase to lowercase 2-2703
  - usejava 2-5000
  - user input
    - from a button menu 2-2808
  - UserData
    - areaseries property 2-268
    - Axes property 2-357
    - barseries property 2-436
    - contour property 2-1027
    - errorbar property 2-1431
    - Figure property 2-1604
    - hggroup property 2-2088
    - hgtransform property 2-2119
    - Image property 2-2198
    - Light property 2-2587
    - Line property 2-2612
    - lineseries property 2-2629
    - Patch property 2-3410
    - quivergroup property 2-3676
    - rectangle property 2-3850
    - Root property 2-3972
    - scatter property 2-4038
    - stairs series property 2-4252
    - stem property 2-4284
    - Surface property 2-4456
    - surfaceplot property 2-4481
    - Text property 2-4573
    - Uicontextmenu property 2-4734
    - Uicontrol property 2-4772
    - Uimenu property 2-4812
    - Uipushtool property 2-4859
    - Uitable property 2-4915
    - Uitoggletool property 2-4929
    - Uitoolbar property 2-4940
  - userpath 2-5002
- V**
- Value, Uicontrol property 2-4772
  - vander 2-5023
  - Vandermonde matrix 2-3523
  - var 2-5024
  - varargin 2-5026
  - varargout 2-5028
  - variable numbers of arguments 2-5028
  - variable-order solver (ODE) 2-3299
  - variables
    - checking existence of 2-1459

- global 2-1929
  - in workspace 2-5163
  - keeping some when clearing 2-906
  - linking to graphs with linkdata 2-2647
  - listing 2-5137
  - local 2-1929
  - name of passed 2-2283
  - persistent 2-3465
  - saving 2-3997
  - sizes of 2-5137
- VData**
- quivergroup property 2-3678
- VDataSource**
- quivergroup property 2-3678
- vector**
- dot product 2-1335
  - frequency 2-2700
  - product (cross) 2-1086
- vector field, plotting** 2-986
- vectorize** 2-5030
- vectorizing ODE function (BVP)** 2-572
- vectors, creating**
- logarithmically spaced 2-2700
  - regularly spaced 2-73 2-2664
- velocity vectors, plotting** 2-986
- verctrl** function (Windows) 2-5035
- verLessThan** 2-5039
- version** 2-5041
- version numbers**
- comparing 2-5039
  - displaying 2-5031
- vertcat** 2-5043
- vertcat** (function equivalent for [ 2-71
- vertcat** (tscollection) 2-5045
- VertexNormals**
- Patch property 2-3410
  - Surface property 2-4456
  - surfaceplot property 2-4481
- VerticalAlignment, Text** property 2-4573
- VerticalAlignment, textbox** property 2-233 2-244
- Vertices, Patch** property 2-3410
- video**
- saving in AVI format 2-314
- view** 2-5061
- azimuth of viewpoint 2-5061
  - coordinate system defining 2-5061
  - elevation of viewpoint 2-5061
- view angle, of camera** 2-605
- View, Axes** property (obsolete) 2-357
- viewing**
- a group of object 2-590
  - a specific object in a scene 2-590
- viewmtx** 2-5064
- Visible**
- areaserie property 2-268
  - Axes property 2-358
  - barserie property 2-436
  - contour property 2-1027
  - errorbar property 2-1431
  - Figure property 2-1604
  - hggroup property 2-2088
  - hgtransform property 2-2119
  - Image property 2-2198
  - Light property 2-2587
  - Line property 2-2612
  - lineserie property 2-2629
  - Patch property 2-3410
  - quivergroup property 2-3676
  - rectangle property 2-3850
  - Root property 2-3972
  - scatter property 2-4039
  - stairserie property 2-4252
  - stem property 2-4285
  - Surface property 2-4456
  - surfaceplot property 2-4482
  - Text property 2-4574
  - Uicontextmenu property 2-4734
  - Uicontrol property 2-4773

- Uimenu property 2-4812
  - Uipushtool property 2-4859
  - Uitable property 2-4915
  - Uitoggletool property 2-4929
  - Uitoolbar property 2-4940
  - visualizing
    - cell array structure 2-846
    - sparse matrices 2-4218
  - volumes
    - calculating isosurface data 2-2453
    - computing 2-D stream lines 2-4311
    - computing 3-D stream lines 2-4313
    - computing isosurface normals 2-2450
    - contouring slice planes 2-1032
    - drawing stream lines 2-4315
    - end caps 2-2443
    - reducing face size in isosurfaces 2-4124
    - reducing number of elements in 2-3858
  - voronoi 2-5075
  - Voronoi diagrams
    - multidimensional vizualization 2-5082
    - two-dimensional vizualization 2-5075
  - voronoin 2-5082
- W**
- wait
    - timer object 2-5086
  - waitbar 2-5087
  - waitfor 2-5091
  - waitforbuttonpress 2-5095
  - warndlg 2-5097
  - warning 2-5100
  - warning message (enabling, suppressing, and displaying) 2-5100
  - waterfall 2-5103
  - .wav files
    - reading 2-5109
    - writing 2-5114
  - wavinfo 2-5106
  - wavread 2-5106 2-5109
  - wavwrite 2-5114
  - WData
    - quivergroup property 2-3678
  - WDataSource
    - quivergroup property 2-3679
  - weekday 2-5122
  - well conditioned 2-3720
  - what 2-5124
  - whatsnew 2-5128
  - which 2-5129
  - white space characters, ASCII 2-2472 2-4370
  - whitebg 2-5135
  - wilkinson 2-5152
  - Wilkinson matrix 2-4179 2-5152
  - WindowButtonDownFcn, Figure property 2-1604
  - WindowButtonMotionFcn, Figure
    - property 2-1605
  - WindowButtonUpFcn, Figure property 2-1606
  - WindowKeyPressFcn, Figure property 2-1606
  - WindowKeyReleaseFcn, Figure property 2-1608
  - Windows Paintbrush files
    - writing 2-2243
  - WindowScrollWheelFcn, Figure property 2-1609
  - WindowState, Figure property 2-1612
  - winopen 2-5153
  - winqueryreg 2-5155
  - WK1 files
    - loading 2-5158
    - writing from matrix 2-5161
  - wk1finfo 2-5157
  - wk1read 2-5158
  - wk1write 2-5161
  - workspace 2-5163
    - changing context while debugging 2-1173 2-1196
    - consolidating memory 2-3343
    - predefining variables 2-4258
    - saving 2-3997
    - variables in 2-5137

- viewing contents of 2-5163
- workspace variables
  - reading from disk 2-2672
- WVisual, Figure property 2-1614
- WVisualMode, Figure property 2-1616

## X

### X

- annotation arrow property 2-209 2-214
- annotation line property 2-220
- textarrow property 2-233
- X Windows Dump files
  - writing 2-2244
- x-axis limits, setting and querying 2-5180 2-5184 2-5188

XAxisLocation, Axes property 2-358

XColor, Axes property 2-358

### XData

- areaserie property 2-268
- barseries property 2-436
- contour property 2-1027
- errorbar property 2-1431
- Image property 2-2198
- Line property 2-2612
- lineseries property 2-2629
- Patch property 2-3411
- quivergroup property 2-3679
- scatter property 2-4039
- stairsere property 2-4252
- stem property 2-4285
- Surface property 2-4456
- surfaceplot property 2-4482

### XDataMode

- areaserie property 2-269
- barseries property 2-436
- contour property 2-1027
- errorbar property 2-1431
- lineseries property 2-2629
- quivergroup property 2-3680

- stairsere property 2-4253
- stem property 2-4285
- surfaceplot property 2-4482

### XDataSource

- areaserie property 2-269
- barseries property 2-437
- contour property 2-1028
- errorbar property 2-1432
- lineseries property 2-2630
- quivergroup property 2-3680
- scatter property 2-4039
- stairsere property 2-4253
- stem property 2-4285
- surfaceplot property 2-4482

XDir, Axes property 2-359

XDisplay, Figure property 2-1616

XGrid, Axes property 2-359

xlabel 2-5174 2-5176 2-5178

XLabel, Axes property 2-360

xlim 2-5180 2-5184 2-5188

XLim, Axes property 2-360

XLimMode, Axes property 2-361

### XLS files

- loading 2-5194

xlsfinfo 2-5192

XMinorGrid, Axes property 2-361

xor 2-5215

XScale, Axes property 2-362

xslt 2-5216

XTick, Axes property 2-362

XTickLabel, Axes property 2-362

XTickLabelMode, Axes property 2-363

XTickMode, Axes property 2-363

XVisual, Figure property 2-1617

XVisualMode, Figure property 2-1619

### XWD files

- writing 2-2244

xyz coordinates . *See* Cartesian coordinates

**Y****Y**

annotation arrow property 2-209 2-214 2-220  
textarrow property 2-233

y-axis limits, setting and querying 2-5180 2-5184  
2-5188

YAxisLocation, Axes property 2-358

YColor, Axes property 2-358

**YData**

areaserie property 2-270  
barseries property 2-437  
contour property 2-1028  
errorbar property 2-1432  
Image property 2-2199  
Line property 2-2613  
lineseries property 2-2630  
Patch property 2-3411  
quivergroup property 2-3681  
scatter property 2-4040  
stairsereis property 2-4253  
stem property 2-4286  
Surface property 2-4457  
surfaceplot property 2-4483

**YDataMode**

contour property 2-1029  
quivergroup property 2-3681  
surfaceplot property 2-4483

**YDataSource**

areaserie property 2-270  
barseries property 2-438  
contour property 2-1029  
errorbar property 2-1433  
lineseries property 2-2631  
quivergroup property 2-3681  
scatter property 2-4040  
stairsereis property 2-4254  
stem property 2-4286  
surfaceplot property 2-4483

YDir, Axes property 2-359

YGrid, Axes property 2-359

ylabel 2-5174 2-5176 2-5178

YLabel, Axes property 2-360

ylim 2-5180 2-5184 2-5188

YLim, Axes property 2-360

YLimMode, Axes property 2-361

YMinorGrid, Axes property 2-361

YScale, Axes property 2-362

YTick, Axes property 2-362

YTickLabel, Axes property 2-362

YTickLabelMode, Axes property 2-363

YTickMode, Axes property 2-363

**Z**

z-axis limits, setting and querying 2-5180 2-5184  
2-5188

ZColor, Axes property 2-358

**ZData**

contour property 2-1030  
Line property 2-2613  
lineseries property 2-2631  
Patch property 2-3411  
quivergroup property 2-3682  
scatter property 2-4041  
stemsereis property 2-4287  
Surface property 2-4457  
surfaceplot property 2-4484

**ZDataSource**

contour property 2-1030  
lineseries property 2-2631 2-4287  
scatter property 2-4041  
surfaceplot property 2-4484

ZDir, Axes property 2-359

zero of a function, finding 2-1804

zeros 2-5218

ZGrid, Axes property 2-359

Ziggurat 2-3702 2-3709

zlabel 2-5174 2-5176 2-5178

zlim 2-5180 2-5184 2-5188

ZLim, Axes property 2-360



ZLimMode, Axes property 2-361  
ZMinorGrid, Axes property 2-361  
zoom 2-5223  
zoom mode objects 2-5224  
ZScale, Axes property 2-362

ZTick, Axes property 2-362  
ZTickLabel, Axes property 2-362  
ZTickLabelMode, Axes property 2-363  
ZTickMode, Axes property 2-363