Sysquake

User Manual

Sysquake User Manual, Dec. 2019.
Calerga Sàrl, Vevey, Switzerland.

Most of the material in Sysquake User Manual has first been written as a set of XHTML files, with lots of cross-reference links. Since (X)HTML is not very well suited for printing, it has been converted to LaTeX with the help of a home-made conversion utility. Additional XML tags have been used to benefit from LaTeX features: e.g. raster images have been replaced with EPS images, equations have been converted from text to real mathematic notation, and a table of contents and an index have been added. The same method has been used to create the material for the `help` command. Thanks to the `make` utility, the whole process is completely automatic. This system has proved to be very flexible to maintain three useful formats in parallel: two for on-line help, and one for high-quality printing.

World Wide Web:	https://calerga.com
E-mail:	sysquake@calerga.com
Mail:	Calerga Sàrl Bd Saint-Martin 21 1800 Vevey Switzerland

Typesetting: 2019-12-14

Contents

Chapter 1

Introduction

This chapter introduces to the application Sysquake, the interactive design CAD tool for getting insight into complicated scientific problems and designing advanced technical devices. You should read it to know more about what Sysquake is, what it may be used for, and how to use it for simple tasks.

1.1 Introduction

To design technical devices, or to understand the physical and mathematical laws which describe their behavior, engineers and scientists frequently use computers to calculate and represent graphically different quantities, such as the sample sequence and the frequency response of a digital audio filter, or the trajectory and the mass of a rocket flying to Mars. Usually, these quantities are related to each other; they are different views of the same reality. Understanding these relationships is the key to a good design. In some cases, especially for simple systems, an intuitive understanding can be acquired. For more complicated systems, it is often difficult or impossible to "guess", for instance, whether increasing the thickness of a robot arm will increase or decrease the frequency of the oscillations.

Traditionally, the design of a complicated system is performed in several iterations. Specifications can seldom be used directly to calculate the value of the parameters of the system, because there is no explicit formula to link them. Hence each iteration is made of two phases. The first one, often called *synthesis*, consists in calculating the unknown parameters of the system based on a set of *design variables*. The design variables are more or less loosely related to the specifications. During the second phase, called *analysis*, the performance of the system is evaluated and compared to the specifications. If it does not match them, the design variables are modified and a new iteration is carried out.

When the relationship between the criteria used for evaluating the performance and the design parameters is not very well known, modifications of the design parameters might lead as well to poorer performance as to better one. Manual trial and error may work but is cumbersome. This is where interactive design may help. Instead of splitting each iteration between synthesis and analysis, both phases are merged into a single one where the effect of modifying a parameter results immediately in the update of graphics. The whole design procedure becomes really dynamic; the engineer perceives the gradient of the change of performance criteria with respect to what he manipulates, and the compromises which can be obtained are easily identified.

Sysquake's purpose is to support this kind of design in fields such as automatic control and signal processing. Several graphics are displayed simultaneously, and some of them contain elements which can be manipulated with the mouse. During the manipulation, all the graphics are updated to reflect the change. What the graphics show and how their update is performed are not fixed, but depend on programs written in an easy-to-learn language specialized for numeric computation. Several programs are included with Sysquake for common tasks, such as the design of PID controllers; but you are free to modify them to better suit your needs and to write new ones for other design methods or new applications.

Another area where Sysquake shines is teaching. Replacing the static figures you find in books or the animations you see on the Web with interactive graphics, where the student can manipulate himself the curves to acquire an intuitive understanding of the theory they represent, accelerates and improves the learning process tremendously.

1.2 How Sysquake can be used

Sysquake is expected to be used mainly for three different purposes.

Understanding basic concepts In science as well as in engineering, theory is often not very intuitive at first, because it relates quantities from different domains: energies and positions, time and frequency, temperatures and entropies. In automatic control, where feedback is used to improve performance of any kind of system, transient behavior, such as settling time, overshoot, and risk of actuator saturation, is typically analyzed in the time domain; while the stability, noise rejection, and different kinds of robustness are more easily expressed in the frequency domain. The basic mechanisms which relate these quantities can be illustrated very effectively with Sysquake.

Designing systems The quality of a device is always the result of a compromise. Multiple objectives, such as the speed, the accuracy and the cost, must be taken into account simultaneously and have contradictory requirements. Sysquake helps a lot to find which compromises are feasible, how to push the design in the desired direction, and how to modify the specifications in case they do not permit a satisfying solution.

Developing and testing new design methods Some design methods still in use were developed at the time when computers were either nonexistent or too expensive, or too slow for interaction. While they can also benefit from Sysquake, they are better complemented or replaced by approaches which offer more degrees of freedom for better performances. A lot of work can undoubtedly be done in this field.

There is more or less a one-to-one correspondence between these application fields and the ways Sysquake can be used.

Ready-to-use interactive figures A static figure which illustrates a basic concept in a book or a course on the World Wide Web can be replaced with a dynamic figure, where the reader manipulates an element and sees its effect. For instance, to show how the height of a building influences the amplitude of its oscillations during an earthquake, the student is invited to change the building height with the mouse and see how the earthquake simulation is modified. To support this (and much more complicated interactive figures), Sysquake loads an *SQ file*, a text file which contains the description of the figures and the code necessary to support the interaction. SQ files do not have to handle advanced features provided automatically by Sysquake, like multiple undo, zoom, and scale options. They rely on built-in functions for graphics related to linear dynamic systems, but can also compute and display arbitrary data with a complete language.

Set of programs for different design or analysis methods SQ files are not limited to a fixed set of figures and interaction. They can implement all that is needed for a given design or analysis approach. Then the user chooses what he wants to display, may enter numeric parameters which characterize the problem, and manipulates the graphics until he obtains the desired results. He can also save the parameters of his design in an *SQD file* (Sysquake Data file) and load them later.

Programs written from scratch The applications described above are based on SQ files, which either come from those distributed with Sysquake or are contributed by other users, or are

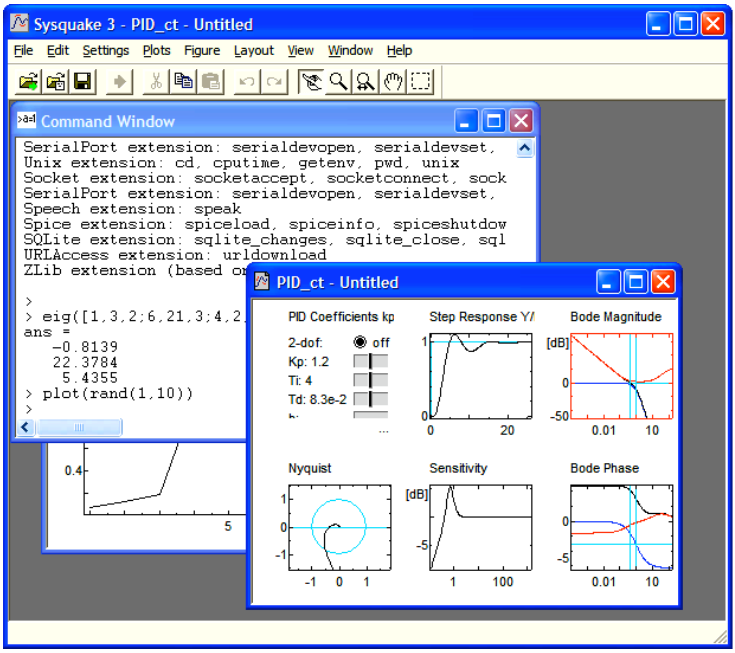

Figure 1.1 Sysquake on Windows XP

written from scratch. Developing an SQ file is not difficult; however, it requires some programming knowledge and experience. For small problems, a simpler programming model is available, *SQ scripts*.

These purposes are the *raison-d'être* of Sysquake which make it different from any other general-purpose math software. However, Sysquake can *also* be used as a powerful calculator for quick interactive evaluation of expressions.

Another advantage of Sysquake is that it runs on both Windows (see Fig. 1.1) and Mac OS X (see Fig. 1.2), with a polished user interface. Other Calerga products share the programming language of Sysquake. For example Sysquake Remote provides scientific computing and graphics for Web applications (more informations can be found on the Web site of Calerga, https://calerga.com). The part of the application which does not rely on Sysquake-specific features (interactive graphics) can be easily reused in these products.

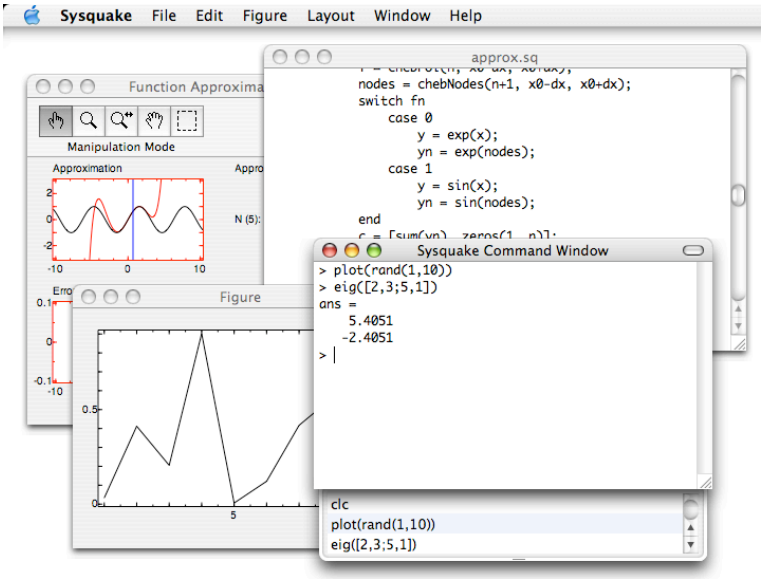

Figure 1.2 Sysquake on Mac OS X

Chapter 2

Registration

Sysquake Pro and Sysquake Application Builder require a registration key file in order to run. Please follow the instructions you have received with Sysquake.

Licenses for multiple computers and site licenses do not require separate registration for each computer; the same registration file is used on each computer. If your company or university has acquired such a license, a single person should be responsible for the registration file deployment; Calerga cannot easily accept and validate requests from multiple persons for the same license.

2.1 Where SQ_Reg.key is located

The registration information is stored in the file SQ_Reg.key on the hard disk. Its location depends on the operating system.

Windows

On Windows computers, SQ_Reg.key is located in the same folder as Sysquake.exe, typically in the directory C:\Program Files\Sysquake.

If the installer application finds the file SQ_Reg.key in the same directory where it is located, it copies it automatically to its final location.

macOS

On macOS computers, SQ_Reg.key can be located in three different places:

- in the same folder as the Sysquake application;
- in ~/Library/Calerga, where ~ stands for your home folder;

– in /Library/Calerga (starting in the boot volume).

If Sysquake Pro or Sysquake Application Builder cannot find a valid registration file at startup, it will request one and copy it to ~/Library/Calerga. If Sysquake is used by different users on the same computer, you can move it to /Library/Calerga (you need administrator privileges to do that).

The folder of Sysquake is usually not recommended to reduce clutter in the Applications folder and to avoid deleting it when you replace Sysquake with a newer version.

Linux

On Linux computers, SQ_Reg.key is located in the base directory of Sysquake, typically /usr/local/sysquake.

2.2 Remark

The serial number is linked to your name and your company's name. The protection system is designed to permit you to reinstall the serial number easily, on the same or a different computer. You may even copy the registration file onto different computers if you have a multiple license for the same software, or to your private laptop computer. License files for Windows and Mac OS are compatible. Simply make sure you comply with the license terms.

When you perform a minor upgrade of Sysquake, you can keep the same registration file without running the registration application or contacting Calerga again.

2.3 What's in the Serial Number

The serial number is not a random string of letters and digits. It can be decoded as follows. A typical serial number looks like

```
500549a-999999-0a20.a392
```

To decode it, split it into six parts:

```
vvvwwwf-yyyymm-ssss.ssss
```

Each group of characters has the following meaning:

– **vvv**, a group of three digits, is the first version number for which the license is valid; `500` means 5.0.

- **www**, a group of three digits, is the last version number for which the license is valid; 549 means 5.4.9. Note that this is not a commitment to offer free upgrades, or any upgrade at all.
- **f**, a single letter, identifies additional capabilities of Sysquake; a means none. It is not used in current releases.
- **yyyy**, a group of four digits, is the last year of the validity of the license; 9999 means no expiration.
- **mm**, a group of two digits, is the last month for which the license is valid; 99 means no expiration.
- **ssss.ssss**, a group of eight hexadecimal digits, is an encrypted checksum which binds the serial number to other bits of registration information.

Chapter 3

Getting Started with Sysquake

This chapter describes the user interface of Sysquake, i.e. how you manipulate its windows and what you can find in its menus.

3.1 First steps

Launch Sysquake, then choose File/Open (i.e. click the File menu, then the Open item) and select the file sampling.sq, which should be located in the "SQ_Files" folder, to learn basic manipulations. The purpose of this SQ file is to show the effect of sampling (and other method for converting systems from continuous time to discrete time) on the frequency response and step response.

In the figure window, two graphics should be displayed (see Fig. 3.1). At the top, the frequency response amplitude of a continuous-time third-order system is represented as a black line. The red curve represents the frequency response amplitude of the same system, but sampled. The sampling frequency is represented by the blue vertical line, and the Nyquist frequency (half of the sampling frequency) in green. The discrete-time curve has a periodicity equal to the sampling frequency. The bottom figure represents the step response of the system, again both in the continuous-time domain and in the discrete-time domain (red circles).

If you click one of the vertical lines and hold down the mouse button, you can move the line to the right to increase the sampling frequency or to the left to decrease it. More interesting is the effect on the discrete-time responses. Both the frequency responses and the step responses are updated during the move. If you decrease enough the sampling frequency, the frequency response matches more and more loosely the continuous-time response. In the time domain, you

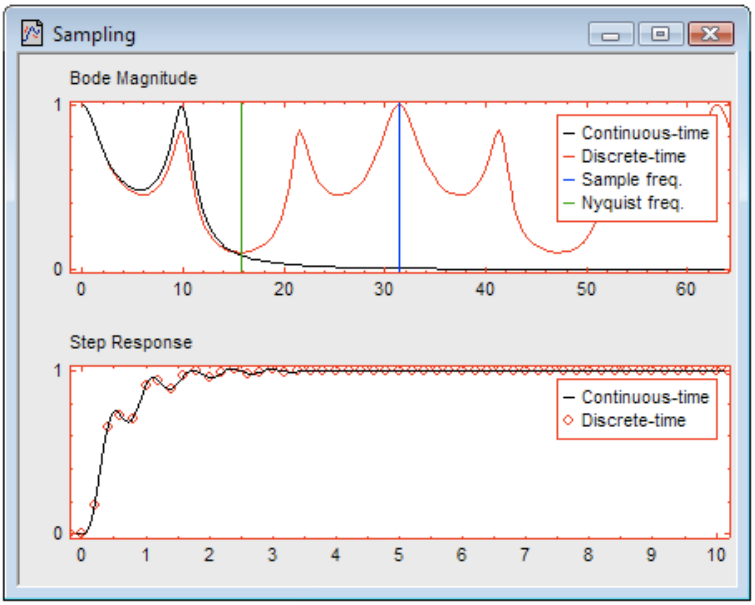

Figure 3.1 SQ file sampling.sq

can see that the samples do not follow the oscillations. You can also click and move (or *drag*) one of the samples in the step response plot.

A very useful feature of Sysquake is the Undo command. You can undo and redo any interactive manipulation, change of settings, zoom, and choice of figures to display, as many times as memory permits. These commands can be found in the Edit menu. They are your friends; learn to use them frequently!

3.2 Files

Sysquake uses several kinds of files: SQ files, SQD (or SQ Data) files, libraries, and files created by programs. SQ files are kinds of *programs*, or *scripts*, which implement figures, menus and computation code for a given topic. Several SQ files are provided with Sysquake. Some of them are suitable for many problems in the same area and can be customized with your own data, for example for the design of a digital filter, while others have more narrow applications.

You can also write your own SQ files with an external text editor. If you use a word processor, make sure that you save SQ files as text only, or ASCII; many word processors add formatting information (such as margins or character style) which is not understood by Sysquake. On Macintosh, SQ files are associated with Sysquake and get an SQ file icon when you open them with Sysquake for the first time, either

Figure 3.2 Toolbar (Windows Vista)

Figure 3.3 Toolbar (Windows Vista)

from Sysquake (menu File/Open) or by dragging the document icon onto Sysquake's; then you can simply double-click the SQ file icon in the Finder to open it in Sysquake.

SQD files store the state of a session with Sysquake. They are always associated with an SQ file. They only contain the data necessary to load back the SQ file and restore the parameters, settings and figures which were defined when they were saved. Typically, they are written and read by Sysquake; but since they are also text files, you can easily create them by hand, read the values, or use them to exchange data with other applications.

Libraries are collections of functions which complement the built-in functions of Sysquake's language. They are made available explicitly to SQ files or other libraries which request them.

With its rich built-in language, Sysquake can also read and write arbitrary text and binary files. What they contain and when they are created and read depend only on the programmer. For example, an image processing SQ file could read TIFF files (a popular file format for raster images) by adding a custom entry in the Settings menu. This is actually what image.sq, one of the SQ files provided with Sysquake, does.

3.3 Manipulation modes

There are five manipulation modes, which can be chosen in the Figure menu or in the toolbar (toolbars differ slightly on Mac OS (see Fig. 3.2), Windows (see Fig. 3.3) and Linux).

Manipulate You can drag graphical elements in one of the subplots and see the effect this has on other figures. Depending on the figure, you can drag graphical elements horizontally, vertically, or in any direction. Not all the elements can be dragged; the shape of the cursor usually indicates whether dragging is enabled (hand with index finger) or disabled (standard arrow cursor). For some

figures, holding down the Shift key modifies the action performed by the drag.

In automatic control and filtering, the dynamic of linear systems is often represented by poles and zeros symmetric with respect to the horizontal axis (real axis). To enforce this symmetry, dragging a complex pole or zero (a pole or zero not on the real axis) also moves the symmetric one; attempting to drag a complex pole or zero to the other side of the real axis makes it stick to the real axis; and dragging a pole or zero from the real axis can be done only upward if there is at least one other pole or zero on the real axis, which becomes its symmetric.

Zoom You can click in a subplot to double the scale in both x and y directions, or select the area you want to display by holding down the mouse button. To zoom out, hold down the Shift key and click. To revert to the default scale, select the subplot (see below) and select Figure/Automatic Scale. Many figures have automatic scaling by default; to fix the scale without zooming, unselect Figure/Automatic Scale.

In 3D figures, a click zooms in, or zooms out if the Shift key is held down.

Zoom X Axis The zoom is constrained to the x axis. The scale remains automatic for the y axis if it already was before the zoom. Note that figures where the scale for both axis is constrained to be the same cannot be zoomed in or out in this mode.

In 3D figures, a click zooms in or out with orthographic projection. With perspective projection, a mouse drag toward the center of the figure moves the point of view closer to the target point, keeping the zoom factor such that the image size is preserved. A mouse drag from the center of the figure has the opposite effect, moving the point of view further from the target point.

Drag You can click and drag a figure to move it in its subplot region. The limits of the displayed area of the plot are changed such that the point under the mouse cursor moves with it. Hold down the Shift key to drag along the x axis.

In 3D figures, a mouse drag moves the point of view around the target point, as if the mouse cursor dragged a sphere around the object. If the Shift key is held down, both the point of view and the target point are moved (i.e. the camera dollies parallel to the image).

Select You can select a subplot by clicking it (this is necessary only if several subplots are displayed simultaneously), and change what is displayed by choosing an entry of the Plot menu or some options

in the Figure menu. By holding down the Shift key, you can select more than one subplot. You can also drag a figure from one subplot slot to another one. The figure which was in the target subplot replaces the dragged figure. The zoom factor and scale options are preserved. Some commands are available from a contextual menu (click in the figure with the right button, or hold down the Control key and click in a figure on Macintosh with a single-button mouse); you do not need to select a subplot before using it.

It is sometimes useful to synchronize the area displayed in two subplots. For instance, if you display simultaneously the amplitude and the phase of a frequency response, you may want to zoom along the X axis to display a smaller frequency range, identical for both figures. To do that, first select both subplots (switch to Select mode, click the first subplot, hold down the Shift key, and click the second subplot). Then switch to Zoom, Zoom X, or Shift mode, and change the scale of one of the selected subplot. All the selected subplots will follow, provided that their scale was already the same before the change.

Remark: with some versions of Sysquake, the middle button of mouses with three buttons has the same effect as holding down the Shift key and clicking with the left button.

3.4 Menus

This section describes the commands you can find in the menus. The most important ones have keyboard shortcuts which are more efficient for the experienced user.

Note that some menu entries are enabled or disabled by the contents of the SQ file currently loaded. For example, in many figures, such as those with negative values, the logarithmic scale is disabled.

File

The File menu (see Fig. 3.4) contains the commands which handle SQ files as a whole (file operations and reset to the default values) and quit Sysquake itself.

New Opens a new text editor window, where you can type the source code of an SQ file or the contents of any text file. This is used mainly to develop new Sysquake applications, but can be convenient as a general-purpose text editor.

Open and Run Opens either an SQ file or an SQD file, i.e. a file which contains customized values and settings as well as a reference to the related SQ file. When you open an SQD file, Sysquake

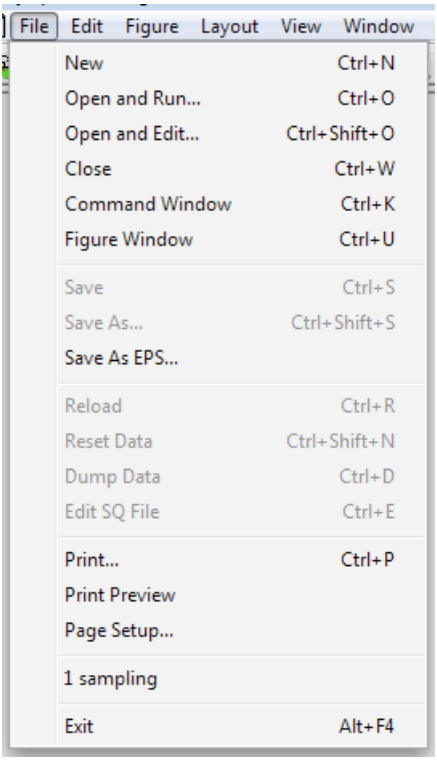

Figure 3.4 File menu

loads its related SQ file first, then restores the state of the data and the figures which prevailed when the SQD file was saved.

Open and Edit Opens a text file in the editor window without attempting to running it as an SQ file. You can then inspect it, edit it, save it, and (if it is an SQ file) run it.

Command Window Displays the command window. The command window is a window where you can type commands and evaluate expressions, as well as see the textual output (if any) of the execution of SQ files. This is especially useful when you develop your own SQ files. On Macintosh, Command Window is in the Window menu.

Figure Window Displays the figure window. The figure window is the window where graphics created by SQ scripts or from the Command window are displayed. On Macintosh, Figure Window is in the Window menu.

Close Closes the active window. The Figure window and the Command window are only hidden and can be shown again with the menu entries described above.

Save Saves the current values and settings with a reference to the SQ file in the current SQD file. For an SQD file to be considered as "current", it must have been opened, or already saved with Save As. Otherwise, you must select Save As to provide a file name for the new SQD file.

Save As Same as Save, but a file name and location is requested.

Save As SQ File Same as Save As, except that an SQ file is written instead of an SQD file. The SQ file has the advantage of being self-contained and independent from the original SQ file, and the disadvantage of requiring more disk storage and not benefiting from improvements made to the original SQ file. If the SQ file contains a help text, the mention *Saved as SQ file (default values are new)* is appended to make clear that the SQ file has been modified. You can edit the SQ file to change the help message (as well as other elements) if the SQ file author permits you to do so. On Windows, Save As SQ File is an option of the Save As dialog.

Export Graphics As The contents of the figure window are saved as an EPS (encapsulated PostScript) or PDF (Portable Document Format) file, i.e. a high-quality graphics file which can be imported in many programs.

Reload Reloads the current SQ file. This is especially useful when developing your own SQ files with an external text editor.

Reset Data Reverts to the default values of the SQ file or the values of the SQD file.

Dump Data Writes what would be saved in an SQD file by the Save command to the Command window or panel. Usually, the result corresponds to the variables used by the SQ file and contains the numeric values the figures are based on.

Edit SQ File When the figure window of an SQ file is frontmost, Edit SQ File switches to its source code in a text editor window. You can inspect the code, modify it and reload it.

Print Prints the contents of the figure window. Depending on the operating system, it may also be possible to print the command window or panel, to specify printing options, and to preview what would be printed.

Recent files The most recent files can be opened without having to find them with the Open menu entry.

Quit or Exit Quits Sysquake. On macOS, the Quit entry is located in the Sysquake menu.

Edit

The Edit menu contains the commands which manipulate the clipboard for data exchange within Sysquake and with other programs, and the Undo/Redo commands. Note that the command-line interface (the text window where you can type direct commands) may support only Undo or no Undo at all, depending on the platform.

Undo Reverts to the situation which prevailed before the last user action. Most figure interactive manipulation and setting in the Settings menu can be undone. Undo can be used as many times as memory permits.

Redo Undoes the last undo.

Cut, Copy, Paste, and Clear Standard editing operations. In the figure window, only Copy is supported; it makes a copy of the selected subplot, or of all the subplots, as graphics (the exact format depends on the operating system).

Select All Selects all the subplots.

Copy As For SQ files which can export data by copying it to the clipboard, the submenu items show what can be exported. Once copied, data can be pasted in another SQ file or another application. For example, an SQ file for identifying the parameters of a model based on experiments performed on a real process may be copied to the clipboard and pasted into another SQ file for the design of a feedback controller.

Paste As For SQ files which can import data by pasting it from the clipboard, the submenu items show what can be imported. Data which can be imported have typically been copied in another SQ file, but can also come from another application.

Preferences

Depending on the platform, preferences are grouped in a single dialog window or available separately from a submenu. On macOS, preferences are found in the Sysquake menu. Settings are saved and restored the next time Sysquake is launched.

Set SQ file path When an SQD file is opened, Sysquake looks for the name of the SQ file associated with it. Then it looks first in the folder of the SQD file, then in prespecified folders to load the SQ file and set the variables based on the contents of the SQD file. With Set SQ File Path, you can check and change where Sysquake looks for SQ files. By default, this is in the folder "SQ_files", itself located

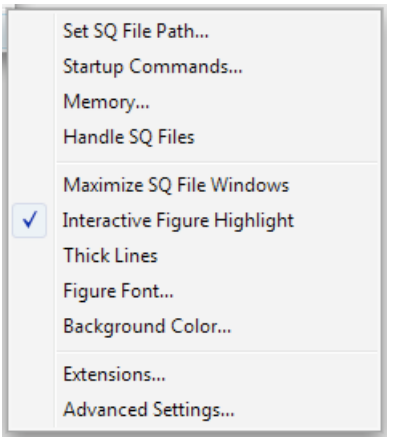

Figure 3.5 Preferences submenu

at the same place as Sysquake; but you can change it, or add your personal folders. Paths also specify where libraries and help files are searched. Each path is relative to Sysquake, and folders are separated with new lines.

A path can be:

- The directory which contains the files; the path of the file is obtained by concatenating this directory and the file name, with a separator such as / or \ if it is missing.
- A string with character sequences `%b`, `%s` and/or `%f` which are replaced respectively by the file base name without suffix, the suffix (extension such as `sq` for SQ files or `lml` for libraries), and the file name with suffix.

The syntax of paths depends on the operating system. On Windows, relative paths begin with a backslash ('\'), and absolute paths begin with the volume letter followed by a colon (e.g. 'C:'); each element is separated with a backslash. On macOS, folders are selected in a dialog window.

On most platforms, Sysquake also supports URL. SQ files and functions defined in libraries obtained with a URL path are executed in a sandbox. See function `path` for more details.

Startup commands Startup commands are LME commands executed every time Sysquake is launched. Some global settings can be changed by calling functions: for instance, `format` sets the way results are displayed, and `defaultstyle` sets the default style of new figures.

Another use is to import libraries at startup. If you often call functions from the same libraries (such as `stdlib` for basic

function which extend the built-in functions of Sysquake or `stat` for advanced functions related to statistics), you can add use statements as startup commands to make them always available: `use stdlib, stat`. Functions are imported only for usage in the command-line interface, not in SQ files which must specify explicitly the libraries they use.

Memory The memory usage is adjusted automatically when required. However, depending on your needs, it may be better to allocate a large amount of memory at startup or to limit the maximum amount of memory. We recommend that you keep the default values, except for the minimum memory which can be increased but should be kept lower than your physical memory for best performances.

Handle SQ files (Windows only) Windows stores information about which application should be invoked when a document icon is double-clicked in a central location called the Registry. If you move Sysquake, or if you install a new version (e.g. by upgrading from Sysquake to Sysquake Pro), you should tell Windows if you want the Sysquake which is currently running to open and print SQ files. You do so by choosing Handle SQ Files. Then the menu entry is checked and disabled, because you cannot revert your action. To select another version of Sysquake, run it and select Handle SQ Files from it.

Interactive figure hilight When selected, the frame around interactive figures (subplots where there are elements which can be manipulated with the mouse) is displayed in red instead of black. In many cases, the mouse cursor also changes when the mouse is over an element which can be manipulated.

Thick lines When selected, all lines in graphics are displayed with thicker lines. This may be useful for demonstrations.

Figure font A dialog box offers the choice of font for the figures.

Background color A dialog box is displayed to change the background of figures, between subplots.

Ignore assert In Sysquake programs, the `assert` function can help in reporting errors during development. If Ignore Assert is on, the evaluation of `assert` is skipped, which can provide slightly improved performance. Usually, you should switch it on during development, and off when using SQ files whose you trust the quality.

Ignore private and hideimplementation attributes of functions Functions stored in library files can be public, i.e. accessible from other libraries and SQ files, or private, i.e.

available only to other functions in the same library. Their implementation can be hidden, so that error messages are the same as for native functions and do not contain information about the error location; and debugging cannot step into them. If Ignore private and hideimplementation attributes is on, private functions can be executed as if they were public and hideimplementation is ignored. This can be useful for development, to debug functions from the Command window.

Code Optimization Programming code of SQ files is converted to an intermediate code for faster execution. Code optimization further speeds up its execution by replacing some sequences of code with faster alternatives. Most of the time, you should keep this option set.

Rate Limit to Mouse Drag and Move Handlers Sysquake programs often perform repeated computation when the mouse is moved. When Rate Limit is on, the rate of these computations is limited to 25 times per second; otherwise, it is limited only by the processing power of the computer. Limiting the rate can reduce the power consumption, with increased battery autonomy on laptop computers and less fan noise on desktop computers.

SQ File Possible Error Warnings When you develop new SQ files, Sysquake can help you to find potential problems. In addition to errors which prevent the SQ file to run at all, which are always reported, Sysquake can analyze your code and find programming patterns which are often not intended and cause errors difficult to find. You should leave this option off for SQ files which are known to be correct, because warnings do not always correspond to errors and do not mean that the SQ file has a lower quality.

Default Sandbox Mode for SQ Files SQ files can not only perform mathematical calculation, but also access files, network, or other devices. This makes the execution of SQ files obtained from trusted sources potentially dangerous. The sandbox is a secure environment where all commands which could be harmful are disabled. It can be accessed by code with the sandbox function, or enforced globally for SQ files. The Default Sandbox Mode specifies whether the sandbox mode is enabled when new SQ files are opened.

Ask Before Closing If you have change the state of an SQ file (typically by manipulating the graphics interactively) and Ask Before Closing is on, Sysquake will ask you if you want to save it to an SQD file when you close the window.

Settings

The Settings menu, available only for some SQ files, contains actions defined in the current SQ file. These actions typically modify the system in a noninteractive way, for example to enter numeric values for the coefficients of a model or to change the structure of a controller. SQ files can also redefine the menu name and define several menus.

When a dialog box is displayed, the edit field contains the current values of one or several parameters, separated by commas. Values can be real or complex, scalar, vector, matrices, lists, structures, or inline or anonymous functions. Polynomials are represented by their coefficients, in decreasing power, in a row vector. Here are some examples:

Type	**Example**
Real scalar	`1, 2.3, -3.2e5`
Complex scalar	`3i, 1.2-5.4j`
Row vector	`[4,2,6]`
Column vector	`[4;2;6]`
Polynomial	`[1,2,5]`
Matrix	`[1,3;2,8]`
Set of polynomials	`[1,4.9,3.1;1,6.2,2.6]`
Identity matrix	`eye(3)`
Range	`1:10`
List	`{1,2:5,'abc'}`
Structure	`struct('A',[1,2;3,-1],'B',[3;5])`
Anonymous function	`@(t,tau) 2*exp(-t/tau)`

Plots

The Plots menu, available for most SQ files, contains the list of figures which can be displayed. General-purpose SQ files usually define figures for everything you might want to observe, but only display a few of them by default. More specialized SQ files can have one or two plots which are displayed by default; in this case, the Plots menu is less useful.

To change one of the figures, first select it (click the selection button in the toolbar or choose Select in the Figure menu), then choose one of the entries of the Plots menu. You can change the number of figures which are displayed simultaneously with the Layout menu (see below).

Figure

The Figure menu (see Fig. 3.6) permits to select one of the five modes of operation on the figures, as described above, and to change display options for the selected subplot(s). A subplot is selected either if it has

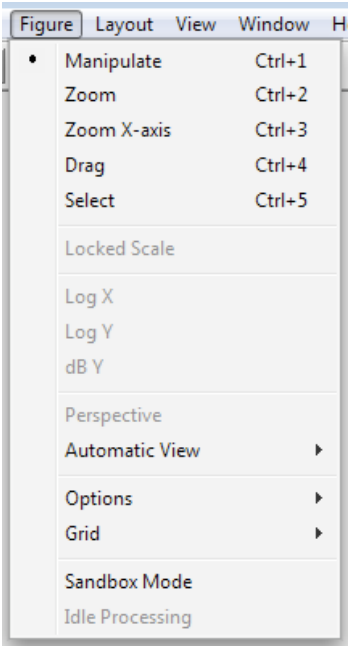

Figure 3.6 Figure menu

been clicked in Select mode, or if it is alone. Depending on the figure, some options may be disabled.

Manipulate, Zoom, Zoom X, Drag, and Select Selects one of the modes of operation.

Locked Scale SQ files typically define a default scale for each figure. The scale may adapt to the figure contents. When you zoom or drag the figure, the scale is locked. You can unlock the scale of the selected subplot and revert to the default scale by unselecting Locked Scale; you can also lock an unlocked figure, e.g. if you want to better observe small amplitude changes when you manipulate another figure. Note that the figure which is manipulated has always its scale locked during the manipulation.

Log X, Log Y, and dB Y With Log X, the horizontal scale becomes logarithmic, and negative values are discarded. With Log Y, the vertical scale becomes logarithmic. dB Y is the same as Log Y as far as the contents of the figure are concerned; however, the axis is labeled in dB's, i.e. a linear scale where a difference of 20 represents a factor 10 for the data: y'[dB] = 20 log10(y) (decibels are defined with 10 instead of 20 for powers; the definition used by Sysquake is valid for voltages, currents, mechanical displacements, etc. which are proportional to the square root of powers).

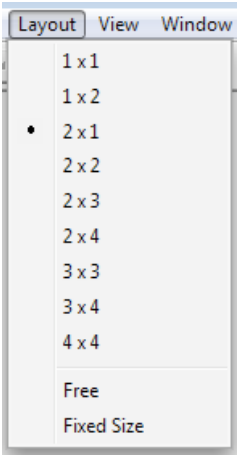

Figure 3.7 Layout menu

Options The Options submenu has three items which may be enabled or disabled. Frame displays a rectangular frame around the graphics with ticks and a white background. Margin leaves room around the frame for the title and axis labels. Label displays the title of the figure. Legend displays the meaning of symbols and colors in figures, provided the feature is implemented in the SQ file. Legends are located in one of the four corners of figures; they can be moved with the mouse.

Grid The Grid submenu sets the level of detail of the grid which is displayed in the back of the selected subplot(s). The availability and the kind of grid depend on the figure.

Sandbox Mode Sandbox Mode specifies whether potentially harmful commands (commands which give access to files, network and devices) are disabled (see above).

Idle Processing Some SQ files perform computations even when you do not interact with them, for example for animated simulations. The computation may slow down the whole computer. To suspend it, unselect Idle Processing. Note that the menu entry is disabled if the SQ file does not implement idle processing.

Layout

The Layout menu (see Fig. 3.7) permits to choose the number of subplots to be displayed simultaneously.

1x1, 1x2, 2x1, 2x2, 2x3, 2x4, 3x2, 3x4, 3x4, 4x4 The corresponding number of subplots is displayed. If some subplots were

selected, they are preserved in the new layout; otherwise, the subplots remain at the same position.

Free Instead of being constrained on a regular grid, the subplots can be freely moved and resized with the mouse (in Select mode). When the window is resized, all subplots are scaled the same way.

Fixed Size The subplots can be freely moved and resized with the mouse like with Free layout, but their size and position remain fixed when the window is resized.

View (Windows)

On Windows, the View menu permits to show or hide toolbars and panels.

Command Panel The command line interface is available in a subpanel of the main window. You can show it or hide it with the menu entry View/Command Panel. When is it shown, you can resize it by dragging the separation with the mouse.

Toolbar The toolbar can be hidden or shown. It can also been torn off or docked with the mouse.

Status Bar The status bar (the region at the bottom of the main window where status messages are displayed) can be shown or hidden.

Help

The Help menu provides information about Sysquake with access to the online version of its user manual and a simple integrated SQ file; and about the SQ file in the front window. The entry "About Sysquake" is located in the Sysquake menu on macOS. Selecting it displays information about the version of Sysquake and whom it is registered to.

Contextual menu

Some commands which are related to the currently selected figure(s) are available from a contextual menu, obtained by clicking with the right button of the mouse (on Macintosh with a single-button mouse, hold down the Control key and click). The figure becomes selected, and a contextual menu appears right under the mouse cursor with commands for choosing the figure and changing the scale and the grid. This is very convenient to avoid switching to and from the Select mode.

3.5 Command-Line Interface

The command-line interface is useful for two purposes:

- as a powerful calculator, where you can type expressions and get answers;
- as an help for developing, testing and debugging new SQ files.

You can ignore it if you use existing SQ files. Note also that you cannot add interactivity from the command line; interactivity requires SQ files (or SQ scripts, which are very close to the commands you can type in the command-line interface).

The command-line interface, SQ scripts, and SQ files are detailed in other chapters.

3.6 Interruption Key

The goal of Sysquake is to be as interactive as possible. However, nothing in its design prevents it from doing long computations, from the command line as well as in SQ files. You can interrupt it by pressing the following keys:

On Windows: Break (Control-Pause).

On macOS: Esc or Command-dot (hold down the Command (Apple) key, then type a dot).

On Linux: Shift-Esc.

When the computations take more than half a second in an SQ file, a mark is displayed at the bottom left of the window.

3.7 Memory

LME uses a block of memory of fixed size. If a function call requires more space than the current size of the block allows, e.g. `magic(500)` for a 500x500 magic array, an error occurs. However, after the error, an attempt is made to increase the available memory. If you typed the command in the command-line interface (see below), you can retry (use the Up arrow key to retrieve the previous command from the history buffer). But if the error occurred during the execution of a handler (a function defined in an SQ file), Sysquake will retry automatically. This procedure is usually transparent for the user, unless a dialog box has been presented; in that case, it may be displayed several times before either enough space is allocated or the maximum amount of memory is reached.

3.8 Extensions

Sysquake is an self-contained application which does not rely on other files to run, except for the registration file SQ_Reg.key for versions which require it.

With Sysquake Pro, it is possible to add optional functionality with the help of extension files. For instance, additional high-quality numeric functions are provided by the file "LMELapack".

At startup, Sysquake scans the folder "LMEExt" located in the same folder as Sysquake Pro itself and loads all extensions it finds there. Other files are ignored. Extensions may be removed without harm; note however that some libraries and SQ files may require them to run correctly.

3.9 Preference Files

Sysquake retains information about the user preferences between invocations. The location where this information is stored depends on the platform.

Windows

Preferences are stored in the system registry, in HKEY_CLASSES_ROOT for information related to the association between Sysquake, its files and icons (preference "Handle SQ Files" as described above) and in HKEY_CURRENT_USER/Software/Calerga for other preferences.

macOS

Preferences are stored in the file "com.calerga.sysquake.plist" located in the preference folder, "~/Library/Preferences". The file has the structure of standard preference files on macOS and can be edited with the application *Property List Editor* which comes with the Apple developer tools.

Linux

Preferences are stored in the home directory in the following files:

.sysquakeprefs Main preferences (text file with pairs *name*="*value*")

.sysquakehistory Past commands, as they can be retrieved with the up arrow key in the command window, as a text file. Entries are separated with lines containing two exclamation marks.

.sysquakestartupcmd Startup commands (text file)

Preferences of other applications of the Sysquake software suite are stored in ".sqappbuilderprefs", ".sqruntimeprefs", and ".sysquakeleprefs".

3.10 Environment Variables

Environment variables are named strings which can be specified for each running application. They are supported on many platforms, including Windows, macOS and Linux. How they are used and which name is meaningful depend on the application, its libraries and the operating system.

In Sysquake, environment variable values are obtained with the function `getenv` defined in the Shell extension. In addition, Sysquake for Linux uses the following environment variables:

HOME Home directory, where preference files are found. This variable is set automatically by Linux.

SYSQUAKEDIR Base directory of Sysquake, where the standard directories "SQ_files", "Lib" and "LMEExt" are located. If this variable is not defined, Sysquake attempts to use its own name as it is provided by the operating system (this works only if Sysquake is launched by specifying a relative or absolute path, not if the path is implicitly found in the PATH environment variable); or as a last resort, the fixed directory "/usr/local/sysquake" or "/usr/local/sysquakepro".

SYSQUAKEKEY Path of the registration file, whose name is usually "SQ_Reg.key" or "SQ.key". If this variable is not defined, the registration file is searched successively in "*SYSQUAKEDIR*/SQ_Reg.key", "*HOME*/.Sysquake/SQ_Reg.key", "*HOME*/SQ_Reg.key", and "./SQ_Reg.key", where *SYSQUAKEDIR* and *HOME* are the values of environment variables SYSQUAKEDIR and HOME respectively, and "." is the current working directory.

X11BROWSER or BROWSER Path of the HTML application to use to display the documentation. If neither of these variables is defined, Sysquake tries to execute `htmlview`, `firefox`, `mozilla`, `netscape`, `opera`, and finally `konqueror`. In versions of Sysquake which support it, the `launchurl` command uses the same browser.

Chapter 4

SQ Files

This chapter describes the main SQ files provided with Sysquake. For other SQ files with a more limited scope, please consult the on-line documentation.

Automatic Control

PID_ct.sq Continuous-time PID controller.

PID_dt.sq Discrete-time PID controller

RST_ct.sq Continuous-time two-degrees-of-freedom linear controller.

RST_dt.sq Discrete-time two-degrees-of-freedom linear controller.

LQR_ct.sq Continuous-time two-degrees-of-freedom linear-quadratic regulator.

Signal Processing

filter.sq Design of analog and digital filters.

id_p.sq Parametric identification.

id_np.sq Non-parametric identification.

4.1 PID_ct.sq

Continuous-time PID controller

PID controllers, or proportional-integral-derivative controllers, are probably the most popular kind of linear single-input single-output

controllers. This is justified by their simplicity and their effectiveness for a large class of systems. Taking as input the difference between the desired set-point $r(t)$ and the measured system output $y(t)$ ("error" $e(t) = r(t) - y(t)$), they have three terms with easy-to-understand effects which are added up, and three parameters to adjust their weights:

- a proportional term (the larger the error, the larger the control signal to reduce it);
- an integral term (if a nonzero control signal is required to cancel out the error, the control signal is increased until the error vanishes);
- a derivative term (the evolution of the error is anticipated to increase damping).

Weights can be specified either separately for the three terms, or as a global gain k_P and two time values T_I and T_D which do not depend on the gain of the system. PID_ct.sq uses the latter parameterization. The control signal $u(t)$ is

$$u(t) = k_\mathrm{P}\left(e(t) + \frac{1}{T_\mathrm{I}}\int_0^t e(t)\mathrm{d}t + T_\mathrm{D}\,\frac{\mathrm{d}e}{\mathrm{d}t}(t)\right)$$

The transfer function of the controller $K(s) = U(s)/E(s)$, where $U(s)$ and $E(s)$ are the Laplace transforms of $y(t)$ and $e(t)$, respectively, is

$$K(s) = k_\mathrm{P}\left(1 + \frac{1}{T_\mathrm{I}s} + T_\mathrm{D}s\right)$$

Translating the conceptual simplicity of the PID into an effective design is not always straightforward. PID_ct.sq displays the graphics where common specifications can be checked (see Fig. 4.1); you can manipulate the PID parameters, the controller gain k_P in the Bode, Nyquist, or root locus diagram, or the time values of the integrator and the derivator in the Bode, root locus, or open-loop poles diagram.

For set-point tracking, filtering the same way the measured output and the set-point by considering only the error $e(t) = y(t) - r(t)$ does not give a good transient behavior when the set-point is discontinuous. The set-point is usually not differentiated. In addition, the proportional term of the controller k_P applied to the set-point can be reduced by a factor b smaller than 1. A third common improvement is to filter the derivative term to limit the amplification of noise at high frequencies (this is actually required to have a causal controller); the filter is parameterized with a number N, typically between 10 and 20, which is the bandwidth of the effect of the derivator term. In the Laplace domain, the control signal is

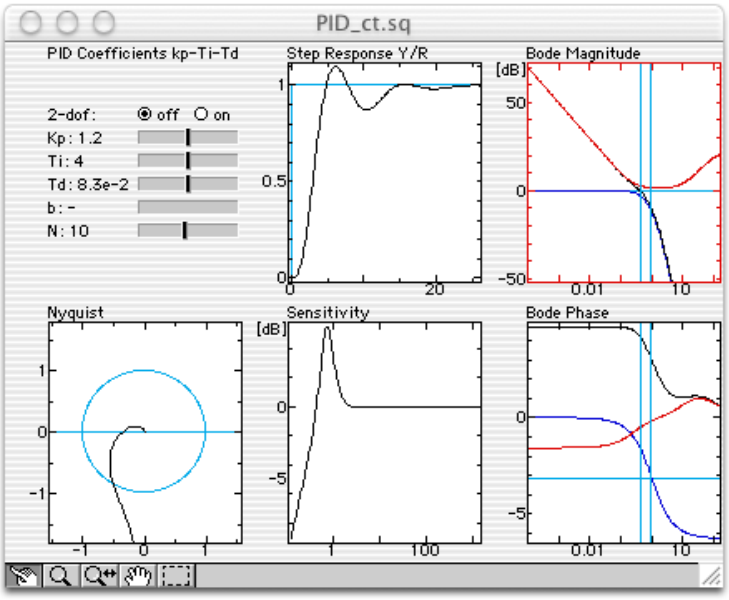

Figure 4.1 PID_ct.sq

$$U(s) = k_P \left(bR(s) - Y(s) + \frac{1}{T_I s} E(s) - \frac{T_D s}{1 + T_D s/N} Y(s) \right)$$

Figures

The figures are the same as those defined for RST_ct.sq, except for the Open-Loop Zeros and Poles and the Closed-Loop Poles which are not defined.

Settings

The System, Sampling Period, method for converting to digital controller, and Damping Specification have the same effect as the corresponding menu entries defined in RST_ct.sq. Two new entries are defined.

PID Coefficients

The three parameters of the PID (k_P, T_I and T_D) can be edited in a dialog box. For P, PI, or PD controllers, set the parameter of the missing component to the empty matrix [].

No Derivator On Reference

When the input of the PID controller is the error between the set-point and the measured output, discontinuities of the set-point are differentiated by the derivator component of the PID and yield infinite values for the control signal (see above).

When No Derivator On Reference is checked, the set-point is not differentiated.

Display Frequency Line

When selected, moving the mouse above a frequency response (Bode or sensitivity) will display a corresponding line in other frequency responses, Nyquist diagrams, and zero/pole diagrams.

4.2 PID_dt.sq

Discrete-time PID controller

The PID controller is fundamentally a continuous-time controller. However, it is often implemented with digital electronic devices (such as microcomputers, microcontrollers, or FPGA). Sampling effects may change performance in subtle ways, especially when the sampling frequency is not very high with respect to the bandwidth of the controlled system. Instead of converting a continuous-time PID controller, it is possible to design a PID directly in the discrete-time domain, approximating the integration and derivation by sums and differences, respectively. The parameters of the PID keep their standard meaning. PID_dt.sq does for discrete-time PID controllers what PID_ct.sq does for continuous-time PID controllers (see Fig. 4.2). In its simplest form, the transfer function $K(z)$ of the PID is

$$K(z) = k_\mathrm{P}\left(1 + \frac{T_s}{T_\mathrm{I}(z-1)} + T_\mathrm{D}\frac{z-1}{T_s z}\right)$$

where T_s is the sampling period.

Like the continuous-time PID controller, the discrete-time controller is usually not implemented like this: the derivative term is not applied to the set-point, the proportional gain is reduced for the set-point, and the derivative action is filtered. The transfer function used for feedback is

$$K(z) = k_\mathrm{P}\left(1 + \frac{T_s}{T_\mathrm{I}(z-1)} + \frac{NT_\mathrm{D}}{T_\mathrm{D}+NT_s}\cdot\frac{z-1}{z-T_\mathrm{D}/(T_\mathrm{D}+NT_s)}\right)$$

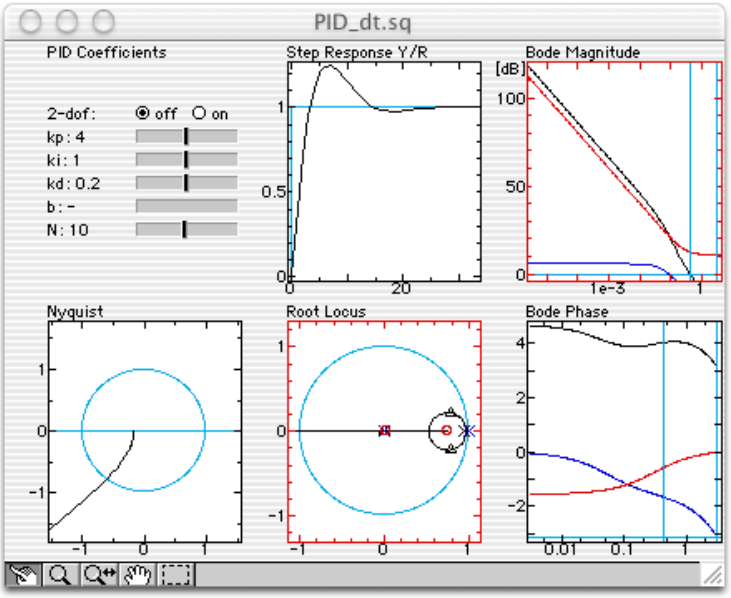

Figure 4.2 PID_dt.sq

Figures

The figures are the same as those defined for RST_dt.sq, except for the Open-Loop Zeros and Poles and the Closed-Loop Poles which are not defined.

Settings

The System (continuous-time model), System (discrete-time model), Sampling Period, and Damping Specification have the same effect as the corresponding menu entries defined in RST_dt.sq. Two new entries are defined.

PID Coefficients

The three parameters of the PID (k_P, T_I and T_D) can be edited in a dialog box. For P, PI, or PD controllers, set the parameter of the missing component to the empty matrix [].

No Derivator On Reference

When the input of the PID controller is the error between the set-point and the measured output, discontinuities of the set-point are differentiated by the derivator component of the PID and yield infinite values

(or very large values in the case of a discrete-time PID controller) for the control signal. To avoid that, the set-point is usually not differentiated. The control signal is

$$u(k) = k_\mathrm{P}\left(e(k) + \frac{T_s}{T_\mathrm{I}}\sum_{i=0}^{k} e(i) - \frac{T_\mathrm{D}}{T_s}\left(y(k) - y(k-1)\right)\right)$$

Display Frequency Line

When selected, moving the mouse above a frequency response (Bode or sensitivity) will display a corresponding line in other frequency responses, Nyquist diagrams, and zero/pole diagrams.

4.3 RST_ct.sq

Continuous-time two-degrees-of-freedom linear controller

RST controllers, or two-degrees-of-freedom linear single-input single-output controllers, are a more general form of linear controller than the popular PID controller. Their name comes from the three polynomials which characterize them. In addition to the feedback S/R, which permits to reduce the sensitivity to model uncertainties and to disturbances, the reference signal is filtered by T/R, which permits better tracking. The two degrees of freedom refer to the independence between those two filters (their common denominator does not constitute a constraint). While many linear single-input single-output controllers (including PID) can be expressed as RST, the SQ file RST_ct.sq preferred design method is the direct manipulation of closed-loop poles. Taking into account other quantities, such as step responses and sensitivities, enables easy and robust design.

First contact

When you open RST_ct.sq from Sysquake (menu File/Open), four figures are displayed: the closed-loop poles, the Bode magnitude, the step response, and the Nyquist diagram (see Fig. 4.3). They correspond to a first-order system and a second-order controller with a scalar feedforward calculated to remove steady-state error. You can move the closed-loop poles by dragging them with the mouse. You can also change the gain and the cut-off frequency of the feedback by dragging the Bode magnitude. Observe what happens when you drag the poles to the left of the imaginary axis: the closed-loop system becomes unstable, the step response becomes very large, and the Nyquist diagram crosses the critical point -1.

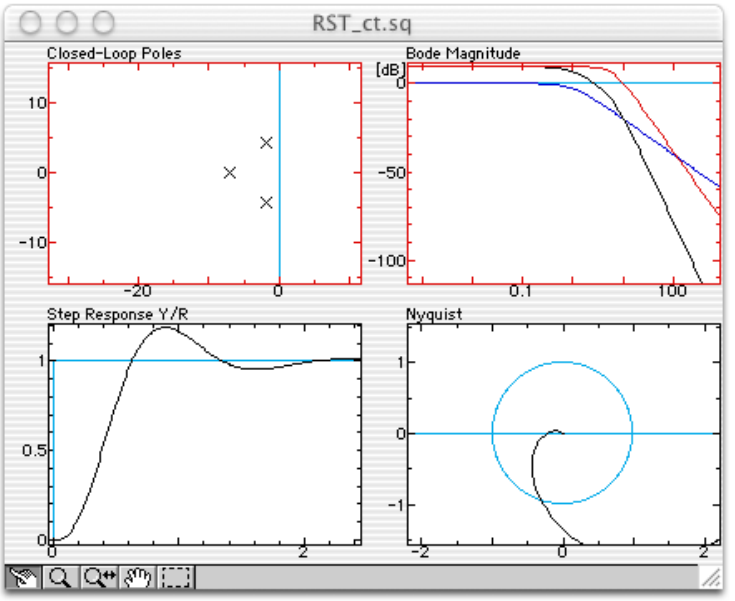

Figure 4.3 RST_ct.sq

Figures

Step Response Y/U

Open-loop step response, useful to get an idea of the dynamics of the system.

Impulse Response Y/U

Open-loop impulse response. Depending on the system and the preferences of the user, the impulse response may be better to represent the dynamics of the system. The presence of an integrator, for instance, may make the step response more difficult to understand.

Step Response Y/R

Tracking closed-loop step response. This step response shows important transient behavior of the controlled system, such as the overshoot, the rise time, the settling time. The tracking steady-state error (or lack of it) is also visible. The input/output stability is usually immediately visible, unless a very slow unstable mode is hidden by the limited range of time. Beware of potential internal unstable modes.

Step Response U/R

Tracking closed-loop step response between the reference signal and the system input. Risks of saturation, high-frequency modes (ringing),

and slow or unstable internal modes are clearly visible and complete the time-domain information obtained with the step response of Y/R.

Step Response Y/W

Input disturbance rejection step response. The disturbance step is added to the input of the system. This response may be very different from the tracking step response, especially with two-degrees-of-freedom controllers where the prefilter polynomial T(z) is tuned to cancel closed-loop poles or to make the tracking faster.

Step Response Y/D

Output disturbance rejection step response. The disturbance step is added to the output of the system. This response may be very different from the tracking step response, especially with two-degrees-of-freedom controllers where the prefilter polynomial T(z) is tuned to cancel closed-loop poles or to make the tracking faster.

Step Response U/D

Step response between an output disturbance and the system input.

Ramp Response Y/R

Tracking closed-loop ramp response. This response may be better suited to the study of transient behavior and permanent error than the step response if the real set-point changes with a fixed rate.

Ramp Response Y/D

Tracking closed-loop ramp response between a disturbance and the system output.

Bode Magnitude and Phase

Open-loop frequency response, displayed as functions of the frequency expressed in radians per time unit. The cross-over slope of the magnitude, and the low- and high-frequency open-loop gains give important insights about the robustness of the controller.

The Bode magnitude can be dragged up and down to change the gain of the controller.

Nyquist

Open-loop frequency response, displayed in the complex plane. The phase is expressed in radians. The gain and phase margins are clearly visible. With high-order systems and controllers, make sure that the system is stable by inspecting the closed-loop poles, the robustness margins (where the stability is explicitly checked) or at least a time-domain response.

Nichols

Logarithm of the frequency response, displayed in the complex plane. The phase is expressed in radians. The gain and phase margins are clearly visible.

The Nichols diagram can be dragged up and down to change the gain of the controller.

Sensitivity

Closed-loop frequency response between an output disturbance and the output. Only the amplitude is displayed, which is enough to give important information about the robustness of the design. Its supremum is the inverse of the modulus margin, which is defined as the distance between the Nyquist diagram and the critical point -1 in the complex plane. Peaks and large values of the sensitivity should be avoided. The sensitivity should be small at low frequency, to make the behavior of the system insensitive with respect to model uncertainties in the bandwidth.

Clicking in any sensitivity diagram highlights the corresponding frequency in all the sensitivity diagrams, the Nyquist diagram, the Nichols diagram, the Bode diagrams, and the open-loop and close-loop poles plots.

Complementary Sensitivity

Closed-loop frequency response between measurement noise and the output. Its name comes from the fact that the sum of the sensitivity and the complementary sensitivity is 1 for any frequency (however, this does not apply to their amplitude). In the case of a one-degree-of-freedom controller, the complementary sensitivity is also the frequency response between the set-point and the output. It should be close to 1 at low frequency, and small at high frequency.

Clicking in any sensitivity diagram highlights the corresponding frequency in all the sensitivity diagrams, the Nyquist diagram, the Nichols diagram, the Bode diagrams, and the open-loop and close-loop poles plots.

Perturbation-Input Sensitivity

Closed-loop frequency response between output disturbance and the system input. Small values at high frequency reduce the excitation of the actuators in presence of measurement noise.

Clicking in any sensitivity diagram highlights the corresponding frequency in all the sensitivity diagrams, the Nyquist diagram, the Nichols diagram, the Bode diagrams, and the open-loop and close-loop poles plots.

Open-Loop Zeros/Poles

All the open-loop zeros and poles are represented. The zeros and poles of the system are represented by black circles and crosses, respectively. The zeros and poles of the free part of the feedback are represented by red circles and crosses; the zeros and poles of the fixed part of the feedback are represented by green circles and crosses; the fixed part of the feedforward polynomial is represented by green squares. All the zeros and poles of the controller can be manipulated with the mouse. The system cannot be changed. As an help to cancel some of the dynamic of the closed-loop system with the feedforward zeros, the closed-loop poles are displayed as magenta (pink) dots.

Closed-Loop Poles

The closed-loop poles are displayed as black crosses. If there are as many closed-loop poles as free coefficients in the feedback, they can be moved; a new controller is calculated by pole placement.

Root Locus

The root locus is the locus of the closed-loop poles when the gain of the feedback is a positive real number. The zeros and poles of the feedback are preserved. The open-loop zeros and poles are represented by black circles and crosses for the system, and red circles and crosses for the feedback. Feedback zeros and poles can be dragged to change the controller. The closed-loop poles are represented by triangles. They can be moved on the root locus to change the feedback gain. If they move beyond open-loop zeros and poles, the sign of the feedback changes, and the root locus is inverted.

Robustness Margins

The gain margin (in dB) and phase margin (in degrees) are displayed with the corresponding frequencies (in radians per time unit). For unstable open-loop systems, the gain margin can be negative and is a lower stability limit for the feedback gain. If the closed-loop system is unstable, no margin is displayed. If the open-loop gain is smaller or larger than 1 at all frequencies, the phase margin is not displayed.

Discrete-Time Step Resp. Y/R and Y/D

Comparison between the step responses of the closed-loop system with an analog controller (in light blue) and with a digital controller (in black). The sampling period can be set by choosing "Sampling Period" in the Settings menu, or adjusted interactively in the figure "Nyquist Frequency" (see below).

Nyquist Frequency

Once an analog RST controller has been designed, it is possible to choose a sampling frequency for a digital implementation. Then the dynamic behavior of the closed-loop system will differ from the initial design. The figure "Nyquist Frequency" displays a Bode diagram of various transfer functions: the continuous-time system is displayed in blue, the continuous-time open-loop response in black, and the discrete-time response of the sampled system (obtained with a zero-order hold) in red. The Nyquist frequency is displayed as a vertical line in red, and can be manipulated interactively. If the closed-loop system is stable, the cross-over frequency is displayed in light blue. Typically, the Nyquist frequency should be 5-10 times larger.

Settings

System

A continuous-time model can be given as two row vectors which contain the numerator and denominator of a transfer function. Multiple models can be provided; each model corresponds to a row.

Feedback Coefficients

The coefficients of the numerator and denominator of the feedback are given as two row vectors. If they are not factors of the feedback fixed parts (see below), the user is asked whether he wants to modify them.

Feedback Fixed Parts

The coefficients of the fixed parts of the numerator and denominator of the feedback are given as two row vectors. The fixed parts can be used to impose some poles and zeros in the feedback (for instance an integrator with [1,0] in the denominator); they are enforced during pole placement. The gain is meaningless; only the zeros are used. When the fixed parts are changed, a new controller is computed such that the closed-loop poles are preserved. If this is not possible because there are not enough closed-loop poles to permit pole placement, the variable part of the controller is preserved. If the resulting controller (product of fixed and variable parts) is non-causal, fast poles are added.

Two DOFs

The *Two DOFs* setting is a binary value which enables an arbitrary feedforward polynomial. Otherwise, the feedforward is set to the same value as the feedback numerator; this means that the error between

the system output and the set-point is used as a whole to compute the system input. When Two DOFs is enabled, the feedforward contains the zeros of its fixed part (see below), and its gain is calculated to have a unit gain between the set-point and the system output.

Feedforward Fixed Part

The feedforward fixed part is given as a row vector. It provides all the zeros of the feedforward; its gain is ignored. The *Feedforward Fixed Part* setting is enabled only for two-degrees-of-freedom controllers.

Characteristic Polynomial

The controller can be calculated by specifying directly the characteristic polynomial, i.e. the denominator of all the closed-loop transfer functions which can be defined, whose roots are the closed-loop poles. To enter the closed-loop poles, use the `poly` function (e.g. `poly([-0.8,-1.3+0.3j,-1.3-0.3j])`).

In order to obtain a solution for any value of the coefficients, the degree of the characteristic polynomial must be larger than or equal to `2 deg A + deg Rf + deg Sf - 1`, where `A`, `Rf` and `Sf` are respectively the system denominator, the fixed part of the feedback denominator and the fixed part of its numerator. This lower limit is displayed in the dialog box. There is no upper limit (from a mathematical point of view).

Sampling Period

The sampling period is given as a positive pure number. It is used for the discrete-time step response to show the difference between the responses with purely continuous-time elements and a digital implementation with a zero-order hold D/A converter.

Bilinear/Back Rect/For Rect Method

Method used for converting the controller from continuous time to discrete time. Usually, the bilinear method is the best one and permits lower sampling frequencies.

Damping Specifications

Absolute and relative damping can be specified; they are represented in the complex plane of the closed-loop poles and the root locus by red lines. For a stable system, the absolute damping is the time constant of the envelope of the slowest mode; the relative damping is the absolute damping divided by the oscillation time.

Display Frequency Line
When selected, moving the mouse above a frequency response (Bode or sensitivity) will display a corresponding line in other frequency responses, Nyquist diagrams, and zero/pole diagrams.

4.4 RST_dt.sq

Discrete-time two-degrees-of-freedom linear controller
RST_dt.sq implements the basic tools for the design of classical controllers for linear SISO discrete-time systems. As a useful extension to design more robust controllers, the system can be modeled by several transfer functions. Even if only one nominal model is used, a very robust controller may be easily obtained with pole placement if the sensitivity functions are taken into account during the design.

The user can provide either a continuous-time model or a discrete-time model. In the second case, the discrete-time model is used as is; the sampling period is used only to scale the times and the frequencies on the display. In the first case, the continuous-time model is sampled with a zero-order hold at the sampling frequency. The continuous-time model is used for the hybrid time responses "Continuous-Time Step Resp. Y/R" and "Continuous-Time Step Resp. Y/D", which better reflects the reality, especially when the sampling period is large with respect to the system; the curve displayed represents the output of the continuous-time system when the controller is digital and interfaced with the system through a zero-order hold and a sampler.

First contact

When you open RST_dt.sq from Sysquake (menu File/Open), four figures are displayed: the closed-loop poles, the Bode magnitude, the step response, and the Nyquist diagram (see Fig. 4.4). They correspond to a first-order system and a second-order controller with a scalar feedforward calculated to remove steady-state error. You can move the closed-loop poles by dragging them with the mouse. You can also change the gain of the feedback by dragging the Bode magnitude up and down. Observe what happens when you drag the poles outside the unit circle: the closed-loop system becomes unstable, the step response becomes very large, and the Nyquist diagram crosses the critical point -1.

How to use it

Many standard graphics are implemented. Among the possible designs, the easiest to use is pole placement, where the closed-loop

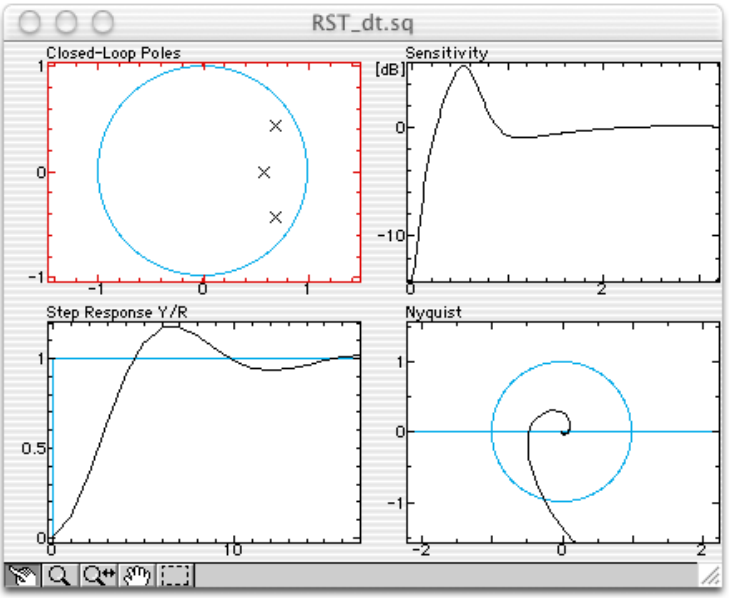

Figure 4.4 RST_dt.sq

poles are moved into the desired region inside the unit circle, the stability region which plays the same role for discrete-time systems as the left half-plane for continuous-time systems. Caution must be exercised about the robustness of the controller, because pole placement itself can lead to extremely unrobust designs. Fortunately, by observing simultaneously robustness indicators such as the sensitivity function, the Nyquist diagram or robustness margins, it is easy to avoid this pitfall.

Figures

Step Response Y/U

Open-loop step response, useful to get an idea of the dynamics of the system.

Impulse Response Y/U

Open-loop impulse response. Depending on the system and the preferences of the user, the impulse response may be better to represent the dynamics of the system. The presence of an integrator, for instance, may make the step response more difficult to understand.

Step Response Y/R

Tracking closed-loop step response. This step response shows important transient effects of the controlled system, such as the overshoot, the rise time, the settling time. The tracking steady-state error (or lack of it) is also visible. The input/output stability is usually immediately visible, unless a very slow unstable mode is hidden by the limited range of time. Beware of potential internal unstable modes.

Step Response U/R

Tracking closed-loop step response between the reference signal and the system input. Risks of saturation, high-frequency modes (ringing), and slow or unstable internal modes are clearly visible and complete the time-domain information obtained with the step response of Y/R.

Step Response Y/D

Disturbance rejection step response. The disturbance step is applied to the output of the system. This response may be very different from the tracking step response, especially with two-degrees-of-freedom controllers where the prefilter polynomial T(z) is tuned to cancel closed-loop poles or to make the tracking faster.

Step Response U/D

Step response between a disturbance and the system input.

Ramp Response Y/R

Tracking closed-loop ramp response. This response may be better suited to the study of transient effects and permanent error than the step response if the real set-point changes with a fixed rate.

Ramp Response Y/D

Tracking closed-loop ramp response between a disturbance and the system output.

Continuous-Time Step Response Y/R and Y/D

If a continuous-time model of the system is provided, what happens between the samples can be displayed by the continuous-time step responses. The continuous-time system input is obtained by converting the discrete-time samples with zero-order hold. The discrete-time step responses representing the output is displayed with straight lines joining each samples. This approximates correctly the continuous-time responses if the sampling period is small enough with respect to the dynamics of the open-loop system. Otherwise, the continuous-time responses should be used.

Bode Magnitude and Phase

Open-loop frequency response, displayed as functions of the frequency expressed in radians per time unit. The cross-over slope of the magnitude, and the low- and high-frequency open-loop gains give important insights about the robustness of the controller.

The Bode magnitude can be dragged up and down to change the gain of the controller.

Nyquist

Open-loop frequency response, displayed in the complex plane. The phase is expressed in radians. The gain and phase margins are clearly visible. With high-order systems and controllers, make sure that the system is stable by inspecting the closed-loop poles, the robustness margins (where the stability is explicitly checked) or at least a time-domain response.

Nichols

Logarithm of the frequency response, displayed in the complex plane. The phase is expressed in radians. The gain and phase margins are clearly visible.

The Nichols diagram can be dragged up and down to change the gain of the controller.

Sensitivity

Closed-loop frequency response between an output disturbance and the output. Only the amplitude is displayed, which is enough to give important information about the robustness of the design. Its supremum is the inverse of the modulus margin, which is defined as the distance between the Nyquist diagram and the critical point -1 in the complex plane. Peaks and large values of the sensitivity should be avoided. The sensitivity should be small at low frequency, to make the behavior of the system insensitive with respect to model uncertainties in the bandwidth.

Complementary Sensitivity

Closed-loop frequency response between measurement noise and the output. Its name comes from the fact that the sum of the sensitivity and the complementary sensitivity is 1 for any frequency (however, this does not apply to their amplitude). In the case of a one-degree-of-freedom controller, the complementary sensitivity is also the frequency response between the set-point and the output. It should be close to 1 at low frequency, and small at high frequency.

Perturbation-Input Sensitivity

Closed-loop frequency response between output disturbance and the system input. Small values at high frequency reduce the excitation of the actuators in presence of measurement noise.

Open-Loop Zeros/Poles

All the open-loop zeros and poles are represented. The zeros and poles of the system are represented by black circles and crosses, respectively. The zeros and poles of the free part of the feedback are represented by red circles and crosses; the zeros and poles of the fixed part of the feedback are represented by green circles and crosses; the fixed part of the feedforward polynomial is represented by green squares. All the zeros and poles of the controller can be manipulated with the mouse. The system cannot be changed. As an help to cancel some of the dynamic of the closed-loop system with the feedforward zeros, the closed-loop poles are displayed as magenta (pink) dots.

Closed-Loop Poles

The closed-loop poles are displayed as black crosses. If there are as many closed-loop poles as free coefficients in the feedback, they can be moved; a new controller is calculated by pole placement.

Root Locus

The root locus is the locus of the closed-loop poles when the gain of the feedback is a positive real number. The zeros and poles of the feedback are preserved. The open-loop zeros and poles are represented by black circles and crosses for the system, and red circles and crosses for the feedback. Feedback zeros and poles can be dragged to change the controller. The closed-loop poles are represented by triangles. They can be moved on the root locus to change the feedback gain. If they move beyond open-loop zeros and poles, the sign of the feedback changes, and the root locus is inverted.

Robustness Margins

The gain margin (in dB) and phase margin (in degrees) are displayed with the corresponding frequencies (in radians per time unit). For unstable open-loop systems, the gain margin can be negative and is a lower stability limit for the feedback gain. If the closed-loop system is unstable, no margin is displayed. If the open-loop gain is smaller or larger than 1 at all frequencies, the phase margin is not displayed.

Settings

System (Continuous-Time Model)

A continuous-time model can be given as two row vectors which contain the numerator and denominator of a transfer function. Multiple models can be provided; each model corresponds to a row. The continuous-time model is converted to a discrete-time model sampled at the sampling period given in the setting *Sampling Period* using a zero-order hold. The continuous-time model is used only by the continuous-time step responses; all the other figures are based on the discrete-time models.

System (Discrete-Time Model)

A discrete-time model is given as two row vectors which contain the numerator and denominator of a transfer function. Multiple models can be provided; each model corresponds to a row. If a discrete-time model is provided, the continuous-time model is ignored and the continuous-time step responses cannot be displayed.

Sampling Period

The sampling period is given as a positive pure number. It is used for the scale of time- and frequency-domain responses. The unit is implicit and could be seconds or anything else (minutes, hours, days, etc.)

Feedback Coefficients

The coefficients of the numerator and denominator of the feedback are given as two row vectors. If they are not factors of the feedback fixed parts (see below), the user is asked whether he wants to modify them.

Feedback Fixed Parts

The coefficients of the fixed parts of the numerator and denominator of the feedback are given as two row vectors. The fixed parts can be used to impose some poles and zeros in the feedback (for instance an integrator with [1,-1] in the denominator); they are enforced during pole placement. The gain is meaningless; only the zeros are used. When the fixed parts are changed, a new controller is computed such that the closed-loop poles are preserved. If this is not possible because there are not enough closed-loop poles to permit pole placement, the variable part of the controller is preserved. If the resulting controller (product of fixed and variable parts) is non-causal, it is delayed with additional poles at 0.

Two DOFs

The *Two DOFs* setting is a binary value which enables an arbitrary feedforward polynomial. Otherwise, the feedforward is set to the same value as the feedback numerator; this means that the error between the system output and the set-point is used as a whole to compute the system input. When Two DOFs is enabled, the feedforward contains the zeros of its fixed part (see below), and its gain is calculated to have a unit gain between the set-point and the system output.

Feedforward Fixed Part

The feedforward fixed part is given as a row vector. It provides all the zeros of the feedforward; its gain is ignored. The *Feedforward Fixed Part* setting is enabled only for two-degrees-of-freedom controllers.

Characteristic Polynomial

The controller can be calculated by specifying directly the characteristic polynomial, i.e. the denominator of all the closed-loop transfer functions which can be defined, whose roots are the closed-loop poles. To enter the closed-loop poles, use the `poly` function (e.g. `poly([0.8,0.6+0.3j,0.6-0.3j])`).

In order to obtain a solution for any value of the coefficients, the degree of the characteristic polynomial must be larger than or equal to `2 deg A + deg Rf + deg Sf - 1`, where `A`, `Rf` and `Sf` are respectively the system denominator, the fixed part of the feedback denominator and the fixed part of the numerator. This lower limit is displayed in the dialog box. There is no upper limit (from a mathematical point of view).

Damping Specifications

Absolute and relative damping can be specified; they are represented in the complex plane of the closed-loop poles and the root locus by red lines. For a stable system, the absolute damping is the absolute value of the slowest pole; the relative damping is the absolute damping divided by the oscillation time expressed in sampling periods.

Display Frequency Line

When selected, moving the mouse above a frequency response (Bode or sensitivity) will display a corresponding line in other frequency responses, Nyquist diagrams, and zero/pole diagrams.

4.5 LQR_ct.sq

Continuous-time linear-quadratic regulator

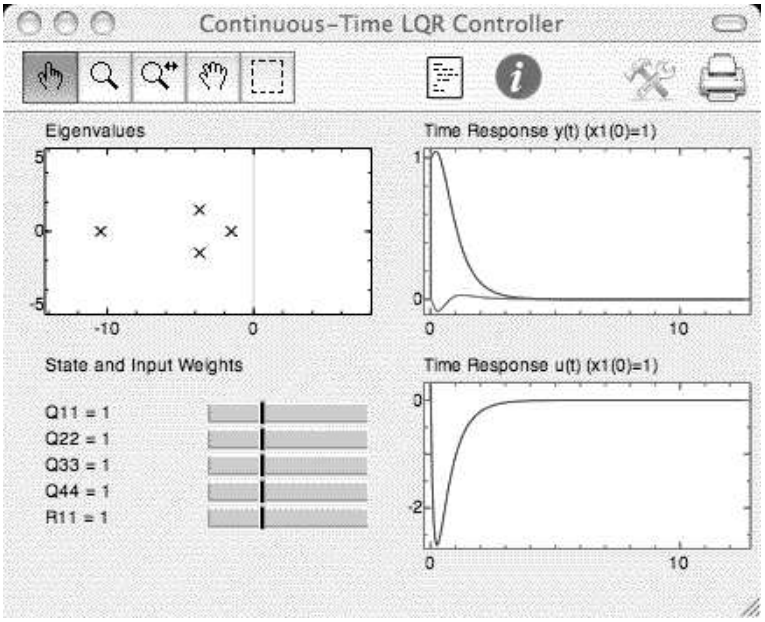

Figure 4.5 LQR_ct.sq

For linear (or linearized) systems described by a state-space model, state-feedback controllers can be designed by minimizing a quadratic cost function which takes into account the error and the input. The problem with arbitrary fixed weights can be written as an algebraic Riccati equation; the function `care` gives its solution if it exists.

First contact

When you open LQR_ct.sq from Sysquake (menu File/Open), four figures are displayed: the closed-loop eigenvalues, the weights on the state and the input as sliders, and the time responses of the output and input of the controlled system with one of the state initial values set to 1 and the others to 0 (see Fig. 4.5). You can change the weights with the mouse. To change the state whose initial value is 1, double-click its figure; a dialog box will be displayed where you can enter the state number. The state number is associated to the figure; you can display different responses if you add other time response figures.

Figures

State and Input Weights

The diagonal elements of the state and input weights are displayed as sliders. You can change them with the mouse. Non-diagonal weights

are zero.

Time Response to Initial Condition y(t)

Output time response of the controlled system. The initial value of all states is zero, except for one of them which is 1. You can change which one is 1 by double-clicking the figure.

Time Response to Initial Condition u(t)

Input time response of the controlled system. The initial value of all states is zero, except for one of them which is 1. You can change which one is 1 by double-clicking the figure.

Sensitivity

Closed-loop frequency response between a state disturbance and the output, as a singular value plot. The singular value plot is the equivalent of the Bode diagram for single-input single-output systems.

Settings

Model

A continuous-time model can be given as four matrices A, B, C, and D:

$$\begin{aligned} sX(s) &= AX(s) + BU(s) \\ Y(s) &= CX(s) + DU(s) \end{aligned}$$

Display Frequency Line

When selected, moving the mouse above a sensitivity plot will display a corresponding line in other sensitivity and eigenvalue plots.

4.6 filter.sq

Design of analog and digital filters

Different filters, defined in continuous time or discrete time. Low-pass, high-pass, band-pass and band-stop can be adjusted interactively in the frequency magnitude diagram.

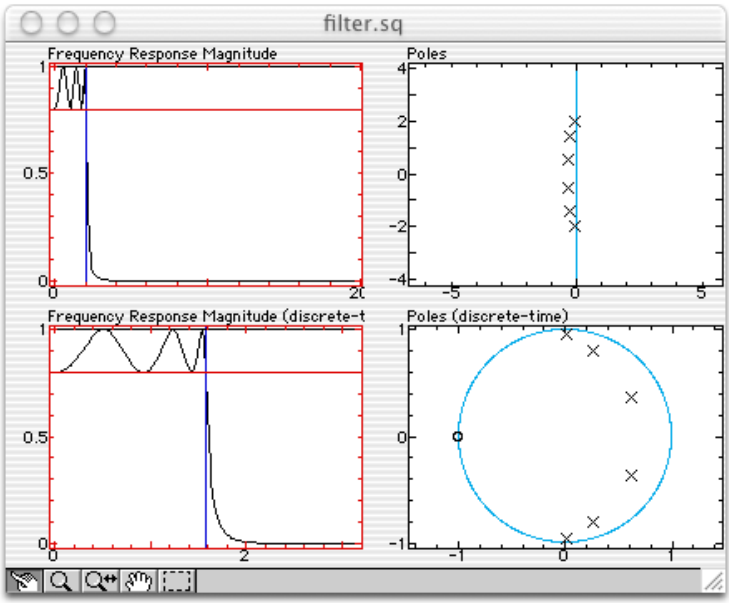

Figure 4.6 filter.sq

First contact

A two-by-two array of figures is displayed (see Fig. 4.6). The top row represents a continuous-time filter, while the bottom row represents the same filter converted with the bilinear transform. The left column shows the magnitude of the frequency response of the filter; the right column shows the zeros (as circles) and the poles (as crosses) of the filters in the complex plane of the Laplace transform for the continuous-time filter and of the z transform for the discrete-time filter. Initially, the filter is a Chebyshev filter, whose bandwidth (vertical blue line) and bandwidth ripples (horizontal red line) can be manipulated.

Figures

Frequency Response Magnitude

The magnitude of the frequency response of the continuous-time filter is displayed in black. The limits of the bandwidth (lower and/or higher, depending on the kind of filter) are displayed as vertical blue lines and can be manipulated; if the Shift key is held down, both lines are moved together to keep their ratio constant. If the upper limit is moved to the left of the lower limit, or if the lower limit is moved to the right of the upper limit, bandwidth filters become bandstop and vice-versa. For

Chebyshev filters, the lower or upper limit of the ripples is displayed as a horizontal red line and can be manipulated.

Frequency Response Phase

The phase of the frequency response of the continuous-time filter is displayed in black.

Poles

In the complex plane of the Laplace transform, the poles of the continuous-time filter are displayed as crosses, and the zeros as circles.

Frequency Response Magnitude (discrete-time)

The magnitude of the frequency response of the discrete-time filter is displayed in black. The limits of the bandwidth (lower and/or higher, depending on the kind of filter) are displayed as vertical blue lines and can be manipulated. For Chebyshev filters, the lower or upper limit of the ripples is displayed as a horizontal red line and can be manipulated.

Frequency Response Phase (discrete-time)

The phase of the frequency response of the discrete-time filter is displayed in black.

Poles (discrete-time)

In the complex plane of the z transform, the poles of the discrete-time filter are displayed as crosses, and the zeros as circles.

Settings

Kind of filter

The kind of filter can be chosen between Butterworth, Chebyshev (ripples of the magnitude in the bandwidth), or Inverse Chebyshev (ripples of the magnitude outside the bandwidth).

Lowpass Filter

High frequencies are filtered out.

Highpass Filter

Low frequencies are filtered out.

Bandpass Filter
Low and high frequencies are filtered out, leaving medium frequencies.

Bandstop Filter
Medium frequencies are filtered out, leaving low and high frequencies.

Filter Order
The order of the filter can be entered in a dialog box. Note that large orders (>10) often result in inaccurate figures, because of numeric problems.

Transition Frequencies
The lower and upper limits of the bandwidth can be entered in a dialog box as a vector of two elements. If the lower limit is 0, the filter is low-pass; if the upper limit is `inf`, the filter is high-pass. If the lower limit is larger than the upper limit, the filter is bandstop.

Sampling Period
The sampling period can be entered in a dialog box.

4.7 id_par.sq

Parametric identification

First contact

SQ file id_p.sq provides the identification methods to obtain an ARX parametric model based on the measured input and output of an unknown system (see Fig. 4.7). The data can be retrieved from a file (typically created by an external real-time acquisition program) or generated by the SQ file.

For didactic purposes, synthetic data can be created for input $u(t)$ and output $y(k)$. Data are obtained by simulating system $G(s) = 1/(s^2 + 2s + 3)$ sampled at $T_s = 0.1$ with a square input and noise filtered by the model denominator (ARX model). For applications to real systems, experimental data can be read from files.

The parameters of the following model are identified:

$$y(t) = q^{-d}\,\frac{B(q^{-1})}{A(q^{-1})}\,u(t) + \frac{1}{A(q^{-1})}\,n(t)$$

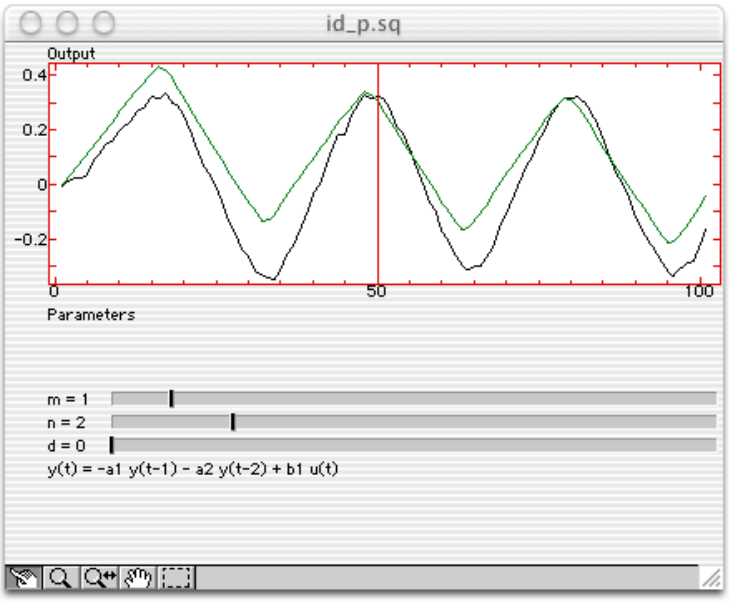

Figure 4.7 id_p.sq

where $A(q^{-1}) = 1 + a_1 q^{-1} + ... + a_n q^{-n}$ and $B(q^{-1}) = b_0 + b_1 q^{-1} + ... + b_{m-1} q^{-m+1}$. The order of polynomials m and n, and the delay d, must be specified.

Settings

Create Synthetic Data

Synthetic data are created from scratch. A dialog box allows to choose the number of samples.

Read Data File

The measurements are read from a text file, typically created by an acquisition program. This file should contain an array of two column (separated by spaces or tabulators) by n row (separated by carriage returns and/or line feeds). The first column corresponds to the system input, and the second column to the system output. Each row corresponds to a sample.

Model Degree

Samples Used For Identification

Not all samples are used for identifications. Remaining samples are useful for validating the model (cross-validation). The number of sam-

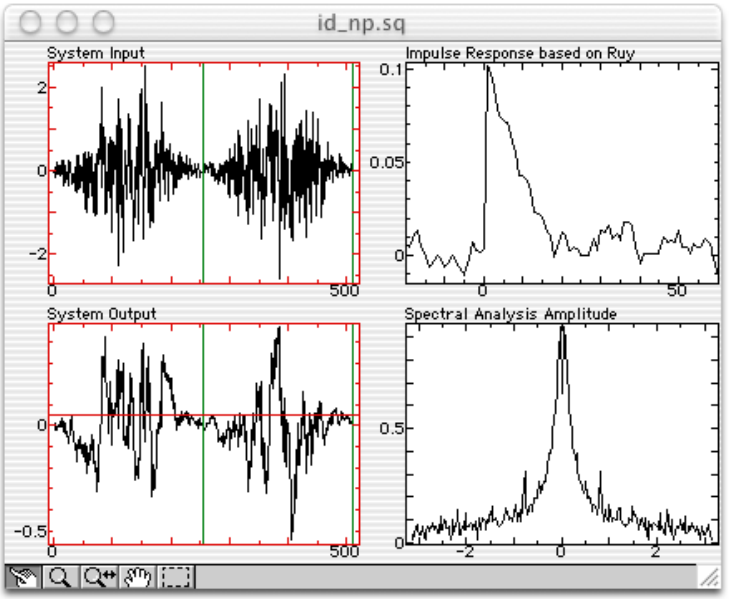

Figure 4.8 id_np.sq

ples used for identification can also be set by manipulating the red vertical line in figure "Output".

4.8 id_npar.sq

Non-parametric identification

First contact

SQ file id_np.sq provides the identification methods to obtain a non-parametric model based on the measured input and output of an unknown system (see Fig. 4.8). Identification can be performed in the time domain or the frequency domain. The data can be retrieved from a file (typically created by an external real-time acquisition program) or generated by the SQ file.

For didactic purposes, synthetic data can be created for input $u(k)$ and output $y(k)$. The input is either white noise (each sample is a pseudo-random number chosen from a normal distribution), a pseudo-random binary sequence (each sample is 1 or -1 with a probability of 0.5), or a pseudo-random binary sequence where the probability to switch at each sample is 0.2 or 0.05 (colored noise). The output is $y(k) = g(k) \star u(k) + n(k)$, where $g(k)$ is the impulse response of a

transfer function specified by the user ($0.1/(z-0.9)$ by default) and $n(k)$ some white noise whose level can be adjusted.

Time-domain identification with correlation analysis

The covariance of the input R_{uu} and the cross covariance between the input and the output R_{uy} are calculated. If the input is white, R_{uy} is an approximation of the impulse response of the system, times a scalar factor. The input can be whitened with a finite-impulse response (FIR) filter.

Frequency-domain identification with spectral analysis

The spectrum of the system can be approximated either by dividing the output's discrete Fourier transform (FFT) by the input's, or (better if the output is disturbed by noise) by dividing R_{uy}'s FFT (cross spectrum) by R_{uu}'s (input's spectrum). The data can be split, so that the average of the FFT of each block of data is used. A time window is applied to each block to reduce the effect of the finite number of samples.

Remark: the splitting in n sequences is used only by the spectral analysis. The time window is used for all frequency-domain analysis methods, while the whitening filter is used only for correlation analysis.

Settings

System

The system used to create synthetic sampled data is given as the numerator and denominator of a discrete-time transfer function in positive powers of z.

Number of samples

Total number of samples which should be created.

White Noise or Pseudo-Random Binary Sequence

For synthetic data, the system input can be chosen among white noise, where each sample is the result of a normally distributed pseudo-random generator, or a pseudo-random binary sequence where the probability to switch the level at each sample is 50%, 20% or 5%. With 20% or 5%, the signal is significantly different from white noise.

Whitening Filter

For the correlation analysis, a whitening filter should be used if the system input is significantly different from white noise. The whitening filter is a finite-impulse response (FIR) filter whose inverse is the auto-regressive (AR) model which gives the non-white input. The whitening filter is not used for frequency-domain identification.

Rectangular/Triangular/Hann/Hamming Window

To reduce the effect of the finite number of samples (aliasing), a non-rectangular window can be applied to the input and output samples. The Hann and Hamming windows have a sinusoidal shape. The windows are not used for time-domain identification.

Multiple Sequences

For frequency-domain identification, it may be better to split the available data and use the different sequences to reduce the variance of the estimation. The price to pay is the lower resolution of the estimate. The number of sequences is set by moving the green vertical lines in the input or output figure.

Read Data File

The measurements are read from a text file, typically created by an acquisition program. This file should contain an array of two column (separated by spaces or tabulators) by n row (separated by carriage returns and/or line feeds). The first column corresponds to the system input, and the second column to the system output. Each row corresponds to a sample.

Chapter 5

Introduction to LME

This chapter describes the command-line interface, where you can type expressions and commands expressed in LME, the language of Sysquake, and observe the results immediately.

In Sysquake, the Command window offers an alternate user interface which complements interactive graphics programmed with SQ files. Here are some of the tasks you can perform:

- evaluate expressions;
- store intermediate results in variables;
- call functions defined in libraries or SQ files to debug them, or simply check their syntax;
- have a quick access to online help.

You can type commands, cut to or paste from the clipboard after the prompt, and copy from anywhere. When you hit the Return key, everything after the last prompt is interpreted. The following keys have a special meaning:

Return Interprets everything from the last prompt.

Shift-Return (Option-Return on macOS) Line break (e.g. for i=1:10<shift-return> i<shift-return> end<return> to display the integer numbers from 1 to 10)

Esc Clears the current command.

Up Retrieves the previous command.

Down Retrieves the next command.

Commands you can type in the Command window or panel include:

- Simple expressions. The result is displayed unless they end with a semicolon. It is assigned automatically to variable ans (answer), which lets you reuse it in the next expression.
- Assignments to variables (new variables are created as required, and are local to the command-line interface).
- Complete loops and conditional constructs.
- Calls to user functions.

The commands you can type are described in the chapter LME Reference (LME is the name of the language used by Sysquake). You can also type expressions; LME evaluates them and displays their result unless you add a semicolon. When you evaluate an expression, its result is assigned to the variable ans, so that you can reuse it easily. Here is a simple example which displays four times the sine of three radians, and then adds 5 to the result (type what is in bold; the plain characters correspond to what Sysquake displays itself when you hit the Return key):

```
> 4*sin(3)
 ans =
  0.5645
> ans+5
 ans =
  5.5645
```

Calls to graphical functions are permitted. On platforms where a single graphics window is displayed, if an SQ file is already loaded, you can clear the figures first with the `clf` command:

```
> clf
> plot(sin(0:0.1:2*pi))
```

Functions and constants are usually defined in SQ files or libraries. You can also do it directly in the command-line interface:

```
> define g = 9.81;
> t = 3;
> g * t^2
 ans =
  88.29
```

You must type the whole definition before evaluating it with the Return key. Separate the statements either with semicolons or with Shift-Return (Option-Return on macOS).

```
> function r=range(x); r=max(x)-min(x);
> range(1:10)
 ans =
  9
```

Functions defined in the command-line interface are in the scope of library `_cli`:

```
> which range
 ans =
  _cli/range
```

If you import the definitions in library `stat`, your definition of `range` will be hidden:

```
> use stat
> which range
 ans =
  stat/range
```

Sysquake always use the definition in the library which was imported the most recently. The order can be checked with `info u`:

```
> info u
  _cli
  stat
```

To let Sysquake search in `_cli` before `stat`, type `use _cli`:

```
> use _cli
> info u
  stat
  _cli
```

This chapter introduces LME(TM) (Lightweight Math Engine), the interpreter for numeric computing used by Sysquake, and shows you how to perform basic computations. It supposes you can type commands to a command-line interface. You are invited to type the examples as you read this tutorial and to experiment on your own. For a more systematic description of LME, please consult the LME Reference chapter.

In the examples below, we assume that LME displays a prompt >. This is not the case for all applications. You should never type it yourself. Enter what follows the prompt on the same line, hit the Return key (or tap the Eval or Execute button), and observe the result.

5.1 Simple operations

LME interprets what you type at the command prompt and displays the result unless you end the command with a semicolon. Simple expressions follow the syntactic rules of many programming languages.

```
> 2+3*4
ans =
```

```
 14
> 2+3/4
ans =
 2.75
```

As you can see, the evaluation order follows the usual rules which state that the multiplication (denoted with a star) and division (slash) have a higher priority than the addition and subtraction. You can change this order with parenthesis:

```
> (2+3)*4
ans =
 20
```

The result of expressions is automatically assigned to variable `ans` (more about variables later), which you can reuse in the next expression:

```
> 3*ans
ans =
 60
```

Power is represented by the ^ symbol:

```
> 2^5
ans =
 32
```

LME has many mathematical functions. Trigonometric functions assume that angles are expressed in radians, and `sqrt` denotes the square root.

```
> sin(pi/4) * sqrt(2)
ans =
 1
```

5.2 Complex Numbers

In many computer languages, the square root is defined only for non-negative arguments. However, it is extremely useful to extend the set of numbers to remove this limitation. One defines i such that $i^2 = -1$, and applies all the usual algebraic rules. For instance, $\sqrt{-1} = \sqrt{i^2} = i$, and $\sqrt{-4} = \sqrt{4}\sqrt{-1} = 2i$. Complex numbers of the form $a + bi$ are the sum of a real part a and an imaginary part b. It should be mentioned that i, the symbol used by mathematicians, is called j by engineers. LME accepts both symbols as input, but it always writes it `j`. You can use it like any function, or stick an `i` or `j` after a number:

```
> 2+3*j
ans =
 2+3j
> 3j+2
ans =
 2+3j
```

Many functions accept complex numbers as argument, and return a complex result when the input requires it even if it is real:

```
> sqrt(-2)
ans =
 0+1.4142i
> exp(3+2j)
ans =
 -8.3585+18.2637j
> log(-8.3585+18.2637j)
ans =
 3+2j
```

To get the real or imaginary part of a complex number, use the functions `real` or `imag`, respectively:

```
> real(2+3j)
ans =
 2
> imag(2+3j)
ans =
 3
```

Complex numbers can be seen as vectors in a plane. Then addition and subtraction of complex numbers correspond to the same operations applied to the vectors. The absolute value of a complex number, also called its magnitude, is the length of the vector:

```
> abs(3+4j)
ans =
 5
> sqrt(3^2+4^2)
ans =
 5
```

The argument of a complex number is the angle between the x axis ("real axis") and the vector, counterclockwise. It is calculated by the `angle` function.

```
> angle(2+3j)
ans =
 0.9828
```

The last function specific to complex numbers we will mention here is `conj`, which calculates the conjugate of a complex number. The conjugate is simply the original number where the sign of the imaginary part is changed.

```
> conj(2+3j)
ans =
 2-3j
```

Real numbers are also complex numbers, with a null imaginary part; hence

```
> abs(3)
ans =
 3
> conj(3)
ans =
 3
> angle(3)
ans =
 0
> angle(-3)
ans =
 3.1416
```

5.3 Vectors and Matrices

LME manipulates vectors and matrices as easily as scalars. To define a matrix, enclose its contents in square brackets and use commas to separate elements on the same row and semicolons to separate the rows themselves:

```
> [1,2;5,3]
ans =
 1 2
 5 3
```

Column vectors are matrices with one column, and row vectors are matrices with one row. You can also use the colon operator to build a row vector by specifying the start and end values, and optionally the step value. Note that the end value is included only if the range is a multiple of the step. Negative steps are allowed.

```
> 1:5
ans =
 1 2 3 4 5
> 0:0.2:1
ans =
```

```
 0 0.2 0.4 0.6 0.8 1
> 0:-0.3:1
ans =
 0 -0.3 -0.6 -0.9
```

There are functions to create special matrices. The `zeros`, `ones`, `rand`, and `randn` functions create matrices full of zeros, ones, random numbers uniformly distributed between 0 and 1, and random numbers normally distributed with a mean of 0 and a standard deviation of 1, respectively. The `eye` function creates an identity matrix, i.e. a matrix with ones on the main diagonal and zeros elsewhere. All of these functions can take one scalar argument n to create a square n-by-n matrix, or two arguments m and n to create an m-by-n matrix.

```
> zeros(3)
ans =
 0 0 0
 0 0 0
 0 0 0
> ones(2,3)
ans =
 1 1 1
 1 1 1
> rand(2)
ans =
 0.1386 0.9274
 0.3912 0.8219
> randn(2)
ans =
 0.2931  1.2931
 -2.3011 0.9841
> eye(3)
ans =
 1 0 0
 0 1 0
 0 0 1
> eye(2,3)
ans =
 1 0 0
 0 1 0
```

You can use most scalar functions with matrices; functions are applied to each element.

```
> sin([1;2])
ans =
 0.8415
 0.9093
```

There are also functions which are specific to matrices. For example, `det` calculates the determinant of a square matrix:

```
> det([1,2;5,3])
ans =
 -7
```

Arithmetic operations can also be applied to matrices, with their usual mathematical behavior. Additions and subtractions are performed on each element. The multiplication symbol ∗ is used for the product of two matrices or a scalar and a matrix.

```
> [1,2;3,4] * [2;7]
ans =
 16
 34
```

The division symbol / denotes the multiplication by the inverse of the right argument (which must be a square matrix). To multiply by the inverse of the left argument, use the symbol \. This is handy to solve a set of linear equations. For example, to find the values of x and y such that $x + 2y = 2$ and $3x + 4y = 7$, type

```
> [1,2;3,4] \ [2;7]
ans =
 3
 -0.5
```

Hence $x = 3$ and $y = -0.5$. Another way to solve this problem is to use the `inv` function, which return the inverse of its argument. It is sometimes useful to multiply or divide matrices element-wise. The `.*`, `./` and `.\` operators do exactly that. Note that the + and - operators do not need special dot versions, because they perform element-wise anyway.

```
> [1,2;3,4] * [2,1;5,3]
ans =
 12 7
 26 15
> [1,2;3,4] .* [2,1;5,3]
ans =
 2  2
 15 12
```

Some functions change the order of elements. The transpose operator (tick) reverses the columns and the rows:

```
> [1,2;3,4;5,6]'
ans =
 1 3 5
 2 4 6
```

When applied to complex matrices, the complex conjugate transpose is obtained. Use dot-tick if you just want to reverse the rows and columns. The `flipud` function flips a matrix upside-down, and `fliplr` flips a matrix left-right.

```
> flipud([1,2;3,4])
ans =
 3 4
 1 2
> fliplr([1,2;3,4])
ans =
 2 1
 4 3
```

To sort the elements of each column of a matrix, or the elements of a row vector, use the `sort` function:

```
> sort([2,4,8,7,1,3])
ans =
 1 2 3 4 7 8
```

To get the size of a matrix, you can use the `size` function, which gives you both the number of rows and the number of columns unless you specify which of them you want in the optional second argument:

```
> size(rand(13,17))
ans =
 13 17
> size(rand(13,17), 1)
ans =
 13
> size(rand(13,17), 2)
ans =
 17
```

5.4 Polynomials

LME handles mostly numeric values. Therefore, it cannot differentiate functions like $f(x) = sin(e^x)$. However, a class of functions has a paramount importance in numeric computing, the polynomials. Polynomials are weighted sums of powers of a variable, such as $2x^2 + 3x - 5$. LME stores the coefficients of polynomials in row vectors; i.e. $2x^2 + 3x - 5$ is represented as `[2,3,-5]`, and $2x^5 + 3x$ as `[2,0,0,0,3,0]`.

Adding two polynomials would be like adding the coefficient vectors if they had the same size; in the general case, however, you had better use the function `addpol`, which can also be used for subtraction:

```
> addpol([1,2],[3,7])
ans =
 4 9
> addpol([1,2],[2,4,5])
ans =
 2 5 7
> addpol([1,2],-[2,4,5])
ans =
 -2 -3 -3
```

Multiplication of polynomials corresponds to convolution (no need to understand what it means here) of the coefficient vectors.

```
> conv([1,2],[2,4,5])
ans =
 2 8 13 10
```

Hence $(x+2)(2x^2+4x+5) = 2x^3+8x^2+13x+10$.

5.5 Strings

You type strings by delimiting them with single quotes:

```
> 'Hello, World!'
ans =
 Hello, World!
```

If you want single quotes in a string, double them:

```
> 'Easy, isn''t it?'
ans =
 Easy, isn't it?
```

Some control characters have a special representation. For example, the line feed, used in LME as an end-of-line character, is \n:

```
> 'Hello,\nWorld!'
ans =
 Hello,
 World!
```

Strings are actually matrices of characters. You can use commas and semicolons to build larger strings:

```
> ['a','bc';'de','f']
ans =
 abc
 def
```

5.6 Variables

You can store the result of an expression into what is called a variable. You can have as many variables as you want and the memory permits. Each variable has a name to retrieve the value it contains. You can change the value of a variable as often as you want.

```
> a = 3;
> a + 5
ans =
 8
> a = 4;
> a + 5
ans =
 9
```

Note that a command terminated by a semicolon does not display its result. To see the result, remove the semicolon, or use a comma if you have several commands on the same line. Implicit assignment to variable `ans` is not performed when you assign to another variable or when you just display the contents of a variable.

```
> a = 3
 a =
  3
> a = 7, b = 3 + 2 * a
 a =
  7
 b =
  17
```

5.7 Loops and Conditional Execution

To repeat the execution of some commands, you can use either a `for/end` block or a `while/end` block. With `for`, you use a variable as a counter:

```
> for i=1:3;i,end
 i =
  1
 i =
  2
 i =
  3
```

With `while`, the commands are repeated as long as some expression is true:

```
> i = 1; while i < 10; i = 2 * i, end
 i =
  2
 i =
  4
 i =
  8
```

You can choose to execute some commands only if a condition holds true :

```
> if 2 < 3;'ok',else;'amazing...',end
ans =
 ok
```

5.8 Functions

LME permits you to extend its set of functions with your own. This is convenient not only when you want to perform the same computation on different values, but also to make you code clearer by dividing the whole task in smaller blocks and giving names to them. To define a new function, you have to write its code in a file; you cannot do it from the command line. In Sysquake, put them in a function block.

Functions begin with a header which specifies its name, its input arguments (parameters which are provided by the calling expression) and its output arguments (result of the function). The input and output arguments are optional. The function header is followed by the code which is executed when the function is called. This code can use arguments like any other variables.

We will first define a function without any argument, which just displays a magic square, the sum of each line, and the sum of each column:

```
function magicsum3
  magic_3 = magic(3)
  sum_of_each_line = sum(magic_3, 2)
  sum_of_each_column = sum(magic_3, 1)
```

You can call the function just by typing its name in the command line:

```
> magicsum3
 magic_3 =
   8 1 6
   3 5 7
   4 9 2
 sum_of_each_line =
   15
   15
```

```
    15
  sum_of_each_column =
    15 15 15
```

This function is limited to a single size. For more generality, let us add an input argument:

```
function magicsum(n)
  magc = magic(n)
  sum_of_each_line = sum(magc, 2)
  sum_of_each_column = sum(magc, 1)
```

When you call this function, add an argument:

```
> magicsum(2)
 magc =
   1 3
   4 2
 sum_of_each_line =
   4
   6
 sum_of_each_column =
   5 5
```

Note that since there is no 2-by-2 magic square, `magic(2)` gives something else... Finally, let us define a function which returns the sum of each line and the sum of each column:

```
function (sum_of_each_line, sum_of_each_column) = magicSum(n)
  magc = magic(n);
  sum_of_each_line = sum(magc, 2);
  sum_of_each_column = sum(magc, 1);
```

Since we can obtain the result by other means, we have added semicolons after each statement to suppress any output. Note the uppercase S in the function name: for LME, this function is different from the previous one. To retrieve the results, use the same syntax:

```
> (sl, sc) = magicSum(3)
 sl =
   15
   15
   15
 sc =
   15 15 15
```

You do not have to retrieve all the output arguments. To get only the first one, just type

```
> sl = magicSum(3)
 sl =
   15
   15
   15
```

When you retrieve only one output argument, you can use it directly in an expression:

```
> magicSum(3) + 3
ans =
   18
   18
   18
```

One of the important benefits of defining function is that the variables have a limited scope. Using a variable inside the function does not make it available from the outside; thus, you can use common names (such as x and y) without worrying about whether they are used in some other part of your whole program. For instance, let us use one of the variables of `magicSum`:

```
> magc = 77
 magc =
   77
> magicSum(3) + magc
ans =
   92
   92
   92
> magc
 magc =
   77
```

5.9 Local and Global Variables

When a value is assigned to a variable which has never been referenced, a new variable is created. It is visible only in the current context: the base workspace for assignments made from the command-line interface, or the current function invocation for functions. The variable is discarded when the function returns to its caller.

Variables can also be declared to be global, i.e. to survive the end of the function and to support sharing among several functions and the base workspace. Global variables are declared with keyword `global`:

```
global x
global y z
```

A global variable is unique if its name is unique, even if it is declared in several functions.

In the following example, we define functions which implement a queue which contains scalar numbers. The queue is stored in a global variable named QUEUE. Elements are added at the front of the vector with function queueput, and retrieved from the end of the vector with function queueget.

```
function queueput(x)
  global QUEUE;
  QUEUE = [x, QUEUE];

function x = queueget
  global QUEUE;
  x = QUEUE(end);
  QUEUE(end) = [];
```

Both functions must declare QUEUE as global; otherwise, the variable would be local, even if there exists also a global variable defined elsewhere. The first time a global variable is defined, its value is set to the empty matrix [ ]. In our case, there is no need to initialized it to another value.

Here is how these functions can be used.

```
> queueput(1);
> queueget
ans =
   1
> queueput(123);
> queueput(2+3j);
> queueget
ans =
   123
> queueget
ans =
   2 + 3j
```

To observe the value of QUEUE from the command-line interface, QUEUE must be declared global there. If a local variable QUEUE already exists, it is discarded.

```
> global QUEUE
> QUEUE
QUEUE =
   []
> queueput(25);
> queueput(17);
> QUEUE
QUEUE =
   17 25
```

Chapter 6

SQ Script Tutorial

This chapter shows you how to develop a new SQ script for Sysquake. SQ scripts are the simplest way to make interactive graphics you can manipulate with the mouse for new problems. Basically, they are made from the commands you would type in the command window to create static graphics, with small changes to support interactive manipulation.

In the remaining of this chapter, we will develop an SQ script which displays the quadratic function $ax^2 + bx + c$ and its tangent at a point the user can manipulate. In the next chapter, we will write an equivalent SQ file, which will be more complicated but support undo/redo and allow us to add menus.

6.1 Displaying a Plot

In this section, we will write what is necessary to display in the same graphics the quadratic function, a vertical line which defines a value for `x0`, and the straight line which is tangent to the quadratic function at `x0`.

An SQ script is written as a text file using any text editor. If you prefer a word processor, make sure that you save the SQ script as raw text, or ASCII, and *not* as styled text. Sysquake handles end of lines in a sensible fashion; do not worry about the different conventions between Mac OS, Unix, Windows and other operating systems. For cross-platform compatibility, restrict yourself to the ASCII character set, and avoid two-bytes characters like Unicode and Japanese kanji (depending on the platform, bytes are interpreted as the native encoding, such as Latin-1 or Shift-JIS, or UTF-8). Once you have written and saved a file you want to test, simply open it in Sysquake. Make sure that the Command window or panel is visible, so that you can see error messages.

We can now begin to write the SQ script.

Step 1: Display the quadratic function

Type the following commands in the Command window or panel:

```
a = 1;
b = 2;
c = 4;
x = -10:0.1:10;
plot(x, a*x.^2+b*x+c);
```

The assignments set variables `a`, `b` and `c` to the coefficients of the quadratic function `ax^2+bx+c`, and variable `x` to an array of values for which the function is evaluated. Command `plot` displays the function.

Step 2: Calculate the tangent line

Let `dx+ey=f` be the tangent line at `x0`. Let us calculate `d`, `e` and `f`:

```
x0 = 1;
d = 2*a*x0+b;
e = -1;
f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);
```

Step 3: Display the tangent line

To display the tangent line, we use command `line`. Note that the quadratic function is not erased.

```
line([d,e], f);
```

Step 4: Write an SQ script

Now that we know how to display our graphics, we can store the commands in an SQ script. We group at the beginning the statements which initialize the variables we might want to manipulate interactively, for reasons which will be explained in the next step.

```
a = 1;
b = 2;
c = 4;
x0 = 1;

x = -10:0.1:10;
plot(x, a*x.^2+b*x+c);
```

```
d = 2*a*x0+b;
e = -1;
f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);

line([d,e], f);
```

Save this in a file named "tut_scf.sq" and open it in Sysquake. The graphics will be displayed automatically. Note that you can zoom and shift it interactively, something you could not do when you created the graphics from the command line.

6.2 Adding Interactivity

Step 5: Initialize variables only once

Sysquake evaluates the SQ script each time it has to update the graphics. However, there is no need to assign initial values to the variables. While it does not matter much now, it will become problematic when we add interactive manipulation of `x0`. Let us use function `firstrun`, which tells whether the script is executed for the first time, to make the initialization conditional:

```
if firstrun
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
end

...
```

Step 6: Add a vertical line at x0

To manipulate the tangency point, we add a vertical line at `x0`. We draw it in red, and add an identifier (the last argument) to tell Sysquake to display a hand when the cursor is near the line.

```
line([1,0], x0, 'r', 1);
```

Step 7: Change x0 when the user manipulates the line

Now comes the interesting part: interactivity. When the user clicks the figure, Sysquake evaluates the SQ script continuously until the mouse button is released. Functions are available to know which object the mouse is over and where the mouse is. In our case, we will use `_id`,

which gives the identifier of the manipulated object (i.e. 1 for the vertical line), and `_x1`, which gives the horizontal position of the mouse in the coordinates of the graphics. We use a `switch` block to change `x0` only when the user manipulates the vertical line, and we change `x0` before we use it to calculate the tangent line.

The complete SQ script is now

```
if firstrun
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
end

switch _id
  case 1
    x0 = _x1;
end

x = -10:0.1:10;
plot(x, a*x.^2+b*x+c);

d = 2*a*x0+b;
e = -1;
f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);

line([d,e], f);

line([1,0], x0, 'r', 1);
```

Click the red line and drag it to the right: the tangent line will follow.

Chapter 7

SQ Script Reference

There are two ways to program interactive graphics for Sysquake: SQ scripts and SQ files. Both are text files based on LME, Sysquake's language. For small programs, SQ scripts are simpler than SQ files, because they do not require the declarations of variables, figures and functions; but they have limitations which make them less suitable for large applications. They should be used only for interactive graphics when no other form of user interface is necessary. The table below summaries the differences.

	SQ scripts	**SQ files**
Interactive graphics	x	x
Sliders and buttons	x	x
Zoom and Shift	x	x
Multiple synchronized graphics	x	x
Easy access to variables	x	
Figure menu		x
Settings menu		x
Undo/Redo		x
Save		x
Functions	libraries only	x
Suitable for long computation		x
Help		x
Multiple instances		on some platforms

Structure of an SQ script

An SQ script is a sequence of LME commands and expressions, very similar to what could be typed in the command-line interface. The single command

```
plot(sin(0:0.1:2*pi));
```

is enough to display a sine evaluated for angles between 0 and 2π and let the user change the axis limits with the Zoom, Zoom-X, and Shift interactive commands. When the user clicks in the figure window, Sysquake interprets the mouse action and executes the whole script again. The script may check whether a graphical element was manipulated (clicked or dragged with the mouse) and react accordingly.

The typical structure of an SQ script which supports the interactive manipulation of graphical element(s) is described below. Code samples show a typical implementation for manipulating the vertical position of points; but of course, many variants are possible.

Variable initialization Graphics depend on the value of one or more variables. This dependence enables interaction. But before any interaction occurs, the variables must be assigned initial values. Since the whole script is executed each time the user clicks with the mouse in the graphics or when the window is resized, the variable initialization must be performed only once, the first time the SQ script is run, which can be determined with function `firstrun`.

```
if firstrun
  x = 1:10;
  y = rand(2,10);
end
```

Interaction handling The script checks if is called as the result of the manipulation of a graphical element. Graphical elements which can be manipulated are usually associated with an identifier (or ID), an arbitrary positive integer given in the command used to draw the element. When the user manipulates this element, the SQ script can retrieve its ID with function `_id`. When the click occurs far away from any graphical element with an ID, `_id` returns the empty array `[]`. Typically, `_id` is used in a `switch` construct. Other functions give more information about the click, such as its coordinates:

Name	Description
_z	initial position of the mouse as a complex number
_x	initial horizontal position of the mouse
_y	initial vertical position of the mouse
_z0	initial position of the clicked element as a complex number
_x0	initial horizontal position of the clicked element
_y0	initial vertical position of the clicked element
_p0	initial position of the clicked element as a 2D or 3D vector
_z1	current position of the mouse as a complex number
_x1	current horizontal position of the mouse
_y1	current vertical position of the mouse
_p1	current position of the mouse as a 2D or 3D vector
_str1	current string parameter
_kx	factor the horizontal position is multiplied by (_x1/_x)
_ky	factor the horizontal position is multiplied by (_y1/_y)
_kz	complex factor the position is multiplied by in the complex plane (_z
_q	additional data specific to the plot
_m	true if the modifier key (Shift key) is held down
_id	ID of the manipulated object
_nb	number of the manipulated trace (1-based)
_ix	index of the manipulated point (1-based)

In our example, the vertical coordinate of the point being manipulated in array y is replaced by the vertical position of the mouse.

```
switch _id
  case 1
    y(_nb,_ix) = _y1;
end
```

Computation The SQ script can call any of the functions and commands of LME to compute the data required for drawing the graphics.

Graphics display All graphics commands of Sysquake are available, such as `plot`, `line`, `text`, and `image`. The SQ script should not call `clf` to clear the figure window; Sysquake will take care of this.

Our example just displays in red the lines of matrix y with the `plot` command, and it gives them an ID of 1.

```
plot(x, y, 'r', 1);
```

Several subplots can be displayed with command `subplot`. Commands `label` and `title` may be used to add labels.

Here is the complete program, which is probably not very useful but shows the basic elements of an SQ script (see Fig. 7.1).

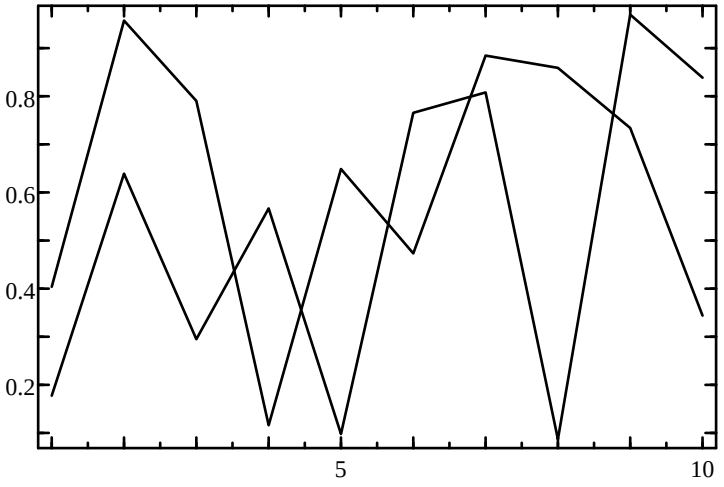

Figure 7.1
Basic SQ script. Each point can be moved up and down with the mouse.

```
if firstrun
  x = 1:10;
  y = rand(2,10);
end

switch _id
  case 1
    y(_nb,_ix) = _y1;
end

plot(x, y, 'r', 1);
```

Chapter 8

SQ File Tutorial

This chapter shows you how to develop an SQ file for Sysquake. Like SQ scripts we saw in the previous chapter, SQ files are programs for Sysquake. But they are built in a stricter framework which provides automatically undo/redo, save/restore, and subplot layout; it also supports menus, choice of plots, periodic processing, a mechanism to compute the minimal amount of information required for the plots displayed at a given time, and an import/export system.

Follow each step of this tutorial, and starting from scratch, you will end up with an SQ file which lets you move the tangent of a quadratic function, and the knowledge to write your own SQ files. When you will need more information, you can refer to the Sysquake Reference chapter.

This tutorial assumes you have a basic knowledge of a procedural programming language, such as C, Pascal, Fortran, a modern Basic dialect, or MATLAB(R). The concepts of variable and function are supposed to be well known.

Structure of an SQ file

This tutorial will describe each element when they are used. However, it is important to notice an important difference between SQ scripts and libraries, which contain LME code, and SQ files, which begin with static declarations of the application components, such as data, figures and menus. In an SQ file, LME code is located in function blocks, i.e. between the lines

```
functions
{@
```

and

```
@}
```

Everything else consists of declarations and comments. Some of the declarations refer to functions defined in function blocks; for instance, a figure declaration includes the function called to draw it, together with its arguments. Sysquake reads all the declarations when the SQ file is loaded in memory; it uses them to reserve space for data, to set up user interface elements, and execute LME functions when required, for instance as a consequence of user actions.

8.1 Displaying a Plot

In this section, we will write what is necessary to display in the same graphics the quadratic function, a vertical line which defines a value for `x0`, and the straight line which is tangent to the quadratic function at `x0`.

An SQ file is written as a text file using any text editor. If you prefer a word processor, make sure that you save the SQ file as raw text, or ASCII, and *not* as styled text. On some versions of Sysquake, a built-in editor is available; check if there is a New item in the File menu (Load lets Sysquake load or reload the text of the front window, while Open reads an SQ file from a files). Sysquake handles end of lines in a sensible fashion; do not worry about the different conventions between Mac OS, Unix, Windows and other operating systems. For cross-platform compatibility, restrict yourself to the ASCII character set, and avoid two-bytes characters like Unicode and Japanese kanji (depending on the platform, bytes are interpreted as the native encoding, such as Latin-1 or Shift-JIS, or UTF-8). Once you have written and saved a file you want to test, simply open it in Sysquake. Make sure that the Command window or panel is visible, so that you can see error messages.

We can now begin to write the SQ file.

Step 1: Choosing variables

The most important concept in Sysquake is the set of variables. Variables define the state of the system (we use the word "system" in a broad meaning as what the user perceives from the graphics). Everything that can be changed, be it interactively, by specifying parameters in a dialog box, or by loading an SQ data file, must be stored in variables. In addition, auxiliary variables can be used as a convenience to avoid repetitive computations or to transmit values between handler functions (more about them later). Each variable can contain a real or complex array, a string, a list, or a structure. Variables are identified by a name of up to 32 letters, digits, and the underscore character "_" which begins with a letter (names beginning with the underscore are reserved). As everything else in Sysquake, names are

case-sensitive; x and X are two different names and identify two separate variables.

You can declare as many variables as you need. Do not use a big array to pack as many different things as you can; it is much more efficient to have a clean set of variables, so that you can use them and change them more easily.

Sysquake considers the values of the variables as a set. Each time the user changes a variable (interactively or otherwise), Sysquake creates a new set and changes the new values. The value of unmodified variables is retained. The command Undo reverts to the previous set. This is why you should usually not use global variables, which exist only in one copy.

For our example, we define variables a, b, and c for the coefficients of the quadratic function; variables d, e, and f for the tangent $dx+ey=f$; and variable x0 for the horizontal position where the line is tangent to the function.

To let Sysquake know about our choice, we write the following lines at the beginning of the SQ file:

```
variable a b c   // coefficients of the quadratic function
                 //  y=ax^2+bx+c
variable d e f   // coefficients of the tangent dx+ey=f
variable x0      // value of x where the line is tangent
```

The keyword `variable` is required; it is followed on the same line by one or more variable names, separated by spaces or tabulators. Everything following the two slashes `//` is a comment which is ignored by Sysquake.

Step 2: Giving initial values

At the beginning, each variable is set to the empty matrix `[]`. Drawing functions could recognize them and not display anything, but it is nicer for the user to start immediately with default values. In Sysquake, variables are set and used by *handler* functions. Functions are written in the LME language, and declared to Sysquake by a handler declaration. Handler declarations and function definitions are very similar. They both use variables, which do not necessarily have matching names. Variables in the handler declaration correspond to the set of variables declared at the level of the SQ file; variables in the function definition are meaningful only in the function itself. The input arguments in the handler declarations must be variables or integer numbers; they cannot be expressions. The handler declaration begins with a keyword, for example `init` to define default values. Here is an init handler for our SQ file:

```
init (a,b,c,x0,d,e,f) = init
```

We will use parenthesis for functions with several output arguments. You may use square brackets if you prefer. The function declared above is defined in a function block. We also write a function `calcTangent` to calculate the tangent of the quadratic function.

```
function
{@
function (a,b,c,x0,d,e,f) = init
  // initial values for the function coefficients
  //  and the x0 value
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
  (d,e,f) = calcTangent(a,b,c,x0);

function (d,e,f) = calcTangent(a,b,c,x0)
  // tangent to y=f(x) at x0 is y-f(x0)=f'(x0)(x-x0),
  //  where f' is der f
  // derivative of ax^2+bx+c is 2ax+b
  d = 2*a*x0+b;
  e = -1;
  f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);
@}
```

Notice the block in {@ @} (function block); now it contains only the `init` and `calcTangent` functions, but we will add more functions in the next sections. The function block does not need to follow a particular handler declaration; handlers are identified only by their name. Usually, we will put the function block after all the declarations. In LME code it contains (but not in declarations), the percent symbol can be used instead of the two slashes to begin a comment.

Errors in an SQ file are detected when you open or load it in Sysquake. To let Sysquake analyze your code and catch constructs which might be errors, you can select SQ File Possible Error Warnings in the Preferences. It would be the case if we do not provide initial values for all variables, or if the order of variables in the init handler declaration does not match the one in its implementation, here in function `init`.

Step 3: Displaying a plot

Each figure is declared by a figure declaration line which contains a name between quotes, and one or more lines declaring handlers for drawing the plot and processing the manipulations with the mouse. For now, we just declare a draw handler (outside the function block), which needs to know the value of the seven variables.

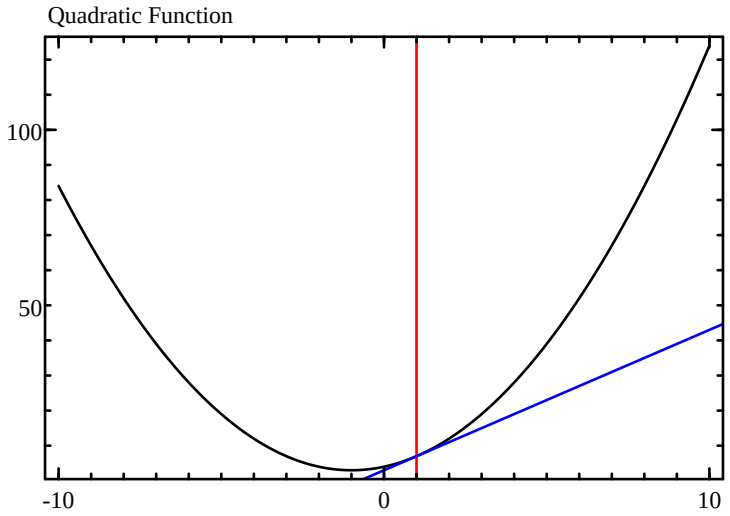

Figure 8.1 SQ file figure

```
figure "Quadratic Function"
  draw drawFunc(a,b,c,x0,d,e,f)
```

The figure displays the quadratic function $ax^2 + bx + c$. In addition, the tangent line is displayed in blue, and a red vertical line represents the position where the blue line is tangent to the function. Do not worry about the mysterious fourth argument of `line`; it is an arbitrary positive identifier which will be used for interactive manipulation.

```
function drawFunc(a,b,c,x0,d,e,f)
  // values of x where the function is evaluated
  x = -10:0.1:10;
  // plot the function
  plot(x, a*x.^2+b*x+c);
  // plot in red ('r') a vertical line at x0
  line([1,0],x0,'r',1);
  // plot in blue ('b') the tangent at x0
  line([d,e],f,'b');
```

If you have typed all the code above, not forgetting to put the function `drawFunc` in the function block, you can open it in Sysquake and observe your first graphics (see Fig. 8.1). Congratulations!

If you do not specify which figures you want to display when the SQ file is opened, Sysquake displays the first one. With more than one, you may want to specify explicitly which one(s) to show. Add a command `subplot` to the init handler with the name of the figure:

```
subplots('Quadratic Function');
```

Make sure that the string matches exactly the name of the figure. To display several figures, you would separate their names with tabulators (’\t’) for figures on the same row, and with line feeds (’\n’) for separating each row. The complete SQ file is shown below.

```
variable a b c   // coefficients of the quadratic function
                 //  y=ax^2+bx+c
variable d e f   // coefficients of the tangent dx+ey=f
variable x0      // value of x where the line is tangent

init (a,b,c,x0,d,e,f) = init

figure "Quadratic Function"
  draw drawFunc(a,b,c,x0,d,e,f)

function
{@
function (a,b,c,x0,d,e,f) = init
  // initial values for the function coefficients
  //  and the x0 value
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
  (d,e,f) = calcTangent(a,b,c,x0);
  subplots('Quadratic Function');

function (d,e,f) = calcTangent(a,b,c,x0)
  // tangent to y=f(x) at x0 is y-f(x0)=f'(x0)(x-x0),
  //  where f' is der f
  // derivative of ax^2+bx+c is 2ax+b
  d = 2*a*x0+b;
  e = -1;
  f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);

function drawFunc(a,b,c,x0,d,e,f)
  // values of x where the function is evaluated
  x = -10:0.1:10;
  // plot the function
  plot(x, a*x.^2+b*x+c);
  // plot in red ('r') a vertical line at x0
  line([1,0],x0,'r',1);
  // plot in blue ('b') the tangent at x0
  line([d,e],f,'b');
@}
```

8.2 Adding Interactivity

The plot of the previous section is static; until now, we have not seen anything which makes Sysquake different, except for a slightly more complicated set-up. We will now dip into interactivity by allowing the user to move the tangent point and observe the tangent.

Step 4: Writing a mouse drag handler

To enable the manipulation of a graphical element, a mouse drag handler must be declared under the same `figure` heading as the draw handler. The mouse drag handler is also a function defined in the function block. There is an important difference, however: it returns new values for one or several variables (not necessarily the same as the input).

Values related to the user interaction are obtained as special variables which begin with an underscore "_". We want to drag the vertical red line at `x0`; hence we need the current `x` coordinate of the mouse, and an indication about whether the user selected the line. The horizontal position of the mouse during the drag is given by `_x1`. A graphic ID is given by `_id`; it corresponds to the last argument of graphical commands like `line` or `plot`. If the user clicks far away from any object drawn by a command with an ID, `_id` is the empty matrix `[]`. Since we want the user to drag the vertical line, we expect to have `_id` set to 1, the value passed to `line` in the draw handler.

Special variables can be passed to the handler as input arguments, or used directly in the handler without any declaration. This is what we shall do here to reduce the number of arguments to the minimum.

```
mousedrag (x0,d,e,f) = dragX0(a,b,c)
```

The mousedrag handler should calculate not only the new value of `x0`, but also all other variables which depend on it, i.e. the coefficients of the tangent. The update of the graphics is totally automatic, and you get a multilevel Undo/Redo for free!

```
function (x0,d,e,f) = dragX0(a,b,c)
  if isempty(_id)
    cancel;
  end
  x0 = _x1;
  (d,e,f) = calcTangent(a,b,c,x0);
```

In this definition (located in the function block), we note the check for an empty `id`. If we do not click the red line, the handler should not terminate normally; even if we kept the previous values, a new Undo frame would be created, and the first execution of Undo would have no visible effect.

If you type the code above, you have a figure where you can manipulate the vertical line with the mouse and see the tangent move.

Step 5: The final touch, a mouseover handler

Interactive manipulation is much easier if subtle hints about what can be manipulated are displayed. Such hints include the shape of the cursor, which should be a finger only if a click permits the manipulation of an element, and messages in the status bar at the bottom of the window. The mouseover handler, which is called in Manipulate mode when the mouse is over a figure, gives this kind of information to Sysquake. The input arguments are similar to the mousedrag handler. The output arguments are special: they should be `_msg`, `_cursor`, or both. `_msg` should be set to a string which is displayed in the status bar. `_cursor` should be set to true to have a finger cursor, and to false to have the plain arrow cursor. Canceling the mouseover handler is like setting `_msg` to the empty string `''` and `_cursor` to false. Note also that if a figure has a mousedown, mousedrag, and/or mouseup handler, but no mouseover handler, the cursor will be set to the finger.

In our case, the user can manipulate the only object with a non-empty id. There is no need to define a function for such a simple task:

```
mouseover _cursor = ~isempty(id)
```

Adding messages is not much more complicated, but now we must define a function. To display the value of `x0`, we can use either the position of the vertical line or the value of `x0`. The special variable `_x0` is the position of the line, not the position of the mouse as in the declaration of the mousedrag handler. The early cancellation of the execution of the handler is easier (and faster) to handle the case where the mouse in not over an object. The handler definition is

```
function (_msg, _cursor) = overFunc
  if isempty(_id)
    cancel;
  end
  _msg = sprintf('x0: %g', _x0);
  _cursor = true;
```

and its declaration is

```
mouseover (_msg, _cursor) = overFunc
```

There is still a problem: the message is not displayed when the user actually drags the vertical line, because the mouseover handler is not called when the mouse button is held down. For this, `_msg` must be added to the mousedrag handler. One way to do this is to declare the handler as

```
mousedrag (x0,d,e,f,_msg) = dragX0(a,b,c)
```

and to define it as

```
function (x0,d,e,f,msg) = dragX0(a,b,c)
  if isempty(_id)
    cancel;
  end
  x0 = _x1;
  (d,e,f) = calcTangent(a,b,c,x0);
  msg = sprintf('x0: %g', x0);
```

8.3 Menu Entries

It may be useful to set the value of some parameters with a menu entry. In our case, it would be difficult to specify in a figure the coefficients of the quadratic function. An SQ file can define *menu handlers*; new entries are installed in the Settings menu (which appears only if it is not empty), and the corresponding handler is executed when the entry is selected in the menu.

Let us add a menu entry which displays a dialog box where we can change the coefficients. First, we declare it with

```
menu "Quadratic Function..."
     (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
```

The input arguments allow to display the current values and to calculate the new tangent for the current value of `x0`. Note how lines can be split between its components. Here is the handler definition:

```
function (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
  (a,b,c) ...
    = dialog('Coefficients a,b,c of ax^2+bx+c:',a,b,c);
  (d,e,f) = calcTangent(a,b,c,x0);
```

The dialog function displays three kinds of alert or dialog boxes, depending on the number of input and output arguments. As we use it here, the first argument is a description, and the remaining input arguments are initial values which are displayed in an edit field. They can be modified by the user. When the OK button is clicked, the dialog box is dismissed and the output arguments receive the new values. If the Cancel button is clicked, the execution of the handler is aborted exactly as if the cancel command had been executed.

Each menu entry can be decorated in two ways: a checkmark can be displayed on the left, and the entry can be disabled (it cannot be selected and the text is written in gray). It does not make sense to use these possibilities with our first menu. Let us add support to choose

whether the position of x0 is displayed with a vertical line or a small diamond. First, we add a variable whose value is true for a line and false for a diamond.

```
variable x0Line
```

We initialize it to true in the init handler, whose declaration becomes

```
init (a,b,c,x0,d,e,f,x0Line) = init
```

and definition

```
function (a,b,c,x0,d,e,f,x0Line) = init
  // initial values for the function coefficients
  //  and the x0 value
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
  (d,e,f) = calcTangent(a,b,c,x0);
  subplots('Quadratic Function');
  // x0 is displayed as a line
  x0Line = true;
```

The draw handler should get the new variable and act accordingly. Here is the new drawFunc handler declaration:

```
draw drawFunc(a,b,c,x0,d,e,f,x0Line)
```

and its definition:

```
function drawFunc(a,b,c,x0,d,e,f,x0Line)
  // values of x where the function is evaluated
  x = -10:0.1:10;
  // plot the function
  plot(x, a*x.^2+b*x+c);
  if x0Line
    // plot in red ('r') a vertical line at x0
    line([1,0],x0,'r',1);
  else
    // plot in red ('r') a diamond ('<') at (x0,f(x0))
    plot(x0,a*x0^2+b*x0+c,'r<',1);
  end
  // plot in blue ('b') the tangent at x0
  line([d,e],f,'b');
```

The mousedrag handler needs no modification. Now the most interesting part. We add two menu entries, declared as

```
separator
menu "Line" _checkmark(x0Line) x0Line = 1
menu "Diamond" _checkmark(~x0Line) x0Line = 0
```

The `separator` adds a horizontal line or a space between the first menu entry and these two new elements. Between the entry title and the handler declaration, the `_checkmark` keyword is used to tell Sysquake to display a check mark if the expression is parenthesis is true. This expression may be more complicated than a variable; for the second entry, we use the *not* operator, so that depending on the value of `x0Line`, either one or the other is checked. No handler definition is needed here, because we set `x0Line` to a constant. In handler declarations, only integers are permitted; fortunately, setting `x0Line` to 1 or 0 works fine.

Here is the complete SQ file:

```
variable a b c   // coefficients of the quadratic function
                 //  y=ax^2+bx+c
variable d e f   // coefficients of the tangent dx+ey=f
variable x0      // value of x where the line is tangent
variable x0Line

init (a,b,c,x0,d,e,f,x0Line) = init

menu "Quadratic Function..."
     (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
separator
menu "Line" _checkmark(x0Line) x0Line = 1
menu "Diamond" _checkmark(~x0Line) x0Line = 0

figure "Quadratic Function"
  draw drawFunc(a,b,c,x0,d,e,f,x0Line)
  mousedrag (x0,d,e,f,_msg) = dragX0(a,b,c)
  mouseover (_msg,_cursor) = overFunc

function
{@
function (a,b,c,x0,d,e,f,x0Line) = init
  // initial values for the function coefficients
  //  and the x0 value
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
  (d,e,f) = calcTangent(a,b,c,x0);
  subplots('Quadratic Function');
  // x0 is displayed as a line
  x0Line = true;

function (d,e,f) = calcTangent(a,b,c,x0)
  // tangent to y=f(x) at x0 is y-f(x0)=f'(x0)(x-x0),
  //  where f' is der f
  // derivative of ax^2+bx+c is 2ax+b
  d = 2*a*x0+b;
```

```
  e = -1;
  f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);

function (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
  (a,b,c) ...
    = dialog('Coefficients a,b,c of ax^2+bx+c:',a,b,c);
  (d,e,f) = calcTangent(a,b,c,x0);

function drawFunc(a,b,c,x0,d,e,f,x0Line)
  // values of x where the function is evaluated
  x = -10:0.1:10;
  // plot the function
  plot(x, a*x.^2+b*x+c);
  if x0Line
    // plot in red ('r') a vertical line at x0
    line([1,0],x0,'r',1);
  else
    // plot in red ('r') a diamond ('<') at (x0,f(x0))
    plot(x0,a*x0^2+b*x0+c,'r<',1);
  end
  // plot in blue ('b') the tangent at x0
  line([d,e],f,'b');

function (x0,d,e,f,msg) = dragX0(a,b,c)
  if isempty(_id)
    cancel;
  end
  x0 = _x1;
  (d,e,f) = calcTangent(a,b,c,x0);
  msg = sprintf('x0: %g', x0);

function (_msg,_cursor) = overFunc
  if isempty(_id)
    cancel;
  end
  _msg = sprintf('x0: %g', _x0);
  _cursor = true;
@}
```

8.4 More about graphic ID

Graphic ID have an important role: they permit to link drawing code in the draw handler with the code which handles user interactions in the mousedrag and mouseover handlers. Graphic are arbitrary positive integer numbers. Their value is not important, provided they are unique in each figure and they are used in a consistent way.

Graphic ID in declarations

In the SQ file of the tutorial, the ID is used only to detect if the mouse is located near the corresponding graphical object or not. In more complicated cases where multiple graphical objects with different ID are displayed in the same figure, mousedrag and mouseover handlers would typically have a `switch` statement to react in a different way for different objects. There is an alternative way to let Sysquake choose which part of code to execute, which often leads to simpler SQ files: specify the ID in the handler declaration, right after the `mousedrag` or `mouseover` declaration. In our SQ file, the figure declaration would become

```
figure "Quadratic Function"
  draw drawFunc(a,b,c,x0,d,e,f,x0Line)
  mousedrag 1 (x0,d,e,f,_msg) = dragX0(a,b,c,_x1)
  mouseover 1 _msg = overFunc(_x0)
```

and the definition of function `dragX0`

```
function (x0,d,e,f) = dragX0(a,b,c,x1)
  x0 = x1;
  (d,e,f) = calcTangent(a,b,c,x0);
```

If there were multiple graphical objects with different ID, the figure declaration would have multiple mousedrag handlers. It is also possible to keep a default mousedrag handler (without ID) for remaining objects and for mouse clicks elsewhere in the figure.

Mouseover handlers can also have a specific ID. But there is an additional benefit: the cursor is set automatically to the finger over objects with an ID for which a mousedrag is declared, and to a plain arrow elsewhere. This is why the declaration of the mouseover above does not produce a `_cursor` output argument anymore.

Constant naming

In programs, a good practice is to give names to all significant constants, especially if they are reused at different locations. LME provides the `define` programming construct to create named constants. In SQ files, `define` can also be used outside any function block, so that it has a scope in both declarations and LME code. The special value `_auto` is set successively to 1, 2, etc.; its main purpose is to produce unique values for constants used as graphic ID. For instance

```
define kLowId = _auto
define kHighId = _auto
```

defines `kLowId` as 1 and `kHighId` as 2. Here is again the complete code of the tutorial SQ file.

```
variable a b c   // coefficients of the quadratic function
                 //  y=ax^2+bx+c
variable d e f   // coefficients of the tangent dx+ey=f
variable x0      // value of x where the line is tangent
variable x0Line

define kLineId = _auto

init (a,b,c,x0,d,e,f,x0Line) = init

menu "Quadratic Function..."
     (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
separator
menu "Line" _checkmark(x0Line) x0Line = 1
menu "Diamond" _checkmark(~x0Line) x0Line = 0

figure "Quadratic Function"
  draw drawFunc(a,b,c,x0,d,e,f,x0Line)
  mousedrag kLineId (x0,d,e,f,_msg) = dragX0(a,b,c)
  mouseover kLineId _msg = overFunc(_x0)

function
{@
function (a,b,c,x0,d,e,f,x0Line) = init
  // initial values for the function coefficients
  //  and the x0 value
  a = 1;
  b = 2;
  c = 4;
  x0 = 1;
  (d,e,f) = calcTangent(a,b,c,x0);
  // x0 is displayed as a line
  x0Line = true;

function (d,e,f) = calcTangent(a,b,c,x0)
  // tangent to y=f(x) at x0 is y-f(x0)=f'(x0)(x-x0),
  //  where f' is der f
  // derivative of ax^2+bx+c is 2ax+b
  d = 2*a*x0+b;
  e = -1;
  f = (2*a*x0+b)*x0-(a*x0^2+b*x0+c);

function (a,b,c,d,e,f) = menuFunc(a,b,c,x0)
  (a,b,c) ...
    = dialog('Coefficients a,b,c of ax^2+bx+c:',a,b,c);
  (d,e,f) = calcTangent(a,b,c,x0);

function drawFunc(a,b,c,x0,d,e,f,x0Line)
  // values of x where the function is evaluated
  x = -10:0.1:10;
```

```
  // plot the function
  plot(x, a*x.^2+b*x+c);
  if x0Line
    // plot in red ('r') a vertical line at x0
    line([1,0],x0,'r',kLineId);
  else
    // plot in red ('r') a diamond ('<') at (x0,f(x0))
    plot(x0,a*x0^2+b*x0+c,'r<',kLineId);
  end
  // plot in blue ('b') the tangent at x0
  line([d,e],f,'b');

function (x0,d,e,f) = dragX0(a,b,c,x1)
  x0 = x1;
  (d,e,f) = calcTangent(a,b,c,x0);

function msg = overFunc(x0)
  msg = sprintf('x0: %g', x0);
@}
```

8.5 Saving Data

Once the user has changed the tangent point, he might find convenient to save it to a file and read it back later. In the SQ file, nothing more is required; the contents of all the variables as well as the information necessary to restore the subplots are written to an SQ data file with the Save command. Opening such a file reloads everything provided that the original file is found. If more control is desired on what is stored in the SQ data file and how it is read back, input and output handlers can be added.

Chapter 9

SQ File Reference

This chapter describes the syntax of SQ files, the "programs" run by Sysquake, and the contents of SQ Data files (files with a .sqd suffix), which store the state of a session with an SQ file.

9.1 SQ Files

SQ files define the set of interactive plots which can be displayed. Each SQ file corresponds to a specific kind of problem; for instance, one could have SQ files for the design of a digital filter or a PID controller for continuous-time systems, for the study of the effect of initial conditions on a simulation, and so on.

SQ files are text files. You can write them using any text editor or word processor. Make sure that you save them as plain ASCII files, without any style or formatting information; depending on the application, this option is called Text Document, Text Only, ASCII File, or an equivalent expression. Also, do not rely on automatic word wrapping; make sure that all lines end with an explicit end-of-line character. Sysquake accepts carriage returns, line feeds, or both, to accommodate for the text file formats found on different operating systems.

SQ files contain different kinds of elements:

Declaration of variables Variables are used as parameters for the figures, menu item actions, etc. Manipulating a figure changes some of the variables, and all the figures currently displayed are updated to reflect the changes. These variables, called *Sysquake variables*, must not be confounded with LME variables (variables used without declaration in LME functions).

Definition of constants Integer literal values can be given a name to make the code clearer. These definitions are visible in the declaration part of SQ files as well as in function definitions.

Declaration of handlers Handlers are expressions executed to perform different tasks managed by Sysquake, such as initialization, figure manipulation, menu selection, etc. They have the same syntax as LME assignments, or expressions if no result is to be used. As input and output, they use Sysquake variables as well as values managed directly by Sysquake, such as the position of the mouse. Variables in the left-hand side of assignments cannot use indexing or structure field access. Values managed by Sysquake are identified with a name beginning with an underscore; they can be used either directly in the handler declaration, or indirectly in a function called in a handler declaration.

Function definitions Handlers are implemented by functions written in LME, an interpreted language well suited for numeric computation.

Help A textual description can be provided in SQ files. Sysquake displays it upon user request.

Syntax

SQ files contain declarations, blocks of text, and comments. Declarations are always placed at the beginning of a line; they start with a keyword, and are usually followed by other elements. Here is the list of keywords:

```
beginmenu       functions          mousescroll
beginsubmenu    help               mouseup
data            idle               output
define          import             publichandler
draw            init               resize
dragin          input              separator
dragout         keydown            terminate
embeddedfile    make               title
endmenu         menu               use
endsubmenu      mousedown          userinterface
export          mousedoubleclick   variable
extension       mousedrag          variables
fighandler      mousedragcont      version
figure          mouseout           watch
function        mouseover
```

Remark: all these keywords are supported on all versions of Sysquake to assure the compatibility of SQ files. However, `import`, `export` and `extension` declarations have an effect only on the full version of Sysquake.

Declarations must be either contained on a single line or split into several lines with the continuation characters `...` at the end of the first line(s):

```
variable a, b, ...
         c, d, e
```

Many keywords are followed by handler declarations. These handlers are implemented in LME code in a functions block and may accept input arguments and output arguments. Arguments may be variables declared in the SQ file, special arguments beginning with an underscore, or (for the input arguments) integer numbers or named constants. In some case, handler declarations may be reduced to a simple assignment such as y=x or num=3. Expressions are not allowed.

Comments are either enclosed between `/*` and `*/` or span from `//` or `%` to the next end of line. They cannot be nested.

Declaration of Variables

Variables defined at the level of the SQ file play a very important role in Sysquake. They are global; all handlers can use them as input and/or output arguments. They are preserved between handler executions. They contain the whole state of Sysquake; together with the SQ file itself, they permit to restore what appears in the Figure window, and are used in the Save mechanism. Finally, they are the base of the Undo system, which enables the user to cancel the last operations (such as manipulating a figure with the mouse, changing a subplot or entering numeric values in a dialog box) and redo them.

Variables are declared with the `variable` (or `variables`) keyword, followed by one or several variable names separated by commas or spaces. They can contain any data type supported by LME, such as numbers, arrays, strings, structures, etc. Variables are always defined; variables whose value has not been set explicitly contain the empty array `[]`. Variable names are made of letters, digits and underscores; they begin with a letter, and case is significant, like everywhere else in Sysquake. Names beginning with an underscore are reserved for special purposes and should not be used for variables.

Five variables are managed by Sysquake and cannot be used in handlers: `_plots`, a string which contains the name of the subplots (see function subplots); `_plotprops`, an array which contains the properties and scaling of each subplot (see function subplotprops); `_plotpos`, an array which contains the position of subplots when Free Position is enabled (see function subplotpos); `_plotsync`, an array which defines which and how subplot scales are synchronized (see function subplotsync); and `_plotparam`, a list whose elements are arbitrary data specific to each subplot (see function subplotparam).

They are revealed when the state of Sysquake is saved to SQ Data files.

Example

```
variable x
variables num den
```

Variables can be initialized in the init handler(s) (see below), or directly in the variable declaration; in that case, only one variable can be declared and initialized for each declaration statement, and the expression on the right of the equal character must not depend on other variables.

Example

```
variable x = 2
variable vec = 1:10
```

If the expression on the right of the equal character depends on other variables, the variable on the left is declared, and the assignment is treated as a make handler (see below), i.e. the assignment is performed everytime the variable is used and its dependencies have changed.

Variables can be declared *nondumpable* to prevent Sysquake from saving and restoring them, be it with the Undo/Redo mechanism or with Save and Open. Nondumpable variables can be used and modified like any other variable; however, modifying them with any handler will prevent Undo from restoring their previous state, and Save (or Dump Data) will skip their value (the saving mechanism is detailed later in this chapter). Variables which should be declared as nondumpable include file descriptors for files and serial port connections: a file descriptor is only valid if the corresponding file or device has been opened with `fopen` or similar functions for devices.

Nondumpable variables are declared with the keyword `_nondumpable` following immediately `variable`. No comma may precede or follow `_nondumpable`. Dumpable and nondumpable variables may not be mixed in the same declaration.

Example

```
variables _nondumpable fd, connection
```

Declaring variables explicitly incites to plan how they will be used as the glue between the different handlers. It makes easier the documentation; each variable can be commented, and their enumeration, if self-explanatory names are chosen, is itself an important help for future maintenance. But in case where small scripts are preferred (maybe as a compact version for on-line distribution), implicit declarations are also possible. Implicit declarations are enabled as follows:

```
variable _implicit
```

When implicit declarations are enabled, all the variables appearing in handler declarations are managed by Sysquake exactly as if they had been declared explicitly in `variable` statements. Implicit variables cannot have the `_nondumpable` attribute; however, you can declare only the nondumpable variables and have a `variable _implicit` declaration for all other variables.

Definition of constants

Some numbers are used as identifiers at different places. For instance graphical objects which can be manipulated with the mouse are tagged with an identifier when they are displayed, and this identifier is used to recognize them when the user clicks them. Numeric ID can be replaced with more meaningful names with constant definitions. Constant definitions begin with the keyword `define`, followed by the identifier, the equal character, and the integer value or the keyword `_auto`. Constants defined with `_auto` are numbered consecutively from 1. In the example below, `kGainID` is 1, `kDelayID` is 2, etc.

Example

```
define kGainID = _auto
define kDelayID = _auto
define kMassID = _auto
define kYMaxID = 1000
```

The definition is valid not only in the declaration part of SQ files everywhere an integer number could appear, but also in functions defined in function blocks. If the definition is placed inside a function block, it cannot be referenced in handler declarations.

Init and Terminate Handlers

The purpose of init handlers is to provide default values for the variables and the plot options. Variables are initialized with the values returned by the init handler(s); other variables (those not enumerated in the left-hand part of init handler declarations) are set to the empty array []. Initial plots are set up with commands `subplots`, `subplotprops`, `subplotpos`, `scalesync` or `subplotsync`, and `subplotparam`. By default, the first figure is displayed. The same functions can be used without any argument to retrieve the corresponding values; this is especially useful to store these settings in a data file.

Example

```
function [A,B,R,S] = init
  // default values for A, B, R and S
  A = [1,1];
  B = 1;
  R = 1;
  S = 5;

  // default plots
  subplots('roots\tstep');
    // kind of plots, separated with tabs and line feeds
    // (one row of two subplots)

  props = [0,-2,0,-1,1         // lin. scale, zoom on [-2,0,-1,1]
           0,nan,nan,nan,nan]; // default lin. scale
  subplotprops(props);
```

When variables are initialized separately, i.e. when the init handler does not return multiple output arguments, and the initialization expression does not depend on another variable, initialization can be done in the variable declaration statement. The following declaration

```
variable x = 2
```

is equivalent to

```
variable x
init x = 2
```

Should resources be allocated by the init handler, a terminate handler can be defined to release them. This is necessary only if commands with side effects are used, for instance if a file is opened by the init handler. The terminate handler is declared like other handlers. Output variables are useless, because the variables are discarded immediately after its execution.

```
terminate termFn(var)
```

Example

Here is an example of init and terminate handlers which open and close a file, respectively. Other handlers can read from or write to the file. In this example, a custom input handler should be written to avoid resetting the value of the file descriptor `fd` (see below).

```
variable fd

init fd = init
terminate terminate(fd)
```

```
functions
{@
function fd = init
    fd = fopen('data.txt', 'rt');

function terminate(fd)
    fclose(fd);
@}
```

Multiple init and terminate handlers can be declared; they are all executed in the same order as they are declared.

Resize handler

The resize handler is a function called when the dimensions of the area where the figures are displayed are changed. This occurs typically when the user resizes the window. The resize handler is declared with the keyword `resize`. Two special input arguments can be used to retrieve the new size of the subplots area:

Name	Description
`_height`	height of the subplots area in pixels
`_width`	width of the subplots area in pixels

The height and width are usually given in pixels for the screen, and in points (1/72 inch) for high-resolution devices such as printers.

The purpose of the resize handler is to change the subplot layout, for instance to reduce their number when the display area is too small. The resize handler is not called at startup; arguments `_height` and `_width` can also be passed to the init handler.

Example

In the fragment of SQ file below, the subplot layout is set in the function `setSubplots`, which is called as an init handler at startup and as a resize handler when the window dimensions are changed. Depending on the width of the window, one or two figures are displayed. A separate init handler is declared for variable initialization.

```
variable a, b, c
init (a, b, c) = initVar
init setSubplots(_width)
resize setSubplots(_width)
...

functions
{@
function (a, b, c) = initVar
  ...
```

```
function setSubplots(width)
  if width > 300
    subplots('Fig. 1\tFig. 2');
  else
    subplots('Fig. 1');
  end
...
@}
```

Figure Declaration

Each figure is declared by a figure declaration, introduced by the `figure` keyword:

```
figure "Figure Name"
```

The string which follows the `figure` keyword is the figure name, which can contain any character and space. It is displayed as a title above the figure and in the Plots menu, and is used by the `subplots` command as an identifier. The figure declaration is followed by the draw, mousedown, mousedoubleclick, mousedrag, mousedragcont, mouseup, mouseover, mouseout, mousescroll, dragin, and dragout handlers corresponding to the figure. Only the draw handler is required.

The Plots menu follows the order of the figure declarations. Separators (horizontal lines in the menu) can be inserted to make the menu easier to read, with the `separator` keyword.

Related figures can be grouped in submenus. Figure entries are enclosed between `beginsubmenu "`*name*`"` and `endsubmenu` lines, where *name* is the name of the submenu.

Example

```
figure "Input Signal"
  draw drawInputSignal(u);
  mousedrag u = dragInputSignal(u, _id, _y1);
  mouseover _cursor = overInputSignal(_id)

figure "Output Signal"
  draw drawOutputSignal(y);

separator

figure "Model"
  draw drawModel(u, y);

separator

beginsubmenu "Response"
```

```
  figure "Impulse"
    draw drawImpulse(u, y)

  figure "Step"
    draw drawStep(u, y)

endsubmenu
```

Draw Handler

Each figure has one draw handler, declared with the `draw` keyword:

```
draw drawFn(v1, v2, ...)
```

The handler draws the figure with graphical commands such as `plot`, `circle`, or `step`. The `scale` command may be used to set the default scale and the scale options.

The draw handler typically has input arguments; but it never has any output argument. It also accepts the special input argument `_param` (see below).

Mousedown, Mousedoubleclick, Mousedrag, Mousedragcont, Mouseup, Mouseover, and Mouseout, and Mousescroll Handlers

The mousedown, mousedoubleclick, mousedrag, mouseup, mouseover, mouseout handlers are called when the mouse is over the figure they are defined for and the mouse button is pressed, double-clicked, held down, released, left unpressed, or move outside respectively. The mousedragcont handler is an alternative to the mousedrag handler (see below). The mousescroll handler is called when the mouse if over the figure and the scroll wheel or scroll ball is moved. The dragin and dragout handlers are used for drag operations between different figures. The table below summaries their differences. For example, the mousedrag handler accepts variables (declared with the `variable` keyword) as input and output arguments; it can set the special variable `_msg` with an output argument, and its role is to modify variables during a mouse drag.

Handler	**Var.**	**_msg**	**_cursor**	**Role**
mousedown	in/out	-	-	Prepares a drag
mousedoubleclick	in/out	-	-	2nd click of a dble-click
mousedrag	in/out	out	-	Performs a drag
mousedragcont	in/out	out	-	Performs a drag
mouseup	in/out	-	-	Cleans up after a drag
mouseover	in/out	out	out	Gives visual feedback
mouseout	in/out	-	-	Cleans up visual fdback
mousescroll	in/out	out	-	Performs a change

The purpose of the mouseover handler is to give to the user visual feedback about what is below the cursor, with a message or the shape of the cursor. It is also possible to change the variable, and though them all the displayed graphics, when the mouse is moved over a figure. This should be used only to give some additional hint about what is below the cursor (such as highlighting a matching element in another graphics), not as a substitute of mousedrag, because the user has no way to select what he wants to manipulate. The mouseout handler should be used to restore the state of the variables.

The purpose of the mousedown, mousedoubleclick, mousedrag, mousedragcont, and mouseup handlers is to handle interactions. They receive as input arguments the position of the mouse and the value of variables, and return as output arguments new values for the variables. Unless an error occurs, the mousedown, mousedrag, mouseup, and draw handlers are called with the following arguments. The placeholder `S0` represents the set of variables before the mouse down, and `S1` the set after the mouse up.

```
S0 = mousedown(S0)
S0 = draw(S0)
S1 = S0
while the mouse button is down
  S1 = mousedrag(S0)
  S1 = draw(S1)
end
S1 = mouseup(S0)
S1 = draw(S1)
```

The Undo command discards `S1` and reverts to `S0`. Hence the changes caused by mousedown should be limited to computing auxiliary values used for the drag operation, but invisible to the user. If an error occurs, the sequence is aborted, and `S1` is discarded.

The sequence above can be reduced to much simpler forms. For dragging an element, it is often sufficient to define mousedrag and draw:

```
while the mouse button is down
  S1 = mousedrag(S0)
```

```
  S1 = draw(S1)
end
```

To change the controller on a simple mouse click, only the mouseup handler is used (as explained above, the mousedown function does not change S1).

```
S1 = mouseup(S0)
S1 = draw(S1)
```

The mousedrag handler is always executed immediately after the mousedown handler if there is one, even if the mouse is not moved. To display some information without changing anything when you drag the mouse and remove any trace afterwards, you can define a mousedrag handler which sets variables used by the draw handlers, then discard S1 with a mouseup handler which just contains the `cancel` command.

The mousedrag handler uses as input arguments the values the variables had before the drag. After the drag, the variables will be affected only by the position of the mouse at the beginning and at the end of the drag operation; the trajectory of the mouse is lost. This behavior is often desirable. In the infrequent cases where it is not, the mousedrag handler should be replaced with a mousedragcont handler. Sysquake calls the handlers in the following sequence:

```
S0 = mousedown(S0)
S0 = draw(S0)
S1 = S0
while the mouse button is down
  S1 = mousedragcont(S1)
  S1 = draw(S1)
end
S1 = mouseup(S0)
S1 = draw(S1)
```

The only difference with the mousedrag handler is that mousedragcont handlers use the new values of the variables as input arguments and modify them continuously. They can record the whole trajectory of the mouse (or any information derived from the trajectory). For example, a mousedragcont handler could be used to draw in a pixmap.

The mousedoubleclick handler is called when the mouse button is pressed down for the second time shortly after the first one. The mousedown, mousedrag, mouseup and draw handlers (if they exist) have been called for the first click.

The purpose of the mousescroll handler is to to change some quantity related to a figure or an object. Since all mouses do not have wheels, tracking balls, or other scrolling device, the mousescroll handler should be a shortcut to some other way of performing the same

action, for instance with a dialog box or a slider. This is espacially true for horizontal scroll which is not supported by most wheels. Note also that some wheels have a coarse resolution.

Predefined variables

In addition to the variables defined with the `variable` keyword, you can use the following predefined variables as input argument for the mousedown, mousedoubleclick, mousedrag, mouseup, and mouseover handlers.

Name	Purpose
`_z`	initial position of the mouse as a complex number
`_x`	initial horizontal position of the mouse
`_y`	initial vertical position of the mouse
`_rho`	initial distance between the position of the mouse and the origin
`_theta`	initial angle of the vector from the origin to the position of the mouse
`_z0`	initial position of the clicked element as a complex number
`_x0`	initial horizontal position of the clicked element
`_y0`	initial vertical position of the clicked element
`_p0`	initial position of the clicked element as a 2D or 3D vector
`_rho0`	initial distance between the position of the clicked element and the origin
`_theta0`	initial angle of the vector from the origin to the position of the clicked ele
`_z1`	current position of the mouse as a complex number
`_x1`	current horizontal position of the mouse
`_y1`	current vertical position of the mouse
`_p1`	current position of the mouse as a 2D or 3D vector
`_rho1`	current distance between the position of the mouse and the origin
`_theta1`	current angle of the vector from the origin to the position of the mouse
`_str1`	current string parameter
`_dx`	horizontal displacement (`_x1-_x`) or horizontal scrolling
`_dy`	vertical displacement (`_y1-_y`) or vertical scrolling
`_dz`	displacement (`_z1-_z`) or scrolling as a complex number
`_kx`	factor the horizontal position is multiplied by (`_x1/_x`)
`_ky`	factor the vertical position is multiplied by (`_y1/_y`)
`_kz`	complex factor the position is multiplied by in the complex plane (`_z1/_z`
`_q`	additional data specific to the plot
`_m`	true if the modifier key (Shift key) is held down
`_id`	ID of the manipulated object
`_nb`	number of the manipulated trace (1-based)
`_ix`	index of the manipulated point (1-based)
`_param`	subplot parameter

The value of `_z0` is constrained to existing elements of the plot, contrary to `_z` which represents the actual position of the mouse click. The value of `_z1` is not constrained. If you need the original position of the manipulated object, you should usually use `_z0` (or `_x0` or `_y0`) and replace it with `_z1`. The value of `_z` is used when the amplitude of

the move is considered, not the initial and final positions, or when no object is selected.

The value of `_id` is the value you define in plot commands for elements you want to manipulate. You can use it to recognize different graphical objects in the same figure. The value of `_nb` defines which line is manipulated among those plotted by the same command (most graphical commands can draw multiple lines or multiple responses). The value of `_ix`, an integer starting from 1, is always defined when the user selected an element of the figure; it is useful mainly for traces where you define explicitly each point, e.g. `plot` and `line`. Note that you need not use IDs; all clicks generate calls to the mouse handler(s), and `_z`, `_z1`, `_x`, `_x1`, `_y`, and `_y1` are always defined.

For each subplot, Sysquake maintains a piece of data which can be used freely for any purpose. This data is passed to and from all figure-related handlers with `_param`. It is initialized to the empty array `[]`. It is useful in cases where the same figure is displayed in different subplots; `_param` may contain for instance initial conditions for a simulation. It may also be used as input or output argument of menu, import and export handlers and in their `_enable` and `_checkmark` expression; in these cases, the handlers are enabled when a single subplot is selected.

As output argument of the mousedrag and mouseover handlers, you can use the following special variables:

Name	**Purpose**
`_msg`	string displayed in the status bar at the bottom of the window
`_param`	new value of the subplot parameter

The variable `_msg` is typically set to describe what is below the cursor, with possibly some numeric values using `sprintf`. If it contains a linefeed (`'\n'`) or carriage return (`'\r'`) character, it is truncated.

As output argument of the mouseover handler, you can also use the following special variable:

Name	**Purpose**
`_cursor`	true to display the cursor as a finger in manipulate mode

The shape of the cursor gives a hint to the user whether he can manipulate the object below the cursor. This can makes the use of SQ files much easier. By default in manipulate mode, the finger cursor is displayed in figures where mousedown, mousedrag and/or mouseup handlers are defined; otherwise, the cursor is the standard arrow. Sliders, buttons, and other controls (see command `slider`) are a special case; the finger is displayed only when the cursor is over a slider. In case you define mouse handlers, though, you often want to specify explicitly whether any manipulation is possible.

Example

Here are handlers for displaying the roots of a polynomial A that you can manipulate.

```
variable A = poly([-1, -2-2j, -2+2j])

figure "Roots"
  draw drawRoots(A)
  mousedrag A = dragRoots(A, _z0, _z1)
  mouseover _cursor = overRoots(_id)

functions
{@
function drawRoots(A)
  // plot the roots of polynomial A
  plotroots(A, 'x', 1);
  scale lock;

function A = dragRoots(A, z0, z1)
  // if the click is too far away from a root of A, z0 is empty
  if isempty(z0)
    cancel;                 // discard the dragging operation
  end
  A = movezero(A, z0, z1);  // move the root

function cursor = overRoots(id)
  // displays a finger if the cursor is over a root,
  // and a plain cursor otherwise
  cursor = ~isempty(id);
@}
```

One thing to note about cursors, if your mouseover handler declaration specifies `_cursor` as the (or one of the) output(s), and your mouseover handler is canceled by `cancel`, the cursor is set to the plain arrow. Hence you can have code like

```
if isempty(id)
  cancel;
end
```

early in your mouseover handler.

Here is an example which shows how the mousedragcont handler can be used to accumulate the position of the mouse during a drag.

```
variable x, y  // position of the mouse

figure "Plot"
  draw draw(x, y)
  mousedragcont (x, y) = drag(x, y, _x1, _y1)
```

```
functions
{@

function draw(x, y)
  // display the trace of the mouse
  scale('equal', [0,10,0,10]);
  plot(x, y);

function (x, y) = drag(x, y, _x1, _y1)
  // add the current position of the mouse to x and y
  x(end + 1) = _x1;
  y(end + 1) = _y1;
@}
```

Handlers for a specific ID

Instead of a single handler which is called when required whatever there is under the mouse, you can restrict the handler to a specific ID value. The handler is called only if the mouse is over an object with that ID. You can have multiple handlers of the same type with different ID. This can simplify the code: the handler does not have to check that the ID is not empty and has the correct value, and multiple handlers can have a smaller number of arguments.

The ID is placed directly after the handler keyword, either as an integer or a constant name defined with `define`. The example below is a complete SQ file where you can move two points, the first one horizontally and the second one vertically. Note that the mousedrag handlers are so simple they do not need a function. Since there are mouseover handlers with specific ID, but no generic one, the cursor is displayed as a finger only over one of the points.

```
variable p1x = 0.3
variable p2y = 0.6

define kP1Id = 1
define kP2Id = 2

figure "Points"
  draw drawPoints(p1x, p2y)
  mousedrag kP1Id p1x = _x1
  mouseover kP1Id _msg = overP1
  mousedrag kP2Id p2y = _y1
  mouseover kP2Id _msg = overP2

functions
{@
function drawPoints(p1x, p2y)
  scale([0, 1, 0, 1]);
  plot(p1x, 0.5, 'x', kP1Id);
```

```
  plot(0.5, p2y, 'o', kP2Id);

function msg = overP1
  msg = 'Point 1';

function msg = overP2
  msg = 'Point 2';
@}
```

Fighandler Handler

Separate handlers for the different events which can occur result typically in small functions. While the implementation of these functions can be very simple, code reuse is complicated: only the function definitions can be stored in separate libraries, while the multiple handler declarations (often at least a draw handler, a mousedrag handler and a mouseover handler) must be inserted in the SQ file.

Fighandler handlers replace all the handlers related to figures, i.e. draw, mousedown, mousedoubleclick, mousedrag, mousedragcont, mouseup, mouseover, mouseout, mousescroll, dragin, and dragout. In the handler, the action to perform is given by `_event`, which is typically used in a `switch` construct. Upon events to be ignored, `cancel(false)` should be executed.

Example

Here is the example for figure "Roots" given above, rewritten with a single fighandler function instead of separate draw, drag and over handlers.

```
variable A = poly([-1, -2-2j, -2+2j])

figure "Roots"
  fighandler A = figRoots(A)

functions
{@
function A = figRoots(A)
  switch _event
    case 'draw'
      plotroots(A, 'x', 1);
      scale lock;
    case 'drag'
      if isempty(_z0)
        cancel;
      end
      A = movezero(A, _z0, _z1);
    case 'over'
      _cursor(~isempty(_id));
```

```
    otherwise
      cancel(false);
  end
@}
```

Menu Handler

The interactive manipulation of graphical elements is not suited for all the modifications of the variables. To change the structure of a controller or to specify numeric values, a command triggered by a menu entry, possibly with the help of a dialog box, is more convenient. This is the purpose of menu handlers, which are installed in the Settings menu. Menu handlers are declared as follows:

```
menu "Title" (out1, out2, ...) = function(in1, in2, ...)
```

The string is the menu entry title. The function typically gives new values to one or more variables. If numeric values must be entered or modified by the user, or if a confirmation is required, the `dialog` command should be used. To cancel the action, the `cancel` command must be used to prevent a new set of variables from being created in the Undo buffer.

Like figures, menu entries are listed in the Settings menu in the same order as their declaration in the SQ file. Separators can be added with the `separator` keyword, and entries can be grouped in submenus with `beginsubmenu` and `endsubmenu` keywords. In addition, instead of the default Settings menu, menu entries can be grouped in different menus with the `beginmenu` and `endmenu` keywords. On versions of Sysquake which do not support multiple menus, these keywords are also accepted for compatibility; separators are inserted automatically.

The appearance of the menu entries can be modified in two ways: they can be disabled (they are written in gray and cannot be selected) with the `_enable` keyword, and they can be decorated with a check mark with the `_checkmark` keyword. Both keywords must appear between the menu handler title and the handler function declaration. They use a single LME boolean expression (typically based on the variables) as argument.

Examples

In the following example, the variable `color` has the value 0 for black, 1 for blue and 2 for red.

```
variable color, polygon, sides

init color = 0
init polygon = 0
init sides = 3
```

```
beginsubmenu "Color"
  menu "Black" _checkmark(color==0) color=0
  menu "Blue" _checkmark(color==1) color=1
  menu "Red" _checkmark(color==2) color=2
endsubmenu
separator
menu "Polygon"
     _checkmark(polygon) polygon=toggle(polygon)
menu "Number of Sides..."
     _enable(polygon) sides=setNumberOfSides(sides)

functions
{@
function b = toggle(b)
  b = ~b;

function sides = setNumberOfSides(sides)
  (ok, sides) = dialog('Number of sides:', sides);
  if ~ok
    cancel;
  end
@}
```

In the fragment below, two menus are declared:

```
beginmenu "Color"
  menu "Black" _checkmark(color==0) color=0
  menu "Blue" _checkmark(color==1) color=1
  menu "Red" _checkmark(color==2) color=2
endmenu
beginmenu "Parameters"
  menu "Polygon"
     _checkmark(polygon) polygon=toggle(polygon)
  menu "Number of Sides..."
     _enable(polygon) sides=setNumberOfSides(sides)
endmenu
```

Sysquake will display two menus whose titles are "Color" and "Parameters".

Keydown Handler

To react to a key pressed down, multiple specific keydown handlers and a single default keydown handler can be installed. Specific keydown handlers are declared as follows:

```
keydown "k" (out1, out2, ...) = function(in1, in2, ...)
```

The string "k" contains the key which triggers the handler when pressed. Most keys are denoted by the corresponding character; arrow keys are denoted by `"up"`, `down`, `left`, and `right`.

The default keydown handler is declared as follows:

```
keydown (out1, out2, ...) = function(in1, in2, ...)
```

It is triggered if a key without a specific keydown handler is pressed. In both kinds of keydown handlers, the function typically gives new values to one or more variables. In addition to the variables defined with the `variable` keyword, you can use the following predefined variable as input argument:

Name	**Purpose**
_key	key pressed as a string

Example

In the following example, the variable `n` is incremented or decremented when the user type "+" or "-", respectively. When another key is pressed, `cancel` is executed, so that no new undo frame is created.

```
variable n

init n = 1
keydown n = keydownHandler(n, _key)

functions
{@
function n = keydownHandler(n, key)
  switch key
    case '-'
      n = n - 1
    case '+'
      n = n + 1
    otherwise
      cancel;
  end
@}
```

Make Handler

It is often useful to have variables which hold the result of a computation based on other variables. If several figures depend on it, this avoids the need to calculate it in each draw handler and reduce the computation time. The computation may be performed in the handlers which change the independent variables. However, this forces

to add arguments; handlers become more difficult to write, especially when dependencies are complicated. Make handlers describe the way to calculate variables not as the direct effect of some user action, but when their output arguments are required by another handler. The name *make* is borrowed from the programming utility which builds a complicated project based on a set of rules and dependencies. Make handlers do the same for variables. Their declaration is simply

```
make (output_variables) = makeHandler(input_variables)
```

Suppose that a variable x is changed, and that the draw handler of a figure uses variable y as input argument. If the following make handler is declared,

```
make y = f(x)
```

Sysquake calls the function f to compute y based on x. More complex dependencies may be defined using several make handlers. There must not be circular dependencies; no variable may depend on itself through one or more make handlers.

When a make handler gives the value of a single variable, it can be specified in the variable declaration statement. The following declaration

```
variable x, y
variable z = x + y
```

is equivalent to

```
variable x, y, z
make z = x + y
```

Examples

The examples below represent three different ways to implement the same behavior. A variable x may be modified either in a menu handler or by interactive manipulation of a figure. Two figures uses variable y, which is a function of x.

The first implementation does not store y in a variable declared to Sysquake. It is computed in each draw handler.

```
variable x
init x = init
menu "Change x" x = changeX(x)
figure "Figure 1"
  draw drawFig1(x)
  mousedrag x = dragFig1(_x1)
figure "Figure 2"
  draw drawFig2(x)
```

```
functions
{@
function x = init
  x = 3;
function x = changeX(x)
  // (dialog box to let the user edit x)
function drawFig1(x)
  y = f(x);
  // (plot based on x and y)
function x = dragFig1(x1)
  x = x1;
function drawFig2(x)
  y = f(x);
  // (other plot based on x and y)
function y = f(x)
  // computation of y
@}
```

The second implementation declares y, which is computed in all handlers which change the value of x. When both figures are displayed, y is computed only once when the menu or mousedrag handler is invoked.

```
variable x, y
init (x, y) = init
menu "Change x" (x, y) = changeX(x)
figure "Figure 1"
  draw drawFig1(x, y)
  mousedrag (x, y) = dragFig1(_x1)
figure "Figure 2"
  draw drawFig2(x, y)
functions
{@
function (x, y) = init
  x = 3;
  y = f(x);
function (x, y) = changeX(x)
  // (dialog box to let the user edit x)
  y = f(x);
function drawFig1(x, y)
  // (plot based on x and y)
function (x, y) = dragFig1(x1)
  x = x1;
  y = f(x);
function drawFig2(x, y)
  // (other plot based on x and y)
function y = f(x)
  // computation of y
@}
```

The third implementation also declares y, which is computed by a

make handler when it is needed by another handler. Note how each handler is as simple as it can be. If none of the two figures is displayed, y is not computed at all.

```
variable x, y
init x = init
make y = f(x)
menu "Change x" x = changeX(x)
figure "Figure 1"
  draw drawFig1(x, y)
  mousedrag x = dragFig1(_x1)
figure "Figure 2"
  draw drawFig2(x, y)
functions
{@
function x = init
  x = 3;
function x = changeX(x)
  // (dialog box to let the user edit x)
function drawFig1(x, y)
  // (plot based on x and y)
function x = dragFig1(x1)
  x = x1;
function drawFig2(x, y)
  // (other plot based on x and y)
function y = f(x)
  // computation of y
@}
```

Function Definitions

Functions declared by the different handlers must be built-in, defined in separate libraries, or defined in one or several function blocks.

A library (.lml file) must be referenced with the `use` keyword:

```
use library
```

The library name does not include the file suffix `.lml`. Libraries are searched in the same folders or directories as SQ files.

Functions defined directly in the SQ files are placed in a function block, with the following syntax:

```
function
{@
...
@}
```

The keyword `functions` is a synonym of `function`. The functions can be defined in any order and in any number of blocks. In addition

to functions declared as handlers, other functions can be defined to extend the set of built-in commands.

Function blocks can also include `use` statements. Whether to place `use` statements outside or inside function blocks is a matter of style. SQ-level `use` statements should be preferred for libraries which define functions called directly as handlers.

Embedded files

In some cases, especially when a large amount of constant data is required in an SQ file, it may be convenient to store these data in the SQ file itself and to use the standard input functions of LME, such as `fread` and `fscanf`, to retrieve them. This avoids the need of separate files. Blocks of texts, introduced by keyword `embeddedfile`, provide such a facility:

```
embeddedfile "name"
{@
Embedded file contents...
@}
```

The contents of the block may be as large as required. They can be either plain text, or binary data encoded with base64. The name is used to identify the embedded file; it corresponds to the argument of function `efopen`. The number of embedded files in an SQ file is not limited.

Title, Version, and Help

SQ files may include a title, version information, and explanations about their purpose and how they can be used. The title is specified by a string:

```
title "..."
```

It is used instead of the file name in some windows and menus titles.

The version and help are provided in blocks of text:

```
version
{@
Text...
@}

help
{@
Text...
@}
```

In the block of text, paragraphs are separated by one or more empty lines. Initial and trailing empty lines and spaces are ignored. For cases where preformatted text is preferred, such as for program code or equations, the HTML tags <pre>/</pre> are used. Here is an example:

```
help
{@
Here is an identity matrix:

<pre>
    [ 1 0 0 ]
I = [ 0 1 0 ]
    [ 0 0 1 ]
</pre>
@}
```

Upon user request, Sysquake displays the version or the help of the current SQ file.

The purpose of the version text is to give any version number, release date, and copyright information relevant to the SQ file.

User interface

The standard user interface of Sysquake has menus to customize the layout of figures and their options. Some of them can be disabled once the layout has been carefully tuned. This is done with the `userinterface` keyword, followed by comma-separated options:

```
userinterface  option1,  options, ...
```

Here is the list of supported options:

Name	**Purpose**
`appmenus` (default)	Standard menus
`noappmenus`	No standard menus
`figoptions` (default)	Submenu "Figure>Options" for margin, title, etc.
`nofigoptions`	No submenu "Figure>Options"
`plotchoice` (default)	Menus "Plots" and "Layout"
`noplotchoice`	No menus "Plots" and "Layout"
`resize` (default)	The figure window can be resized
`noresize`	The figure window has a fixed size
`selectall` (default)	Menu entry "Edit>Select All"
`noselectall`	No menu entry "Edit>Select All"
`toolbar` (default)	Figure toolbar
`notoolbar`	No figure toolbar

Depending on the version of Sysquake, the menus or menu entries are disabled or completely removed. Option `noplotchoice` also

disables the dragging of subplots. Option `noappmenus` removes most menus and menu entries.

To have finer control on exactly which plot options should be available, the Figure menu should be removed with `nofigoptions` and the appropriate options added back in menus with `sqguicmd`. The code below shows a reduced menu with logarithmic scale for x and y axis, and labels and legends.

```
userinterface nofigoptions

beginmenu "Options"
  menu "Log x"
    _enable(sqguicmd('scale/log-x','e'))
    _checkmark(sqguicmd('scale/log-x','c'))
    sqguicmd('scale/log-x')
  menu "Log y"
    _enable(sqguicmd('scale/log-y','e'))
    _checkmark(sqguicmd('scale/log-y','c'))
    sqguicmd('scale/log-y')
  separator
  menu "Label"
    _enable(sqguicmd('figure/label','e'))
    _checkmark(sqguicmd('figure/label','c'))
    sqguicmd('figure/label')
  menu "Legend"
    _enable(sqguicmd('figure/legend','e'))
    _checkmark(sqguicmd('figure/legend','c'))
    sqguicmd('figure/legend')
endmenu
```

9.2 SQ Data Files and Input/Output Handlers

SQ Data files (or SQD files) are composed of a succession of LME statements which define variables. The contents of these variables are used by an input function which translates them to the variables used by Sysquake itself and which restores the subplots and their settings. This filtering step serves two purposes:

- Sysquake can use more or other variables than what the SQ Data files define. For instance, a proportional-integral-derivative controller (PID) could be defined by three parameters, but described by a transfer function in the Sysquake variables.

- A validity check is often useful, because the data files are text files which can be written by the user.

To avoid the hassle to write input and output handlers when these advanced features are not needed, Sysquake has a default behavior if the handlers are not declared in the SQ file. If the output function is missing, all the variables and the settings are written to the file. If the input function is missing, the variables declared in the SQ file are set to the values defined in the SQD file, or to the empty array `[]` if they are not found.

To permit the user to simply open the SQ data file without specifying the SQ file which can make use of it, the first line should form a valid LME comment with the name of the associated SQ file:

```
%SQ sqfile.sq
...
```

When the user opens a file, the first line is read. If it corresponds to a data file, the associated SQ file is read and its init function processed. Finally, the data file is executed and the contents of its variables converted by the input function. If the file opened by the user is a SQ file, it is read and its init function executed.

A typical data file could be

```
%SQ greatDesign.sq

A = [1,2];
B = 2;
kp = 20;
_plots = 'greatView\tstep';
_plotprops = [0,-1,1,-1,1;
0,10,50,0,0];
```

Variables `A`, `B`, and `kp` correspond directly to variables defined in the SQ file. Variable `_plots` is the list of the subplots which were displayed when the file was saved; it corresponds to the input or output argument of the command `subplots`. Variable `_plotprops` is the properties of the subplots (options like logarithmic scaling and grids, and zoom); it corresponds to the input or output argument of the command `subplotprops`. The names `_plots` and `_plotprops` are those used by the default input and output handlers; it is better to use them also if you write your own handlers.

Input and Output Handlers

To generate and read back the data file above, the following handlers can be declared:

```
output outputHandler(_fd, v1, v2, ...)
input (v1, v2, ...) = inputHandler(_fd)
```

The output handler must write to file descriptor `_fd` the contents of the variables. You need not save all the variables; for example, if `pol` is a polynomial and you have defined a variable `r` to store its roots, saving `pol` is enough; you can restore `r` in the input handler with `r=roots(pol)`. The variable `_fd` represents the file descriptor of the output file:

Name	**Purpose**
`_fd`	file descriptor of the output or input file

Like all special arguments, `_fd` can also be used directly in the function definitions, without being passed from the handler declaration.

You can use it with functions such as `fprintf` and `dumpvar`. You can write the data using any format. However, it is better to follow the format of standard SQD files, as described above, with perhaps calls to the built-in functions of LME. The first line, which contains the name of the SQ file, is required to permit Sysquake to find the appropriate SQ file.

The input handler reads the SQD file using the file descriptor `_fd`, and produces values stored in Sysquake variables. The input handler is always called after the init handler; variables which are not output arguments of the input handler keep the value set by the init handler.

If the SQD file contains variable assignments, the easiest way to parse it is to read the whole file with `fread`, and to interpret the string with `sandbox`.

The following example shows how to write input and output handlers which save and restore three variables, check the validity of the variables in the SQD file, and save and restore the subplots.

```
output myOutputHandler(A,B,kp)
input (A,B,kp) = myInputHandler()

function
{@
function myOutputHandler(A,B,kp)

    // write header line
    fprintf(_fd, '%SQ greatDesign.sq\n\n');

    // write variables
    dumpvar(_fd, 'A', A);
    dumpvar(_fd, 'B', B);
    dumpvar(_fd, 'gain', kp);  // different name

    // write current subplot settings
    dumpvar(_fd, '_plots', subplots);
    dumpvar(_fd, '_plotprops', subplotprops);
    if ~isempty(subplotpos)
        dumpvar(_fd, '_plotpos', subplotpos);
```

```
    end

function (A,B,kp) = myInputHandler()

 // read and interpret SQD file into struct s
 str = fread(_fd, inf, '*char');
 s = sandbox(str);

    // map the SQD variables to the SQ variable
    kp = s.gain;

    // check the validity of the transfer function
    if length(s.A) >= length(s.B)
        dialog(['The transfer function of the system B/A ',
            'must be strictly causal.');
        cancel;
    end
 A = s.A;
 B = s.B;

    // restore subplot settings
    subplots(s._plots);
    subplotprops(s._plotprops);
    subplotpos(s._plotpos);
@}
```

The contents of what might be a data file can also be written in the data block of a SQ file. The user may find more convenient to save the data in the SQ file itself, either to have an SQ file with new default values or to distribute it to other people. The data block should be at the end of the SQ file, so that all the variables and the functions used by the input handler are defined. You can write it manually, but its purpose is to be written by Sysquake when the user chooses to save the data as an SQ file. Its syntax is

```
data
{@
 contents of a data file
@}
```

9.3 Error Messages

Here is the list of error messages; some of them may appear in a message box when you read a bad file, but many are internal errors. LME errors are listed in the LME Reference chapter and are caused by a bad function either at compile time or at execution time.

Exhausted resources Not enough memory for the current operation.

Too small name buffer Not enough memory for the table of variable names.

No more name slot Too many variable names.

Too small Sysquake buffer Not enough memory to launch Sysquake.

Not enough memory for decoding SQ file Class file too complex.

Too small Undo buffer Not enough memory to create the Undo structures.

Too many variables for the Undo buffer Undo structures too small for the large number of variables.

Variable too large for the Undo buffer The new value given to a variable is too large.

Variable not found Attempt to retrieve a variable which has not been defined (should never occur).

Variable already exists Attempt to redefine a variable.

Standard variable unexpected here A standard variable, such as _x1, is not supported here.

Syntax error in class file Unbalanced quotes or parenthesis, missing element, or other syntax error in the class file.

Bad block of text Block of text (such as a block of function definitions) without end mark.

Bad variable definition The variable keyword is not followed by a valid variable name.

Bad function definition The function keyword is not followed by a valid function call.

Undefined element The keyword is unknown.

Nonexistent set of variables in the Undo buffer Attempt to revert to an undo state which is not available.

Not a data file The file does not begin with the expected "% SQ" characters.

Cannot undo Too many attempts to undo.

Cannot redo Too many attempts to redo.

9.4 Advanced Features of SQ Files

This section describes the more advanced features of SQ files.

Dragin and Dragout Handlers

The dragin and dragout handlers are used for drag operations between different figures. A dragin handler must be defined in the scope of the figure which can be the target of cross-figure drag operations, and a dragout handler must be defined in the scope of the figure from where cross-figure drag operations are initiated.

No dragin or dragout is called unless a drag operation starts from a figure which has a dragout handler (source) and the mouse is moved over a figure which has a dragin handler (target). In that case, when the mouse enters the target figure, the source dragout handler is called first, then the target dragin handler. Then the dragging continues exactly as if it had started from the target figure, with calls to the mousedrag, mousedragcont, and/or mouseup handlers of the target figure. Other cross-figure drag operations can follow if the first target figure has a dragout handler.

Here is the sequence of calls during a drag, extended to take into account one or several drags between figures.

```
Sw = S0
Sw = mousedown_i(Sw)
Sw = draw_i(Sw)
S1 = Sw
while the mouse is down
  while the mouse in figure i
    S1 = mousedrag_i(Sw)
    S1 = draw(S1)
  end
  if the mouse is in figure j
    Sw = dragout_i(Sw)
    Sw = dragin_j(Sw)
    i = j
  end
end
S1 = mouseup_i(Sw)
S1 = draw(S1)
discard Sw
```

Example

Here is an example where a red cross can be moved between Figure One and Figure Two. The location of the cross is stored in Sysquake variables `x` and `y` for its coordinates, and `i` for the figure (1 for Figure One, 2 for Figure Two).

```
variable i, x, y

init (i, x, y) = init

figure "Figure one"
  draw draw(1, i, x, y)
  mousedrag (x, y) = drag(1, i, _id, _x1, _y1)
  dragin i = dragin(1)
  dragout dragout(1)

figure "Figure two"
  draw draw(2, i, x, y)
  mousedrag (x, y) = drag(2, i, _id, _x1, _y1)
  dragin i = dragin(2)
  dragout dragout(2)

function
{@
function (i, x, y) = init
  i = 1; x = 0; y = 0;
  subplots('Figure one\tFigure two');

function draw(fig, i, x, y)
  scale('equal', [-5, 5, -5, 5]);
  if i == fig
    plot(x, y, 'xR', 1);
  else
    text(0, 0, '(empty)');
  end

function (x, y) = drag(fig, i, _id, _x1, _y1)
  if isempty(_id); cancel; end
  x = _x1; y = _y1;

function fig = dragin(fig)
  fprintf('dragin %d\n', fig);

function dragout(fig)
  fprintf('dragout %d\n', fig);
@}
```

Here is what happens when the cross is dragged from Figure One to Figure Two:

- Figure One's mousedrag handler is called with `fig=1` as long as the mouse is not moved over Figure Two
- When the mouse enters Figure Two, since Figure One (the source figure) has a dragout handler and Figure Two (the target figure) has a dragin handler, Sysquake calls Figure One's dragout handler with `fig=1`, then Figure Two's dragin handler with `fig=2`; the

dragin handler is declared in such a way that Sysquake variable `i` is set to the target figure number, i.e. 2. The cross is displayed in Figure Two.

– Figure Two's mousedrag handler is called with `fig=2`. Figure One's mousedrag handler is not called anymore.

A dragout operation is prevented by the dragout handler if it calls `cancel`, either without argument to abort completely the mousedrag operation or as `cancel(false)` to continue the mousedrag in the initial figure. Calling `cancel` in the dragin handler has an effect only on the drag in the second figure.

User Interface Options

By default, Sysquake lets the user of an SQ file change the layout of figures with the Layout menu, the Plots menu which is filled with all the figures defined in the SQ file, and the ability to drag figures with the mouse. The developer of the SQ file can disable these features by adding the following statement to the SQ file:

```
userinterface noplotchoice
```

The supported options of `userinterface` are enumerated in the table below:

Option	**Effect**
`figoptions`	menu items for figure options (default)
`nofigoptions`	no menu items for figure options
`noplotchoice`	fixed set of figures
`noselectall`	no menu entry "Select All"
`plotchoice`	options to change figures
`selectall`	menu entry "Select All" (default)

Languages

Some elements contain text. A single SQ file can contain multiple translations of these elements. When the SQ file is loaded, only elements corresponding to a single language are displayed; the language can be selected from a menu which enumerates all the translations it contains.

Elements declared for a specific language must be enclosed between `beginlanguage` and `endlanguage`:

```
beginlanguage "Language;code"
// declarations
endlanguage
```

The string following `beginlanguage` contains the language name (such as `Fran■ais`), followed optionally by a semicolon and the ISO 639 code of the language (such as `fr`). The language name is used in the user interface; the code of the language is used on some platforms to obtain the default language.

A single SQ file can contain multiple language specifications for the same language. When the first specification contains both the language name and the language code, successive declarations for the same language can omit the language name.

Any declaration can be placed under the scope of `beginlanguage`; then it becomes language-specific, contrary to declarations outside of any `beginlanguage` section which are valid for all the languages. Language-specific init handlers are used in a special way: contrary to non-language-specific init handlers, they are executed when the language is changed. This is especially useful to change the figures.

Instead of having different handlers for different languages, it is also possible to test in handlers which is the current language with function

Examples

In the following example, two languages are defined, English and French. Function definitions are omitted.

```
variable a, b
init (a, b) = init

beginlanguage "English;en"
  title "Amounts"
  menu "Value of a" a = setA(a)
  init initFigEn
  figure "Chart"
    draw drawChart(a, b)
endlanguage

beginlanguage "Fran■ais;fr"
  title "Grandeurs"
  menu "Valeur de a" a = setA(a)
  init initFigFr
  figure "Graphique"
    draw drawChart(a, b)
endlanguage

functions
{@
function initFigEn
  subplots('Chart');
function initFigFr
  subplots('Graphique');
```

```
// ...
@}
```

The following SQ file is equivalent, with multiple declarations for the same languages.

```
variable a, b
init (a, b) = init

beginlanguage "English;en"
  title "Amounts"
endlanguage
beginlanguage "Fran■ais;fr"
  title "Grandeurs"
endlanguage

beginlanguage "en"
  menu "Value of a" a = setA(a)
endlanguage
beginlanguage "fr"
  menu "Valeur de a" a = setA(a)
endlanguage

beginlanguage "en"
  figure "Chart"
    draw drawChart(a, b)
endlanguage
beginlanguage "fr"
  figure "Graphique"
    draw drawChart(a, b)
endlanguage

functions
{@
// ...
@}
```

Idle Handler

In addition to menu and mouse handlers, which modify Sysquake variables when the user performs some action, SQ files can define an idle handler, which is called periodically. Variables are modified without adding a new undo level. An idle handler is declared as follows:

```
idle (output_variables) = idleHandler(input_variables)
```

The rate at which the idle handler is executed depends on the operating system. It usually depends on the activity of the computer, such as the user interface or the network. You can give hints to Sysquake by adding a special output argument to the idle handler:

Name	Description
`_idlerate`	time rate, in seconds, for calling the idle handler
`_idleamount`	proportion of time for idle processing (between 0 and 1)

Only one of these arguments should be used. It should be stressed that the use of `_idlerate` does not guarantee real-time execution. Sysquake limits the time it gives to the idle handler to permit user interaction. It also enforces a maximum delay to make sure the idle handler is not blocked forever in case `_idlerate` is set to 0.

There may be only one idle handler per SQ file. It is useful when some external data can be acquired and displayed continuously, for long optimization procedures to display their progress, or for simple animations.

Example

```
variable x
init x = init
idle (x, _idlerate) = idle(x)
functions
{@
function x = init
    x = 0;
function (x, _idlerate) = idle(x)
    // increment and display x
 x = x + 1
 // should be called about every second
    _idlerate = 1;
@}
```

Import and Export Handlers

The most obvious way to exchange data between different SQ files and with other applications is with Copy/Paste in dialog boxes or via files. This has four drawbacks:

- it involves several steps;
- the amount of data is limited;
- since no common interchange format has been defined, editing (manually or via computation in the command line) may be required;
- a significant development effort is required.

Sysquake solves this problem with a generalized import/export mechanism. There are two handler types to support this mechanism, export and import, with a syntax similar to menu handlers. They are

collected respectively in submenus "Copy As" and "Paste As" in the Edit menu. Export handlers implement the conversion from variables to some well-defined LME data type, such as a matrix or a structure. Import handlers do the same in the opposite way. There is no limit to the number of import and export handlers. Handlers should be defined for complete sets of data which can easily be grasped by the user rather than variables which make sense only to the programmer: for instance, a set of points is better than their X-axis coordinates.

Import handlers are declared as

```
import "menu entry name" (outvar) = fn(invar,_xdatatype,_xdata)
```

or

```
import "menu entry name" _enabled(b) ...
        (outvar) = fn(invar,_xdatatype,_xdata)
```

The menu entry name is inserted in the Edit/Paste As submenu. The entry is enabled when the expression invoked with `_enabled` is true, or always if `_enabled` is missing. When the menu entry is selected, the handler is executed the usual way (i.e. a new undo frame is created, the handler is executed, and the changes are accepted unless an error occurs or the `cancel` function is executed by the handler). Both the `_enabled` expression and the handler arguments may include the special variable `_xdatatype`, which contains a string describing the type of the data in the clipboard (see below); the handler typically also uses as input `_xdata`, which contains the clipboard contents as an LME object.

Name	**Description**
`_xdata`	clipboard converted to an LME object
`_xdatatype`	type of the contents of the clipboard

The import handlers may be laid out like the figures and the menu entries, with separators. If an entry has an empty name, it is merged with the previous entry with a nonempty name: a single menu entry is displayed, it is enabled if any of the `_enabled` expressions is true, and the first handler for which the `_enabled` expression is true is executed. In the following example, points stored in arrays x and y are set by importing either an n-my-2 real array (type `'xy'`) or an n-by-1 complex array (type `'complex array'`).

```
import "Points" _enable(strcmp(_xdatatype,'xy')) ...
        (x,y)=importXY(_xdata)
import "" _enable(strcmp(_xdatatype,'complex array')) ...
        (x,y)=importC(_xdata)
functions
{@
function (x,y)=importXY(xdata)
```

```
  x = xdata(:,1);
  y = xdata(:,2);
function (x,y)=importC(xdata)
  x = real(xdata(:));
  y = imag(xdata(:));
@}
```

Export handlers are declared as

```
export "menu entry name" (_xdata,_xdatatype) = fn(invar)
```

or

```
export "menu entry name" _enabled(b) ...
        (_xdata,_xdatatype) = fn(invar)
```

The menu entry name is inserted in the Edit/Copy As submenu. The entry is enabled when the expression invoked with `_enabled` is true, or always if `_enabled` is missing. When the menu entry is selected, the handler is executed, but no new stack frame is created. If the handler does not through any error or cancel, the contents of `_xdata` is copied to the clipboard, as data of type `_xdatatype` if the argument exists or as native data (tab-separated array for a real matrix or text for a string) if `_xdatatype` is missing.

Format of the interchange data

There are two formats of data used for exchange with the export and import handlers: raw data and structured data. *Raw data* used when the export handler does not output an `_xdatatype` argument. They also contain text: strings are output literally, and the real part of matrices is output as tabbed arrays of numbers. In both cases, end-of-lines are platform-dependent (LF for OS X and Linux, and CRLF for Windows). Other data types (such as structures) are not supported.

Structured data represents a complex type exported with an explicit `_xdatatype`; it contains text, whose first line is

```
%SQ xdata " datatype"
```

where *datatype* corresponds directly to the value of the argument `_xdatatype` set by the export handler; the remaining line(s) is a textual representation (as generated with dumpvar) of `_xdata`.

Data types are strings which are used to ensure the compatibility between export and import handlers. Official types, defined by Calerga, do not begin with `"x-"`; user types should begin with `"x-"` to avoid type name clash in the future. Here is a list of the current official types.

```
unknown    any LME type
```

`string` string

`real array` matrix of real numbers

`complex array` matrix of complex numbers

`polynomial` polynomial whose real coefficients are given in descending powers as a row vector

`samples`

`tf s` transfer function in the Laplace domain given by a structure of two fields:

Field	Description
`num`	numerator, row vector of coefficients in descending powers of *s*
`den`	numerator, row vector of coefficients in descending powers of *s*

`tf z` transfer function in the *z* domain given by a structure of three fields:

Field	Description
`num`	numerator, row vector of coefficients in descending powers of *z*
`den`	numerator, row vector of coefficients in descending powers of *z*
`Ts`	sampling period

`ss c` state-space linear time-invariant continuous-time model with nu inputs, ny outputs and n states, given by a structure of four fields:

Field	Description
`A`	n-by-n real matrix
`B`	n-by-nu real matrix
`C`	ny-by-n real matrix
`D`	ny-by-nu real matrix

`ss d` state-space linear time-invariant discrete-time model with nu inputs, ny outputs and n states, given by a structure of five fields:

Field	Description
`A`	n-by-n real matrix
`B`	n-by-nu real matrix
`C`	ny-by-n real matrix
`D`	ny-by-nu real matrix
`Ts`	sampling period

`rst c` polynomial linear time-invariant continuous-time controller given by the structure of two or three fields described below; control signal $U(s)$ is given by $R(s)U(s) = S(s)\big(Y_{ref}(s) - Y(s)\big)$ or

$R(s)U(s) = T(s)Y_{ref}(s) - S(s)Y(s)$, where $Y_{ref}(s)$ is the system output, $Y(s)$ is the set-point, and $R(s)$, $S(s)$, and $T(s)$ are three polynomials

Field	Description
R	row vector of coefficients in descending powers of s
S	row vector of coefficients in descending powers of s
T	row vector of coefficients in descending powers of s (optional)
Ts	sampling period

`rst d` polynomial linear time-invariant discrete-time controller given by the structure of two or three fields described below; control signal $u(t)$ is given by $R(q)u(t) = S(q)\big(y_{ref}(t) - y(t)\big)$ or $R(q)u(s) = T(q)y_{ref}(t) - S(q)y(t)$, where $y_{ref}(t)$ is the system output, $y(t)$ is the set-point, and $R(q)$, $S(q)$, and $T(q)$ are three polynomials in the forward-shift operator q

Field	Description
R	row vector of coefficients in descending powers of z
S	row vector of coefficients in descending powers of z
T	row vector of coefficients in descending powers of z (optional)

`function` inline or anonymous function

xy n-by-2 array of n points [X,Y]

Public handler

Public handlers are used to interact with an SQ file from another context. In most cases, Sysquake executes handlers defined in an SQ file in the context of this SQ file. For example, Sysquake knows which SQ file instance to address when a menu handler is executed, and which Sysquake variables and figures are involved. But in some advanced applications, it can be useful to execute the handler of another instance. This is the case when you want to transmit information across two SQ file instances, or to update graphics from a separate thread created with `threadnew`. Unlike libraries, the purpose of public handlers is not code reuse, but code execution in the context of another SQ file instance to change its SQ variables and usually trigger the update of graphics.

Public handlers, like all other handlers, are implemented as a single assignment. In addition to SQ variables, the special variable `_xdata` is used as input to provide data from the caller, and as output to give back data to the caller.

Here is a typical public handler declaration, with SQ variables `in_sq` and `out_sq`:

```
publichandler "phname" (out_sq, _xdata) = phfun(in_sq, _xdata)
```

The caller of the public handler provides and gets only the data passed in `_xdata` by calling `sqcall`:

```
out_xdata = sqcall(instanceid, 'phname', in_xdata);
```

Watch handler

Watch handlers are functions called when an SQ variable has been changed, for instance to send an updated parameter to a device you want to control from an SQ file. They can be declared either for a list of variables or for all SQ variables:

```
watch {var1,var2,...} watchfun(...)
watch _all watchfun(...)
```

The watch handler is called after another handler has changed one of the watched variables. During mouse drag, calls to watch handlers are delayed until the end of the drag, even if no mouseup handler is declared, to reduce the number of calls.

The arguments of the watch handler function do not have to match the watched variables.

The watched variables themselves do not trigger calls to make handlers; but watch handler input arguments do, like any other handler.

Extension declaration

Extensions required by the SQ file can be declared as follows:

```
extension "extensionname"
```

The extension name is the name of the extension file without its suffix, such as `longint` for long integer support. Multiple extensions must be declared separately.

Currently, Sysquake Application Builder is the only application which requires the declaration of the extensions which are used. Extension declarations are ignored by other applications.

Chapter 10

LME Reference

This chapter describes LME (Lightweight Math Engine), the interpreter for numeric computing used by Sysquake.

10.1 Program format

Statements

An LME program, or a code fragment typed at a command line, is composed of statements. A statement can be either a simple expression, a variable assignment, or a programming construct. Statements are separated by commas, semicolons, or end of lines. The end of line has the same meaning as a comma, unless the line ends with a semicolon. When simple expressions and assignments are followed by a comma (or an end of line), the result is displayed to the standard output; when they are followed by a semicolon, no output is produced. What follows programming constructs does not matter.

When typed at the command line, the result of simple expressions is assigned to the variable ans; this makes easy reusing intermediate results in successive expressions.

Continuation characters

A statement can span over several lines, provided all the lines but the last one end with three dots. For example,

```
1 + ...
2
```

is equivalent to 1 + 2. After the three dots, the remaining of the line, as well as empty lines and lines which contain only spaces, are ignored.

Inside parenthesis or braces, line breaks are permitted even if they are not escaped by three dots. Inside brackets, line breaks are matrix row separators, like semicolons, unless they follow a comma or a semicolon where they are ignored.

Comments

Unless when it is part of a string enclosed between single ticks, a single percent character or two slash characters mark the beginning of a comment, which continues until the end of the line and is ignored by LME. Comments must follow continuation characters, if any.

```
a = 2;    % comment at the end of a line
x = 5;    // another comment
% comment spanning the whole line
b = ...   % comment after the continuation characters
    a;
a = 3%   no need to put spaces before the percent sign
s = '%';  % percent characters in a string
```

Comments may also be enclosed between `/*` and `*/`; in that case, they can span several lines.

Pragmas

Pragmas are directives for the LME compiler. They can be placed at the same location as LME statements, i.e. in separate lines or between semicolons or commas. They have the following syntax:

```
_pragma name arguments
```

where `name` is the pragma name and `arguments` are additional data whose meaning depends on the pragma.

Currently, only one pragma is defined. Pragmas with unknown names are ignored.

Name	**Arguments**	**Effect**
`line`	*n*	Set the current line number to *n*

`_pragma line 120` sets the current line number as reported by error messages or used by the debugger or profiler to 120. This can be useful when the LME source code has been generated by processing another file, and line numbers displayed in error messages should refer to the original file.

10.2 Function Call

Functions are fragments of code which can use *input arguments* as parameters and produce *output arguments* as results. They can be built in LME (*built-in functions*), loaded from optional extensions, or defined with LME statements (*user functions*).

A *function call* is the action of executing a function, maybe with input and/or output arguments. LME supports different syntaxes.

```
fun
fun()
fun(in1)
fun(in1, in2,...)
out1 = fun...
(out1, out2, ...) = fun...
[out1, out2, ...] = fun...
[out1 out2 ...] = fun...
```

Input arguments are enclosed between parenthesis. They are passed to the called function by value, which means that they cannot be modified by the called function. When a function is called without any input argument, parenthesis may be omitted.

Output arguments are assigned to variables or part of variables (structure field, list element, or array element). A single output argument is specified on the left on an equal character. Several output arguments must be enclosed between parenthesis or square brackets (arguments can simply be separated by spaces when they are enclosed in brackets). Parenthesis and square brackets are equivalent as far as LME is concerned; parenthesis are preferred in LME code, but square brackets are available for compatibility with third-party applications.

Output arguments can be discarded without assigning them to variables either by providing a shorter list of variables if the arguments to be discarded are at the end, or by replacing their name with a tilde character. For example to get the index of the maximum value in a vector and to discard the value itself:

```
(~, index) = max([2, 1, 5, 3]);
```

10.3 Named input arguments

Input arguments are usually recognized by their position. Some functions also differentiate them by their data type. This can lead to code which is difficult to write and to maintain. A third method to distinguish the input arguments of a function is to tag them with a name, with a syntax similar to an assignment. Named arguments must follow unnamed arguments.

```
fun(1, [2,3], dim=2, order=1);
```

For some functions, named arguments are an alternative to a sequence of unnamed arguments.

10.4 Command syntax

When a function has only literal character strings as input arguments, a simpler syntax can be used. The following conditions must be satisfied:

- No output argument.
- Each input argument must be a literal string
  - without any space, tabulator, comma or semicolon,
  - beginning with a letter, a digit or one of '-/.:*' (minus, slash, dot, colon, or star),
  - containing at least one letter or digit.

In that case, the following syntax is accepted; left and right columns are equivalent.

Command	**Function**
`fun str1`	`fun('str1')`
`fun str1 str2`	`fun('str1','str2')`
`fun abc,def`	`fun('abc'),def`

Arguments can also be quoted strings; in that case, they may contain spaces, tabulators, commas, semicolons, and escape sequences beginning with a backslash (see below for a description of the string data type). Quoted and unquoted arguments can be mixed:

```
fun 'a bc\n'      fun('a bc\n')
fun str1 'str 2'  fun('str1','str 2')
```

The command syntax is especially useful for functions which accept well-known options represented as strings, such as `format loose`.

10.5 Libraries

Libraries are collections of user functions, identified in LME by a name. Typically, they are stored in a file whose name is the library name with a ".lml" suffix (for instance, library `stdlib` is stored in file "stdlib.lml"). Before a user function can be called, its library must be loaded with the `use` statement. `use` statements have an effect only in the context where they are placed, i.e. in a library, or the command-line interface,

or a Sysquake SQ file; this way, different libraries may define functions with the same name provided they are not used in the same context.

In a library, functions can be public or private. Public functions may be called from any context which use the library, while private functions are visible only from the library they are defined in.

10.6 Types

Numerical, logical, and character arrays

The basic type of LME is the two-dimensional array, or matrix. Scalar numbers and row or column vectors are special kinds of matrices. Arrays with more than two dimensions are also supported. All elements have the same type, which are described in the table below. Two non-numeric types exist for character arrays and logical (boolean) arrays. Cell arrays, which contain composite types, are described in a section below.

Type	**Description**
double	64-bit IEEE number
complex double	Two 64-bit IEEE numbers
single	32-bit IEEE number
complex single	Two 32-bit IEEE numbers
uint32	32-bit unsigned integer
int32	32-bit signed integer
uint16	16-bit unsigned integer
int16	16-bit signed integer
uint8	8-bit unsigned integer
int8	8-bit signed integer
uint64	64-bit unsigned integer
int64	64-bit signed integer

64-bit integer numbers are not supported by all applications on all platforms.

These basic types can be used to represent many mathematic objects:

Scalar One-by-one matrix.

Vector n-by-one or one-by-n matrix. Functions which return vectors usually give a column vector, i.e. n-by-one.

Empty object 0-by-0 matrix (0-by-n or n-by-0 matrices are always converted to 0-by-0 matrices).

Polynomial of degree d 1-by-(d+1) vector containing the coefficients of the polynomial of degree d, highest power first.

List of n polynomials of same degree d n-by-(d+1) matrix containing the coefficients of the polynomials, highest power at left.

List of n roots n-by-1 matrix.

List of n roots for m polynomials of same degree n n-by-m matrix.

Single index One-by-one matrix.

List of indices Any kind of matrix; the real part of each element taken row by row is used.

Sets Numerical array, or list or cell array of strings (see below).

Boolean value One-by-one logical array; 0 means false, and any other value (including nan) means true (comparison and logical operators and functions return logical values). In programs and expressions, constant boolean values are entered as `false` and `true`. Scalar boolean values are displayed as `false` or `true`; in arrays, respectively as F or T.

String Usually 1-by-n char array, but any shape of char arrays are also accepted by most functions.

Unless a conversion function is used explicitly, numbers are represented by double or complex values. Most mathematical functions accept as input any type of numeric value and convert them to double; they return a real or complex value according to their mathematical definition.

Basic element-wise arithmetic and comparison operators accept directly integer types ("element-wise" means the operators + - .* ./ .\ and the functions `mod` and `rem`, as well as operators * / \ with a scalar multiplicand or divisor). If their arguments do not have the same type, they are converted to the size of the largest argument size, in the following order:

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

Literal two-dimensional arrays are enclosed in brackets. Rows are separated with semicolons or line breaks, and row elements with commas or spaces. Here are three different ways to write the same 2-by-3 double array.

```
A = [1, 2, 3; 4, 5, 6];
A = [1 2 3
 4 5 6];
A = [1, 2,
3;
4, 5 6];
```

Functions which manipulate arrays (such as `reshape` which changes their size or `repmat` which replicates them) preserve their type.

To convert arrays to numeric, char, or logical arrays, use functions + (unary operator), `char`, or `logical` respectively. To convert the numeric types, use functions `double`, `single`, or `uint8` and similar functions.

Numbers

Double and complex numbers are stored as floating-point numbers, whose finite accuracy depends on the number magnitude. During computations, round-off errors can accumulate and lead to visible artifacts; for example, `2-sqrt(2)*sqrt(2)`, which is mathematically `0`, yields `-4.4409e-16`. Integers whose absolute value is smaller than 2^52 (about 4.5e15) have an exact representation, though.

Literal double numbers (constant numbers given by their numeric value) have an optional sign, an integer part, an optional fractional part following a dot, and an optional exponent. The exponent is the power of ten which multiplies the number; it is made of the letter 'e' or 'E' followed by an optional sign and an integer number. Numbers too large to be represented by the floating-point format are changed to plus or minus infinity; too small numbers are changed to 0. Here are some examples (numbers on the same line are equivalent):

```
123 +123 123. 123.00 12300e-2
-2.5 -25e-1 -0.25e1 -0.25e+1
0 0.0 -0 1e-99999
inf 1e999999
-inf -1e999999
```

Literal integer numbers may also be expressed in hexadecimal with prefix `0x`, in octal with prefix `0`, or in binary with prefix `0b`. The four literals below all represent 11, stored as double:

```
0xb
013
0b1011
11
```

Literal integer numbers stored as integers and literal single numbers are followed by a suffix to specify their type, such as `2int16` for the number 2 stored as a two-byte signed number or `0x300uint32` for the number whose decimal representation is 768 stored as a four-byte unsigned number. All the integer types are valid, as well as `single`. This syntax gives the same result as the call to the corresponding function (e.g. `2int16` is the same as `int16(2)`), except when the integer number cannot be represented with a double; then the number is rounded

to the nearest value which can be represented with a double. Compare the expressions below:

Expression	**Value**
`uint64(123456789012345678)`	`123456789012345696`
`123456789012345678uint64`	`123456789012345678`

Literal complex numbers are written as the sum or difference of a real number and an imaginary number. Literal imaginary numbers are written as double numbers with an `i` or `j` suffix, like `2i`, `3.7e5j`, or `0xffj`. Functions `i` and `j` can also be used when there are no variables of the same name, but should be avoided for safety reasons.

The suffices for single and imaginary can be combined as `isingle` or `jsingle`, in this order only:

```
2jsingle
3single + 4isingle
```

Command `format` is used to specify how numbers are displayed.

Strings

Strings are stored as arrays (usually row vectors) of 16-bit unsigned numbers. Literal strings are enclosed in single quotes:

```
'Example of string'
''
```

The second string is empty. For special characters, the following escape sequences are recognized:

Character	**Escape seq.**	**Character code**
Null	\0	0
Bell	\a	7
Backspace	\b	8
Horizontal tab	\t	9
Line feed	\n	10
Vertical tab	\v	11
Form feed	\f	12
Carriage return	\r	13
Single tick	\'	39
Single tick	'' (two ')	39
Backslash	\\	92
Hexadecimal number	\xhh	hh
Octal number	\ooo	ooo
16-bit UTF-16	\uhhhh	1 UTF-16 code
21-bit UTF-32	\Uhhhhhhhh	1 or 2 UTF-16 codes

For octal and hexadecimal representations, up to 3 (octal) or 2 (hexadecimal) digits are decoded; the first non-octal or non-hexadecimal

digit marks the end of the sequence. The null character can conveniently be encoded with its octal representation, `\0`, provided it is not followed by octal digits (it should be written `\000` in that case). It is an error when another character is found after the backslash. Single ticks can be represented either by a backslash followed by a single tick, or by two single ticks.

Depending on the application and the operating system, strings can contain directly Unicode characters encoded as UTF-8, or MBCS (multi-byte character sequences). 16-bit characters encoded with `\uhhhh` escape sequences are always accepted and handled correctly by all built-in LME functions (low-level input/output to files and devices which are byte-oriented is an exception; explicit UTF-8 conversion should be performed if necessary).

UTF-32 sequences `\Uhhhhhhhh` assume UTF-16 encoding. In sequences `\uhhhh` and `\Uhhhhhhhh`, up to 4 or 8 hexadecimal digits can be provided, respectively, but the first non-hexadecimal character marks the end of the sequence.

Inline data

For large amounts of text or binary data, the syntax described above is impractical. Inline data is a special syntax for storing strings as raw text or `uint8` arrays as base64.

Strings (char arrays of dimension 1-by-*n*) can be defined in the source code as raw text without any escape sequence with the following syntax:

```
@/text marker
text
marker
```

where `@/text` is that literal sequence of six characters followed or not by spaces and tabs, `marker` is an arbitrary sequence of characters without spaces, tabs or end-of-lines which does not occur in the text, and `text` is the text itself. The spaces, tabs and first end-of-line which follow the first marker are ignored. The last marker must be at the beginning of a line; this means that the string always ends with an end-of-line. The whole text inline data is equivalent to a string with the corresponding characters and can be located in an assignment or any expression. End-of-line sequences (`\n`, `\r` or `\r\n`) are replaced by a single linefeed character.

Here is an example of a short fragment of C code, assigned to variable `src`. The sequence `\n` is not interpreted as an escape sequence by LME; it results in the two characters `\` and `n` in `src`. The trailing semicolon suppresses the display of the assignment, like in any LME expression.

```
src = @/text"""
int main() {
  printf("Hello, data!\n");
}
""";
```

Arrays of `uint8`, of dimension *n*-by-1 (column vectors), can be defined in the source code in a compact way using the base64 encoding in *inline data*:

```
@/base64 data
```

where `@/base64` is that literal sequence of eight characters, followed by spaces and/or line breaks, and the data encoded with base64 (see RFC 2045). The base64-encoded data can contain lowercase and uppercase letters a-z and A-Z, digits 0-9, and characters / (slash) and + (plus), and is followed by 0, 1 or 2 characters = (equal) for padding. Spaces, tabs and line breaks are ignored. Comments are not allowed.

The first character which is not a valid base64 character signals the end of the inline data and the beginning of the next token of source code. Inline data can be a part of any expression, assignment or function call, like any other literal value. In the case where the inline data is followed by a character which would erroneously be interpreted as more base64 codes (e.g. neither padding with = nor statement terminator and a keyword at the beginning of the following line), it should be enclosed in parenthesis.

Inline data can be generated with the `base64encode` function. For example, to encode `uint8(0:255).'` as inline data, you can evaluate

```
base64encode(uint8(0:255))
```

Then copy and paste the result to the source code, for instance as follows to set a variable `d` (note how the semicolon will be interpreted as the delimiter following the inline data, not the data iteself):

```
d = @/base64
AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8gISIjJCUmJygpKiss
LS4vMDEyMzQ1Njc4OTo7PD0+P0BBQkNERUZHSElKS0xNTk9QUVJTVFVWV1hZ
WltcXV5fYGFiY2RlZmdoaWprbG1ub3BxcnN0dXZ3eHl6e3x9fn+AgYKDhIWG
h4iJiouMjY6PkJGSk5SVlpeYmZqbnJ2en6ChoqOkpaanqKmqq6ytrq+wsbKz
tLW2t7i5uru8vb6/wMHCw8TFxsfIycrLzM3Oz9DR0tPU1dbX2Nna29zd3t/g
4eLj5OXm5+jp6uvs7e7v8PHy8/T19vf4+fr7/P3+/w== ;
```

Lists and cell arrays

Lists are ordered sets of other elements. They may be made of any type, including lists. Literal lists are enclosed in braces; elements are separated with commas.

```
{1,[3,6;2,9],'abc',{1,'xx'}}
```

Lists can be empty:

```
{}
```

List's purpose is to collect any kind of data which can be assigned to variables or passed as arguments to functions.

Cell arrays are arrays whose elements (or cells) contain data of any type. They differ from lists only by having more than one dimension. Most functions which expect lists also accept cell arrays; functions which expect cell arrays treat lists of n elements as 1-by-n cell arrays.

To create a cell array with 2 dimensions, cells are written between braces, where rows are separated with semicolons and row elements with commas:

```
{1, 'abc'; 27, true}
```

Since the use of braces without semicolon produces a list, there is no direct way to create a cell array with a single row, or an empty cell array. Most of the time, this is not a problem since lists are accepted where cell arrays are expected. To force the creation of a cell array, the `reshape` function can be used:

```
reshape({'ab', 'cde'}, 1, 2)
```

Structures

Like lists and cell arrays, structures are sets of data of any type. While list elements are ordered but unnamed, structure elements, called *fields*, have a name which is used to access them.

There are three ways to make structures: with field assignment syntax inside braces, with the `struct` function, or by setting each field in an assignment. `s.f` refers to the value of the field named `f` in the structure `s`. Usually, `s` is the name of a variable; but unless it is in the left part of an assignment, it can be any expression which evaluates to a structure.

```
a = {label = 'A', position = [2, 3]};

b = struct(name = 'Sysquake',
           os = {'Windows', 'macOS', 'Linux'});

c.x = 200;
c.y = 280;
c.radius = 90;

d.s = c;
```

With the assignments above, `a.os{3}` is `'Linux'` and `c.s.radius` is `90`.

While the primary way to access structure fields is by name, field order is still preserved, as can be seen by displaying the strcture, getting the field names with `fieldnames`, or converting the structure to a cell array with `struct2cell`. The fields can be reordered with `orderfields`.

Structure arrays

While structure fields can contain any type of array and cell arrays can have structures stored in their cells, structure arrays are arrays where each element has the same named fields. Plain structures are structure arrays of size `[1,1]`, like scalar numbers are arrays of size `[1,1]`.

Values are specified first by indices between parenthesis, then by field name. Braces are invalid to access elements of structure arrays (they can be used to access elements of cell arrays stored in structure array fields).

Structure arrays are created from cell arrays with functions `structarray` or `cell2struct`, or by assigning values to fields.

```
A = structarray('name', {'dog','cat'},
                'weight', {[3,100],[3,18]});

B = cell2struct({'dog','cat';[3,100],[3,18]},
                {'name','weight'});

C(1,1).name = 'dog';
C(1,1).weight = [3,100];
C(1,2).name = 'cat';
C(1,2).weight = [3,18];
```

Column struct arrays (1-dimension) can be defined with field assignments inside braces by separating array elements with semicolons. Missing fields are set to the empty array `[]`.

```
D = {a = 1, b = 2; a = 5, b = 3; b = 8};
```

Value sequences

Value sequences are usually written as values separated with commas. They are used as function input arguments or row elements in arrays or lists.

When expressions involving lists, cell arrays or structure arrays evaluate to multiple values, these values are considered as a value sequence, or part of a value sequence, and used as such in context

where value sequences are expected. The number of values can be known only at execution time, and may be zero.

```
L = {1, 2};
v = [L{:}];  // convert L to a row vector
c = complex(L{:});  // convert L to a complex number
```

Value sequences can arise from element access of list or cell arrays with brace indexing, or from structure arrays with field access with or without parenthesis indexing.

Function references

Function references are equivalent to the name of a function together with the context in which they are created. Their main use is as argument to other functions. They are obtained with operator @.

Inline and anonymous functions

Inline and anonymous functions encapsulate executable code. They differ only in the way they are created: inline functions are made with function `inline`, while anonymous functions have special syntax and semantics where the values of variables in the current context can be captured implicitly without being listed as argument. Their main use is as argument to other functions.

Sets

Sets are represented with numeric arrays of any type (integer, real or complex double or single, character, or logical), or lists or cell arrays of strings. Members correspond to an element of the array or list. All set-related functions accept sets with multiple values, which are always reduced to unique values with function `unique`. They implement membership test, union, intersection, difference, and exclusive or. Numerical sets can be mixed; the result has the same type as when mixing numeric types in array concatenation. Numerical sets and list or cell arrays os strings cannot be mixed.

Null

Null stands for the lack of data. It is both a data type and the only value it can represent. It can be assigned to a variable, be contained in a list or cell array element or a structure field, or passed as an input or output argument to/from a function.

Null is a recent addition to LME, where the lack of data is usually represented by the empty matrix `[]`. It is especially useful when LME

is interfaced with languages or libraries where the null value has a special meaning, such as SQL (Structured Query Language, used with relational databases) or the DOM (Document Object Model, used with XML).

Objects

Objects are the basis of *Object-Oriented Programming* (OOP), an approach of programming which puts the emphasis on encapsulated data with a known programmatic interface (the objects). Two OOP languages in common use today are C++ and Java.

The exact definition of OOP varies from person to person. Here is what it means when it relates to LME:

Data encapsulation Objects contain data, but the data cannot be accessed directly from the outside. All accesses are performed via special functions, called *methods*. What links a particular method to a particular object is a `class`. Class are identified with a name. When an object is created, its class name is specified. The names of methods able to act on objects of a particular class are prefixed with the class name followed with two colons. Objects are special structures whose contents are accessible only to its methods.

Function and operator overloading Methods may have the same name as regular functions. When LME has to call a function, it first checks the type of the input arguments. If one of them is an object, the corresponding method is called, rather than the function defined for non-object arguments. A method which has the same name as a function or another method is said to *overload* it. User functions as well as built-in ones can be overloaded. Operators which have a function name (for instance `x+y` can also be written `plus(x,y)`) can also be overloaded. Special functions, called object *constructors*, have the same name as the class and create new objects. They are also methods of the class, even if their input arguments are not necessarily objects.

Inheritance A class (*subclass*) may extend the data and methods of another class (*base class* or *parent*). It is said to *inherit* from the parent. In LME, objects from a subclass contain in a special field an object of the parent class; the field name has the same name as the parent class. If LME does not find a method for an object, it tries to find one for its parent, great-parent, etc. if any. An object can also inherit from several parents.

Here is an example of the use of `polynom` objects, which (as can be guessed from their name) contain polynomials. Statement `use polynom` imports the definitions of methods for class `polynom` and others.

```
use polynom;
p = polynom([1,5,0,1])
  p =
    x^3+5x^2+1
q = p^2 + 3 * p / polynom([1,0])
  q =
    x^6+10x^5+25x^4+2x^3+13x^2+15x+1
```

10.7 Input and Output

LME identifies channels for input and output with non-negative integer numbers called *file descriptors*. File descriptors correspond to files, devices such as serial port, network connections, etc. They are used as input argument by most functions related to input and output, such as `fprintf` for formatted data output or `fgets` for reading a line of text.

Note that the description below applies to most LME applications. For some of them, files, command prompts, or standard input are irrelevant or disabled; and standard output does not always correspond to the screen.

At least four file descriptors are predefined:

Value	**Input/Output**	**Purpose**
0	Input	Standard input from keyboard
1	Output	Standard output to screen
2	Output	Standard error to screen
3	Output	Prompt for commands

You can use these file descriptors without calling any opening function first, and you cannot close them. For instance, to display the value of π, you can use `fprintf`:

```
fprintf(1, 'pi = %.6f\n', pi);
  pi = 3.141593
```

Some functions use implicitly one of these file descriptors. For instance `disp` displays a value to file descriptor 1, and `warning` displays a warning message to file descriptor 2.

File descriptors for files and devices are obtained with specific functions. For instance `fopen` is used for reading from or writing to a file. These functions have as input arguments values which specify what to open and how (file name, host name on a network, input or output mode, etc.), and as output argument a file descriptor. Such file descriptors are valid until a call to `fclose`, which closes the file or the connection.

10.8 Error Messages

When an error occurs, the execution is interrupted and an error message explaining what happened is displayed, unless the code is enclosed in a try/catch block. The whole error message can look like

```
> use stat
> iqr(123)

Index out of range for variable 'M' (stat/prctile;61)
-> stat/iqr;69
```

The first line contains an error message, the location in the source code where the error occurred, and the name of the function or operator involved. Here `stat` is the library name, `prctile` is the function name, and 61 is the line number in the file which contains the library. If the function where the error occurs is called itself by another function, the whole chain of calls is displayed; here, `prctile` was called by `iqr` at line 69 in library `stat`.

Here is the list of errors which can occur. For some of them, LME attempts to solve the problem itself, e.g. by allocating more memory for the task.

Stack overflow Too complex expression, or too many nested function calls.

Data stack overflow Too large objects on the stack (in expressions or in nested function calls).

Variable overflow Not enough space to store the contents of a variable.

Code overflow Not enough memory for compiling the program.

Not enough memory Not enough memory for an operation outside the LME core.

Algorithm does not converge A numeric algorithm does not converge to a solution, or does not converge quickly enough. This usually means that the input arguments have invalid values or are ill-conditioned.

Incompatible size Size of the arguments of an operator or a function do not agree together.

Bad size Size of the arguments of a function are invalid.

Non-vector array A row or column vector was expected, but a more general array was found.

Not a column vector A column vector was expected, but a more general array was found.

Not a row vector A row vector was expected, but a more general array was found.

Non-matrix array A matrix was expected, but an array with more than 2 dimensions was found.

Non-square matrix A square matrix was expected, but a rectangular matrix was found.

Index out of range Index negative or larger than the size of the array.

Wrong type String or complex array instead of real, etc.

Non-integer argument An argument has a fractional part while an integer is required.

Non positive integer argument An argument is not a positive integer as expected.

Argument out of range An argument is outside the permitted range of values.

Non-scalar argument An argument is an array while a scalar number is required.

Non-object argument An object is required as argument.

Not a permutation The argument is not a permutation of the integers from 1 to n.

Bad argument A numeric argument has the wrong site or the wrong value.

Unknown option A string option has an invalid value.

Object too large An object has a size larger than some fixed limit.

Undefined variable Attempt to retrieve the contents of a variable which has not been defined.

Undefined input argument Attempt to retrieve the contents of an input argument which was neither provided by the caller nor defined in the function.

Undefined function Attempt to call a function not defined.

Too few or too many input arguments Less or more arguments in the call than what the function accepts.

Too few or too many output arguments Less or more left-side variables in an assignment than the function can return.

Syntax error Unspecified compile-time error.

"function" keyword without function name Incomplete function header.

Bad function header Syntax error in a function header

Missing expression Statement such as `if` or `while` without expression.

Unexpected expression Statement such as end or `else` followed by an expression.

Incomplete expression Additional elements were expected during the compilation of an expression, such as right parenthesis or a sub-expression at the right of an operator.

"for" not followed by a single assignment `for` is followed by an expression or an assignment with multiple variables.

Bad variable name The left-hand part of an assignment is not a valid variable name (e.g. 2=3)

String without right quote The left quote of a string was found, but the right quote is missing.

Unknown escape character sequence In a string, the backslash character is not followed by a valid escape sequence.

Unexpected right parenthesis Right parenthesis which does not match a left parenthesis.

Unexpected right bracket Right bracket which does not match a left bracket.

Unrecognized or unexpected token An unexpected character was found during compilation (such as (1+))

"end" not in an index expression end was used outside of any index sub-expression in an expression.

"beginning" not in an index expression `beginning` was used outside of any index sub-expression in an expression.

"matrixcol" not in an index expression `matrixcol` was used outside of any index sub-expression in an expression.

"matrixrow" not in an index expression `matrixrow` was used outside of any index sub-expression in an expression.

"matrixrow" or "matrixcol" used in the wrong index `matrixrow` was used in an index which was not the first one, or `matrixcol` was used in an index which was not the only one or the second one.

Compilation overflow Not enough memory during compilation.

Too many nested subexpressions The number of nested of subexpressions is too high.

Variable table overflow A single statement attempts to define too many new variables at once.

Expression too large Not enough memory to compile a large expression.

Too many nested (), [] and {} The maximum depth of nested subexpressions, function argument lists, arrays and lists is reached.

Too many nested programming constructs Not enough memory to compile that many nested programming constructs such as `if`, `while`, `switch`, etc.

Wrong number of input arguments Too few or too many input arguments for a built-in function during compilation.

Wrong number of output arguments Too few or too many output arguments for a built-in function during compilation.

Too many indices More than two indices for a variable.

Variable not found A variable is referenced, but appears neither in the arguments of the function nor in the left part of an assignment.

Unbounded language construct `if`, `while`, `for`, `switch`, or `try` without end.

Unexpected "end" The `end` statement does not match an `if`, `switch`, `while`, `for`, or `catch` block.

"case" or "otherwise" without "switch" The `case` or `otherwise` statement is not inside a `switch` block.

"catch" without "try" The `catch` statement does not match a `try` block.

"break" or "continue" not in a loop The `break` or `continue` statement is not inside a `while` or `for block`.

Variable name reused Same variable used twice as input or as output argument.

Too many user functions Not enough memory for that many user functions.

Attempt to redefine a function A function with the same name already exists.

Can't find function definition Cannot find a function definition during compilation.

Unexpected end of expression Missing right parenthesis or square bracket.

Unexpected statement Expression expected, but a statement is found (e.g. `if`).

Null name Name without any character (when given as a string in functions like `feval` and `struct`).

Name too long More than 32 characters in a variable or function name.

Unexpected function header A function header (keyword "function") has been found in an invalid place, for example in the argument of `eval`.

Function header expected A function header was expected but not found.

Bad variable in the left part of an assignment The left part of an assignment does not contain a variable, a structure field, a list element, or the part of an array which can be assigned to.

Bad variable in a for loop The left part of the assignment of a for loop is not a variable.

Source code not found The source code of a function is not available.

File not found `fopen` does not find the file specified.

Bad file ID I/O function with a file descriptor which neither is standard nor corresponds to an open file or device.

Cannot write to file Attempt to write to a read-only file.

Bad seek Seek out of range or attempted on a stream file.

Too many open files Attempt to open too many files.

End of file Attempt to read data past the end of a file.

Timeout Input or output did not succeed before a too large amount of time elapsed.

No more OS memory The operating system cannot allocate more memory.

Bad context Call of a function when it should not (application-dependent).

Not supported The feature is not supported, at least in the current version.

10.9 Character Set

There exist different standards to represent characters. In LME, characters are stored as 16-bit unsigned integer numbers. The mapping between these codes and the actual characters they represent depends on the application and the operating system. Currently, on macOS, Windows and Linux, Sysquake uses the UTF-16 character encoding (i.e. Unicode characters encoded in one or two 16-bit words).

To make the exchange of files possible without manual conversion, all text files used by LME applications can have their character set specified explicitly. In Sysquake, this includes library files (.lml), SQ files (.sq), and SQ data files (.sqd). Versions of Sysquake using Unicode (currently macOS and Linux) convert automatically files with a charset specification.

The character set specification is a comment line with the following format:

`// charset=`*`charsetname`*

or

`% charset=`*`charsetname`*

Spaces between the comment mark and the keyword `charset` are ignored. The comment line must be the first or the second line of the text file. The character set *`charsetname`* must be one of the following:

Name	**Description**
`ascii` or `usascii`	ASCII
`utf-8` or `utf8`	UTF-8 (unicode)
`iso-8859-1` or `iso-latin-1`	ISO-Latin-1 (Windows 1252)
`macintosh` or `macosroman`	Mac OS Classic

Here are advices about the use of character set specifications, both for the current transition phase where Sysquake for Windows does not use Unicode and for the future.

If you need only ASCII (typically because you work in English, or for files without text or where unaccented letters are acceptable), do not add any character set specification (ASCII is a subset of all supported

character sets) or add `charset=ascii` as an indication that the file should contain only 7-bit characters.

If you need accented characters found in western European languages, use ISO-8859-1 with an explicit character set specification on Windows and other platforms if you need cross-platform compatibility, or any character set with a character set specification otherwise.

If you need another native character set on Windows, do not add any character set specification, and switch to UTF-8 as soon as a unicode version of Sysquake becomes available.

10.10 Formatted text

Like most text-based programming languages, LME primary text format is plain text, without any character or paragraph formatting. To improve readability, it also supports formatted text. Formatting information used to change the character font and size and how paragraphs are aligned is embedded in the text itself as markup, i.e. special sequences of characters unlikely to occur in normal text. This is similar to HTML or LaTeX, but with a simpler syntax similar to what is used in wikis and blogs. The markup which has been chosen for LME is Creole, a collaborative effort to create a common markup language to be used across different wikis, and more precisely the open-source NME implementation.

Formatted text output channel

In addition to the standard output channel (file descriptor 1) and the standard error channel (file descriptor 2), LME has a special channel for formatted output (file descriptor 4). Anything written to that channel is parsed so that markup constructs are interpreted; the result is displayed in the command window. Each write command should contain a whole block of text with markup; there is no buffering across output commands.

Not all versions of LME support formatted output, and how formatted output is displayed depends on the application and the platform. For instance, in shell applications, word-wrap is applied to paragraphs, but bold or italic text is rendered as plain text.

Example

```
T = 27.3;
fprintf(4, '=Report=\nTemperature is %.1f\n', T);
```

The same markup is used in LME applications at other places. For example, in Sysquake, the version and help information can contain markup.

Markup reference

Text with markup is plain text with a few character sequences which are recognized by the markup processor and change the format of the result. The result is styled text with titles, paragraphs with justification, bold and italic faces, etc.

There are two kinds of markup constructs: blocks and inline. Blocks are paragraphs (indented or not), headings, list items, table cells, horizontal rules and block extensions. Inline constructs are character styles, verbatim text and links; they are part of blocks. Except in preformatted blocks, sequences of spaces or tabs are replaced with single spaces.

Paragraphs

Paragraphs are made of lines whose first character is not one of `*#:;=` nor sequence `{{{`, `---`, or `<<<`. Lines can begin with sequences `**` and `##`.

Paragraphs end with blank lines or next heading, list, table, indented paragraph, preformatted block of text, horizontal rule or block extension. They can contain styled text, links, spans of verbatim text, and inline extensions.

Example

```
This is a paragraph
written in two lines.

This is another paragraph.
```

Indenting

Indented paragraphs are paragraphs which begin with a colon. Multiple colons define the level of indenting. Indented paragraphs can contain styled text, links, spans of verbatim text, and inline extensions.

Example

```
This is a normal paragraph.
:This is an indented
paragraph in two lines.
::This is more indented.
```

Headings

Headings are made of a single line beginning with an equal character. The number of equal characters defines the level of title, from main title (=) to sub-sub-subtitle (====). Headings may end with a sequence of equal characters.

Examples

```
=Level 1 heading=
Paragraph
=Another level 1 heading, without trailing equal character
==Level 2 heading==
===Level 3 heading===
```

Lists

Lists are collections of items. There are three kinds of lists: unnumbered lists, numbered lists, and definition lists. Lists can be nested; they end with the next heading, indented paragraph, table, or blank line.

Unnumbered lists are represented as indented paragraphs with a bullet. Each item begins with a star character (∗); it can span multiple lines.

Numbered lists are represented as indented paragraphs with a number. Items are numbered automatically with consecutive integers starting at 1. Each item begins with a sharp character (#); it can span multiple lines.

Definition lists are made of two kinds of items: title, typically displayed in bold font, and definition, typically displayed indented. Titles begin with a semicolon at the beginning of a line. Definitions either follow the title, separated with a colon; or they begin on a new line beginning with a colon.

List nesting can combine different kinds of lists. Sublist items begin with multiple markers, the first one(s) corresponding to the enclosing list(s). For instance, items of an unnumbered list nested in a numbered list should start with #∗ at the beginning of the line, without any preceding space. List markers must be used in a consistent way; for example, ## at the beginning of a line in an unnumbered list is not interpreted as a second-level numbered list item, but as monospace style (see below).

Examples

```
∗ First item of unnumbered list.
∗ Second
item, in two lines.
∗Third item without space (spaces are optional).

# First item of numbered list.
# Second item.
#∗ First item on unnumbered sublist.
#∗ Second item.
# Thirst item of top-level numbered list.

; First title of definition list
```

```
: Definition of first item.
; Second title: Second definition
beginning on the same line.

Paragraph separated with a blank line.
```

Tables

Tables are rectangular array of cells. They are made of one line per row. Each cell begins with character |. Heading cells (typically displayed in bold face) begin with |=. Rows may end with a trailing |.

Example

In the table below, the first row and the first column contain headings. The very first cell is empty.

```
||=First column|=Second column
|=First row|Cell 1,1|Cell 1,2
|=Second row|Cell 2,1|Cell 2,2
```

Preformatted

Preformatted text is a block of text displayed literally, including line feeds. Preformatted text is preceded with a line containing {{{ and is followed by a line containing }}}.

Example

```
This is some C code:
{{{
double f(double x)
{
  return 2 * x;  // f(x) = 2x
}
}}}
```

In preformatted text, lines which begin with }}}, with leading spaces or not, must have an additional space which is discarded in the output.

Horizontal rules

Horizontal rules are single lines containing four hyphens.

Example

```
Paragraph.
---
Paragraph following a horizontal rule.
```

Extensions

Sequences << and <<< are reserved for extensions.

Character style

Inside paragraphs, indented paragraphs, headings, list elements, and table cells, the following two-character sequences toggle on or off the corresponding style. It is not mandatory to nest spans of styled characters between matching style sequences. Style is reset at the end of text block.

Markup	Style
**	Bold
//	Italic
##	Monospace
,, (two commas)	Subscript
^^	Superscript
__ (two underscores)	Underlined

Double stars and double sharps are interpreted as the beginning of list items when they are the first characters of a line in the context of lists. To be interpreted as style markup, they can be preceded by a space.

Example

```
This is //italic text//, **bold text**,
and //**bold italic text**//.
```

Escape character

The tilde character, when followed by any character except space, tab or line feed, is an escape character; it is not displayed and the next character loses its special meaning, if any.

Example

```
Two stars: ~*~* or ~** or *~*; tilde: ~~.
```

is rendered as "Two stars: ** or ** or **; tilde: ~."

Verbatim

Verbatim text is a sequence of characters enclosed between {{{ and }}}. After {{{, all characters are output verbatim, without any markup interpreting, until the next }}} or the end of text block. Multiple spaces and tabs and single line feeds are still converted to single spaces, though. Verbatim text is an alternative to the escape character; it is more convenient for sequences of characters.

Example

```
{{{**}}} //{{{{{{xx}}}}}}//
```

is rendered as "** *{{{xx}}}*".

Line break

Except in preformatted blocks, line breaks are not preserved. The sequence \\ forces a line break.

Example

```
The next line of this paragraph begins...\\here!
```

Links

Hypertext links (URLs) are enclosed between [[and]]. The text displayed as the link is either the same as the URL itself if there is no | character, or it is what follows |. No markup is recognized in the URL part; what follows | can contain styled text and verbatim text. Spaces surrounding | are ignored.

Examples

```
* Simple link: [[https://calerga.com]]
* Link with link text: [[https://calerga.com | Calerga]]
* Link with styled link text: [[https://calerga.com | **Calerga**]]
```

10.11 List of Commands, Functions, and Operators

Programming keywords

```
break          for                  rethrow
case           function             return
catch          global               switch
clear          hideimplementation   try
continue       if                   until
define         otherwise            use
endfunction    persistent           useifexists
else           private              while
elseif         public
error          repeat
```

Programming operators and functions

```
assert                fun2str       sandbox
Variable assignment   inline        sandboxtrust
Operator ()           isdefined     str2fun
Operator @            isfun         str2obj
builtin               isglobal      subsasgn
deal                  lasterr       subsref
dumpvar               lasterror     varargin
eval                  namedargin    varargout
feval                 nargin
fevalx                nargout
```

Platform

```
exist     iskeyword    lookfor
help      ismac        variables
info      ispc         which
inmem     isunix
```

Debugging

```
dbclear    dbstack     dbtype
dbcont     dbstatus    echo
dbhalt     dbstep      profile
dbquit     dbstop
```

Arrays

```
[]          inthist     permute
,           ipermute    rand
;           isempty     randi
:           length      randn
arrayfun    linspace    repmat
beginning   logspace    reshape
cat         magic       rng
diag        matrixcol   rot90
end         matrixrow   size
eye         meshgrid    sort
find        ndgrid      squeeze
flipdim     ndims       sub2ind
fliplr      nnz         unique
flipud      numel       unwrap
ind2sub     ones        zeros
```

Strings

```
base32decode    length        strfind
base32encode    lower         strmatch
base64decode    mathml        strrep
base64encode    mathmlpoly    strtok
char            regexp        strtrim
deblank         regexpi       unicodeclass
ischar          setstr        upper
isdigit         split         utf32decode
isempty         sprintf       utf32encode
isletter        sscanf        utf8decode
isspace         strcmp        utf8encode
latex2mathml    strcmpi
```

Hash

```
hmac    sha1
md5     sha1
```

Lists

```
{}          islist      num2list
apply       length      replist
join        list2num
isempty     map
```

Cell arrays

```
cell        iscell
cellfun     num2cell
```

Structures and structure arrays

```
cell2struct    isstruct       struct2cell
cellfun        orderfields    structarray
fieldnames     rmfield        structmerge
getfield       setfield
isfield        struct
```

Null value

```
isnull     null
```

Objects

```
class          isobject         superiorto
inferiorto     methods
isa            superclasses
```

Logical operators

```
==      >       |
===     <=      &&
~=      >=      ||
~==     ~       ?
<       &
```

Logical functions

```
all          isfinite      isprime
any          isfloat       isrow
false        isinf         isscalar
find         isinteger     isspace
ischar       isletter      isvector
iscolumn     islogical     logical
isdigit      ismatrix      true
isempty      isnan         xor
isequal      isnumeric
```

Bitwise functions

```
bitall     bitget       bitxor
bitand     bitor        graycode
bitany     bitset       igraycode
bitcmp     bitshift
```

Integer functions

```
int8      int64      uint16
int16     map2int    uint32
int32     uint8      uint64
```

Set functions

```
intersect    setdiff    union
ismember     setxor     unique
```

Constants

```
eps            inf       pi
false          intmax    realmax
flintmax       intmin    realmin
goldenratio    j         true
i              nan
```

Arithmetic functions

```
+      \          diff
-      .\         kron
*      ^          mod
.*     .^         prod
/      cumprod    rem
./     cumsum     sum
```

Trigonometric functions in radians

```
acos     atan     sec
acot     atan2    sin
acsc     cos      tan
asec     cot
asin     csc
```

Trigonometric functions in degrees

```
acosd     atand     secd
acotd     atan2d    sind
acscd     cosd      tand
asecd     cotd
asind     cscd
```

Hyperbolic functions

```
acosh     asinh     csch
acoth     atanh     sech
acsch     cosh      sinh
asech     coth      tanh
```

Other scalar math functions

```
abs         erfcinv      log
angle       erfcx        log10
beta        erfinv       log1p
betainc     exp          log2
betaln      expm1        nchoosek
conj        factor       nthroot
diln        factorial    rat
ellipam     gamma        real
ellipe      gammainc     reallog
ellipf      gammaln      realpow
ellipj      gcd          realsqrt
ellipke     hypot        sign
erf         imag         sinc
erfc        lcm          sqrt
```

Type conversion functions

```
cast       fix        single
ceil       floor      swapbytes
complex    round      typecast
double     roundn
```

Matrix math functions

```
'          fft                null
.'         funm               orth
balance    hess               pinv
care       householder        qr
chol       householderapply   rank
cond       ifft               schur
conv2      inv                sqrtm
dare       linprog            svd
det        logm               trace
dlyap      lu                 tril
eig        lyap               triu
expm       norm
```

Geometry functions

```
cart2pol    cross    pol2cart
cart2sph    dot      sph2cart
```

Probability distribution functions

```
cdf     pdf
icdf    random
```

Statistic functions

```
cov          max       moment
cummax       mean      skewness
cummin       median    std
kurtosis     min       var
```

Polynomial math functions

```
addpol     filter     polyint
conv       poly       polyval
deconv     polyder    roots
```

Interpolation and triangulation functions

```
delaunay      interp1      voronoi
delaunayn     interpn      voronoin
griddata      tsearch
griddatan     tsearchn
```

Quaternion operators

```
,      *      ^
;      .*     .^
==     /      '
~=     ./     .'
+      \
-      .\
```

Quaternion math functions

```
abs       q2mat          real
conj      q2rpy          rpy2q
cos       q2str          sign
cumsum    qimag          sin
diff      qinv           sqrt
exp       qnorm          sum
log       qslerp
mean      quaternion
```

Other quaternion functions

```
beginning    fliplr          permute
cat          flipud          repmat
char         ipermute        reshape
disp         isempty         rot90
dumpvar      isquaternion    size
double       length          squeeze
end          ndims           subsasgn
flipdim      numel           subsref
```

Non-linear numeric functions

```
fminbnd       integral       ode45
fminsearch    lsqcurvefit    odeset
fsolve        lsqnonlin      optimset
fzero         ode23
```

Dynamical systems functions

```
c2dm       margin      tf2ss
d2cm       movezero    zp2ss
dmargin    ss2tf
```

Input/output

```
bwrite      format     redirect
disp        fprintf    sprintf
error       fread      sread
fclose      frewind    sscanf
feof        fscanf     swrite
fgetl       fseek      warning
fgets       ftell
fionread    fwrite
```

Files

```
efopen    fopen
```

Path manipulation

```
fileparts    filesep    fullfile
```

XML

```
getElementById          saxnew        xmlreadstring
getElementsByTagName    saxnext       xmlrelease
saxcurrentline          saxrelease
saxcurrentpos           xmlread
```

LME threads

```
semaphoredelete    semaphoreunlock    threadset
semaphorelock      threadkill         threadsleep
semaphorenew       threadnew
```

Parallel execution

```
batch          delete          pardefaultcluster
cancel         fetchOutputs    submit
createJob      findTask        wait
createTask     parcluster
```

Basic graphics

```
activeregion    fplot         polar
altscale        image         quiver
area            label         scale
bar             legend        scalefactor
barh            line          subplotstyle
circle          math          text
colormap        pcolor        tickformat
contour         plot          ticks
figurestyle     plotoption    title
fontset         plotset
```

3D graphics

```
camdolly      camup         material
camorbit      camva         mesh
campan        camzoom       plot3
campos        contour3      plotpoly
camproj       daspect       sensor3
camroll       lightangle    surf
camtarget     line3
```

Graphics for dynamical systems

```
bodemag       dsigma     nichols
bodephase     dstep      nyquist
dbodemag      erlocus    plotroots
dbodephase    hgrid      rlocus
dimpulse      hstep      sgrid
dinitial      impulse    sigma
dlsim         initial    step
dnichols      lsim       zgrid
dnyquist      ngrid
```

User interface controls

```
button         settabs    textfield
popupmenu      slider
pushbutton     text
```

Figures and subplots

```
currentfigure    subplot         subplotsize
defaultstyle     subplotparam    subplotspring
figure           subplotpos      subplotsync
scaleoverview    subplotprops
scalesync        subplots
```

Dialog box

```
dialog       getfile
dialogset    putfile
```

Date and time

```
cal2julian    julian2cal    tic
clock         posixtime     toc
```

Extensions loaded on demand

```
exteval     extload     extunload
```

Interactivity

```
firstrun    _p0         _x
_dx         _p1         _x0
_dy         _q          _x1
_dz         _rho        _y
_id         _rho0       _y0
_ix         _rho1       _y1
_kx         _str1       _z
_ky         _theta      _z0
_kz         _theta0     _z1
_nb         _theta1
_m          _v
```

Sysquake instances

```
sqcall                sqguicmd        sqinstancetitle
sqcurrentinstance     sqinstances     sqselect
```

Miscellaneous

```
cancel        idlestate    redraw
clf           progress     sqcurrentlanguage
hasfeature    quit         sqfilepath
```

10.12 Variable Assignment and Subscripting

Variable assignment

Assignment to a variable or to some elements of a matrix variable.

Syntax

```
var = expr
(var1, var2, ...) = function(...)
```

Description

`var = expr` assigns the result of the expression `expr` to the variable `var`. When the expression is a naked function call, `(var1,var2,...) = function(...)` assigns the value of the output arguments of the function to the different variables. Usually, providing less variables than the function can provide just discards the superfluous output arguments; however, the function can also choose to perform in a different way (an example of such a function is `size`, which returns the number of rows and the number of columns of a matrix either as two numbers if there are two output arguments, or as a 1-by-2 vector if there is a single output argument). Providing more variables than what the function can provide is an error.

Variables can store any kind of contents dynamically: the size and type can change from assignment to assignment.

A subpart of a matrix variable can be replaced with the use of parenthesis. In this case, the size of the variable is expanded when required; padding elements are 0 for numeric arrays and empty arrays `[]` for cell arrays and lists.

See also

Operator (), operator {}, `clear`, `exist`, `for`, `subsasgn`

beginning

First index of an array.

Syntax

```
v(...beginning...)
A(...beginning...)
function e = C::beginning(obj, i, n)
```

Description

In an expression used as an index to access some elements of an array, `beginning` gives the index of the first element (line or column, depending of the context). It is always 1 for native arrays.

`beginning` can be overloaded for objects of used-defined classes. Its definition should be have a header equivalent to `function e=C::beginning(obj,i,n)`, where `C` is the name of the class, `obj` is the object to be indexed, `i` is the position of the index expression where `beginning` is used, and `n` is the total number of index expressions.

See also

Operator (), operator `{}`, `beginning`, `end`, `matrixcol`, `matrixrow`

end

Last index of an array.

Syntax

```
v(...end...)
A(...end...)
function e = C::end(obj, i, n)
```

Description

In an expression used as an index to access some elements of an array, `end` gives the index of the last element (line or column, depending of the context).

`end` can be overloaded for objects of used-defined classes. Its definition should be have a header equivalent to `function e=C::end(obj,i,n)`, where `C` is the name of the class, `obj` is the object to be indexed, `i` is the position of the index expression where `end` is used, `n` is the total number of index expressions.

Examples

Last 2 elements of a vector:

```
a = 1:5; a(end-1:end)
    4 5
```

Assignment to the last element of a vector:

```
a(end) = 99
  a =
    1 2 3 4 99
```

Extension of a vector:

```
a(end + 1) = 100
  a =
    1 2 3 4 99 100
```

See also

Operator `()`, operator `{}`, `size`, `length`, `beginning`, `matrixcol`, `matrixrow`

global persistent

Declaration of global or persistent variables.

Syntax

```
global x y ...
persistent x y ...
```

Description

By default, all variables are *local* and created the first time they are assigned to. Local variables can be accessed only from the body of the function where they are defined, but not by any other function, even the ones they call. They are deleted when the function exits. If the function is called recursively (i.e. if it calls itself, directly or indirectly), distinct variables are defined for each call. Similarly, local variables defined in the workspace using the command-line interface cannot be referred to in functions.

On the other hand, *global variables* can be accessed by multiple functions and continue to exist even after the function which created them exits. Global variables must be declared with `global` in each function which uses them. They can also be declared in the workspace. There exists only a single variable for each different name.

Declaring a global variable has the following result:

- If a previous local variable with the same name exists, it is deleted.
- If the global variable does not exist, it is created and initialized with the empty array `[]`.

- Every access which follows the declaration in the same function or workspace uses the global variable.

Like global variables, *persistent variables* are preserved between function calls; but they cannot be shared between different functions. They are declared with `persistent`. They cannot be declared outside a function. Different persistent variables can have the same name in different functions.

Examples

Functions to reset and increment a counter:

```
function reset
  global counter;
  counter = 0;

function value = increment
  global counter;
  counter = counter + 1;
  value = counter;
```

Here is how the counter can be used:

```
reset;
i = increment
  i =
    1
j = increment
  j =
    2
```

See also

`function`

matrixcol

First index in a subscript expression.

Syntax

```
A(...matrixcol...)
function e = C::matrixcol(obj, i, n)
```

Description

In an expression used as a single subscript to access some elements of an array `A(expr)`, `matrixcol` gives an array of the same size as `A` where each element is the column index. For instance for a 2-by-3 matrix, `matrixcol` gives the 2-by-3 matrix `[1,2,3;1,2,3]`.

In an expression used as the second of multiple subscripts to access some elements of an array `A(...,expr)` or `A(...,expr,...)`, `matrixcol` gives a row vector of length `size(A,2)` whose elements are the indices of each column. It is equivalent to the range `(beginning:end)`.

`matrixcol` is useful in boolean expressions to select some elements of an array.

`matrixcol` can be overloaded for objects of used-defined classes. Its definition should have a header equivalent to `function e=C::matrixcol(obj,i,n)`, where `C` is the name of the class, `obj` is the object to be indexed, `i` is the position of the index expression where `matrixcol` is used, and `n` is the total number of index expressions.

Example

Set to 0 the NaN values which are not in the first column:

```
A = [1, nan, 5; nan, 7, 2; 3, 1, 2];
A(matrixcol > 1 & isnan(A)) = 0
  A =
      1   0   5
    nan   7   2
      3   1   2
```

See also

`matrixrow`, `beginning`, `end`

matrixrow

First index in a subscript expression.

Syntax

```
A(...matrixrow...)
function e = C::matrixrow(obj, i, n)
```

Description

In an expression used as a single subscript to access some elements of an array `A(expr)`, `matrixrow` gives an array of the same size as `A`

where each element is the row index. For instance for a 2-by-3 matrix, `matrixrow` gives the 2-by-3 matrix `[1,1,1;2,2,2]`.

In an expression used as the first of multiple subscripts to access some elements of an array `A(expr,...)`, `matrixrow` gives a row vector of length `size(A,1)` whose elements are the indices of each row. It is equivalent to the range `(beginning:end)`.

`matrixrow` is useful in boolean expressions to select some elements of an array.

`matrixrow` can be overloaded for objects of used-defined classes. Its definition should be have a header equivalent to `function e=C::matrixrow(obj,i,n)`, where `C` is the name of the class, `obj` is the object to be indexed, `i` is the position of the index expression where `matrixrow` is used, and `n` is the total number of index expressions.

See also

`matrixcol`, `beginning`, `end`

subsasgn

Assignment to a part of an array, list, or structure.

Syntax

```
A = subsasgn(A, s, B)
```

Description

When an assignment is made to a subscripted part of an object in a statement like `A(s1,s2,...)=B`, LME executes `A=subsasgn(A,s,B)`, where `subsasgn` is a method of the class of variable `A` and `s` is a structure with two fields: `s.type` which is `'()'`, and `s.subs` which is the list of subscripts `{s1,s2,...}`. If a subscript is the colon character which stands for all elements along the corresponding dimensions, it is represented with the string `':'` in `s.subs`.

When an assignment is made to a subscripted part of an object in a statement like `A{s}=B`, LME executes `A=subsasgn(A,s,B)`, where `subsasgn` is a method of the class of variable `A` and `s` is a structure with two fields: `s.type` which is `'{}'`, and `s.subs` which is the list containing the single subscript `{s}`.

When an assignment is made to the field of an object in a statement like `A.f=B`, LME executes `A=subsasgn(A,s,B)`, where `s` is a structure with two fields: `s.type` which is `'.'`, and `s.subs` which is the name of the field (`'f'` in this case).

While the primary purpose of `subsasgn` is to permit the use of subscripts with objects, a built-in implementation of `subsasgn` is provided

for arrays when `s.type` is `'()'`, for lists and cell arrays when `s.type` is a list or a cell array, and for structures when `s.type` is `'.'`. In that case, the second argument `s` can be reduced to the list of subscripts or the field name; and a single subscripts can be given directly instead of a list of length 1.

Examples

```
A = [1,2;3,4];
subsasgn(A, {type='()',subs={1,':'}}, 999)
  999 999
    3   4
subsasgn(A, {type='()',subs={':',1}}, [])
    2
    4
```

Same result when the indices are given directly as the second argument:

```
subsasgn(A, {1,':'}, 999)
  999 999
    3   4
s = {a=2, b=1:5};
subsasgn(s, 'b', 'abc')
  a: 2
  b: 'abc'
```

See also

Operator `()`, operator `{}`, `subsref`, `beginning`, `end`

subsref

Reference to a part of an array, list, or structure.

Syntax

```
B = subsref(A, s)
```

Description

When an object variable is subscripted in an expression like `A(s1,s2,...)`, LME evaluates `subsref(A,s)`, where `subsref` is a method of the class of variable `A` and `s` is a structure with two fields: `s.type` which is `'()'`, and `s.subs` which is the list of subscripts `{s1,s2,...}`. If a subscript is the colon character which stands for all elements along the corresponding dimensions, it is represented with the string `':'` in `s.subs`.

When an object variable is subscripted in an expression like `A{s}`, LME evaluates `subsref(A,s)`, where `subsref` is a method of the class of variable `A` and `s` is a structure with two fields: `s.type` which is `'{}'`, and `s.subs` which is the list containing the single subscript `{s}`.

When the field of an object variable is retrieved in an expression like `A.f`, LME executes `subsref(A,s)`, where `s` is a structure with two fields: `s.type` which is `'.'`, and `s.subs` which is the name of the field (`'f'` in this case).

While the primary purpose of `subsref` is to permit the use of subscripts with objects, a built-in implementation of `subsref` is provided for arrays when `s.type` is `'()'`, for lists when `s.type` is `'{}'`, and for structures when `s.type` is `'.'`. In that case, the second argument `s` can be reduced to the list of subscripts or the field name; and a single subscripts can be given directly instead of a list of length 1.

Examples

```
A = [1,2;3,4];
subsref(A, {type='()',subs={1,':'}})
  1 2
```

Same result when the indices are given directly as the second argument:

```
subsref(A, {1,':'})
  1 2
s = {a='abc', b=1:5};
subsref(s, 'b')
  1 2 3 4 5
```

See also

Operator `()`, operator `{}`, `subsasgn`, `beginning`, `end`

10.13 Programming Constructs

Programming constructs are the backbone of any LME program. Except for the variable assignment, all of them use reserved keywords which may not be used to name variables or functions. In addition to the constructs described below, the following keywords are reserved for future use:

```
classdef    parfor
goto        spmd
```

break

Terminate loop immediately.

Syntax

```
break
```

Description

When a `break` statement is executed in the scope of a loop construct (`while`, `repeat` or `for`), the loop is terminated. Execution continues at the statement which follows `end`. Only the innermost loop where `break` is located is terminated.

The loop must be in the same function as `break`. It is an error to execute `break` outside any loop.

See also

`while`, `repeat`, `for`, `continue`, `return`

case

Conditional execution of statements depending on a number or a string.

See also

`switch`, `otherwise`

catch

Error recovery.

See also

`try`

continue

Continue loop from beginning.

Syntax

```
continue
```

Description

When a `continue` statement is executed in the scope of a loop construct (`while`, `repeat` or `for`), statements following `continue` are ignored and a new loop is performed if the loop termination criterion is not fulfilled.

The loop must be in the same function as `continue`. It is an error to execute `continue` outside any loop.

See also

`while`, `repeat`, `for`, `break`

define

Definition of a constant.

Syntax

```
define c = expr
define c = expr;
```

Description

`define c=expr` assign permanently expression `expr` to `c`. It is equivalent to

```
function y = c
  y = expr;
```

Since `c` does not have any input argument, the expression is usually constant. A semicolon may follow the definition, but it does not have any effect. `define` must be the first element of the line (spaces and comments are skipped).

Examples

```
define e = exp(1);
define g = 9.81;
define c = 299792458;
define G = 6.672659e-11;
```

See also

`function`

for

Loop controlled by a variable which takes successively the value of the elements of a vector or a list.

Syntax

```
for v = vect
    s1
    ...
end

for v = list
    s1
    ...
end
```

Description

The statements between the `for` statement and the corresponding end are executed repeatedly with the control variable v taking successively every column of `vect` or every element of list `list`. Typically, `vect` is a row vector defined with the range operator.

You can change the value of the control variable in the loop; however, next time the loop is repeated, that value is discarded and the next column of `vect` is fetched.

Examples

```
for i = 1:3; i, end
  i =
    1
  i =
    2
  i =
    3
for i = (1:3)'; i, end
  i =
    1
    2
    3
for i = 1:2:5; end; i
  i =
    5
for i = 1:3; break; end; i
  i =
    1
for el = {1,'abc',{2,5}}; el, end
  el =
    1
  el =
    abc
  el =
    {2,5}
```

See also

`while`, `repeat`, `break`, `continue`, variable assignment

function endfunction

Definition of a function, operator, or method.

Syntax

```
function f
   statements

function f(x1, x2, ...)
   statements

function f(x1, x2 = expr2, ...)
   statements

function y = f(x1, x2, ...)
   statements

function (y1,y2,...) = f(x1,x2,...)
   statements

function ... class::method ...
   statements

function ...
   statements
endfunction
```

Description

New functions can be written to extend the capabilities of LME. They begin with a line containing the keyword `function`, followed by the list of output arguments (if any), the function name, and the list of input arguments between parenthesis (if any). The output arguments must be enclosed between parenthesis or square brackets if they are several. One or more variable can be shared in the list of input and output arguments. When the execution of the function terminates (either after the last statement or because of the command `return`), the current value of the output arguments, as set by the function's statements, is given back to the caller. All variables used in the function's statements are local; their value is undefined before the first assignment (and it is illegal to use them in an expression), and is not shared with variables in other functions or with recursive calls of the same function. Different kinds of variables can be declared explicitly with `global` and `persistent`.

When multiple functions are defined in the same code source (for instance in a library), the body of a function spans from its header to the next function or until the `endfunction` keyword, whichever comes first. Function definitions cannot be nested. `endfunction` is required only when the function definition is followed by code to be executed outside the scope of any function. This includes mixed code and function definitions entered in one large entry in a command-line interface, or applications where code is mainly provided as statements, but where function definitions can help and separate libraries are not wished (note that libraries cannot contain code outside function definitions; they do not require `endfunction`). Both `function` and `endfunction` appear usually at the beginning of a line, but are also permitted after a semicolon or a comma.

Variable number of arguments

Not all of the input and output arguments are necessarily specified by the caller. The caller fixes the number of input and output arguments, which can be obtained by the called function with `nargin` and `nargout`, respectively. Unspecified input arguments (from nargin+1 to the last one) are undefined, unless a default value is provided in the function definition: with the definition `function f(x,y=2)`, y is 2 when `f` is called with a single input argument. Unused output arguments (from nargout+1 to the last one) do not have to be set, but may be.

Functions which accept an unspecified number of input and/or output arguments can use the special variables `varargin` and `varargout`, which are lists of values corresponding to remaining input and output arguments, respectively.

Named arguments

The caller can pass some or all of the input arguments by name, such as `f(x=2)`. Named arguments must follow unnamed ones. Their order does not have to match the order of the input arguments in the function declaration, and some arguments can be missing. Missing arguments are set to their default value if it exists, or left undefined. Undefined arguments can be detected with `isdefined`, or the error caused by their use caught by `try`.

Functions which accept unspecified named arguments or which do not want to expose the argument names used in their implementation can use the special variable `namedargin`, which is a structure containing all named arguments passed by the caller.

Unused arguments

Character ˜ stands for an unused argument. It can be used as a placeholder for an input argument name in the function definition, or in the

list of output arguments specified for the function call.

If function `f` is defined with function header `function f(x,~)`, it accepts two input arguments, the first one assigned to `x` and the second one discarded. This can be useful if `f` is called by code which expects a function with two input arguments.

In `(a,~,c)=f`, function `f` is called to provide three output arguments (`nargout==3`), but the second output argument is discarded.

Operator overloading

To redefine an operator (which is especially useful for object methods; see below), use the equivalent function, such as `plus` for operator +. The complete list is given in the section about operators.

To define a method which is executed when one of the input arguments is an object of class `class` (or a child in the classes hierarchy), add `class::` before the method (function) name. To call it, use only the method name, not the class name.

Examples

Function with optional input and output arguments:

```
function (Sum, Prod) = calcSumAndProd(x, y)
  if nargout == 0
    return;           % nothing to be computed
  end
  if nargin == 0      % make something to be computed...
    x = 0;
  end
  if nargin <= 1       % sum of elements of x
    Sum = sum(x);
  else                % sum of x and y
    Sum = x + y;
  end
  if nargout == 2     % also compute the product
    if nargin == 1    % product of elements of x
      Prod = prod(x);
    else              % product of x and y
      Prod = x .* y;
    end
  end
```

Two equivalent definitions:

```
function S = area(a, b = a, ellipse = false)
  S = ellipse ? pi * a * b / 4 : a * b;

function S = area(a, b, ellipse)
  if ~isdefined(b)
    b = a;
```

```
  end
  if ~isdefined(ellipse)
    ellipse = false;
  end
  S = ellipse ? pi * a * b / 4 : a * b;
```

With unnamed arguments only, `area` can be called with values for `a` only, `a` and `b`, or `a`, `b` and `ellipse`. By naming `ellipse`, the second argument can be omitted:

```
S = area(2, ellipse=true)
  S =
    3.1416
```

Function `max` can return the index of the maximum value in a vector. In the following call, the maximum itself is discarded.

```
(~, maxIndex) = max([2,7,3,5])
  maxIndex =
    2
```

See also

`return`, `nargin`, `nargout`, `isdefined`, `varargin`, `varargout`, `namedargin`, `define`, `inline`, `global`, `persistent`

hideimplementation

Hide the implementation of remaining functions in a library.

Syntax

```
hideimplementation
```

Description

In a library, functions which are defined after the `hideimplementation` keyword have their implementation hidden: for errors occuring when they are executed, the error message is the same as if the function was a native function (it does not contain information about the error location in the function or subfunctions), and during debugging, `dbstep in` steps over the function call.

`hideimplementation` may not be placed in the same line of source code as any other command (comments are possible, though).

See also

`public`, `private`, `function`, `use`, `error`, `dbstep`

if elseif else end

Conditional execution depending on the value of one or more boolean expressions.

Syntax

```
if expr
    s1
    ...
end

if expr
    s1
    ...
else
    s2
    ...
end

if expr1
    s1
    ...
elseif expr2
    s2
    ...
else
    s3
    ...
end
```

Description

If the expression following `if` is true (nonempty and all elements different from 0 and false), the statements which follow are executed. Otherwise, the expressions following `elseif` are evaluated, until one of them is true. If all expressions are false, the statements following `else` are executed. Both `elseif` and `else` are optional.

Example

```
if x > 2
  disp('large');
elseif x > 1
  disp('medium');
else
  disp('small');
end
```

See also

`switch`, `while`

include

Include libraries.

Syntax

```
include lib
```

Description

`include lib` inserts the contents of the library file `lib`. Its effect is similar to the `use` statement, except that the functions and constants in `lib` are defined in the same context as the library where `include` is located. Its main purpose is to permit to define large libraries in multiple files in a transparent way for the user. `include` statements must not follow other statements on the same line, and can reference only one library which is searched at the same locations as `use`. They can be used only in libraries.

Since LME replaces `include` with the contents of `lib`, one should be cautious about the public or private context which is preserved between the libraries. It is possible to include a fragment of function without a function header.

See also

`use`, `includeifexists`, `private`, `public`

includeifexists

Include library if it exists.

Syntax

```
includeifexists lib
```

Description

`includeifexists lib` inserts the contents of the library file `lib` if it exists; if the library does not exists, it does nothing.

See also

`include`, `useifexists`, `private`, `public`

otherwise

Conditional execution of statements depending on a number or a string.

See also

`switch`, `case`

private

Mark the beginning of a sequence of private function definitions in a library.

Syntax

```
private
```

Description

In a library, functions which are defined after the `private` keyword are private. `private` may not be placed in the same line of source code as any other command (comments are possible, though).

In a library, functions are either public or private. Private functions can only be called from the same library, while public functions can also be called from contexts where the library has been imported with a `use` command. Functions are public by default.

Example

Here is a library for computing the roots of a second-order polynomial. Only function `roots2` can be called from the outside of the library.

```
private
function d = discr(a, b, c)
  d = b^2 - 4 * a * c;
public
function r = roots2(p)
  a = p(1);
  b = p(2);
  c = p(3);
  d = discr(a, b, c);
  r = [-b+sqrt(d); -b-sqrt(d)] / (2 * a);
```

See also

`public`, `function`, `use`

public

Mark the beginning of a sequence of public function definitions in a library.

Syntax

```
public
```

Description

In a library, functions which are defined after the `public` keyword are public. `public` may not be placed in the same line of source code as any other command (comments are possible, though).

In a library, functions are either public or private. Private functions can only be called from the same library, while public functions can also be called from contexts where the library has been imported with a `use` command. Functions are public by default: the `public` keyword is not required at the beginning of the library.

See also

`private`, `function`, `use`

repeat

Loop controlled by a boolean expression.

Syntax

```
repeat
    s1
    ...
until expr
```

Description

The statements between the `repeat` statement and the corresponding `until` are executed repeatedly (at least once) until the expression of the `until` statement yields true (nonempty and all elements different from 0 and false).

Example

```
v = [];
repeat
  v = [v, sum(v)+1];
until v(end) > 100;
v
    1   2   4   8  16  32  64 128
```

See also

`while`, `for`, `break`, `continue`

return

Early return from a function.

Syntax

```
return
```

Description

`return` stops the execution of the current function and returns to the calling function. The current value of the output arguments, if any, is returned. `return` can be used in any control structure, such as `if`, `while`, or `try`, or at the top level.

Example

```
function dispFactTable(n)
  % display the table of factorials from 1 to n
  if n == 0
    return;  % nothing to display
  end
  fwrite(' i   i!\n');
  for i = 1:n
    fwrite('%2d  %3d\n', i, prod(1:i));
  end
```

See also

`function`

switch

Conditional execution of statements depending on a number or a string.

Syntax

```
switch expr
    case e1
        s1
        ...
    case [e2,e3,...]
        s23
        ...
    case {e4,e5,...}
        s45
        ...
    otherwise
        so
```

```
        ...
end

switch string
    case str1
        s1
        ...
    case str2
        s2
        ...
    case {str3,str4,...}
        s34
        ...
    otherwise
        so
        ...
end
```

Description

The expression of the `switch` statement is evaluated. If it yields a number, it is compared successively to the result of the expressions of the `case` statements, until it matches one; then the statements which follow the `case` are executed until the next `case`, `otherwise` or `end`. If the case expression yields a vector or a list, a match occurs if the switch expression is equal to any of the elements of the case expression. If no match is found, but `otherwise` is present, the statements following `otherwise` are executed. If the switch expression yields a string, a match occurs only in case of equality with a case string expression or any element of a case list expression.

Example

```
switch option
  case 'arithmetic'
    m = mean(data);
  case 'geometric'
    m = prod(data)^(1/length(data));
  otherwise
    error('unknown option');
end
```

See also

`case`, `otherwise`, `if`

try

Error recovery.

Syntax

```
try
  ...
end

try
  ...
catch
  ...
end

try
  ...
catch e
  ...
end
```

Description

The statements after `try` are executed. If an error occurs, execution is switched to the statements following `catch`, if any, or to the statements following `end`. If `catch` is followed by a variable name, a structure describing the error (the result of `lasterror`) is assigned to this variable; otherwise, the error message can be retrieved with `lasterr` or `lasterror`. If no error occurs, the statements between `try` and `end` are ignored.

`try` ignores two errors:

- the interrupt key (Control-Break on Windows, Command-. on macOS, Control-C on other operating systems with a keyboard, timeout in Sysquake Remote);
- an attempt to execute an untrusted function in a sandbox. The error can be handled only outside the sandbox.

Examples

```
a = 1;
a(2), 555
  Index out of range 'a'
try, a(2), end, 555
  555
try, a(2), catch, 333, end, 555
  333
  555
try, a, catch, 333, end, 555
  a =
    1
  555
```

See also
`lasterr`, `lasterror`, `error`

until

End of repeat/until loop.

See also
`repeat`

use

Import libraries.

Syntax
```
use lib
use lib1, lib2, ...
```

Description
Functions can be defined in separate files, called *libraries*. `use` makes them available in the current context, so that they can be called by the functions or statements which follow. Using a library does not make available functions defined in its sublibraries; however, libraries can be used multiple times, in each context where their functions are referenced.

All `use` statements are parsed before execution begins. They can be placed anywhere in the code, typically before the first function. They cannot be skipped by placing them after an `if` statement. Likewise, `try/catch` cannot be used to catch errors; `useifexists` should be used if the absence of the library is to be ignored.

See also
`useifexists`, `include`, `function`, `private`, `public`, `info`

useifexists

Import libraries if they exist.

Syntax
```
useifexists lib
useifexists lib1, lib2, ...
```

Description

`useifexists` has the same syntax and effect as `use`, except that libraries which are not found are ignored without error.

See also

`use`, `include`, `function`, `private`, `public`, `info`

while

Loop controlled by a boolean expression.

Syntax

```
while expr
    s1
    ...
end
```

Description

The statements between the `while` statement and the corresponding `end` are executed repeatedly as long as the expression of the `while` statement yields true (nonempty and all elements different from 0 and false).

If a `break` statement is executed in the scope of the while loop (i.e. not in an enclosed loop), the loop is terminated.

If a `continue` statement is executed in the scope of the while loop, statements following `continue` are ignored and a new loop is performed if the `while` statement yields true.

Example

```
e = 1;
i = 2;
while true  % forever
  eNew = (1 + 1/i) ^ i;
  if abs(e - eNew) < 0.001
    break;
  end
  e = eNew;
  i = 2 * i;
end
e
  2.717
```

See also

`repeat`, `for`, `break`, `continue`, `if`

10.14 Debugging Commands

dbclear

Remove a breakpoint.

Syntax

```
dbclear fun
dbclear fun line
dbclear('fun', line)
dbclear
```

Description

`dbclear fun` removes all breakpoints in function `fun`. `dbclear fun line` or `dbclear('fun',line)` removes the breakpoint in function `fun` at line number `line`.

Without argument, `dbclear` removes all breakpoints.

See also

`dbstop`, `dbstatus`

dbcont

Resume execution.

Syntax

```
dbcont
```

Description

When execution has been suspended by a breakpoint or `dbhalt`, it can be resumed from the command-line interface with `dbcont`.

See also

`dbstop`, `dbhalt`, `dbstep`, `dbquit`

dbhalt

Suspend execution.

Syntax

```
dbhalt
```

Description

In a function, `dbhalt` suspends normal execution as if a breakpoint had been reached. Commands `dbstep`, `dbcont` and `dbquit` can then be used from the command line to resume or abort execution.

See also

`dbstop`, `dbcont`, `dbquit`

dbquit

Abort suspended execution.

Syntax

```
dbquit
```

Description

When execution has been suspended by a breakpoint or `dbhalt`, it can be aborted completely from the command-line interface with `dbquit`.

See also

`dbstop`, `dbcont`, `dbhalt`

dbstack

Chain of function calls.

Syntax

```
dbstack
s = dbstack
dbstack all
s = dbstack('all')
```

Description

`dbstack` displays the chain of function calls which lead to the current execution point, with the line number where the call to the subfunction is made. It can be executed in a function or from the command-line interface when execution is suspended with a breakpoint or `dbhalt`.

`dbstack all` (or `dbstack('all')`) displays the whole stack of function calls. For instance, if two executions are successively suspended at breakpoints, `dbstack` displays only the second chain of function calls, while `dbstack all` displays all functions.

With an output argument, `dbstack` returns the result as a structure array. Field `name` contains the function name (or class and method names), and field `line` the line number. Note that you cannot assign the result of `dbstack` to a new variable in suspended mode.

Examples

```
use stat
dbstop prctile
iqr(rand(1,1000))
  <prctile:45>  if nargin < 3
dbstack
  stat/prctile;45
  stat/iqr;69
```

See also

`dbstop`, `dbhalt`

dbstatus

Display list of breakpoints.

Syntax

```
dbstatus
dbstatus fun
```

Description

`dbstatus` displays the list of all breakpoints. `dbstatus fun` displays the list of breakpoints in function `fun`.

See also

`dbstop`, `dbclear`, `dbtype`

dbstep

Execute a line of instructions.

Syntax

```
dbstep
dbstep in
dbstep out
```

Description

When normal execution is suspended after a breakpoint set with `dbstop` or the execution of function `dbhalt`, `dbstep`, issued from the command line, executes the next line of the suspended function. If the line is the last one of the function, execution resumes in the calling function.

`dbstep in` has the same effect as `dbstep`, except if a subfunction is called. In this case, execution is suspended at the beginning of the subfunction.

`dbstep out` resumes execution in the current function and suspends it in the calling function.

Example

Load library `stdlib` and put a breakpoint at the beginning of function `hankel`:

```
use stdlib
dbstop hankel
```

Start execution of function `hankel` until the breakpoint is reached (the next line to be executed is displayed):

```
hankel(1:3,3:8)
  <hankel:21>   c = c(:);
```

When the execution is suspended, any function can be called. Local variables of the function can be accessed and changed; but no new variable can be created. Here, the list of variables and the value of `c` are displayed:

```
info v
  M (not defined)
  c (1x3)
  r (1x6)
  m (not defined)
  n (not defined)
  ix (not defined)
  M1 (not defined)
c
  c =
    1 2 3
```

Display the stack of function calls:

```
dbstack
  stdlib/hankel;21
```

Execute next line (typing Return with an empty command has the same effect as typing `dbstep`):

```
dbstep
  <hankel:22>   m = length(c);
```

Continue until the end; then normal execution is resumed:

```
dbcont
  ans =
    1 2 3 4 5 6
 2 3 4 5 6 7
 3 4 5 6 7 8
```

Display breakpoint and clear it:

```
dbstatus
  stdlib/hankel;0
dbclear
```

See also

`dbstop`, `dbcont`, `dbquit`

dbstop

Set a breakpoint.

Syntax

```
dbstop fun
dbstop fun line
dbstop('fun', line)
```

Description

`dbstop fun` sets a breakpoint at the beginning of function `fun`. `dbstop fun line` or `dbstop('fun',line)` sets a breakpoint in function `fun` at line `line`.

When LME executes a line where a breakpoint has been set, it suspends execution and returns to the command-line interface. The user can inspect or change variables, executes expressions or other functions, continue execution with `dbstep` or `dbcont`, or abort execution with `dbquit`.

Example

```
use stdlib
dbstop cart2pol
dbstatus
  stdlib/cart2pol;0
dbclear cart2pol
```

See also

`dbhalt`, `dbclear`, `dbstatus`, `dbstep`, `dbcont`, `dbquit`, `dbtype`

dbtype

Display source code with line numbers, breakpoints, and current execution point.

Syntax

```
dbtype fun
dbtype
dbtype('fun', fd=fd)
src = dbtype('fun')
```

Description

`dbtype fun` displays the source code of function `fun` with line numbers, breakpoints, and the position where execution is suspended (if it is in `fun`). Without argument, `dbtype` displays the function which is suspended.

`dbtype` can be used at any time to check the source code of any function known to LME.

By defulat, `dbtype` displays the source code to the standard output channel. A file descriptor can be specified as a named argument `fd`.

With an output argument, `dbtype` returns the function source code as a string.

Example

```
use stdlib
dbstop cart2pol
(phi, r) = cart2pol(1, 2);
  <cart2pol:99>   r = hypot(x, y);
dbstep
  <cart2pol:100>   phi = atan2(y, x);
dbtype
   #  97 function (phi, r, z) = cart2pol(x, y, z)
      98
      99  r = hypot(x, y);
  >  100  phi = atan2(y, x);
```

See also

`dbstatus`, `dbstack`, `echo`

echo

Echo of code before its execution.

Syntax

```
echo on
echo off
echo fun on
echo(state)
echo(state, fd)
echo(fun, state)
echo(fun, state, fd)
```

Description

`echo on` enables the display of an echo of each line of function code before execution. The display includes the function name and the line number. `echo off` disables the echo.

The argument can also be passed as a boolean value with the functional form `echo(state)`: `echo on` is equivalent to `echo(true)`.

`echo fun on` enables echo for function named `fun` only. `echo fun off` disables echo (the function name is ignored); `echo off` has the same effect.

By default, the echo is output to the standard error channel (file descriptor 2). Another file descriptor can be specified as an additional numeric argument, with the functional form only.

Example

Trace of a function:

```
use stdlib
echo on
C = compan([2,5,4]);
  compan  26  if min(size(v)) > 1
  compan  29  v = v(:).';
  compan  30  n = length(v);
  compan  31  M = [-v(2:end)/v(1); eye(n-2, n-1)];
```

Echo stored into a file `'log.txt'`:

```
fd = fopen('log.txt', 'w');
echo(true, fd);
...
echo off
fclose(fd);
```

See also

`dbtype`

10.15 Profiler

profile

Install, remove, or display a function profile.

Syntax

```
profile fun
profile report
profile done

profile function fun
profile off
profile on
profile reset
profile('report', format)
```

Description

The purpose of the profiler is to measure the amount of time spent executing each line of code of a function. This helps in evaluating where effort should be put in order to optimize the code. With LME, a single function can be profiled at any given time. Command `profile` manages all aspects related to profiling, from specifying which function is to be profiled to displaying the results and resuming normal operations.

The time measured for each line includes time spent executing subfunctions called from that line. Only the cumulative times are collected; lines of code in loops are likely to have a larger impact on the overall execution time.

The profile accuracy is limited mainly by two factors:

- The resolution of the timer, which is typically between 1e-9 and 1e-6 second. This has obviously a larger effect on lines executed quickly. Lines which contain scalar assignments or statements like `if` and `for` may completely escape from the timing.
- The time overhead to perform the timing and add the data. Here again, its effect is more dramatic with fast lines.

To profile a function, one usually proceeds in four steps:

Setup `profile fun` sets up profiling for function `fun`. Room in memory is allocated and initialized for collecting the cumulative time of execution for each line in `fun`.

Function execution Each execution of the function adds to the profile data. Since the relative execution times are usually what

is really interesting, you may want to execute the function several times to reduce fluctuations due to rounding errors. Time spent outside the function (such as the time you spend typing the commands at the command-line interface) is not included.

Profile report `profile report` displays a report for the function being profiled. The default format is a listing of all lines with the line number, the cumulative time spent for the line in seconds, its percentage with respect to the time spent in the whole function, and the source code of the line. You can continue executing the function and creating new reports; times are cumulative (but see `profile reset` and `profile off` below).

End `profile done` releases the data structures set up with `profile fun`.

Other options are available. `profile off` suspends profiling, and `profile on` resumes it. When profiling is suspended, calls to the profiled function are not taken into account.

`profile reset` resets all the times and resumes profiling if it was suspended.

`profile function fun` is equivalent to `profile fun`, but it may also be used to profile functions with the same name as one of the options which have a special meaning for `profile`, like `report` or `done`.

`profile('report',format)` produces a report with a special format specified by the string `format`. This string is similar to the format argument of `sprintf`; it is reused for each line of the profiled function. Its characters are output literally, except for sequences which begin with a percent character, whose meaning is given in the table below.

Char.	Meaning
`%%`	single %
`%l`	line number
`%t`	cumulative time
`%p`	percentage of the total time
`%s`	source code of the line

Like with `sprintf`, precision numbers may be inserted between the percent sign and the letter; for instance, `%8.3t` displays the cumulative time in a column of 8 characters with a fractional part of 3 digits. The percentage is displayed only if it is greater than 1 %; otherwise, it is replaced (together with the percent character which may follow it) with spaces. The default format is `'%4l%9.3t%6.1p%% %s\n'`.

Example

We shall profile function `logspace` from library `stdlib` (the source code of this function has been revised since the profiling was done).

```
use stdlib
profile logspace
x = logspace(1,10);
profile report
  13    0.000        function r = logspace(x1, x2, n)
  14    0.000
  15    0.000  14.8%    if nargin < 3
  16    0.000   5.8%       n = 100;
  17    0.000   2.2%    end
  18    0.000  77.1%    r = exp(log(x1)+log(x2/x1)*(0:n-1)/(n-1));
```

While the times spent for all lines are smaller than half a millisecond, the resolution is fine enough to permit relative timing of each line. The function header does not correspond to any code and is not timed. To improve the accuracy of the timing, we repeat the execution 10000 times.

```
for i=1:10000; x = logspace(1,10); end
profile report
  13    0.000        function r = logspace(x1, x2, n)
  14    0.000
  15    0.055   8.9%    if nargin < 3
  16    0.057   9.2%       n = 100;
  17    0.047   7.6%    end
  18    0.458  74.3%    r = exp(log(x1)+log(x2/x1)*(0:n-1)/(n-1));
```

Finally, here is a report with a different format: the first column is the percentage as an integer, a space and the percent sign, followed by spaces and the source code:

```
profile('report', '%3.0p %%  %s\n')
       function r = logspace(x1, x2, n)

  9 %     if nargin < 3
  9 %        n = 100;
  8 %     end
 74 %     r = exp(log(x1) + log(x2/x1) * (0:n-1) / (n-1));
```

See also

`tic`, `toc`, `sprintf`

10.16 Miscellaneous Functions

This section describes functions related to programming: function arguments, error processing, evaluation, memory.

assert

Check that an assertion is true.

Syntax

```
assert(expr)
assert(expr, str)
assert(expr, format, arg1, arg2, ...)
assert(expr, identifier, format, arg1, arg2, ...)
```

Description

`assert(expr)` checks that `expr` is true and throws an error otherwise. Expression `expr` is considered to be true if it is a non-empty array whose elements are all non-zero.

With more input arguments, `assert` checks that `expr` is true and throws the error specified by remaining arguments otherwise. These arguments are the same as those expected by function `error`.

When the intermediate code is optimized, `assert` can be ignored. It should be used only to produce errors at an early stage or as a debugging aid, not to trigger the `try/catch` mechanism. The expression should not have side effects. The most common use of `assert` is to check the validity of input arguments.

Example

```
function y = fact(n)
  assert(length(n)==1 && isreal(n) && n==round(n), 'LME:nonIntArg');
  y = prod(1:n);
```

See also

`error`, `warning`, `try`

builtin

Built-in function evaluation.

Syntax

```
(argout1, ...) = builtin(fun, argin1, ...)
```

Description

`(y1,y2,...)=builtin(fun,x1,x2,...)` evaluates the built-in function `fun` with input arguments `x1`, `x2`, etc. Output arguments are assigned to `y1`, `y2`, etc. Function `fun` is specified by its name as a string.

`builtin` is useful to execute a built-in function which has been redefined.

Example

Here is the definition of operator `plus` so that it can be used with character strings to concatenate them.

```
function r = plus(a, b)
  if ischar(a) && ischar(b)
    r = [a, b];
  else
    r = builtin('plus', a, b);
  end
```

The original meaning of `plus` for numbers is preserved:

```
1 + 2
  3
'ab' + 'cdef'
  abcdef
```

See also

`feval`

clear

Discard the contents of a variable.

Syntax

```
clear
clear(v1, v2, ...)
clear -functions
```

Description

Without argument, `clear` discards the contents of all the local variables, including input arguments. With string input arguments, `clear(v1,v2,...)` discards the contents of the enumerated variables. Note that the variables are specified by strings; `clear` is a normal function which evaluates its arguments if they are enclosed between parenthesis. You can also omit parenthesis and quotes and use command syntax.

`clear` is usually not necessary, because local variables are automatically discarded when the function returns. It may be useful if a large variable is used only at the beginning of a function, or at the command-line interface.

`clear -functions` or `clear -f` removes the definition of all functions. It can be used only from the command-line interface, not in a function.

Examples

In the example below, `clear(b)` evaluates its argument and clears the variable whose name is `'a'`; `clear b`, without parenthesis and quotes, does not evaluate it; the argument is the literal string `'b'`.

```
a = 2;
b = 'a';
clear(b)
a
  Undefined variable 'a'
b
  a
clear b
b
  Undefined variable b
```

See also

variable assignment, `isdefined`

deal

Copy input arguments to output arguments.

Syntax

```
(v1, v2, ...) = deal(e)
(v1, v2, ...) = deal(e1, e2, ...)
```

Description

With a single input argument, `deal` provides a copy of it to all its output arguments. With multiple input arguments, `deal` provides them as output arguments in the same order.

`deal` can be used to assign a value to multiple variables, to swap the contents of two variables, or to assign the elements of a list to different variables.

Examples

Swap variable a and b:

```
a = 2;
b = 'abc';
(a, b) = deal(b, a)
  a =
    abc
  b =
    2
```

Copy the same random matrix to variables x, y, and z:

```
(x, y, z) = deal(rand(5));
```

Assign the elements of list l to variables v1, v2, and v3:

```
l = {1, 'abc', 3:5};
(v1, v2, v3) = deal(l{:})
  v1 =
    1
  v2 =
    abc
  v3 =
    3 4 5
```

See also

varargin, varargout, operator {}

dumpvar

Dump the value of an expression as an assignment to a variable.

Syntax

```
dumpvar(value)
dumpvar(name, value)
dumpvar(fd, name, value)
str = dumpvar(value)
str = dumpvar(name, value)
... = dumpvar(..., fd=fd, NPrec=nPrec)
```

Description

dumpvar(fd,name,value) writes to the channel fd (the standard output by default) a string which would set the variable name to value, if it was evaluated by LME. If name is omitted, only the textual representation of value is written. A file descriptor can also be specified as a named argument fd.

With an output argument, dumpvar stores result into a string and produces no output.

In addition to fd, dumpvar also accepts named argument NPrec for the maximum number of digits in floating-point numbers.

Examples

```
dumpvar(2+3)
  5
a = 6; dumpvar('a', a)
```

```
  a = 6;
s = 'abc'; dumpvar('string', s)
  string = 'abc';
dumpvar('x', 1/3, NPrec=5)
  x = 0.33333;
```

See also

`fprintf`, `sprintf`, `str2obj`

error

Display an error message and abort the current computation.

Syntax

```
error(str)
error(format, arg1, arg2, ...)
error(identifier, format, arg1, arg2, ...)
error(identifier)
error(..., throwAsCaller=b)
```

Description

Outside a try block, `error(str)` displays string `str` as an error message and the computation is aborted. With more arguments, `error` use the first argument as a format string and displays remaining arguments accordingly, like `fprintf`.

In a `try` block, `error(str)` throws a user error without displaying anything.

An error identifier can be added in front of other arguments. It is a string made of at least two segments separated by semicolons. Each segment has the same syntax as variable or function name (i.e. it begins with a letter or an underscore, and it continues with letters, digits and underscores.) The identifier can be retrieved with `lasterr` or `lasterror` in the `catch` part of a `try/catch` construct and helps to identify the error. For errors thrown by LME built-in functions, the first segment is always LME.

The identifier of an internal error (an error which can be thrown by an LME builti-in function, such as `'LME:indexOutOfRange'`), can be used as the only argument; then the standard error message is displayed.

`error` also accepts a boolean named argument `throwAsCaller`. If it is `true`, the context of the error is changed so that the function calling `error` appears to throw the error itself. It is useful for fully debugged functions whose internal operation can be hidden. Keyword `hideimplementation` has a similar effect at the level of a library, by hiding the internal error handling in all its functions.

Examples

```
error('Invalid argument.');
  Invalid argument.
o = 'ground';
error('robot:hit', 'The robot is going to hit %s', o);
  The robot is going to hit ground
lasterror
  message: 'The robot is going to hit ground'
  identifier: 'robot:hit'
```

Definition of a function which checks its input arguments, and a test function which calls it:

```
function xmax = largestRoot(a, b, c)
  // largest root of a x^2 + b x + c = 0
  if b^2 - 4 * a * c < 0
    error('No real root', throwAsCaller=true);
  end
  xmax = (-b + sqrt(b^2 - 4 * a * c)) / (2 * a);
function test
  a = largestRoot(1,1,1);
```

Error message:

```
test
No real root (test;8)
```

Error message without `throwAsCaller=true` in the definition of `largestRoot`:

```
test
No real root (largestRoot;4)
-> test;8
```

See also

`warning`, `try`, `lasterr`, `lasterror`, `assert`, `fprintf`, `hideimplementation`

eval

Evaluate the contents of a string as an expression or statements.

Syntax

```
x = eval(str_expression)
eval(str_statement)
```

Description

If `eval` has output argument(s), the input argument is evaluated as an expression whose result(s) is returned. Without output arguments, the input argument is evaluated as statement(s). `eval` can evaluate and assign to existing variables, but cannot create new ones.

Examples

```
eval('1+2')
  3
a = eval('1+2')
  a = 3
eval('a=2+3')
  a = 5
```

See also

`feval`

exist

Existence of a function or variable.

Syntax

```
b = exist(name)
b = exist(name, type)
```

Description

`exist` returns true if its argument is the name of an existing function or variable, or false otherwise. A second argument can restrict the lookup to builtin functions (`'builtin'`), user functions (`'function'`), or variables (`'variable'`).

Examples

```
exist('sin')
  true
exist('cos', 'function')
  false
```

See also

`info`, `isdefined`

feval

Function evaluation.

Syntax

```
(argout1,...) = feval(fun,argin1,...)
```

Description

`(y1,y2,...)=feval(fun,x1,x2,...)` evaluates function `fun` with input arguments x1, x2, etc. Output arguments are assigned to y1, y2, etc. Function `fun` is specified by either its name as a string, a function reference, or an anonymous or inline function.

If a variable `f` contains a function reference or an anonymous or inline function, `f(arguments)` is equivalent to `feval(f,arguments)`.

Examples

```
y = feval('sin', 3:5)
  y =
    0.1411 -0.7568 -0.9589
y = feval(@(x) sin(2*x), 3:5)
  y =
    -0.2794 0.9894 -0.544
fun = @(x) sin(2*x);
y = fun(3:5)
  y =
    -0.2794 0.9894 -0.544
```

See also

`builtin`, `eval`, `fevalx`, `apply`, `inline`, operator @

fun2str

Name of a function given by reference or source code of an inline function.

Syntax

```
str = fun2str(funref)
str = fun2str(inlinefun)
```

Description

`fun2str(funref)` gives the name of the function whose reference is `funref`.

`fun2str(inlinefun)` gives the source code of the inline function `inlinefun`.

Examples

```
fun2str(@sin)
  sin
fun2str(inline('x+2*y', 'x', 'y'))
  function y=f(x,y);y=x+2*y;
```

See also

operator @, `str2fun`

info

Information about LME.

Syntax

```
info
info builtin
info date
info errors
info functions
info global
info libraries
info methods
info operators
info persistent
info size
info threads
info usedlibraries
info variables
info(kind, fd=fd)
str = info
SA = info(kind)
```

Description

`info` displays the language version. With an output argument, the language version is given as a string.

`info builtin` displays the list of built-in functions with their module name (modules are subsets of built-in functions). A letter `u` is displayed after each untrusted function (functions which cannot be executed in the sandbox). With an output argument, `info('builtin')` gives a structure array which describes each built-in function, with the following fields:

Field	Description
`name`	function name
`module`	module name
`trusted`	true if the function is trusted

`info operators` displays the list of operators. With an output argument, `info('operators')` gives a list of structures, like `info('builtin')`.

`info functions` displays the list of user-defined functions with the library where they are defined and the line number in the source code. Parenthesis denote functions known by LME, but not loaded; they also indicate spelling errors in function or variable names. With an output argument, `info('functions')` gives a structure array which describes each user-defined function, with the following fields:

Field	**Description**
`library`	library name
`name`	function name
`loaded`	true if loaded
`line`	line number if available, or `[]`

`info methods` displays the list of methods. With an output argument, `info('methods')` gives a structure array which describes each method, with the following fields:

Field	**Description**
`library`	library name
`class`	class name
`name`	function name
`loaded`	true if loaded
`line`	line number if available, or `[]`

`info variables` displays the list of variables with their type and size. With an output argument, `info('variables')` gives a structure array which describes each variable, with the following fields:

Field	**Description**
`name`	function name
`defined`	true if defined

`info global` displays the list of all global variables. With an output argument, `info('global')` gives the list of the global variable names.

`info persistent` displays the list of all persistent variables. With an output argument, `info('persistent')` gives the list of the persistent variable names.

`info libraries` displays the list of all loaded libraries with the libraries they have loaded with `use`. The base context in which direct commands are evaluated is displayed as `(base)`; it is not an actual library and contains no function definition. With an output argument, `info('libraries')` gives a structure array with the following fields:

Field	**Description**
`library`	library name, or `'(base)'`
`sublibraries`	list of sublibraries

`info usedlibraries` displays the list of libraries available in the current context. With an output argument, `info('usedlibraries')` gives the list of the names of these libraries.

`info errors` displays the list of error messages. With an output argument, `info('errors')` gives a structure array which describes each error message, with the following fields:

Field	Description
`id`	error ID
`msg`	error message

`info size` displays the size in bytes of integer numbers (as used for indices and most internal computations), double numbers, single numbers, and pointers; the byte ordering in multibyte values (little-endian if the least-significant byte comes first, else big-endian), and whether arrays are stores column-wise or row-wise. With an output argument, `info('size')` gives them in a structure of six fields:

Field	Description
`int`	integer size
`double`	double size
`single`	single size (or 0)
`ptr`	pointer size
`be`	true if big-endian
`columnwise`	true for column-wise array layout

`info date` displays the compilation date. With an output argument, `info('date')` gives it in a structure:

Field	Description
`date`	year, month, and day in a row vector

`info threads` displays the ID of all threads. With an output argument, `info('threads')` gives a structure array which describes each thread, with the following fields:

Field	Description
`id`	thread ID
`totaltime`	execution time in seconds

Only the first character of the argument is meaningful; `info b` is equivalent to `info builtin`.

A named argument `fd` can specify the output channel; in that case, the command syntax cannot be used.

Examples

```
info
  LME 5.2
info s
  int: 4 bytes
```

```
  double: 8 bytes
  ptr: 4 bytes
  little endian
  array layout: row-wise
info b
  LME/abs
  LME/acos
  LME/acosh
  (etc.)
info v
  ans (1x1 complex)
vars = info('v')
  var =
    2x1 struct array (2 fields)
```

List of variables displayed on channel 2 (standard error channel):

```
info('v', fd=2)
```

Library hierarchy in the command-line interface:

```
use lti
info l
  (base): _cli, lti
  _cli: lti
  lti: polynom
  polynom
```

The meaning is as follows: `(base)` is the context where commands are evaluated; functions defined from the command-line interface, stored in `_cli`, and in `lti` can be called from there. Functions defined from the command-line interface also have access to the definitions of `lti`. Library `lti` uses library `polynom`, but functions defined in `polynom` cannot be called directly from commands (`polynom` does not appear as a sublibrary of `(base)` or `_cli`). Finally, library `polynom` does not import a sublibrary itself.

See also

`inmem`, `which`, `exist`, `use`

isequal

Comparison.

Syntax

```
b = isequal(a, b, ...)
```

Description

`isequal` compares its input arguments and returns true if all of them are equal, and false otherwise. Two numeric, logical and/or char arrays are considered to be equal if they have the same size and if their corresponding elements have the same value; an array which has at least one NaN (not a number) element is not equal to any other array. Two lists, cell arrays, structures or structure arrays are equal if the corresponding elements or fields are equal. Structure fields do not have to be in the same order.

`isequal` differs from operator == in that it results in a scalar logical value and arrays do not have to have the same size. It differs from operator === in that it does not require the type or the structure field order to agree, and in the way NaN is interpreted.

See also

operator ==, operator ===

inline

Creation of inline function.

Syntax

```
fun = inline(funstr)
fun = inline(expr)
fun = inline(expr, arg1, ...)
fun = inline(funstr, param)
fun = inline(expr, arg1, ..., paramstruct)
fun = inline(expr, ..., true)
```

Description

Inline function are LME objects which can be evaluated to give a result as a function of their input arguments. Contrary to functions declared with the `function` keyword, inline functions can be assigned to variables, passed as arguments, and built dynamically. Evaluating them with `feval` is faster than using `eval` with a string, because they are compiled only once to an intermediate code. They can also be used as the argument of functions such as `fzero` and `fmin`.

`inline(funstr)` returns an inline function whose source code is `funstr`. Input argument `funstr` follows the same syntax as a plain function. The function name is ignored.

`inline(expr)` returns an inline function with one implicit input argument and one result. The input argument `expr` is a string which evaluates to the result. The implicit input argument of the inline function is a symbol made of a single lower-case letter different from `i` and

`j`, such as `x` or `t`, which is found in `expr`. If several such symbols are found, the one closer to `x` in alphabetical order is picked.

`inline(expr,arg1,...)` returns an inline function with one result and the specified arguments `arg1` etc. These arguments are also given as strings.

Inline functions also accept an additional input argument which correspond to fixed parameters provided when the function is executed. `inline(funstr,param)`, where `funstr` is a string which contains the source code of a function, stores `param` together with the function. When the function is called, `param` is prepended to the list of input arguments.

`inline(expr,args...,paramstruct)` is a simplified way to create an inline function when the code consists of a single expression. `args` is the names of the arguments which must be supplied when the inline function is called, as strings; `paramstruct` is a structure whose fields define fixed parameters.

`inline(expr,...,true)` defines a function which can return as many output arguments as what `feval` (or other functions which call the inline function) expects. Argument `expr` must be a function call itself.

Anonymous functions created with operator `@` are an alternative, often easier way of creating inline functions. The result is the same. Since `inline` is a normal function, it must be used in contexts where fixed parameters cannot be created as separate variables.

Examples

A simple expression, evaluated at x=1 and x=2:

```
fun = inline('cos(x)*exp(-x)');
y = feval(fun, 2)
  y =
    -5.6319e-2
y = feval(fun, 5)
  y =
    1.9113e-3
```

A function of x and y:

```
fun = inline('exp(-x^2-y^2)', 'x', 'y');
```

A function with two output arguments (the string is broken in three lines to have a nice program layout):

```
fun = inline(['function (a,b)=f(v);',...
              'a=mean(v);',...
              'b=prod(v)^(1/length(v));']);
(am, gm) = feval(fun, 1:10)
  am =
```

```
    5.5
  gm =
    4.5287
```

Simple expression with fixed parameter a:

```
fun = inline('cos(a*x)', 'x', struct('a',2));
feval(fun, 3)
  0.9602
```

An equivalent function where the source code of a complete function is provided:

```
fun = inline('function y=f(a,x); y=cos(a*x);', 2);
feval(fun, 3)
  0.9602
```

The same function created with the anonymous function syntax:

```
a = 2;
fun = @(x) cos(a*x);
```

A function with two fixed parameters a and b whose values are provided in a list:

```
inline('function y=f(p,x);(a,b)=deal(p{:});y=a*x+b;',{2,3})
```

An inline function with a variable number of output arguments:

```
fun = inline('eig(exp(x))',true);
e = feval(fun, magic(2))
  e =
   -28.1440
    38.2514
(V,D) = feval(fun, magic(2))
  V =
    -0.5455  -0.4921
     0.8381  -0.8705
  D =
   -28.1440   0.0000
     0.0000  38.2514
```

See also

`function`, operator @, `feval`, `eval`

inmem

List of functions loaded in memory.

Syntax

```
inmem
SA = inmem
```

Description

`inmem` displays the list of user-defined functions loaded in memory with the library where they are defined. With an output argument, `inmem` gives the result as a structure array which describes each user-defined function loaded in memory, with the following fields:

Field	Description
`library`	library name
`class`	class name (`''` for functions)
`name`	function name

See also

`info`, `which`

isdefined

Check if a variable is defined.

Syntax

```
isdefined(var)
```

Description

`isdefined(var)` returns true if variable `var` is defined, and false otherwise. Unlike ordinary functions, `isdefined`'s argument must be a variable known to LME, referenced by name without quotes, and not an arbitrary expression. A variable is undefined in the following circumstances:

- function input argument when the function call does not supply enough values;
- function output argument which has not been assigned to, in the function itself, not in a function call;
- function local variable before its first assignment;
- function local variable after it has been cleared with function `clear`.

At command-line interface, `clear` usually discards completely variables.

Example

Let function `f` be defined as

```
function f(x)
  if isdefined(x)
    disp(x);
  else
    disp('Argument x is not defined.');
  end
```

Then

```
f
  Argument x is not defined.
f(3)
  3
```

See also

`nargin`, `exist`, `which`, `clear`, `function`

isfun

Test for an inline function or function reference.

Syntax

```
b = isfun(obj)
```

Description

`isfun(obj)` returns true if `obj` is an inline function or a function reference, or false otherwise.

See also

`isa`, `class`, `fun2str`

isglobal

Test for the existence of a global variable.

Syntax

```
b = isglobal(str)
```

Description

`isglobal(str)` returns true if the string `str` is the name of a global variable, defined as such in the current context.

See also

info, exist, isdefined, which

iskeyword

Test for a keyword name.

Syntax

```
b = iskeyword(str)
list = iskeyword
```

Description

iskeyword(str) returns true if the string str is a reserved keyword which cannot be used as a function or variable name, or false otherwise. Keywords include if and global, but not the name of built-in functions like sin or i.

Without input argument, iskeyword gives the list of all keywords.

Examples

```
iskeyword('otherwise')
  true
iskeyword
  {'break','case','catch','continue','else','elseif',
   'end','endfunction','for','function','global',
   'hideimplementation','if','otherwise','persistent',
   'private','public','repeat','return','switch','try',
   'until','use','useifexists','while'}
```

See also

info, which

ismac

Check whether computer runs under macOS.

Syntax

```
b = ismac
```

Description

ismac returns true on macOS, false on other platforms.

See also

isunix, ispc

ispc

Check whether platform is a PC.

Syntax

```
b = ispc
```

Description

`ispc` returns true on Windows, false on other platforms.

See also

`isunix`, `ismac`

isunix

Check whether computer runs under unix.

Syntax

```
b = isunix
```

Description

`isunix` returns true on unix platforms (including Mac OS X and unix-like), false on other platforms.

See also

`ispc`, `ismac`

lasterr

Last error message.

Syntax

```
msg = lasterr
(msg, identifier) = lasterr
```

Description

`lasterr` returns a string which describes the last error. With two output arguments, it also gives the error identifier. It can be used in the `catch` part of the `try` construct.

Example

```
x = 2;
x(3)
  Index out of range
(msg, identifier) = lasterr
  msg =
    Index out of range
  identifier =
    LME:indexOutOfRange
```

See also

`lasterror`, `try`, `error`

lasterror

Last error structure.

Syntax

```
s = lasterror
```

Description

`lasterror` returns a structure which describes the last error. It contains the following fields:

Field	Type	Description
`identifier`	string	short tag which identifies the error
`message`	string	error message

The structure can be used as argument to `rethrow` in the `catch` part of a `try/catch` construct to propagate the error further.

Example

```
x = 2;
x(3)
  Index out of range
lasterror
  message: 'Index out of range'
  identifier: 'LME:indexOutOfRange'
```

See also

`lasterr`, `try`, `rethrow`, `error`

namedargin

Named input arguments.

Syntax

```
function ... = fun(..., namedargin)
```

Description

`namedargin` is a special variable which can be used to collect named input arguments. In the function declaration, it must be used as the last (or unique) input argument. When the function is called with named arguments, all of them are collected and stored in `namedargin` as a structure, where field names correspond to the argument names. With `namedargin`, there is no matching between the named arguments and the argument names in the function declaration. If the function is called without any named argument, `namedargin` is set to an empty structure.

In the body of the function, `namedargin` is a normal variable. Its fields can be accessed with the dot notation `namedargin.name` or `namedargin.(name)`. All functions using structures can be used, such as `fieldnames` or `isfield`. `namedargin` can also be modified or assigned to any value of any type.

When both `varargin` (for a variable number of unnamed arguments) and `namedargin` are used in the same function, they must be the last-but-one and the last arguments in the function declaration, respectively.

Example

Here is a function which calculates the volume of a solid of revolution defined by a function `y=f(x)` between `x=a` and `x=b`, rotating around `y=0`. It accepts the same options as `integral`, given as a single option argument, as named values or both.

```
function V = solidRevVolume(fun, a, b, opt=struct, namedargin)
  opt = structmerge(opt, namedargin);
  V = pi * integral(@(x) fun(x)^2, a, b, opt);
```

It can be called without any option (`opt` is set to its default value, an empty structure):

```
cyl = solidRevVolume(@(x) 1, 0, 1)
  cyl = 3.1416
cone = solidRevVolume(@(x) x, 0, 2, RelTol=1e-4)
  cone = 8.3776
```

See also

`varargin`, `function`, `struct`, `fieldnames`, `structmerge`, operator `.`

nargin

Number of input arguments.

Syntax

```
n = nargin
n = nargin(fun)
```

Description

Calling a function with less arguments than what the function expects is permitted. In this case, the trailing variables are not defined. The function can use the `nargin` function to know how many arguments were passed by the caller to avoid accessing the undefined variables. Named arguments (arguments passed as `name=value` by the caller) are not included in the count.

Note that if you want to have an optional argument before the end of the list, you have to interpret the meaning of the variables yourself. LME always sets the `nargin` first arguments.

There are three other ways to let a function accept a variable number of input arguments: to check if an input argument is defined with `isdefined`, to define default values directly in the function header, or to call `varargin` to collect some or all of the input arguments in a list.

With an input argument, `nargin(fun)` returns the (maximum) number of input arguments a function accepts. `fun` can be the name of a built-in or user function, a function reference, or an inline function. Functions with a variable number of input arguments (such as `fprintf`) give `-1`.

Examples

A function with a default value (`pi`) for its second argument:

```
function x = multiplyByScalar(a,k)
if nargin < 2  % multiplyByScalar(x)
  k = pi;         % same as multiplyByScalar(x,pi)
end
x = k * a;
```

A function with a default value (standard output) for its first argument. Note how you have to interpret the arguments.

```
function fprintstars(fd,n)
if nargin == 1  % fprintstars(n) to standard output
  fprintf(repmat('*',1,fd));  % n is actually stored in fd
else
  fprintf(fd, repmat('*',1,n));
end
```

Number of input arguments of function `plus` (usually called as the infix operator "+"):

```
nargin('plus')
  2
```

See also

`nargout`, `varargin`, `isdefined`, `function`

nargout

Number of output arguments.

Syntax

```
n = nargout
n = nargout(fun)
```

Description

A function can be called with between 0 and the number of output arguments listed in the function definition. The function can use `nargout` to check whether some output arguments are not used, so that it can avoid computing them or do something else.

With one argument, `nargout(fun)` returns the (maximum) number of output arguments a function can provide. `fun` can be the name of a built-in or user function, a function reference, or an inline function. Functions with a variable number of output arguments (such as `feval`) give `-1`.

Example

A function which prints nicely its result when it is not assigned or used in an expression:

```
function y = multiplyByTwo(x)
if nargout > 0
  y = 2 * x;
else
  fprintf('The double of %f is %f\n', x, 2*x);
end
```

Maximum number of output arguments of `svd`:

```
nargout('svd')
  3
```

See also

`nargin`, `varargout`, `function`

rethrow

Throw an error described by a structure.

Syntax

```
rethrow(s)
rethrow(s, throwAsCaller=b)
```

Description

`rethrow(s)` throws an error described by structure `s`, which contains the same fields as the output of `lasterror`. `rethrow` is typically used in the `catch` part of a `try/catch` construct to propagate further an error; but it can also be used to initiate an error, like `error`.

`rethrow` also accepts a boolean named argument `throwAsCaller`. If it is `true`, the context of the error is changed so that the function calling `rethrow` appears to throw the error itself. It is useful for fully debugged functions whose internal operation can be hidden.

Example

The error whose identifier is `'LME:indexOutOfRange'` is handled by `catch`; other errors are not.

```
try
  ...
catch
  err = lasterror;
  if err.identifier === 'LME:indexOutOfRange'
    ...
  else
    rethrow(err);
  end
end
```

See also

`lasterror`, `try`, `error`

str2fun

Function reference.

Syntax

```
funref = str2fun(str)
```

Description

`str2fun(funref)` gives a function reference to the function whose name is given in string `str`. It has the same effect as operator @, which is preferred when the function name is fixed.

Examples

```
str2fun('sin')
  @sin
@sin
  @sin
a = 'cos';
str2fun(a)
  @cos
```

See also

operator @, `fun2str`

str2obj

Convert to an object its string representation.

Syntax

```
obj = str2obj(str)
```

Description

`str2obj(str)` evaluates string `str` and gives its result. It has the inverse effect as `dumpvar` with one argument. It differs from `eval` by restricting the syntax it accepts to literal values and to the basic constructs for creating complex numbers, arrays, lists, structures, objects, and other built-in types.

Examples

```
str2obj('1+2j')
  1 + 2j
str = dumpvar({1, 'abc', 1:100})
  str =
    {1, ...
     'abc', ...
     [1:100]}
str2obj(str)
  {1,'abc',real 1x100}
eval(str)
  {1,'abc',real 1x100}
str2obj('sin(2)')
  Bad argument 'str2obj'
eval('sin(2)')
  0.9093
```

See also

`eval`, `dumpvar`

varargin

Remaining input arguments.

Syntax

```
function ... = fun(..., varargin)
function ... = fun(..., varargin, namedargin)
l = varargin
```

Description

`varargin` is a special variable which can be used to collect input arguments. In the function declaration, it must be used after the normal input arguments; if `namedargin` is also present, `varargin` immediately precedes it. When the function is called with more arguments than what can be assigned to the other arguments, remaining ones are collected in a list and assigned to `varargin`. In the body of the function, `varargin` is a normal variable. Its elements can be accessed with the brace notation `varargin{i}`. `nargin` is always the total number of arguments passed to the function by the caller.

When the function is called with fewer arguments than what is declared, `varargin` is set to the empty list, `{}`.

Example

Here is a function which accepts any number of square matrices and builds a block-diagonal matrix:

```
function M = blockdiag(varargin)
  M = [];
  for block = varargin
    // block takes the value of each input argument
    (m, n) = size(block);
    M(end+1:end+m,end+1:end+n) = block;
  end
```

In the call below, `varargin` contains the list `{ones(3),2*ones(2),3}`.

```
blockdiag(ones(3),2*ones(2),3)
     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
     0     0     0     2     2     0
     0     0     0     2     2     0
     0     0     0     0     0     3
```

See also

`nargin`, `namedargin`, `varargout`, `function`

varargout

Remaining output arguments.

Syntax

```
function (..., varargout) = fun(...)
varargout = ...
```

Description

`varargout` is a special variable which can be used to dispatch output arguments. In the function declaration, it must be used as the last (or unique) output argument. When the function is called with more output arguments than what can be obtained from the other arguments, remaining ones are extracted from the list `varargout`. In the body of the function, `varargout` is a normal variable. Its value can be set globally with the brace notation `{...}` or element by element with `varargout{i}`. `nargout` can be used to know how many output arguments to produce.

Example

Here is a function which differentiates a vector of values as many times as there are output arguments:

```
function varargout = multidiff(v)
  for i = 1:nargout
    v = diff(v);
    varargout{i} = v;
  end
```

In the call below, `[1,3,7,2,5,3,1,8]` is differentiated four times.

```
(v1, v2, v3, v4) = multidiff([1,3,7,2,5,3,1,8])
v1 =
     2     4    -5     3    -2    -2     7
v2 =
     2    -9     8    -5     0     9
v3 =
   -11    17   -13     5     9
v4 =
    28   -30    18     4
```

See also

`nargout`, `varargin`, `function`

variables

Contents of the variables as a structure.

Syntax

```
v = variables
```

Description

`variables` returns a structure whose fields contain the variables defined in the current context.

Example

```
a = 3;
b = 1:5;
variables
  a: 3
  b: real 1x5
  ...
```

See also

`info`

warning

Write a warning to the standard error channel.

Syntax

```
warning(msg)
warning(format, arg1, arg2, ...)
```

Description

`warning(msg)` displays the string `msg`. It should be used to notify the user about potential problems, *not* as a general-purpose display function.

With more arguments, `warning` uses the first argument as a format string and displays remaining arguments accordingly, like `fprintf`.

Example

```
warning('Doesn\'t converge.');
```

See also

`error`, `disp`, `fprintf`

which

Library where a function is defined.

Syntax

```
fullname = which(name)
```

Description

`which(name)` returns an indication of where function `name` is defined. If `name` is a user function or a method prefixed with its class and two colons, the result is `name` prefixed with the library name and a slash. If `name` is a built-in function, it is prefixed with `(builtin)`; a variable, with `(var)`; and a keyword, with `(keyword)`. If `name` is unknown, `which` returns the empty string.

Examples

```
which logspace
  stdlib/logspace
which polynom::plus
  polynom/polynom::plus
which sin
  (builtin)/sin
x = 2;
which x
  (var)/x
```

See also

`info`, `isdefined`

10.17 Sandbox Function

sandbox

Execute untrusted code in a secure environment.

Syntax

```
sandbox(str)
sandbox(str, varin)
varout = sandbox(str)
varout = sandbox(str, varin)
```

Description

`sandbox(str)` executes the statements in string `str`. Functions which might do harm if used improperly are disabled; they include those related to the file system, to devices and to the network. Global and persistent variables are forbidden as well; but local variables can be created. The same restrictions apply to functions called directly or

indirectly by statements in `str`. The purpose of `sandbox` is to permit the evaluation of code which comes from untrusted sources, such as the Internet.

`sandbox(str,varin)` evaluates the statements in string `str` in a context with local variables equal to the fields of structure `varin`.

With an output argument, `sandbox` collects the contents of all variables in the fields of a single structure.

An error is thrown when the argument of `sandbox` attempts to execute one of the functions which are disabled. This error can be caught by a `try`/`catch` construct outside `sandbox`, but not inside its argument, so that unsuccessful attempts to circumvent the sandbox are always reported to the appropriate level.

Examples

Evaluation of two assignments; the second value is displayed, and the variables are discarded at the end of the evaluation.

```
sandbox('a=2; b=3:5');
b =
  3 4 5
```

Evaluation of two assignments; the contents of the variables are stored in structure `result`.

```
result = sandbox('a=2; b=3:5;')
  result =
    a: 2
    b: real 1x3
```

Evaluation with local variables `x` and `y` initialized with the field of a structure. Variable `z` is local to the sandbox.

```
in.x = 12;
in.y = 1:10;
sandbox('z = x + y', in);
  z =
    13 14 15 16 17 18 19 20 21 22
```

Attempt to execute the untrusted function `fopen` and to hide it from the outside. Both attempts fail: `fopen` is trapped and the security violation error is propagated outside the sandbox.

```
sandbox('try; fd=fopen('/etc/passwd'); end');
  Security violation 'fopen'
```

See also

`sandboxtrust`, `eval`, `variables`

sandboxtrust

Escape the sandbox restrictions.

Syntax

```
sandboxtrust(fun)
```

Description

`sandboxtrust(fun)` sets a flag associated with function `fun` so that `fun` is executed without restriction, even when called from a sandbox. All functions called directly or indirectly from a trusted function are executed without restriction, except if a nested call to `sandbox` is performed. Argument `fun` can be a function reference or the name of a function as a string; the function must be a user function, not a built-in one.

The purpose of `sandboxtrust` is to give back some of the capabilities of unrestricted code to code executed in a sandbox. For instance, if unsecure code must be able to read the contents of a specific file, a trusted function should be written for that. It is very important for the trusted function to check carefully its arguments, such as file paths or URL.

Example

Function which reads the contents of file `'data.txt'`:

```
function data = readFile
  fd = fopen('data.txt');
  data = fread(fd, inf, '*char');
  fclose(fd);
```

Execution of unsecure code which may read this file:

```
sandboxtrust(@readFile);
sandbox('d = readFile;');
```

See also

`sandbox`

10.18 Help Function

help

Help about an LME function.

Syntax

```
help functionname
help 'operator'
help
help(str, fd=fd)
b = help(str)
```

Description

`help functionname` displays a help message about function whose name is `functionname`. `help 'operator'` displays a help message about an operator. For methods (functions for arguments of a specific class), the class should be specified as `class::method`, except for constructors where both `class` and `class::class` are recognized. Without argument, `help` displays a message about the help command itself.

For user functions, `help` uses the first comment which immediately follows the function header. The comment may be a continuous set of lines beginning with `%` or `//`, or a single block of lines delimited with `/*` and `*/`. Lines which contain `/*` and `*/` cannot have any other character, not even spaces. Alternatively for user functions, or for built-in or extension functions, the help text is found in files with suffix ".hlp" in the same folders as libraries. For functions unknown to LME (such as functions defined in libraries which have not been loaded with `use`), `help` searches in file `other.hlp`, which typically includes each library `hlp` files with `includeifexists` statements.

If no matching function is found, `help` has the same effect as `lookfor`, i.e. it proposes a list of functions whose short description contains the string passed as argument (or just the method name if the argument has the syntax `class::method`).

A named argument `fd` can specify the output channel; in that case, the command syntax cannot be used.

With an output argument, `help` returns `true` if help is found for its input argument, `false` otherwise. Help is not displayed. The `lookfor` fallback is not attempted.

Examples

Help about function `sin`:

```
help sin
Sine.

SYNTAX
y = sin(x)
...
```

Help about operator +:

```
help '+'
Addition.

SYNTAX
x + y
M1 + M2
...
```

Source code of function `dctmtx` with its help comment block:

```
function T = dctmtx(n)
 /*
 Discrete cosine transform matrix.

 SYNTAX
 T = dctmtx(n)

 DESCRIPTION
 dctmtx(n) returns an n-by-n square matrix T such that
 Y=T*y is the discrete cosine transform of the columns
 ...
 */
     T = [repmat(1/sqrt(n),1,n); ...
          sqrt(2/n)*cos(pi/(2*n)*repmat(1:2:2*n,n-1,1)...
              .*repmat((1:n-1)',1,n))];
```

See also

`lookfor`, `which`

lookfor

Search functions.

Syntax

```
lookfor str
lookfor('str')
lookfor('str', fd=fd)
```

Description

`lookfor str` searches the characters `str` in the short description of all commands and functions and displays all the matches. Case is ignored. If `str` contains spaces or non-alphanumeric characters, the syntax `lookfor('str')` must be used.

A named argument `fd` can specify the output channel; in that case, the command syntax cannot be used.

Example

```
lookfor arc
  lookfor   Search functions.
  acos      Arc cosine.
  asin      Arc sine.
  atan      Arc tangent.
```

See also

`help`

10.19 Operators

Operators are special functions with a syntax which mimics mathematical arithmetic operations like the addition and the multiplication. They can be infix (such as x+y), separating their two arguments (called *operands*); prefix (such as `-x`), placed before their unique operand; or postfix (such as `M'`), placed after their unique operand. In Sysquake, their arguments are always evaluated from left to right. Since they do not require parenthesis or comma, their priority matters. Priority specifies when subexpressions are considered as a whole, as the argument of some operator. For instance, in the expression a+b∗c, where ∗ denotes the multiplication, the evaluation could result in `(a+b)*c` or a+(b∗c); however, since operator ∗'s priority is higher than operator +'s, the expression yields a+(b∗c) without ambiguity.

Here is the list of operators, from higher to lower priority:

```
' .'
^ .^
- (unary)
* .* / ./ \ .\
+ -
== ~= < > <= >= ===~==
~
&
|
&&
||
:  ?
,
;
```

Most operators have also a functional syntax; for instance, a+b can also be written `plus(a,b)`. This enables their overriding with new definitions and their use in function references or functions such as `feval` which take the name of a function as an argument.

Here is the correspondence between operators and functions:

`[a;b]`	`vertcat(a,b)`	`a-b`	`minus(a,b)`
`[a,b]`	`horzcat(a,b)`	`a*b`	`mtimes(a,b)`
`a:b`	`colon(a,b)`	`a/b`	`mrdivide(a,b)`
`a:b:c`	`colon(a,b,c)`	`a\b`	`mldivide(a,b)`
`a|b`	`or(a,b)`	`a.*b`	`times(a,b)`
`a&b`	`and(a,b)`	`a./b`	`rdivide(a,b)`
`a<=b`	`le(a,b)`	`a.\b`	`ldivide(a,b)`
`a<b`	`lt(a,b)`	`a^b`	`mpower(a,b)`
`a>=b`	`ge(a,b)`	`a.^b`	`power(a,b)`
`a>b`	`gt(a,b)`	`~a`	`not(a)`
`a==b`	`eq(a,b)`	`-a`	`uminus(a)`
`a~=b`	`ne(a,b)`	`+a`	`uplus(a)`
`a===b`	`same(a,b)`	`a'`	`ctranspose(a)`
`a~==b`	`unsame(a,b)`	`a.'`	`transpose(a)`
`a+b`	`plus(a,b)`		

Operator which do *not* have a corresponding function are `?:`, `&&` and `||` because unlike functions, they do not always evaluate all of their operands.

Operator ( )

Parenthesis.

Syntax

```
(expr)
v(:)
v(index)
v(index1, index2)
v(:, index)
v(index, :)
v(select)
v(select1, select2)
v(:,:)
```

Description

A pair of parenthesis can be used to change the order of evaluation. The subexpression it encloses is evaluated as a whole and used as if it was a single object. Parenthesis serve also to indicate a list of input or output parameters; see the description of the `function` keyword.

The last use of parenthesis is for specifying some elements of an array or list variable.

Arrays: In LME, any numeric object is considered as an array of two dimensions or more. Therefore, at least two indices are required

to specify a single element; the first index specifies the row, the second the column, and so on. In some circumstances, however, it is convenient to consider an array as a vector, be it a column vector, a row vector, or even a matrix whose elements are indexed row-wise (or on some platforms). For this way of handling arrays, a single index is specified.

The first valid value of an index is always 1. The array whose elements are extracted is usually a variable, but can be any expression: an expression like `[1,2;3,4](1,2)` is valid and gives the 2nd element of the first row, i.e. 3.

In all indexing operations, several indices can be specified simultaneously to extract more than one element along a dimension. A single colon means all the elements along the corresponding dimension.

Instead of indices, the elements to be extracted can be selected by the true values in a logical array of the same size as the variable (the result is a column vector), or in a logical vector of the same size as the corresponding dimension. Calculating a boolean expression based on the variable itself used as a whole is the easiest way to get a logical array.

Variable indexing can be used in an expression or in the left hand side of an assignment. In this latter case, the right hand size can be one of the following:

- An array of the same size as the extracted elements.
- A scalar, which is assigned to each selected element of the variable.
- An empty matrix `[]`, which means that the selected elements should be deleted. Only whole rows or columns (or (hyper)planes for arrays of more dimensions) can be deleted; i.e. `a(2:5,:) = []` and `b([3,6:8]) = []` (if b is a row or column vector) are legal, while `c(2,3) = []` is not.

When indices are larger than the dimensions of the variable, the variable is expanded; new elements are set to 0 for numeric arrays, false for logical arrays, the nul character for character array, and the empty array `[]` for cell arrays.

Lists: In LME, lists have one dimension; thus a single index is required. Be it with a single index or a vector of indices, indexed elements are grouped in a list. New elements, also provided in a list, can be assigned to indexed elements; if the list to be assigned has a single element, the element is assigned to every indexed element of the variable.

Cell arrays: cell arrays are subscripted like other arrays. The result, or the right-hand side of an assignment, is also a cell array, or a list for the syntax `v(select)` (lists are to cell arrays what column

vectors are to non-cell arrays). To create a single logical array for selecting some elements, function `cellfun` may be useful. To remove cells, the right-hand side of the assignment can be the empty list `{}` or the empty array `[]`.

Structure arrays: access to structure array fields combines subscripting with parenthesis and structure field access with dot notation. When the field is not specified, parenthesis indexing returns a structure or structure array. When indexing results in multiple elements and a field is specified, the result is a value sequence.

Examples

Ordering evaluation:

```
(1+2)*3
  9
```

Extracting a single element, a row, and a column:

```
a = [1,2,3; 4,5,6];
a(2,3)
  6
a(2,:)
  4 5 6
a(:,3)
  3
  6
```

Extracting a sub-array with contiguous rows and non-contiguous columns:

```
a(1:2,[1,3])
  1 3
  4 6
```

Array elements as a vector:

```
a(3:5)
  3
  4
  5
a(:)
  1
  2
  3
  4
  5
  6
```

Selections of elements where a logical expression is true:

```
a(a>=5)
  5
  6
a(:, sum(a,1) > 6)
  2 3
  5 6
```

Assignment:

```
a(1,5) = 99
  a =
    1 2 3 0 99
    4 5 6 0 0
```

Extraction and assignment of elements in a list:

```
a = {1,[2,7,3],'abc',magic(3),'x'};
a([2,5])
  {[2,7,3],'x'}
a([2,5]) = {'ab','cde'}
  a =
    {1,'ab','abc',[8,1,6;3,5,7;4,9,2],'cde'}
a([2,5]) = {[3,9]}
  a =
    {1,[3,9],'abc',[8,1,6;3,5,7;4,9,2],[3,9]}
```

Removing elements in a list ({} and [] have the same effect here):

```
a(4) = {}
  a =
    {1,[3,9],'abc',[3,9]}
a([1, 3]) = []
  a =
    {[3,9],[3,9]}
```

Replacing NaN with empty arrays in a cell array:

```
C = {'abc', nan; 2, false};
C(cellfun(@(x) any(isnan(x(:))), C)) = {[]};
```

Element in a structure array:

```
SA = structarray('a',{1,[2,3]},'b',{'ab','cde'});
SA(1).a
  2 3
SA(2).b = 'X';
```

When assigning a new field and/or a new element of a structure array, the new field is added to each element and the size of the array is expanded; fields are initialized to the empty array [].

```
SA(3).c = true;
SA(1).c
  []
```

See also

Operator {}, operator ., end, reshape, variable assignment, operator [], subsref, subsasgn, cellfun

Operator [ ]

Brackets.

Syntax

```
[matrix_elements]
```

Description

A pair of brackets is used to define a 2-d array given by its elements or by submatrices. The operator , (or spaces) is used to separate elements on the same row, and the operator ; (or newline) is used to separate rows. Since the space is considered as a separator when it is in the direct scope of brackets, it should not be used at the top level of expressions; as long as this rule is observed, each element can be given by an expression.

Inside brackets, commas and semicolons are interpreted as calls to horzcat and vertcat. Brackets themselves have no other effect than changing the meaning of commas, semicolons, spaces, and new lines: the expression [1], for instance, is strictly equivalent to 1. The empty array [] is a special case.

Since horzcat and vertcat also accept cell arrays, brackets can be used to concatenate cell arrays, too.

Examples

```
[1, 2, 3+5]
  1 2 8
[1:3; 2 5 , 9 ]
  1 2 3
  2 5 9
[5-2, 3]
  3 3
[5 -2, 3]
  5 -2 3
[(5 -2), 3]
  3 3
[1 2
3 4]
  1 2
  3 4
[]
  []
```

Concatenation of two cell arrays:

```
C1 = {1; 2};
C2 = {'ab'; false};
[C1, C2]
  2x2 cell array
```

Compare this with the effect of braces, where elements are not concatenated but used as cells:

```
{C1, C2}
  1x2 cell array
```

See also

Operator {}, operator (), operator ,, operator ;

Operator {}

Braces.

Syntax

```
{list_elements}
{cells}
{struct_elements}
v{index}
v{index1, index2, ...}
v{index} = expr
fun(...,v{:},...)
```

Description

A pair of braces is used to define a list, a cell array, a struct, or an n-by-1 struct array given by its elements. When no element has a name (a named element is written `name=value` where `value` can be any expression), the result is a list or a celle array; when all elements have a name, the result is a struct or a struct array.

In a list, the operator `,` is used to separate elements. In a cell array, the operator `,` is used to separate cells on the same row; the operator `;` is used to separate rows. Braces without semicolons produce a list; braces with semicolon(s) produce a cell array.

In a struct, the operator `,` is used to separate fields. In a struct array, the operator `;` is used to separate elements.

`v{index}` is the element of list variable `v` whose index is given. `index` must be an integer between 1 (for the first element) and `length(v)` (for the last element). `v{index}` may be used in an expression to extract an element, or on the left hand-side of the equal sign to assign a new value to an element. Unless it is the target

of an assignment, v may also be the result of an expression. If v is a cell array, `v{index}` is the element number `index`.

`v{index1,index2,...}` gives the specified cell of a cell array.

v itself may be an element or a field in a larger variable, provided it is a list; i.e. complicated assignments like `a{2}.f{3}(2,5)=3` are accepted. In an assignment, when the index (or indices) are larger than the list or cell array size, the variable is expanded with empty arrays `[]`.

In the list of the input arguments of a function call, `v{:}` is replaced with its elements. v may be a list variable or the result of an expression.

Examples

```
x = {1, 'abc', [3,5;7,1]}
  x =
    {1,string,real 2x2}
x{3}
    3 5
    7 1
x{2} = 2+3j
  x =
    {1,2+3j,real 2x2}
x{3} = {2}
  x =
    {1,2+3j,list}
x{end+1} = 123
  x =
    {1,2+3j,list,123}
C = {1, false; 'ab', magic(3)}
  2x2 cell array
C{2, 1}
  ab
a = {1, 3:5};
fprintf('%d ', a{:}, 99);
  1 3 4 5 99
s = {a=1, b='abc'};
s.a
  1
S = {a=1, b='abc'; a=false, b=1:5};
size(S)
  2 1
S(2).b
  1 2 3 4 5
S = {a=1; b=2};
S(1).b
  []
```

See also

operator `,`, operator `[]`, operator `()`, operator `;`, operator `.`, `subsref`, `subsasgn`

Operator . (dot)

Structure field access.

Syntax

```
v.field
v.field = expr
```

Description

A dot is used to access a field in a structure. In `v.field`, v is the name of a variable which contains a structure, and `field` is the name of the field. In expressions, `v.field` gives the value of the field; it is an error if it does not exist. As the target of an assignment, the value of the field is replaced if it exists, or a new field is added otherwise; if v itself is not defined, a structure is created from scratch.

v itself may be an element or a field in a larger variable, provided it is a structure (or does not exists in an assignment); i.e. complicated assignments like `a{2}.f{3}(2,5)=3` are accepted.

If V is a structure array, `V.field` is a value sequence which contains the specified field of each element of V.

The syntax `v.(expr)` permits to specify the field name dynamically at run-time, as the result of evaluating expression `expr`. `v('f')` is equivalent to `v.f`. This syntax is more elegant than functions `getfield` and `setfield`.

Examples

```
s.f = 2
  s =
    f: 2
s.g = 'hello'
  s =
    f: 2
    s: string
s.f = 1:s.f
  s =
    f: real 1x2
    g: string
```

See also

Operator `()`, operator `{}`, `getfield`, `setfield`, `subsref`, `subsasgn`

Operator +

Addition.

Syntax

```
x + y
M1 + M2
M + x
plus(x, y)
+x
+M
uplus(x)
```

Description

With two operands, both operands are added together. If both operands are matrices with a size different from 1-by-1, their size must be equal; the addition is performed element-wise. If one operand is a scalar, it is added to each element of the other operand.

With one operand, no operation is performed, except that the result is converted to a number if it was a string or a logical value, like with all mathematical operators and functions. For strings, each character is replaced with its numeric encoding. The prefix + is actually a synonym of `double`.

`plus(x,y)` is equivalent to x+y, and `uplus(x)` to +x. They can be used to redefine these operators for objects.

Example

```
2 + 3
  5
[1 2] + [3 5]
  4 7
[3 4] + 2
  5 6
```

See also

operator `-`, `sum`, `addpol`, `double`

Operator -

Subtraction or negation.

Syntax

```
x - y
M1 - M2
M - x
minus(x, y)
-x
-M
uminus(x)
```

Description

With two operands, the second operand is subtracted from the first operand. If both operands are matrices with a size different from 1-by-1, their size must be equal; the subtraction is performed element-wise. If one operand is a scalar, it is repeated to match the size of the other operand.

With one operand, the sign of each element is changed.

`minus(x,y)` is equivalent to `x-y`, and `uminus(x)` to `-x`. They can be used to redefine these operators for objects.

Example

```
2 - 3
  -1
[1 2] - [3 5]
  -2 -3
[3 4] - 2
  1 2
-[2 2-3j]
  -2 -2+3j
```

See also

operator +, `conj`

Operator ∗

Matrix multiplication.

Syntax

```
x ∗ y
M1 ∗ M2
M ∗ x
mtimes(x, y)
```

Description

x∗y multiplies the operands together. Operands can be scalars (plain arithmetic product), matrices (matrix product), or mixed scalar and matrix.

`mtimes(x,y)` is equivalent to x∗y. It can be used to redefine this operator for objects.

Example

```
2 * 3
  6
[1,2;3,4] * [3;5]
  13
  29
[3 4] * 2
  6 8
```

See also

operator `.*`, operator `/`, `prod`

Operator `.*`

Scalar multiplication.

Syntax

```
x .* y
M1 .* M2
M .* x
times(x, y)
```

Description

x.∗y is the element-wise multiplication. If both operands are matrices with a size different from 1-by-1, their size must be equal; the multiplication is performed element-wise. If one operand is a scalar, it multiplies each element of the other operand.

`times(x,y)` is equivalent to x.∗y. It can be used to redefine this operator for objects.

Example

```
[1 2] .* [3 5]
  3 10
[3 4] .* 2
  6 8
```

See also

operator ∗, operator ./, operator .^

Operator /

Matrix right division.

Syntax

```
a / b
A / B
A / b
mrdivide(a, b)
```

Description

a/b divides the first operand by the second operand. If the second operand is a scalar, it divides each element of the first operand. Otherwise, it must be a square matrix; M1/M2 is equivalent to M1∗inv(M2).

mrdivide(x,y) is equivalent to x/y. It can be used to redefine this operator for objects.

Example

```
9 / 3
  3
[2,6] / [1,2;3,4]
  5 -1
[4 10] / 2
  2 5
```

See also

operator \, inv, operator ./, deconv

Operator ./

Scalar right division.

Syntax

```
x ./ y
M1 ./ M2
M ./ x
x ./ M
rdivide(x, y)
```

Description

The first operand is divided by the second operand. If both operands are matrices with a size different from 1-by-1, their size must be equal; the division is performed element-wise. If one operand is a scalar, it is repeated to match the size of the other operand.

`rdivide(x,y)` is equivalent to `x./y`. It can be used to redefine this operator for objects.

Examples

```
 [3 10] ./ [3 5]
   1 2
[4 8] ./ 2
   2 4
10 ./ [5 2]
   2 5
```

See also

operator `/`, operator `.*`, operator `.^`

Operator \

Matrix left division.

Syntax

```
x \ y
M1 \ M2
x \ M
mldivide(x, y)
```

Description

`x\y` divides the second operand by the first operand. If the first operand is a scalar, it divides each element of the second operand. Otherwise, it must be a square matrix; `M1\M2` is equivalent to `inv(M1)*M2`.

`mldivide(x,y)` is equivalent to `x\y`. It can be used to redefine this operator for objects.

Examples

```
3 \ 9
  3
[1,2;3,4] \ [2;6]
  2
  0
2 \ [4 10]
  2 5
```

See also

operator `/`, `inv`, operator `.\`

Operator .\

Scalar left division.

Syntax

```
M1 .\ M2
M1 .\ x
ldivide(x, y)
```

Description

The second operand is divided by the first operand. If both operands are matrices with a size different from 1-by-1, their size must be equal; the division is performed element-wise. If one operand is a scalar, it is repeated to match the size of the other operand.

`ldivide(x,y)` is equivalent to `x.\y`. It can be used to redefine this operator for objects.

Example

```
[1 2 3] .\ [10 11 12]
  10 5.5 4
```

See also

operator \, operator `./`

Operator ^

Matrix power.

Syntax

```
x ^ y
M ^ y
x ^ M
mpower(x, y)
```

Description

x^y calculates x to the y power, provided that either

- both operands are scalar;

- the first operand is a square matrix and the second operand is a scalar;
- or the first operand is a scalar and the second operand is a square matrix.

Other cases yield an error.

`mpower(x,y)` is equivalent to `x^y`. It can be used to redefine this operator for objects.

Examples

```
2 ^ 3
  8
[1,2;3,4] ^ 2
  7  10
  15 22
2 ^ [1,2;3,4]
  10.4827 14.1519
  21.2278 31.7106
```

Algorithms

If the first operand is a scalar and the second a square matrix, the matrix exponential is used. The result is `expm(log(x)*M)`.

If the first operand is a square matrix and the second a scalar, unless for small real integers, the same algorithm as for matrix functions is used, i.e. a complex Schur decomposition followed by the Parlett method. The result is `funm(M, @(x) x^y)`.

See also

operator `.^`, `expm`, `funm`

Operator `.^`

Scalar power.

Syntax

```
M1 .^ M2
x .^ M
M .^ x
power(x, y)
```

Description

`M1.^M2` calculates `M1` to the `M2` power, element-wise. Both arguments must have the same size, unless one of them is a scalar.

`power(x,y)` is equivalent to `x.^y`. It can be used to redefine this operator for objects.

Examples

```
[1,2;3,4].^2
  1  4
  9 16
[1,2,3].^[5,4,3]
  1 16 27
```

See also

operator ^, exp

Operator ’

Complex conjugate transpose.

Syntax

```
M’
ctranspose(M)
```

Description

`M’` is the transpose of the real matrix `M`, i.e. columns and rows are permuted. If `M` is complex, the result is the complex conjugate transpose of `M`. If `M` is an array with multiple dimensions, the first two dimensions are permuted.

`ctranspose(M)` is equivalent to `M’`. It can be used to redefine this operator for objects.

Examples

```
[1,2;3,4]’
  1 3
  2 4
[1+2j, 3-4j]’
  1-2j
  3+4j
```

See also

operator `.’`, `conj`

Operator .’

Transpose.

Syntax

```
M.’
transpose(M)
```

Description

`M.'` is the transpose of the matrix `M`, i.e. columns and rows are permuted. `M` can be real or complex. If `M` is an array with multiple dimensions, the first two dimensions are permuted.

`transpose(M)` is equivalent to `M.'`. It can be used to redefine this operator for objects.

Example

```
[1,2;3,4].'
  1 3
  2 4
[1+2j, 3-4j].'
  1+2j
  3-4j
```

See also

operator ', `permute`, `fliplr`, `flipud`, `rot90`

Operator ==

Equality.

Syntax

```
x == y
eq(x, y)
```

Description

`x == y` is true if x is equal to y, and false otherwise. Comparing NaN (not a number) to any number yields false, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

`eq(x,y)` is equivalent to x==y. It can be used to redefine this operator for objects.

Example

```
1 == 1
  true
1 == 1 + eps
  false
1 == 1 + eps / 2
  true
inf == inf
  true
nan == nan
```

```
  false
[1,2,3] == [1,3,3]
  T F T
```

See also

operator ˜=, operator <, operator <=, operator >, operator >=, operator ===, operator ˜==, `strcmp`

Operator ===

Object equality.

Syntax

```
a === b
same(a, b)
```

Description

`a === b` is true if `a` is the same as `b`, and false otherwise. `a` and `b` must have exactly the same internal representation to be considered as equal; with IEEE floating-point numbers, `nan===nan` is true and `0===-0` is false. Contrary to the equality operator ==, === returns a single boolean even if its operands are arrays.

`same(a,b)` is equivalent to `a===b`.

Example

```
(1:5) === (1:5)
  true
(1:5) == (1:5)
  T T T T T
[1,2,3] === [4,5]
  false
[1,2,3] == [4,5]
  Incompatible size
nan === nan
  true
nan == nan
  false
```

See also

operator ˜==, operator ==, operator ˜=, operator <, operator <=, operator >, operator >=, operator ==, operator ˜=, `strcmp`

Operator ˜=

Inequality.

Syntax

```
x ~= y
ne(x, y)
```

Description

x ~= y is true if x is not equal to y, and false otherwise. Comparing NaN (not a number) to any number yields true, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

ne(x,y) is equivalent to x~=y. It can be used to redefine this operator for objects.

Example

```
1 ~= 1
  false
inf ~= inf
  false
nan ~= nan
  true
[1,2,3] ~= [1,3,3]
  F T F
```

See also

operator ==, operator <, operator <=, operator >, operator >=, operator ===, operator ~==, strcmp

Operator ~==

Object inequality.

Syntax

```
a ~== b
unsame(a, b)
```

Description

a ~== b is true if a is not the same as b, and false otherwise. a and b must have exactly the same internal representation to be considered as equal; with IEEE numbers, nan~==nan is false and 0~==-0 is true. Contrary to the inequality operator, ~== returns a single boolean even if its operands are arrays.

unsame(a,b) is equivalent to a~==b.

Example

```
(1:5) ~== (1:5)
  false
(1:5) ~= (1:5)
  F F F F F
[1,2,3] ~== [4,5]
  true
[1,2,3] ~= [4,5]
  Incompatible size
nan ~== nan
  false
nan ~= nan
  true
```

See also

operator ===, operator ==, operator ~=, operator <, operator <=, operator >, operator >=, `strcmp`

Operator <

Less than.

Syntax

```
x < y
lt(x, y)
```

Description

x < y is true if x is less than y, and false otherwise. Comparing NaN (not a number) to any number yields false, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

`lt(x,y)` is equivalent to x<y. It can be used to redefine this operator for objects.

Example

```
[2,3,4] < [2,4,2]
  F T F
```

See also

operator ==, operator ~=, operator <=, operator >, operator >=

Operator >

Greater than.

Syntax

```
x > y
gt(x, y)
```

Description

`x > y` is true if x is greater than y, and false otherwise. Comparing NaN (not a number) to any number yields false, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

`gt(x,y)` is equivalent to x>y. It can be used to redefine this operator for objects.

Example

```
[2,3,4] > [2,4,2]
  F F T
```

See also

operator ==, operator ~=, operator <, operator <=, operator >=

Operator <=

Less or equal to.

Syntax

```
x <= y
le(x, y)
```

Description

`x <= y` is true if x is less than or equal to y, and false otherwise. Comparing NaN (not a number) to any number yields false, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

`le(x,y)` is equivalent to x<=y. It can be used to redefine this operator for objects.

Example

```
[2,3,4] <= [2,4,2]
  T T F
```

See also

operator ==, operator ~=, operator <, operator >, operator >=

Operator >=

Greater or equal to.

Syntax

```
x >= y
ge(x, y)
```

Description

x >= y is true if x is greater than or equal to y, and false otherwise. Comparing NaN (not a number) to any number yields false, including to NaN. If x and/or y is an array, the comparison is performed element-wise and the result has the same size.

ge(x,y) is equivalent to x>=y. It can be used to redefine this operator for objects.

Example

```
[2,3,4] >= [2,4,2]
  T F T
```

See also

operator ==, operator ~=, operator <, operator <=, operator >

Operator ~

Not.

Syntax

```
~b
not(b)
```

Description

~b is false (logical 0) if b is different from 0 or false, and true otherwise. If b is an array, the operation is performed on each element.

not(b) is equivalent to ~b. It can be used to redefine this operator for objects.

Character ~ can also be used as a placeholder for unused arguments.

Examples

```
~true
  false
~[1,0,3,false]
  F T F T
```

See also

operator ~=, `bitcmp`, `function` (unused arguments)

Operator &

And.

Syntax

```
b1 & b2
and(b1, b2)
```

Description

b1&b2 performs the logical AND operation between the corresponding elements of b1 and b2; the result is true (logical 1) if both operands are different from false or 0, and false (logical 0) otherwise.

`and(b1,b2)` is equivalent to b1&b2. It can be used to redefine this operator for objects.

Example

```
[false, false, true, true] & [false, true, false, true]
  F F F T
```

See also

operator |, `xor`, operator ~, operator &&, `all`

Operator &&

And with lazy evaluation.

Syntax

```
b1 && b2
```

Description

b1&&b2 is b1 if b1 is false, and b2 otherwise. Like with `if` and `while` statements, b1 is true if it is a nonempty array with only non-zero elements. b2 is evaluated only if b1 is true.

b1&&b2&&...&&bn returns the last operand which is false (remaining operands are not evaluated), or the last one.

Example

Boolean value which is true if the vector v is made of pairs of equal values:

```
mod(length(v),2) == 0 && v(1:2:end) == v(2:2:end)
```

The second operand of && is evaluated only if the length is even.

See also

operator ||, operator ?, operator &, `if`

Operator |

Or.

Syntax

```
b1 | b2
or(b1, b2)
```

Description

b1|b2 performs the logical OR operation between the corresponding elements of b1 and b2; the result is false (logical 0) if both operands are false or 0, and true (logical 1) otherwise.

`or(b1,b2)` is equivalent to b1|b2. It can be used to redefine this operator for objects.

Example

```
[false, false, true, true] | [false, true, false, true]
  F T T T
```

See also

operator &, `xor`, operator ˜, operator ||, any

Operator ||

Or with lazy evaluation.

Syntax

```
b1 || b2
```

Description

b1||b2 is b1 if b1 is true, and b2 otherwise. Like with `if` and `while` statements, b1 is true if it is a nonempty array with only non-zero elements. b2 is evaluated only if b1 is false.

b1||b2||...||bn returns the last operand which is true (remaining operands are not evaluated), or the last one.

Example

Boolean value which is true if the vector v is empty or if its first element is NaN:

```
isempty(v) || isnan(v(1))
```

See also

operator &&, operator ?, operator |, `if`

Operator ?

Alternative with lazy evaluation.

Syntax

```
b ? x : y
```

Description

b?x:y is x if b is true, and y otherwise. Like with `if` and `while` statements, b is true if it is a nonempty array with only non-zero elements. Only one of x and y is evaluated depending on b.

Operators ? and : have the same priority; parenthesis or brackets should be used if e.g. x or y is a range.

Example

Element of a vector v, or default value 5 if the index `ind` is out of range:

```
ind < 1 || ind > length(v) ? 5 : v(ind)
```

See also

operator &&, operator ||, `if`

Operator ,

Horizontal matrix concatenation.

Syntax

```
[M1, M2, ...]
[M1 M2 ...]
horzcat(M1, M2, ...)
```

Description

Between brackets, the comma is used to separate elements on the same row in a matrix. Elements can be scalars, vector, arrays, cell arrays, or structures; their number of rows must be the same, unless one of them is an empty array. For arrays with more than 2 dimensions, all dimensions except dimension 2 (number of columns) must match.

Outside brackets or between parenthesis, the comma is used to separate statements or the arguments of functions.

`horzcat(M1,M2,...)` is equivalent to `[M1,M2,...]`. It can be used to redefine this operator for objects. It accepts any number of input arguments; `horzcat()` is the real double empty array `[]`, and `horzcat(M)` is `M`.

Between braces, the comma separates cells on the same row.

Examples

```
[1,2]
  1 2
[[3;5],ones(2)]
  3 1 1
  5 1 1
['abc','def']
  abcdef
```

See also

operator `[]`, operator `;`, `cat`, `join`, operator `{}`

Operator ;

Vertical matrix concatenation.

Syntax

```
[M1; M2]
vertcat(M1, M2)
```

Description

Between brackets, the semicolon is used to separate rows in a matrix. Rows can be scalars, vector, arrays, cell arrays, or structures; their number of columns must be the same, unless one of them is an empty array. For arrays with more than 2 dimensions, all dimensions except dimension 1 (number of rows) must match.

Outside brackets, the comma is used to separate statements; they loose any meaning between parenthesis and give a syntax error.

`vertcat(M1,M2)` is equivalent to `[M1;M2]`. It can be used to redefine this operator for objects.

Between braces, the semicolon separates rows of cells.

Examples

```
[1;2]
  1
  2
[1:5;3,2,4,5,1]
  1 2 3 4 5
  3 2 4 5 1
['abc';'def']
  abc
  def
```

See also

operator `[]`, operator `,`, `join`, operator `{}`

Operator :

Range.

Syntax

```
x1:x2
x1:step:x2
colon(x1,x2)
colon(x1,step,x2)
```

Description

x1:x2 gives a row vector with the elements x1, x1+1, x1+2, etc. until x2. The last element is equal to x2 only if x2-x1 is an integer, and smaller otherwise. If x2<x1, the result is an empty matrix.

x1:step:x2 gives a row vector with the elements x1, x1+step, x1+2*step, etc. until x2. The last element is equal to x2 only if (x2-x1)/step is an integer. With fractional numbers, rounding errors may cause x2 to be discarded even if (x2-x1)/step is "almost"

an integer. If `x2*sign(step)<x1*sign(step)`, the result is an empty matrix.

If `x1` or `step` is complex, a complex vector is produced, with the expected contents. The following algorithm is used to generate each element:

```
z = x1
while real((x2 - z) * conj(step)) >= 0
    append z to the result
    z = z + step
end
```

Values are added until they go beyond the projection of `x2` onto the straight line defined by `x1` and direction `step`. If `x2-x1` and `step` are orthogonal, it is attempted to produce an infinite number of elements, which will obviously trigger an out of memory error. This is similar to having a null step in the real case.

Note that the default step value is always 1 for consistency with real values. Choosing for instance `sign(x2-x1)` would have made the generation of lists of indices more difficult. Hence for a vector of purely imaginary numbers, always specify a step.

`colon(x1,x2)` is equivalent to `x1:x2`, and `colon(x1,step,x2)` to `x1:step:x2`. It can be used to redefine this operator for objects.

The colon character is also used to separate the alternatives of a conditional expression `b?x:y`.

Example

```
2:5
  2 3 4 5
2:5.3
  2 3 4 5
3:3
  3
3:2
  []
2:2:8
  2 4 6 8
5:-1:2
  5 4 3 2
0:1j:10j
  0 1j 2j 3j 4j 5j 6j 7j 8j 9j 10j
1:1+1j:5+4j
  1 2+1j 3+2j 4+3j 5+4j
0:1+1j:5
  0 1+1j 2+2j 3+3j 4+4j 5+5j
```

See also

`repmat`, operator ?

Operator `@`

Function reference or anonymous function.

Syntax

```
@fun
@(arguments) expression
```

Description

`@fun` gives a reference to function `fun` which can be used wherever an inline function can. Its main use is as the argument of functions like `feval` or `integral`, but it may also be stored in lists, cell arrays, or structures. A reference cannot be cast to a number (unlike characters or logical values), nor can it be stored in a numeric array. The function reference of an operator must use its function name, such as `@plus`.

Anonymous functions are an alternative, more compact syntax for inline functions. In `@(args) expr`, `args` is a list of input arguments and `expr` is an expression which contains two kinds of variables:

- input arguments, provided when the anonymous expression is executed;
- captured variables (all variables which do not appear in the list of input arguments), which have the value of variables of the same name existing when and where the anonymous function is created. These values are fixed.

If the top-level element of the anonymous function is itself a function, multiple output arguments can be specified for the call of the anonymous function, as if a direct call was performed. Anonymous functions which do not return any output are also valid.

Anonymous functions may not have input arguments with default values (`@(x=2)x+5` is invalid).

Anonymous functions are a convenient way to provide the glue between functions like `fzero` and `ode45` and the function they accept as argument. Additional parameters can be passed directly in the anonymous function with captured variables, instead of being supplied as additional arguments; the code becomes clearer.

Examples

Function reference:

```
integral(@sin, 0, pi)
  2
```

Anonymous function:

```
a = 2;
fun = @(x) sin(a * x);
fun(3)
  -0.2794
integral(fun, 0, 2)
  0.8268
```

Without anonymous function, parameter a should be passed as an additional argument after all the input arguments defined for `integral`, including those which are optional when parameters are missing:

```
integral(inline('sin(a * x)', 'x', 'a'), 0, 2, [], false, a)
  0.8268
```

Anonymous functions are actually stored as inline functions with all captured variables:

```
dumpvar(fun)
  inline('function y=f(a,x);y=sin(a*x);',2)
```

Anonymous function with multiple output arguments:

```
fun = @(A) size(A);
s = fun(ones(2,3))
  s =
    2 3
(m, n) = fun(ones(2,3))
  m =
    2
  n =
    3
```

See also

`fun2str`, `str2fun`, `inline`, `feval`, `apply`

10.20 Mathematical Functions

abs

Absolute value.

Syntax

```
x = abs(z)
```

Description

abs takes the absolute value of each element of its argument. The result is an array of the same size as the argument; each element is non-negative.

Example

```
abs([2,-3,0,3+4j]
  2 3 0 5
```

See also

`angle`, `sign`, `real`, `imag`, `hypot`

acos

Arc cosine.

Syntax

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

Description

`acos(x)` gives the arc cosine of x, which is complex if x is complex or if `abs(x)>1`.

Examples

```
acos(2)
  0+1.3170j
acos([0,1+2j])
  1.5708 1.1437-1.5286j
```

See also

`cos`, `asin`, `acosh`

acosd acotd acscd asecd asind atand atan2d

Inverse trigonometric functions with angles in degrees.

Syntax

```
y = acosd(x)
y = acotd(x)
y = acscd(x)
y = asecd(x)
y = asind(x)
y = atand(x)
z = atan2d(y, x)
```

Description

Inverse trigonometric functions whose name ends with a d give a result expressed in degrees instead of radians.

Examples

```
acosd(0.5)
  60.0000
acos(0.5) * 180 / pi
  60.0000
```

See also

`cosd`, `cotd`, `cscd`, `secd`, `sind`, `tand`, `acos`, `acot`, `acsc`, `asec`, `asin`, `atan`, `atan2`

acosh

Inverse hyperbolic cosine.

Syntax

```
y = acosh(x)
```

Description

`acosh(x)` gives the inverse hyperbolic cosine of x, which is complex if x is complex or if x<1.

Examples

```
acosh(2)
  1.3170
acosh([0,1+2j])
  0+1.5708j 1.5286+1.1437j
```

See also

`cosh`, `asinh`, `acos`

acot

Inverse cotangent.

Syntax

```
y = acot(x)
```

Description

acot(x) gives the inverse cotangent of x, which is complex if x is.

See also

`cot`, `acoth`, `cos`

acoth

Inverse hyperbolic cotangent.

Syntax

```
y = acoth(x)
```

Description

acoth(x) gives the inverse hyperbolic cotangent of x, which is complex if x is complex or is in the range (-1,1).

See also

`coth`, `acot`, `atanh`

acsc

Inverse cosecant.

Syntax

```
y = acsc(x)
```

Description

acsc(x) gives the inverse cosecant of x, which is complex if x is complex or is in the range (-1,1).

See also

`csc`, `acsch`, `asin`

acsch

Inverse hyperbolic cosecant.

Syntax

```
y = acsch(x)
```

Description

acsch(x) gives the inverse hyperbolic cosecant of x, which is complex if x is.

See also

`csc`, `acsc`, `asinh`

angle

Phase angle of a complex number.

Syntax

```
phi = angle(z)
```

Description

`angle(z)` gives the phase of the complex number `z`, i.e. the angle between the positive real axis and a line joining the origin to `z`. `angle(0)` is 0.

Examples

```
angle(1+3j)
  1.2490
angle([0,1,-1])
  0 0 3.1416
```

See also

`abs`, `sign`, `atan2`

asec

Inverse secant.

Syntax

```
y = asec(x)
```

Description

asec(x) gives the inverse secant of x, which is complex if x is complex or is in the range (-1,1).

See also

`sec`, `asech`, `acos`

asech

Inverse hyperbolic secant.

Syntax

```
y = asech(x)
```

Description

asech(x) gives the inverse hyperbolic secant of x, which is complex if x is complex or strictly negative.

See also

sech, asec, acosh

asin

Arc sine.

Syntax

```
y = asin(x)
```

Description

asin(x) gives the arc sine of x, which is complex if x is complex or if abs(x)>1.

Examples

```
asin(0.5)
  0.5236
asin(2)
  1.5708-1.317j
```

See also

sin, acos, asinh

asinh

Inverse hyperbolic sine.

Syntax

```
y = asinh(x)
```

Description

asinh(x) gives the inverse hyperbolic sine of x, which is complex if x is complex.

Examples

```
asinh(2)
  1.4436
asinh([0,1+2j])
  0 1.8055+1.7359j
```

See also
`sinh`, `acosh`, `asin`

atan

Arc tangent.

Syntax
```
y = atan(x)
```

Description
`atan(x)` gives the arc tangent of x, which is complex if x is complex.

Example
```
atan(1)
  0.7854
```

See also
`tan`, `asin`, `acos`, `atan2`, `atanh`

atan2

Direction of a point given by its Cartesian coordinates.

Syntax
```
phi = atan2(y,x)
```

Description
`atan2(y,x)` gives the direction of a point given by its Cartesian coordinates x and y. Imaginary component of complex numbers is ignored. `atan2(y,x)` is equivalent to `atan(y/x)` if `x>0`.

Examples
```
atan2(1, 1)
  0.7854
atan2(-1, -1)
  -2.3562
atan2(0, 0)
  0
```

See also
`atan`, `angle`

atanh

Inverse hyperbolic tangent.

Syntax

```
y = atanh(x)
```

Description

`atan(x)` gives the inverse hyperbolic tangent of x, which is complex if x is complex or if `abs(x)>1`.

Examples

```
atanh(0.5)
  0.5493
atanh(2)
  0.5493 + 1.5708j
```

See also

`asinh`, `acosh`, `atan`

beta

Beta function.

Syntax

```
y = beta(z,w)
```

Description

`beta(z,w)` gives the beta function of z and w. Arguments and result are real (imaginary part is discarded). The beta function is defined as

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}\,\mathrm{d}t$$

Example

```
beta(1,2)
  0.5
```

See also

`gamma`, `betaln`, `betainc`

betainc

Incomplete beta function.

Syntax

```
y = betainc(x,z,w)
```

Description

`betainc(x,z,w)` gives the incomplete beta function of `x`, `z` and `w`. Arguments and result are real (imaginary part is discarded). `x` must be between 0 and 1. The incomplete beta function is defined as

$$I_x(z,w) = \frac{1}{B(z,w)} \int_0^x t^{z-1}(1-t)^{w-1}\,\mathrm{d}t$$

Example

```
betainc(0.2,1,2)
  0.36
```

See also

`beta`, `betaln`, `gammainc`

betaln

Logarithm of beta function.

Syntax

```
y = betaln(z,w)
```

Description

`betaln(z,w)` gives the logarithm of the beta function of `z` and `w`. Arguments and result are real (imaginary part is discarded).

Example

```
betaln(0.5,2)
  0.2877
```

See also

`beta`, `betainc`, `gammaln`

cart2pol

Cartesian to polar coordinates transform.

Syntax

```
(phi, r) = cart2pol(x, y)
(phi, r, z) = cart2pol(x, y, z)
```

Description

`(phi,r)=cart2pol(x,y)` transforms Cartesian coordinates `x` and `y` to polar coordinates `phi` and `r` such that $x = r\cos(\varphi)$ and $x = r\sin(\varphi)$.

`(phi,r,z)=cart2pol(x,y,z)` transform Cartesian coordinates to cylindrical coordinates, leaving `z` unchanged.

Example

```
(phi, r) = cart2pol(1, 2)
  phi =
    1.1071
  r =
    2.2361
```

See also

`cart2sph`, `pol2cart`, `sph2cart`, `abs`, `angle`

cart2sph

Cartesian to spherical coordinates transform.

Syntax

```
(phi, theta, r) = cart2sph(x, y, z)
```

Description

`(phi,theta,r)=cart2sph(x,y,z)` transforms Cartesian coordinates `x`, `y`, and `z` to polar coordinates `phi`, `theta`, and `r` such that $x = r\cos(\varphi)\cos(\vartheta)$, $y = r\sin(\varphi)\cos(\vartheta)$, and $z = r\sin(\vartheta)$.

Example

```
(phi, theta, r) = cart2sph(1, 2, 3)
  phi =
    1.1071
  theta =
    0.9303
  r =
    3.7417
```

See also

`cart2pol`, `pol2cart`, `sph2cart`

cast

Type conversion.

Syntax

```
Y = cast(X, type)
```

Description

`cast(X,type)` converts the numeric array X to the type given by string `type`, which can be `'double'`, `'single'`, `'int8'` or any other signed or unsigned integer type, `'char'`, or `'logical'`. The number value is preserved when possible; conversion to integer types discards most significant bytes. If X is an array, conversion is performed on each element; the result has the same size. The imaginary part, if any, is discarded only with conversions to integer types.

Example

```
cast(pi, 'int8')
  3int8
```

See also

`uint8` and related functions, `double`, `single`, `typecast`

cdf

Cumulative distribution function.

Syntax

```
y = cdf(distribution,x)
y = cdf(distribution,x,a1)
y = cdf(distribution,x,a1,a2)
```

Description

`cdf(distribution,x)` calculates the integral of a probability density function from $-\infty$ to x. The distribution is specified with the first argument, a string; case is ignored ('t' and 'T' are equivalent). Additional arguments must be provided for some distributions. The distributions are given in the table below. Default values for the parameters, when mentioned, mean that the parameter may be omitted.

Distribution	Name	Parameters
beta	`beta`	a and b
Cauchy	`cauchy`	a and b
χ	`chi`	deg. of freedom ν
χ^2	`chi2`	deg. of freedom ν
γ	`gamma`	shape α and λ
exponential	`exp`	mean
F	`f`	deg. of freedom ν_1 and ν_2
half-normal	`half-normal`	ϑ
Laplace	`laplace`	mean and scale
lognormal	`logn`	mean (0) and st. dev. (1)
Nakagami	`nakagami`	μ and ω
normal	`norm`	mean (0) and st. dev. (1)
Rayleigh	`rayl`	b
Student's T	`t`	deg. of freedom ν
uniform	`unif`	limits of the range (0 and 1)
Weibull	`weib`	shape k and scale λ

Example

```
cdf('chi2', 2.5, 3)
  0.5247
integral(@(x) pdf('chi2',x,3), 0, 2.5, AbsTol=1e-4)
  0.5247
```

See also

`pdf`, `icdf`, `random`, `erf`

ceil

Rounding towards +infinity.

Syntax

```
y = ceil(x)
```

Description

`ceil(x)` gives the smallest integer larger than or equal to `x`. If the argument is a complex number, the real and imaginary parts are handled separately.

Examples

```
ceil(2.3)
  3
ceil(-2.3)
```

```
   -2
ceil(2.3-4.5j)
   3-4j
```

See also

`floor`, `fix`, `round`, `roundn`

complex

Make a complex number.

Syntax

```
z = complex(x, y)
```

Description

`complex(x,y)` makes a complex number from its real part x and imaginary part y. The imaginary part of its input arguments is ignored.

Examples

```
complex(2, 3)
   2 + 3j
complex(1:5, 2)
   1+2j 2+2j 3+2j 4+2j 5+2j
```

See also

`real`, `imag`, `i`

conj

Complex conjugate value.

Syntax

```
w = conj(z)
```

Description

`conj(z)` changes the sign of the imaginary part of the complex number z.

Example

```
conj([1+2j,-3-5j,4,0])
   1-2j -3+5j 4 0
```

See also

`imag`, `angle`, `j`, operator `-`

cos

Cosine.

Syntax

```
y = cos(x)
```

Description

`cos(x)` gives the cosine of x, which is complex if x is complex.

Example

```
cos([0, 1+2j])
  1 2.0327-3.0519j
```

See also

`sin`, `acos`, `cosh`

cosd cotd cscd secd sind tand

Trigonometric functions with angles in degrees.

Syntax

```
y = cosd(x)
y = cotd(x)
y = cscd(x)
y = secd(x)
y = sind(x)
y = tand(x)
```

Description

Trigonometric functions whose name ends with a d have an argument expressed in degrees instead of radians.

Examples

```
cosd(20)
  0.9397
cos(20 * pi / 180)
  0.9397
```

See also

`acosd`, `acotd`, `acscd`, `asecd`, `asind`, `atand`, `atan2d`, `cos`, `cot`, `csc`, `sec`, `sin`, `tan`

cosh

Hyperbolic cosine.

Syntax

```
y = cosh(x)
```

Description

`cos(x)` gives the hyperbolic cosine of x, which is complex if x is complex.

Example

```
cosh([0, 1+2j])
  1 -0.6421+1.0686j
```

See also

`sinh`, `acosh`, `cos`

cot

Cotangent.

Syntax

```
y = cot(x)
```

Description

cot(x) gives the cotangent of x, which is complex if x is.

See also

`acot`, `coth`, `tan`

coth

Hyperbolic cotangent.

Syntax

```
y = coth(x)
```

Description
coth(x) gives the hyperbolic cotangent of x, which is complex if x is.

See also
`acoth`, `cot`, `tanh`

csc

Cosecant.

Syntax
```
y = csc(x)
```

Description
csc(x) gives the cosecant of x, which is complex if x is.

See also
`acsc`, `csch`, `sin`

csch

Hyperbolic cosecant.

Syntax
```
y = csch(x)
```

Description
csch(x) gives the hyperbolic cosecant of x, which is complex if x is.

See also
`acsch`, `csc`, `sinh`

diln

Dilogarithm.

Syntax
```
y = diln(x)
```

Description

`diln(x)` gives the dilogarithm, or Spence's integral, of `x`. Argument and result are real (imaginary part is discarded). The dilogarithm is defined as

$$\mathrm{diln}(x) = \int_1^x \frac{\log(t)}{t-1}\,\mathrm{d}t$$

Example

```
diln([0.2, 0.7, 10])
  -1.0748  -0.3261   3.9507
```

double

Conversion to double-precision numbers.

Syntax

```
B = double(A)
```

Description

`double(A)` converts number or array `A` to double precision. `A` can be any kind of numeric value (real, complex, or integer), or a character or logical array.

To keep the integer type of logical and character arrays, the unitary operator + should be used instead.

Examples

```
double(uint8(3))
  3
double('AB')
  65 66
islogical(double(1>2))
  false
```

See also

`uint8` and related functions, `single`, `cast`, operator +, `setstr`, `char`, `logical`

ellipam

Jacobi elliptic amplitude.

Syntax

```
phi = ellipam(u, m)
phi = ellipam(u, m, tol)
```

Description

`ellipam(u,m)` gives the Jacobi elliptic amplitude `phi`. Parameter `m` must be in [0,1]. The Jacobi elliptic amplitude is the inverse of the Jacobi integral of the first kind, such that $u = F(\varphi|m)$.

`ellipam(u,m,tol)` uses tolerance `tol`; the default tolerance is `eps`.

Example

```
phi = ellipam(2.7, 0.6)
  phi =
    2.0713
u = ellipf(phi, 0.6)
  u =
    2.7
```

See also

`ellipf`, `ellipj`

ellipe

Jacobi elliptic integral of the second kind.

Syntax

```
u = ellipe(phi, m)
```

Description

`ellipe(phi,m)` gives the Jacobi elliptic integral of the second kind, defined as

$$E(\varphi|m) = \int_0^{\varphi} \sqrt{1 - m\sin^2 t}\, dt$$

Complete elliptic integrals of first and second kinds, with `phi=pi/2`, can be obtained with `ellipke`.

See also

`ellipf`, `ellipke`

ellipf

Jacobi elliptic integral of the first kind.

Syntax

```
u = ellipf(phi, m)
```

Description

`ellipf(phi,m)` gives the Jacobi elliptic integral of the first kind, defined as

$$F(\varphi|m) = \int_0^{\varphi} \frac{\mathrm{d}t}{\sqrt{1-m\sin^2 t}}$$

Complete elliptic integrals of first and second kinds, with `phi=pi/2`, can be obtained with `ellipke`.

See also

`ellipe`, `ellipke`, `ellipam`

ellipj

Jacobi elliptic functions.

Syntax

```
(sn, cn, dn) = ellipj(u, m)
(sn, cn, dn) = ellipj(u, m, tol)
```

Description

`ellipj(u,m)` gives the Jacobi elliptic function sn, cn, and dn. Parameter `m` must be in [0,1]. These functions are based on the Jacobi elliptic amplitude φ, the inverse of the Jacobi elliptic integral of the first kind which can be obtained with `ellipam`):

$$u = F(\varphi|m)$$

$$\mathrm{sn}(u|m) = \sin(\varphi)$$

$$\mathrm{cn}(u|m) = \cos(\varphi)$$

$$\mathrm{dn}(u|m) = \sqrt{1-m\,\sin^2\varphi}$$

`ellipj(u,m,tol)` uses tolerance `tol`; the default tolerance is `eps`.

Examples

```
(sn, cn, dn) = ellipj(2.7, 0.6)
  sn =
    0.8773
  cn =
    -0.4799
  dn =
    0.7336
sin(ellipam(2.7, 0.6))
  0.8773
ellipj(0:5, 0.3)
  0.0000    0.8188    0.9713    0.4114   -0.5341   -0.9930
```

See also

`ellipam`, `ellipke`

ellipke

Complete elliptic integral.

Syntax

```
(K, E) = ellipke(m)
(K, E) = ellipke(m, tol)
```

Description

`(K,E)=ellipke(m)` gives the complete elliptic integrals of the first kind K=F(`m`) and second kind E=E(`m`), defined as

$$F(m) = \int_0^{\pi/2} \frac{\mathrm{d}t}{\sqrt{1 - m\sin^2 t}}$$

$$E(m) = \int_0^{\pi/2} \sqrt{1 - m\sin^2 t}\,\mathrm{d}t$$

Parameter `m` must be in [0,1].
`ellipke(m,tol)` uses tolerance `tol`; the default tolerance is eps.

Example

```
(K, E) = ellipke(0.3)
  K =
    1.7139
  E =
    1.4454
```

See also

`ellipj`

eps

Difference between 1 and the smallest number x such that x > 1.

Syntax

```
e = eps
e = eps(x)
e = eps(type)
```

Description

Because of the floating-point encoding of "real" numbers, the absolute precision depends on the magnitude of the numbers. The relative precision is characterized by the number given by eps, which is the smallest double positive number such that 1+eps can be distinguished from 1.

`eps(x)` gives the smallest number e such that x+e has the same sign as x and can be distinguished from x. It takes into account whether x is a double or a single number. If x is an array, the result has the same size; each element corresponds to an element of the input.

`eps('single')` gives the smallest single positive number e such that `1single`+e can be distinguished from `1single`. `eps('double')` gives the same value as eps without input argument.

Examples

```
eps
  2.2204e-16
1 + eps - 1
  2.2204e-16
eps / 2
  1.1102e-16
1 + eps / 2 - 1
  0
```

See also

`inf`, `realmin`, `pi`, `i`, `j`

erf

Error function.

Syntax

```
y = erf(x)
```

Description

`erf(x)` gives the error function of x. Argument and result are real (imaginary part is discarded). The error function is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, \mathrm{d}t$$

Example

```
erf(1)
  0.8427
```

See also

`erfc`, `erfinv`

erfc

Complementary error function.

Syntax

```
y = erfc(x)
```

Description

`erfc(x)` gives the complementary error function of x. Argument and result are real (imaginary part is discarded). The complementary error function is defined as

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, \mathrm{d}t$$

Example

```
erfc(1)
  0.1573
```

See also

`erf`, `erfcx`, `erfcinv`

erfcinv

Inverse complementary error function.

Syntax

```
x = erfcinv(y)
```

Description

`erfcinv(y)` gives the value `x` such that `y=erfc(x)`. Argument and result are real (imaginary part is discarded). `y` must be in the range [0,2]; values outside this range give nan.

Example

```
y = erfc(0.8)
  y =
    0.2579
erfcinv(y)
  0.8
```

See also

`erfc`, `erfinv`

erfcx

Scaled complementary error function.

Syntax

```
y = erfcx(x)
```

Description

`erfcx(x)` gives the scaled complementary error function of `x`, defined as `exp(x^2)*erfc(x)`. Argument and result are real (imaginary part is discarded).

Example

```
erfcx(1)
  0.4276
```

See also

`erfc`

erfinv

Inverse error function.

Syntax

```
x = erfinv(y)
```

Description

`erfinv(y)` gives the value x such that y=`erf(x)`. Argument and result are real (imaginary part is discarded). y must be in the range [-1,1]; values outside this range give nan.

Example

```
y = erf(0.8)
  y =
    0.7421
erfinv(y)
  0.8
```

See also

`erf`, `erfcinv`

exp

Exponential.

Syntax

```
y = exp(x)
```

Description

`exp(x)` is the exponential of x, i.e. `2.7182818284590446...`^x.

Example

```
exp([0,1,0.5j*pi])
  1 2.7183 1j
```

See also

`log`, `expm1`, `expm`, operator `.^`

expm1

Exponential minus one.

Syntax

```
y = expm1(x)
```

Description

`expm1(x)` is `exp(x)-1` with improved precision for small x.

Example

```
expm1(1e-15)
  1e-15
exp(1e-15)-1
  1.1102e-15
```

See also

`exp`, `log1p`

factor

Prime factors.

Syntax

```
v = factor(n)
```

Description

`factor(n)` gives a row vector which contains the prime factors of n in ascending order. Multiple prime factors are repeated.

Example

```
factor(350)
  2  5  5  7
```

See also

`isprime`

factorial

Factorial.

Syntax

```
y = factorial(n)
```

Description

`factorial(n)` gives the factorial `n!` of nonnegative integer `n`. If the input argument is negative or noninteger, the result is NaN. The imaginary part is ignored.

Examples

```
factorial(5)
  120
factorial([-1,0,1,2,3,3.14])
  nan   1   1   2   6 nan
```

See also

gamma, nchoosek

fix

Rounding towards 0.

Syntax

```
y = fix(x)
```

Description

`fix(x)` truncates the fractional part of x. If the argument is a complex number, the real and imaginary parts are handled separately.

Examples

```
fix(2.3)
  2
fix(-2.6)
  -2
```

See also

floor, ceil, round

flintmax

Largest of the set of consecutive integers stored as floating point.

Syntax

```
x = flintmax
x = flintmax(type)
```

Description

`flintmax` gives the largest positive integer number in double precision such that all smaller integers can be represented in double precision.

`flintmax(type)` gives the largest positive integer number in double precision if type is 'double', or in single precision if type is 'single'. `flintmax` is 2^53 and `flintmax('single')` is 2^24.

Examples

```
flintmax
  9007199254740992
flintmax - 1
  9007199254740991
flintmax + 1
  9007199254740992
flintmax + 2
  9007199254740994
```

See also

`realmax`, `intmax`

floor

Rounding towards -infinity.

Syntax

```
y = floor(x)
```

Description

`floor(x)` gives the largest integer smaller than or equal to `x`. If the argument is a complex number, the real and imaginary parts are handled separately.

Examples

```
floor(2.3)
  2
floor(-2.3)
  -3
```

See also

`ceil`, `fix`, `round`, `roundn`

gamma

Gamma function.

Syntax

```
y = gamma(x)
```

Description

`gamma(x)` gives the gamma function of x. Argument and result are real (imaginary part is discarded). The gamma function is defined as

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, \mathrm{d}t$$

For positive integer values, $\Gamma(n) = (n-1)!$.

Examples

```
gamma(5)
  24
gamma(-3)
  inf
gamma(-3.5)
  0.2701
```

See also

`beta`, `gammaln`, `gammainc`, `factorial`

gammainc

Incomplete gamma function.

Syntax

```
y = gammainc(x,a)
```

Description

`gammainc(x,a)` gives the incomplete gamma function of x and a. Arguments and result are real (imaginary part is discarded). x must be nonnegative. The incomplete gamma function is defined as

$$\mathrm{gammainc}(x, a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t}\, \mathrm{d}t$$

Example

```
gammainc(2,1.5)
  0.7385
```

See also

`gamma`, `gammaln`, `betainc`

gammaln

Logarithm of gamma function.

Syntax

```
y = gammaln(x)
```

Description

`gammaln(x)` gives the logarithm of the gamma function of `x`. Argument and result are real (imaginary part is discarded). `gammaln` does not rely on the computation of the gamma function to avoid overflows for large numbers.

Examples

```
gammaln(1000)
  5905.2204
gamma(1000)
  inf
```

See also

`gamma`, `gammainc`, `betaln`

gcd

Greatest common divisor.

Syntax

```
q = gcd(a, b)
```

Description

`gcd(a,b)` gives the greatest common divisor of integer numbers a and b.

Example

```
gcd(72, 56)
  8
```

See also

`lcm`

goldenratio

Golden ratio constant.

Syntax

```
x = goldenratio
```

Description

`goldenratio` is the golden ration $(\sqrt{5}+1)/2$, up to the precision of its floating-point representation.

Example

```
goldenratio
  1.6180
```

See also

`pi`, `eps`

hypot

Hypotenuse.

Syntax

```
c = hypot(a, b)
```

Description

`hypot(a,b)` gives the square root of the square of a and b, or of their absolute value if they are complex. The result is always real. `hypot` avoids overflow when the result itself does not overflow.

Examples

```
hypot(3, 4)
  5
hypot([1,2,3+4j,inf], 5)
  5.099  5.3852  5.831  inf
```

See also

`sqrt`, `abs`, `norm`

i j

Imaginary unit.

Syntax

```
i
j
1.23e4i
1.23e4j
```

Description

`i` or `j` are the imaginary unit, i.e. the pure imaginary number whose square is -1. `i` and `j` are equivalent.

Used as a suffix appended without space to a number, `i` or `j` mark an imaginary number. They must follow the fractional part and the exponent, if any; for single-precision numbers, they must precede the `single` suffix.

To obtain a complex number i, you can write either `i` or `1i` (or `j` or `1j`). The second way is safer, because variables `i` and `j` are often used as indices and would hide the meaning of the built-in functions. The expression `1i` is always interpreted as an imaginary constant number.

Imaginary numbers are displayed with `i` or `j` depending on the option set with the `format` command.

Examples

```
i
  1j
format i
2i
  2i
2single + 5jsingle
  2+5i (single)
```

See also

`imag`, `complex`

icdf

Inverse cumulative distribution function.

Syntax

```
x = icdf(distribution,p)
x = icdf(distribution,p,a1)
x = icdf(distribution,p,a1,a2)
```

Description

`icdf(distribution,p)` calculates the value of `x` such that `cdf(distribution,x)` is `p`. The distribution is specified with the first argument, a string; case is ignored ('t' and 'T' are equivalent). Additional arguments must be provided for some distributions. The distributions are given in the table below. Default values for the parameters, when mentioned, mean that the parameter may be omitted.

Distribution	Name	Parameters
beta	`beta`	a and b
χ^2	`chi2`	deg. of freedom ν
γ	`gamma`	shape α and scale λ
F	`f`	deg. of freedom ν_1 and ν_2
lognormal	`logn`	mean (0) and st. dev. (1)
normal	`norm`	mean (0) and st. dev. (1)
Student's T	`t`	deg. of freedom ν
uniform	`unif`	limits of the range (0 and 1)

Example

```
x = icdf('chi2', 0.6, 3)
  x =
    2.9462
cdf('chi2', x, 3)
  0.6000
```

See also

`cdf`, `pdf`, `random`

imag

Imaginary part of a complex number.

Syntax

```
im = imag(z)
```

Description

`imag(z)` is the imaginary part of the complex number `z`, or 0 if `z` is real.

Examples

```
imag(1+2j)
  2
imag(1)
  0
```

See also

`real`, `complex`, `i`, `j`

inf

Infinity.

Syntax

```
x = inf
x = Inf
x = inf(n)
x = inf(n1,n2,...)
x = inf([n1,n2,...])
x = inf(..., type)
```

Description

`inf` is the number which represents infinity. Most mathematical functions accept infinity as input argument and yield an infinite result if appropriate. Infinity and minus infinity are two different quantities.

With integer non-negative arguments, `inf` creates arrays whose elements are infinity. Arguments are interpreted the same way as `zeros` and `ones`.

The last argument of `inf` can be a string to specify the type of the result: `'double'` for double-precision (default), or `'single'` for single-precision.

Examples

```
1/inf
  0
-inf
  -inf
```

See also

`isfinite`, `isinf`, `nan`, `zeros`, `ones`

iscolumn

Test for a column vector.

Syntax

```
b = iscolumn(x)
```

Description

`iscolumn(x)` is true if the input argument is a column vector (real or complex 2-dimension array of any floating-point or integer type, character or logical value with second dimension equal to 1, or empty array), and false otherwise.

Examples

```
iscolumn([1, 2, 3])
  false
iscolumn([1; 2])
  true
iscolumn(7)
  true
iscolumn([1, 2; 3, 4])
  false
```

See also

`isrow`, `ismatrix`, `isscalar`, `isnumeric`, `size`, `ndims`, `length`

isfinite

Test for finiteness.

Syntax

```
b = isfinite(x)
```

Description

`isfinite(x)` is true if the input argument is a finite number (neither infinite nor nan), and false otherwise. The result is performed on each element of the input argument, and the result has the same size.

Example

```
isfinite([0,1,nan,inf])
  T T F F
```

See also

`isinf`, `isnan`

isfloat

Test for a floating-point object.

Syntax

```
b = isfloat(x)
```

Description

`isfloat(x)` is true if the input argument is a floating-point type (double or single), and false otherwise.

Examples

```
isfloat(2)
  true
isfloat(2int32)
  false
```

See also

`isnumeric`, `isinteger`

isinf

Test for infinity.

Syntax

```
b = isinf(x)
```

Description

`isinf(x)` is true if the input argument is infinite (neither finite nor nan), and false otherwise. The result is performed on each element of the input argument, and the result has the same size.

Example

```
isinf([0,1,nan,inf])
  F F F T
```

See also

`isfinite`, `isnan`, `inf`

isinteger

Test for an integer object.

Syntax

```
b = isinteger(x)
```

Description

`isinteger(x)` is true if the input argument is an integer type (including char and logical), and false otherwise.

Examples

```
isinteger(2int16)
  true
isinteger(false)
  true
isinteger('abc')
  true
isinteger(3)
  false
```

See also

`isnumeric`, `isfloat`

ismatrix

Test for a matrix.

Syntax

```
b = ismatrix(x)
```

Description

`ismatrix(x)` is true if the input argument is a matrix (real or complex 2-dimension array of any floating-point or integer type, character or logical value with, or empty array), and false otherwise.

Examples

```
ismatrix([1, 2, 3])
  true
ismatrix([1; 2])
  true
ismatrix(7)
  true
ismatrix([1, 2; 3, 4])
  true
ismatrix(ones([2,2,1])
  true
ismatrix(ones([1,2,2])
  false
```

See also

`isrow`, `iscolumn`, `isscalar`, `isnumeric`, `isscalar`, `size`, `ndims`, `length`

isnan

Test for Not a Number.

Syntax

```
b = isnan(x)
```

Description

`isnan(x)` is true if the input argument is nan (not a number), and false otherwise. The result is performed on each element of the input argument, and the result has the same size.

Example

```
isnan([0,1,nan,inf])
  F F T F
```

See also

`isinf`, `nan`

isnumeric

Test for a numeric object.

Syntax

```
b = isnumeric(x)
```

Description

`isnumeric(x)` is true if the input argument is numeric (real or complex scalar, vector, or array), and false otherwise.

Examples

```
isnumeric(pi)
  true
isnumeric('abc')
  false
```

See also

`ischar`, `isfloat`, `isinteger`, `isscalar`

isprime

Prime number test.

Syntax

```
b = isprime(n)
```

Description

`isprime(n)` returns `true` if `n` is a prime number, or `false` otherwise. If `n` is a matrix, the test is applied to each element and the result is a matrix the same size.

Examples

```
isprime(7)
  true
isprime([0, 2, 10])
  F T F
```

See also

`factor`

isrow

Test for a row vector.

Syntax

```
b = isrow(x)
```

Description

`isrow(x)` is true if the input argument is a row vector (real or complex 2-dimension array of any floating-point or integer type, character or logical value with first dimension equal to 1, or empty array), and false otherwise.

Examples

```
isrow([1, 2, 3])
  true
isrow([1; 2])
  false
isrow(7)
  true
isrow([1, 2; 3, 4])
  false
```

See also

`iscolumn`, `ismatrix`, `isscalar`, `isnumeric`, `size`, `ndims`, `length`

isscalar

Test for a scalar number.

Syntax

```
b = isscalar(x)
```

Description

`isscalar(x)` is true if the input argument is scalar (real or complex number of any floating-point or integer type, character or logical value), and false otherwise.

Examples

```
isscalar(2)
  true
isscalar([1, 2, 5])
  false
```

See also

`isnumeric`, `isvector`, `ismatrix`, `size`

isvector

Test for a vector.

Syntax

```
b = isvector(x)
```

Description

`isvector(x)` is true if the input argument is a row or column vector (real or complex 2-dimension array of any floating-point or integer type, character or logical value with one dimension equal to 1, or empty array), and false otherwise.

Examples

```
isvector([1, 2, 3])
  true
isvector([1; 2])
  true
isvector(7)
  true
isvector([1, 2; 3, 4])
  false
```

See also

`isnumeric`, `isscalar`, `iscolumn`, `isrow`, `size`, `ndims`, `length`

lcm

Least common multiple.

Syntax

```
q = lcm(a, b)
```

Description

`lcm(a,b)` gives the least common multiple of integer numbers a and b.

Example

```
lcm(72, 56)
  504
```

See also

`gcd`

log

Natural (base e) logarithm.

Syntax

```
y = log(x)
```

Description

`log(x)` gives the natural logarithm of x. It is the inverse of exp. The result is complex if x is not real positive.

Example

```
log([-1,0,1,10,1+2j])
  0+3.1416j inf 0 2.3026 0.8047+1.1071j
```

See also

`log10`, `log2`, `log1p`, `reallog`, `exp`

log10

Decimal logarithm.

Syntax

```
y = log10(x)
```

Description

`log10(x)` gives the decimal logarithm of x, defined by `log10(x) = log(x)/log(10)`. The result is complex if x is not real positive.

Example

```
log10([-1,0,1,10,1+2j])
  0+1.3644j inf 0 1 0.3495+0.4808j
```

See also

`log`, `log2`

log1p

Logarithm of x plus one.

Syntax

```
y = log1p(x)
```

Description

`log1p(x)` is `log(1+x)` with improved precision for small x.

Example

```
log1p(1e-15)
  1e-15
log(1 + 1e-15)
  1.1102e-15
```

See also

`log`, `expm1`

log2

Base 2 logarithm.

Syntax

```
y = log2(x)
```

Description

`log2(x)` gives the base 2 logarithm of `x`, defined as `log2(x)=log(x)/log(2)`. The result is complex if `x` is not real positive.

Example

```
log2([1, 2, 1024, 2000, -5])
  0  1  10  10.9658  2.3219+4.5324j
```

See also

`log`, `log10`

mod

Modulo.

Syntax

```
m = mod(x, y)
```

Description

`mod(x,y)` gives the modulo of `x` divided by `y`, i.e. a number `m` between 0 and `y` such that `x = q*y+m` with integer `q`. Imaginary parts, if they exist, are ignored.

Examples

```
mod(10,7)
  3
mod(-10,7)
  4
mod(10,-7)
  -4
mod(-10,-7)
  -3
```

See also

`rem`

nan

Not a Number.

Syntax

```
x = nan
x = NaN
x = nan(n)
x = nan(n1,n2,...)
x = nan([n1,n2,...])
x = nan(..., type)
```

Description

NaN (Not a Number) is the result of the primitive floating-point functions or operators called with invalid arguments. For example, 0/0, inf-inf and 0*inf all result in NaN. When used in an expression, NaN propagates to the result. All comparisons involving NaN result in false, except for comparing NaN with any number for inequality, which results in true.

Contrary to built-in functions usually found in the underlying operating system, many functions which would result in NaNs give complex numbers when called with arguments in a certain range.

With integer non-negative arguments, `nan` creates arrays whose elements are NaN. Arguments are interpreted the same way as `zeros` and `ones`.

The last argument of `nan` can be a string to specify the type of the result: `'double'` for double-precision (default), or `'single'` for single-precision.

Examples

```
nan
  nan
0*nan
  nan
nan==nan
  false
nan~=nan
  true
log(-1)
  0+3.1416j
```

See also

`inf`, `isnan`, `zeros`, `ones`

nchoosek

Binomial coefficient.

Syntax

```
b = nchoosek(n, k)
```

Description

`nchoosek(n,k)` gives the number of combinations of n objects taken k at a time. Both n and k must be nonnegative integers with k<n.

Examples

```
nchoosek(10,4)
  210
nchoosek(10,6)
  210
```

See also

`factorial`, `gamma`

nthroot

Real nth root.

Syntax

```
y = nthroot(x,n)
```

Description

`nthroot(x,n)` gives the real nth root of real number x. If x is positive, it is `x.^(1./n)`; if x is negative, it is `-abs(x).^(1./n)` if n is an odd integer, or NaN otherwise.

Example

```
nthroot([-2,2], 3)
  -1.2599    1.2599
[-2,2] .^ (1/3)
  0.6300+1.0911i  1.2599
```

See also

operator `.^`, `realsqrt`, `sqrt`

pdf

Probability density function.

Syntax

```
y = pdf(distribution,x)
y = pdf(distribution,x,a1)
y = pdf(distribution,x,a1,a2)
```

Description

`pdf(distribution,x)` gives the probability of a density function. The distribution is specified with the first argument, a string; case is ignored ('t' and 'T' are equivalent). Additional arguments must be provided for some distributions. See `cdf` for the list of distributions.

See also

`cdf`, `random`

pi

Constant π.

Syntax

```
x = pi
```

Description

`pi` is the number π, up to the precision of its floating-point representation.

Example

```
exp(1j * pi)
  -1
```

See also

`goldenratio`, `i`, `j`, `eps`

pol2cart

Polar to Cartesian coordinates transform.

Syntax

```
(x, y) = pol2cart(phi, r)
(x, y, z) = pol2cart(phi, r, z)
```

Description

`(x,y)=pol2cart(phi,r)` transforms polar coordinates `phi` and `r` to Cartesian coordinates x and y such that $x = r\cos(\varphi)$ and $x = r\sin(\varphi)$.

`(x,y,z)=pol2cart(phi,r,z)` transforms cylindrical coordinates to Cartesian coordinates, leaving `z` unchanged.

Example

```
(x, y) = pol2cart(1, 2)
  x =
    1.0806
  y =
    1.6829
```

See also

`cart2pol`, `cart2sph`, `sph2cart`

random

Random generator for distribution function.

Syntax

```
x = random(distribution)
x = random(distribution, a1)
x = random(distribution, a1, a2)
x = random(..., size)
```

Description

`random(distribution,a1,a2)` calculates a pseudo-random number whose distribution function is specified by name `distribution` and parameters `a1` and `a2` (some distributions have a single parameter). The distributions are given in the table below. Unlike in functions `pdf`, `cdf` and `icdf`, parameters do not have default values and must be specified.

Additional input arguments specify the size of the result, either as a vector (or a single scalar for a square matrix) or as scalar values. The result is an array of the specified size where each value is an independent pseudo-random variable. The default size is 1 (scalar).

If the parameters are arrays, the result is an array of the same size and each element is an independent pseudo-random variable whose

distribution has its parameters at the corresponding position. The size, if specified, must be the same.

Distribution	Name	Parameters
beta	`beta`	a and b
Cauchy	`cauchy`	a and b
χ	`chi`	deg. of freedom ν
χ^2	`chi2`	deg. of freedom ν
γ	`gamma`	shape α and λ
exponential	`exp`	mean
F	`f`	deg. of freedom ν_1 and ν_2
half-normal	`half-normal`	ϑ
Laplace	`laplace`	mean and scale
lognormal	`logn`	mean and st. dev.
Nakagami	`nakagami`	μ and ω
normal	`norm`	mean and st. dev.
Rayleigh	`rayl`	b
Student's T	`t`	deg. of freedom ν
uniform	`unif`	limits of the range
Weibull	`weib`	shape a and scale b

Example

Array of 5 pseudo-random numbers whose distribution is χ^2 with 3 degrees of freedom:

```
random('chi2', 3, [1, 5])
  1.6442  0.4164  2.0272  2.7962  4.5896
```

See also

`pdf`, `cdf`, `icdf`, `rand`, `randn`, `rng`

rat

Rational approximation.

Syntax

```
(num, den) = rat(x)
(num, den) = rat(x, tol)
(num, den) = rat(x, tol=tol)
```

Description

`rat(x,tol)` returns the numerator and the denominator of a rational approximation of real number `x` with the smallest integer numerator

and denominator which fulfil absolute tolerance `tol`. If the input argument `x` is an array, output arguments are arrays of the same size. Negative numbers give a negative numerator. The tolerance can be passed as a named argument.

With one input argument, `rat(x)` uses tolerance `tol=1e-6*norm(x,1)`. With one output argument, `rat(x)` gives the rational approximation itself as a floating-point number.

With command `format rat`, all numeric results as displayed as rational aproximations with the default tolerance, including complex numbers.

Example

```
(num,den) = rat(pi)
  num =
    355
  den =
    113
num/den
  3.141592920353982
```

See also

`format`

real

Real part of a complex number.

Syntax

```
re = real(z)
```

Description

`real(z)` is the real part of the complex number `z`, or `z` if `z` is real.

Examples

```
real(1+2j)
  1
real(1)
  1
```

See also

`imag`, `complex`

reallog

Real natural (base e) logarithm.

Syntax

```
y = reallog(x)
```

Description

`reallog(x)` gives the real natural logarithm of x. It is the inverse of exp for real numbers. The imaginary part of x is ignored. The result is NaN if x is negative.

Example

```
reallog([-1,0,1,10,1+2j])
  nan inf 0 2.3026 0
```

See also

`log`, `realpow`, `realsqrt`, `exp`

realmax realmin

Largest and smallest real numbers.

Syntax

```
x = realmax
x = realmax(n)
x = realmax(n1,n2,...)
x = realmax([n1,n2,...])
x = realmax(..., type)
x = realmin
x = realmin(...)
```

Description

`realmax` gives the largest positive real number in double precision. `realmin` gives the smallest positive real number in double precision which can be represented in normalized form (i.e. with full mantissa precision).

With integer non-negative arguments, `realmax` and `realmin` create arrays whose elements are all set to the respective value. Arguments are interpreted the same way as `zeros` and `ones`.

The last argument of `realmax` and `realmin` can be a string to specify the type of the result: `'double'` for double-precision (default), or `'single'` for single-precision.

Examples

```
realmin
  2.2251e-308
realmin('single')
  1.1755e-38
realmax
  1.7977e308
realmax('single')
  3.4028e38single
realmax + eps(realmax)
  inf
```

See also

`inf`, `ones`, `zeros`, `eps`, `flintmax`

realpow

Real power.

Syntax

```
z = realpow(x, y)
```

Description

`realpow(x,y)` gives the real value of x to the power y. The imaginary parts of x and y are ignored. The result is NaN if it is not defined for the input arguments. If the arguments are arrays, their size must match or one of them must be a scalar number; the power is performed element-wise.

See also

operator `.^`, `reallog`, `realsqrt`

realsqrt

Real square root.

Syntax

```
y = realsqrt(x)
```

Description

`realsqrt(x)` gives the real square root of x. The imaginary part of x is ignored. The result is NaN if x is negative.

Example

```
realsqrt([-1,0,1,10,1+2j])
  nan 0 1 3.1623 1
```

See also

`sqrt`, `reallog`, `realpow`, `nthroot`

rem

Remainder of a real division.

Syntax

```
r = rem(x, y)
```

Description

`rem(x,y)` gives the remainder of `x` divided by `y`, i.e. a number `r` between 0 and `sign(x)*abs(y)` such that `x = q*y+r` with integer `q`. Imaginary parts, if they exist, are ignored.

Examples

```
rem(10,7)
  3
rem(-10,7)
  -3
rem(10,-7)
  3
rem(-10,-7)
  -3
```

See also

`mod`

round

Rounding to the nearest integer.

Syntax

```
y = round(x)
```

Description

`round(x)` gives the integer nearest to `x`. If the argument is a complex number, the real and imaginary parts are handled separately.

Examples

```
round(2.3)
  2
round(2.6)
  3
round(-2.3)
  -2
```

See also

`floor`, `ceil`, `fix`, `roundn`

roundn

Rounding to a specified precision.

Syntax

```
y = roundn(x, n)
```

Description

`roundn(x,n)` rounds x to the nearest multiple of 10^n. If argument x is a complex number, the real and imaginary parts are handled separately. `roundn(x,0)` gives the same result as `round(x)`.

Argument n must be a real integer. If x and/or n are arrays, rounding is performed separately on each element.

Examples

```
roundn(pi, -2)
  3.1400
roundn(1000 * pi, 1)
  3140
roundn(pi, [-3, -1])
  3.1420  3.1000
```

See also

`round`, `floor`, `ceil`, `fix`

sign

Sign of a real number or direction of a complex number.

Syntax

```
s = sign(x)
z2 = sign(z1)
```

Description

With a real argument, `sign(x)` is 1 if `x>0`, 0 if `x==0`, or -1 if `x<0`. With a complex argument, `sign(z1)` is a complex value with the same phase as `z1` and whose magnitude is 1.

Examples

```
sign(-2)
  -1
sign(1+1j)
  0.7071+0.7071j
sign([0, 5])
  0 1
```

See also

`abs`, `angle`

sec

Secant.

Syntax

```
y = sec(x)
```

Description

sec(x) gives the secant of x, which is complex if x is.

See also

`asec`, `sech`, `cos`

sech

Hyperbolic secant.

Syntax

```
y = sech(x)
```

Description

acot(x) gives the hyperbolic secant of x, which is complex if x is.

See also

`asech`, `sec`, `cosh`

sin

Sine.

Syntax

```
y = sin(x)
```

Description

`sin(x)` gives the sine of x, which is complex if x is complex.

Example

```
sin(2)
  0.9093
```

See also

`cos`, `asin`, `sinh`

sinc

Sinc.

Syntax

```
y = sinc(x)
```

Description

`sinc(x)` gives the sinc of x, i.e. `sin(pi*x)/(pi*x)` if x~=0 or 1 if x==0. The result is complex if x is complex.

Example

```
sinc(1.5)
  -0.2122
```

See also

`sin`, `sinh`

single

Conversion to single-precision numbers.

Syntax

```
B = single(A)
```

Description

`single(A)` converts number or array `A` to single precision. `A` can be any kind of numeric value (real, complex, or integer), or a character or logical array.

Single literal numbers can be entered as a floating-point number with the `single` suffix.

Examples

```
single(pi)
  3.1416single
single('AB')
  1x2 single array
    65 66
3.7e4single
  37000single
```

See also

`double`, `uint8` and related functions, operator `+`, `setstr`, `char`, `logical`

sinh

Hyperbolic sine.

Syntax

```
y = sinh(x)
```

Description

`sinh(x)` gives the hyperbolic sine of `x`, which is complex if `x` is complex.

Example

```
sinh(2)
  3.6269
```

See also

`cosh`, `asinh`, `sin`

sph2cart

Spherical to Cartesian coordinates transform.

Syntax

```
(x, y, z) = sph2cart(phi, theta, r)
```

Description

`(x,y,z)=sph2cart(phi,theta,r)` transforms polar coordinates `phi`, `theta`, and `r` to Cartesian coordinates `x`, `y`, and `z` such that $x = r\cos(\varphi)\cos(\vartheta)$, $y = r\sin(\varphi)\cos(\vartheta)$, and $z = r\sin(\vartheta)$.

Example

```
(x, y, z) = sph2cart(1, 2, 3)
  x =
    -0.6745
  y =
    -1.0505
  z =
    2.7279
```

See also

`cart2pol`, `cart2sph`, `pol2cart`

sqrt

Square root.

Syntax

```
r = sqrt(z)
```

Description

`sqrt(z)` gives the square root of `z`, which is complex if `z` is not real positive.

Examples

```
sqrt(4)
  2
sqrt([1 4 -9 3+4j])
  1 2 3j 2+1j
```

See also

`realsqrt`, `sqrtm`, `chol`

swapbytes

Conversion between big-endian and little-endian representation.

Syntax

```
Y = swapbytes(X)
```

Description

`swapbytes(X)` swaps the bytes representing number X. If X is an array, each number is swapped separately. The imaginary part, if any, is discarded. X can be of any numeric type. `swapbytes` is its own inverse for real numbers.

Example

```
swapbytes(1uint32)
  16777216uint32
```

See also

`typecast`, `cast`

tan

Tangent.

Syntax

```
y = tan(x)
```

Description

`tan(x)` gives the tangent of x, which is complex if x is complex.

Example

```
tan(2)
  -2.185
```

See also

`atan`, `tanh`

tanh

Hyperbolic tangent.

Syntax

```
y = tanh(x)
```

Description

`tanh(x)` gives the hyperbolic tangent of x, which is complex if x is complex.

Example

```
tanh(2)
  0.964
```

See also

`atanh`, `tan`

typecast

Type conversion with same binary representation.

Syntax

```
Y = typecast(X, type)
```

Description

`typecast(X,type)` changes the numeric array X to the type given by string `type`, which can be `'double'`, `'single'`, `'int8'` or any other signed or unsigned integer type, `'char'`, or `'logical'`. The binary representation in memory is preserved. The imaginary part, if any, is discarded. Depending on the conversion, the number of elements is changed, so that the array size in bytes in preserved. The result is a row vector if X is a scalar or a row vector, or a column vector otherwise. The result depends on the computer architecture.

Example

```
typecast(1uint32, 'uint8')
  1x4 uint8 array
    0    0   0   1
typecast(pi, 'uint8')
  1x8 uint8 array
    64   9  33 251  84  68  45  24
```

See also

`swapbytes`, `bwrite`, `sread`, `cast`

10.21 Linear Algebra

addpol

Addition of two polynomials.

Syntax

```
p = addpol(p1,p2)
```

Description

`addpol(p1,p2)` adds two polynomials `p1` and `p2`. Each polynomial is given as a vector of coefficients, with the highest power first; e.g., $x^2 + 2x - 3$ is represented by `[1,2,-3]`. Row vectors and column vectors are accepted, as well as matrices made of row vectors or column vectors, provided one matrix is not larger in one dimension and smaller in the other one. `addpol` is equivalent to the plain addition when both arguments have the same size.

Examples

```
addpol([1,2,3], [2,5])
  1 4 8
addpol([1,2,3], -[2,5])  % subtraction
  1 0 -2
addpol([1,2,3;4,5,6], [1;1])
  1 2 4
  4 5 7
```

See also

`conv`, `deconv`, operator +

balance

Diagonal similarity transform for balancing a matrix.

Syntax

```
B = balance(A)
(T, B) = balance(A)
```

Description

`balance(A)` applies a diagonal similarity transform to the square matrix `A` to make the rows and columns as close in norm as possible. Balancing may reduce the 1-norm of the matrix, and improves the accuracy of the computed eigenvalues and/or eigenvectors. To avoid round-off errors, `balance` scales `A` with powers of 2.

`balance` returns the balanced matrix `B` which has the same eigenvalues and singular values as `A`, and optionally the diagonal scaling matrix `T` such that `T\A*T=B`.

Example

```
A = [1,2e6;3e-6,4];
(T,B) = balance(A)
  T =
   16384       0
       0       3.125e-2
  B =
       1       3.8147
       1.5729  4
```

See also

`eig`

care

Continuous-time algebraic Riccati equation.

Syntax

```
(X, L, K) = care(A, B, Q)
(X, L, K) = care(A, B, Q, R)
(X, L, K) = care(A, B, Q, R, S)
(X, L) = care(A, S, Q, true)
```

Description

`care(A,B,Q)` calculates the stable solution `X` of the following continuous-time algebraic Riccati equation:

$$A'X + XA - XBB'X + Q = 0$$

All matrices are real; `Q` and `X` are symmetric.

With four input arguments, `care(A,B,Q,R)` (with `R` real symmetric) solves the following Riccati equation:

$$A'X + XA - XBR^{-1}B'X + Q = 0$$

With five input arguments, `care(A,B,Q,R,S)` solves the following equation:

$$A'X + XA - (S + XB)R^{-1}(S' + B'X) + Q = 0$$

With two or three output arguments, `(X,L,K) = care(...)` also returns the gain matrix `K` defined as

$$K = R^{-1}B'X$$

and the column vector of closed-loop eigenvalues

$$L = \text{eig}(A - BK)$$

`care(A,S,Q,true)` with up to two output arguments is equivalent to `care(A,B,Q)` or `care(A,B,Q,false)` with `S=B*B'`.

Example

```
A = [-4,2;1,2];
B = [0;1];
C = [2,-1];
Q = C' * C;
R = 5;
(X, L, K) = care(A, B, Q, R)
  X =
      1.07    3.5169
    3.5169   23.2415
  L =
   -4.3488
   -2.2995
  K =
    0.7034    4.6483
A' * X + X * A - X * B / R * B' * X + Q
  1.7319e-14 1.1369e-13
  8.5265e-14 6.2528e-13
```

See also

dare

chol

Cholesky decomposition.

Syntax

```
M2 = chol(M1)
```

Description

If a square matrix M1 is symmetric (or hermitian) and positive definite, it can be decomposed into the following product:

$M_1 = M_2' M_2$

where M2 is an upper triangular matrix. The Cholesky decomposition can be seen as a kind of square root.

The part of M1 below the main diagonal is not used, because M1 is assumed to be symmetric or hermitian. An error occurs if M1 is not positive definite.

Example

```
M = chol([5,3;3,8])
M =
  2.2361 1.3416
  0      2.4900
M'*M
  5 3
  3 8
```

See also

`inv`, `sqrtm`

cond

Condition number of a matrix.

Syntax

```
x = cond(M)
```

Description

`cond(M)` returns the condition number of matrix `M`, i.e. the ratio of its largest singular value divided by the smallest one, or infinity for singular matrices. The larger the condition number, the more ill-conditioned the inversion of the matrix.

Examples

```
cond([1, 0; 0, 1])
 1
cond([1, 1; 1, 1+1e-3])
 4002.0008
```

See also

`svd`, `rank`

conv

Convolution or polynomial multiplication.

Syntax

```
v = conv(v1,v2)
M = conv(M1,M2)
M = conv(M1,M2,dim)
M = conv(...,kind)
```

Description

`conv(v1,v2)` convolves the vectors v1 and v2, giving a vector whose length is length(v1)+length(v2)-1, or an empty vector if v1 or v2 is empty. The result is a row vector if both arguments are row vectors, and a column vector if both arguments are column vectors. Otherwise, arguments are considered as matrices.

`conv(M1,M2)` convolves the matrices `M1` and `M2` column by columns. `conv(M1,M2,dim)` convolves along the dimension `dim`, 1 for columns and 2 for rows. If one of the matrices has only one column, or one row, it is repeated to match the size of the other argument.

Let `n1` and `n2` be the number of elements in `M1` and `M2`, respectively, along the convolution dimension. By default, the result has `n1+n2-1` elements along the convolution dimension. An additional string argument `kind` can specify a different number of elements in the result: with `kind='same'`, the result has `n1` elements (`M` has the same size as `M1`, i.e. `M1` is filtered by the finite impulse response filter `M2`). With `kind='valid'`, the result has `n1-n2+1` elements, i.e. result elements impacted by boundaries are discarded. `kind='full'` produce the same result as if `kind` is missing.

Examples

```
conv([1,2],[1,2,3])
  1 4 7 6
conv([1,2],[1,2,3;4,5,6],2)
  1  4  7  6
  4 13 16 12
conv([1,2,5,8,3],[1,2,1],'full')
   1  4 10 20 24 14  3
conv([1,2,5,8,3],[1,2,1],'same')
   4 10 20 24 14
conv([1,2,5,8,3],[1,2,1],'valid')
  10 20 24
```

See also

`deconv`, `filter`, `addpol`, `conv2`

conv2

Two-dimensions convolution of matrices.

Syntax

```
M = conv2(M1,M2)
M = conv2(M1,M2,kind)
```

Description

`conv2(M1,M2)` convolves the matrices `M1` and `M2` along both directions. The optional third argument specifies how to crop the result. Let `(nr1,nc1)=size(M1)` and `(nr2,nc2)=size(M2)`. With `kind='full'` (default value), the result `M` has `nr1+nr2-1` lines and `nc1+nc2-1` columns. With `kind='same'`, the result `M` has `nr1` lines

and nc1 columns; this options is very useful if M1 represents equidistant samples in a plane (e.g. pixels) to be filtered with the finite-impulse response 2-d filter M2. With kind='valid', the result M has nr1-nr2+1 lines and nc1-nc2+1 columns, or is the empty matrix []; if M1 represents data filtered by M2, the borders where the convolution sum is not totally included in M1 are removed.

Examples

```
conv2([1,2,3;4,5,6;7,8,9],[1,1,1;1,1,1;1,1,1])
  1  3  6  5  3
  5 12 21 16  9
 12 27 45 33 18
 11 24 39 28 15
  7 15 24 17  9
conv2([1,2,3;4,5,6;7,8,9],[1,1,1;1,1,1;1,1,1],'same')
 12 21 16
 27 45 33
 24 39 28
conv2([1,2,3;4,5,6;7,8,9],[1,1,1;1,1,1;1,1,1],'valid')
 45
```

See also

conv

cov

Covariance.

Syntax

```
M = cov(data)
M = cov(data, false)
M = cov(data, true)
```

Description

cov(data) returns the best unbiased estimate m-by-m covariance matrix of the n-by-m matrix data for a normal distribution. Each row of data is an observation where n quantities were measured. The covariance matrix is symmetric if data is real, and hermitian if data is complex (i.e. M==M'). The diagonal is the variance of each column of data.

cov(data,false) is the same as cov(data).

cov(data,true) returns the m-by-m covariance matrix of the n-by-m matrix data which contains the whole population.

Example

```
A = [1,2;2,4;3,5];
cov(A)
  1 1.5
  1.5 2.3333
```

The diagonal elements are the variance of the columns of A:

```
var(A)
  1 2.3333
```

The covariance matrix can be computed as follows:

```
n = size(A, 1);
A1 = A - repmat(mean(A, 1), n, 1);
(A1' * A1) / (n - 1)
  1 1.5
  1.5 2.3333
```

See also

mean, var

cross

Cross product.

Syntax

```
v3 = cross(v1, v2)
v3 = cross(v1, v2, dim)
```

Description

cross(v1,v2) gives the cross products of vectors v1 and v2. v1 and v2 must be row or columns vectors of three components, or arrays of the same size containing several such vectors. When there is ambiguity, a third argument dim may be used to specify the dimension of vectors: 1 for column vectors, 2 for row vectors, and so on.

Examples

```
cross([1; 2; 3], [0; 0; 1])
  2
 -1
  0
cross([1, 2, 3; 7, 1, -3], [4, 0, 0; 0, 2, 0], 2)
  0  12  -8
  6   0  14
```

See also

dot, operator *, det

cummax

Cumulative maximum.

Syntax

```
M2 = cummax(M1)
M2 = cummax(M1,dim)
M2 = cummax(...,dir)
```

Description

cummax(M1) returns a matrix M2 of the same size as M1, whose elements M2(i,j) are the maximum of all the elements M1(k,j) with k<=i. cummax(M1,dim) operates along the dimension dim (column-wise if dim is 1, row-wise if dim is 2).

An optional string argument dir specifies the processing direction. If it is 'reverse' or begins with 'r', cummax processes elements in reverse order, from the last one to the first one, along the processing dimension. If it is 'forward' or begins with 'f', it processes elements as if not specified, in the forward direction.

Examples

```
cummax([1,2,3;5,1,4;2,8,7])
  1  2  3
  5  2  4
  5  8  7
cummax([1,2,3;5,1,4;2,8,7], 2)
  1  2  3
  5  5  5
  2  8  8
```

See also

max, cummin, cumsum, cumprod

cummin

Cumulative minimum.

Syntax

```
M2 = cummin(M1)
M2 = cummin(M1,dim)
M2 = cummin(...,dir)
```

Description

`cummin(M1)` returns a matrix `M2` of the same size as `M1`, whose elements `M2(i,j)` are the minimum of all the elements `M1(k,j)` with `k<=i`. `cummin(M1,dim)` operates along the dimension `dim` (column-wise if `dim` is 1, row-wise if `dim` is 2).

An optional string argument `dir` specifies the processing direction. If it is `'reverse'` or begins with `'r'`, `cummin` processes elements in reverse order, from the last one to the first one, along the processing dimension. If it is `'forward'` or begins with `'f'`, it processes elements as if not specified, in the forward direction.

See also

`min`, `cummax`, `cumsum`, `cumprod`

cumprod

Cumulative products.

Syntax

```
M2 = cumprod(M1)
M2 = cumprod(M1,dim)
M2 = cumprod(...,dir)
```

Description

`cumprod(M1)` returns a matrix `M2` of the same size as `M1`, whose elements `M2(i,j)` are the product of all the elements `M1(k,j)` with `k<=i`. `cumprod(M1,dim)` operates along the dimension `dim` (column-wise if `dim` is 1, row-wise if `dim` is 2).

An optional string argument `dir` specifies the processing direction. If it is `'reverse'` or begins with `'r'`, `cumprod` processes elements in reverse order, from the last one to the first one, along the processing dimension. If it is `'forward'` or begins with `'f'`, it processes elements as if not specified, in the forward direction.

Examples

```
cumprod([1,2,3;4,5,6])
  1  2  3
  4 10 18
cumprod([1,2,3;4,5,6],2)
  1  2   6
  4 20 120
```

See also

`prod`, `cumsum`, `cummax`, `cummin`

cumsum

Cumulative sums.

Syntax

```
M2 = cumsum(M1)
M2 = cumsum(M1,dim)
M2 = cumsum(...,dir)
```

Description

`cumsum(M1)` returns a matrix `M2` of the same size as `M1`, whose elements `M2(i,j)` are the sum of all the elements `M1(k,j)` with `k<=i`. `cumsum(M1,dim)` operates along the dimension `dim` (column-wise if `dim` is 1, row-wise if `dim` is 2).

An optional string argument `dir` specifies the processing direction. If it is `'reverse'` or begins with `'r'`, `cumsum` processes elements in reverse order, from the last one to the first one, along the processing dimension. If it is `'forward'` or begins with `'f'`, it processes elements as if not specified, in the forward direction.

Examples

```
cumsum([1,2,3;4,5,6])
  1 2 3
  5 7 9
cumsum([1,2,3;4,5,6],2)
  1 3  6
  4 9 15
cumsum([1,2,3;4,5,6],2,'r')
  6   5 3
  15 11 6
```

See also

`sum`, `diff`, `cumprod`, `cummax`, `cummin`

dare

Discrete-time algebraic Riccati equation.

Syntax

```
(X, L, K) = dare(A, B, Q)
(X, L, K) = dare(A, B, Q, R)
```

Description

`dare(A,B,Q)` calculates the stable solution X of the following discrete-time algebraic Riccati equation:

$$X = A'XA - A'XB(B'XB + I)^{-1}B'XA + Q$$

All matrices are real; Q and X are symmetric.

With four input arguments, `dare(A,B,Q,R)` (with R real symmetric) solves the following Riccati equation:

$$X = A'XA - A'XB(B'XB + R)^{-1}B'XA + Q$$

With two or three output arguments, `(X,L,K) = dare(...)` also returns the gain matrix K defined as

$$K = (B'XB + R)^{-1}B'XA$$

and the column vector of closed-loop eigenvalues

$$L = \text{eig}(A - BK)$$

Example

```
A = [-4,2;1,2];
B = [0;1];
C = [2,-1];
Q = C' * C;
R = 5;
(X, L, K) = dare(A, B, Q, R)
  X =
    2327.9552 -1047.113
   -1047.113    496.0624
  L =
     -0.2315
      0.431
  K =
    9.3492   -2.1995
-X + A'*X*A - A'*X*B/(B'*X*B+R)*B'*X*A + Q
    1.0332e-9  -4.6384e-10
   -4.8931e-10  2.2101e-10
```

See also

`care`

deconv

Deconvolution or polynomial division.

Syntax

```
q = deconv(a,b)
(q,r) = deconv(a,b)
```

Description

`(q,r)=deconv(a,b)` divides the polynomial `a` by the polynomial `b`, resulting in the quotient `q` and the remainder `r`. All polynomials are given as vectors of coefficients, highest power first. The degree of the remainder is strictly smaller than the degree of `b`. `deconv` is the inverse of `conv`: `a = addpol(conv(b,q),r)`.

Examples

```
[q,r] = deconv([1,2,3,4,5],[1,3,2])
q =
  1 -1 4
r =
  -6 -3
addpol(conv(q,[1,3,2]),r)
  1 2 3 4 5
```

See also

`conv`, `filter`, `addpol`

det

Determinant of a square matrix.

Syntax

```
d = det(M)
```

Description

`det(M)` is the determinant of the square matrix `M`, which is 0 (up to the rounding errors) if `M` is singular. The function `rank` is a numerically more robust test for singularity.

Examples

```
det([1,2;3,4])
  -2
det([1,2;1,2])
  0
```

See also

`poly`, `rank`

diff

Differences.

Syntax

```
dm = diff(A)
dm = diff(A,n)
dm = diff(A,n,dim)
dm = diff(A,[],dim)
```

Description

`diff(A)` calculates the differences between each elements of the columns of matrix `A`, or between each elements of `A` if it is a row vector.

`diff(A,n)` calculates the n:th order differences, i.e. it repeats `n` times the same operation. Up to a scalar factor, the result is an approximation of the n:th order derivative based on equidistant samples.

`diff(A,n,dim)` operates along dimension `dim`. If the second argument `n` is the empty matrix `[]`, the default value of 1 is assumed.

Examples

```
diff([1,3,5,4,8])
  2 2 -1 4
diff([1,3,5,4,8],2)
  0 -3 5
diff([1,3,5;4,8,2;3,9,8],1,2)
 2  2
 4 -6
 6 -1
```

See also

`cumsum`

dlyap

Discrete-time Lyapunov equation.

Syntax

```
X = dlyap(A, C)
```

Description

`dlyap(A,C)` calculates the solution `X` of the following discrete-time Lyapunov equation:

$$AXA' - X + C = 0$$

All matrices are real.

Example

```
A = [3,1,2;1,3,5;6,2,1];
C = [7,1,2;4,3,5;1,2,1];
X = dlyap(A, C)
  X =
   -1.0505    3.2222   -1.2117
    3.2317   -11.213    4.8234
   -1.4199     5.184   -2.7424
```

See also

`lyap`, `dare`

dot

Scalar product.

Syntax

```
v3 = dot(v1, v2)
v3 = dot(v1, v2, dim)
```

Description

`dot(v1,v2)` gives the scalar products of vectors `v1` and `v2`. `v1` and `v2` must be row or columns vectors of same length, or arrays of the same size; then the scalar product is performed along the first dimension not equal to 1. A third argument `dim` may be used to specify the dimension the scalar product is performed along.

With complex values, complex conjugate values of the first array are multiplied with values of the second array.

Examples

```
dot([1; 2; 3], [0; 0; 1])
  3
dot([1, 2, 3; 7, 1, -3], [4, 0, 0; 0, 2, 0], 2)
  4
  2
dot([1; 2i], [3i; 5])
  0 - 7i
```

See also

`cross`, operator `*`, `det`

eig

Eigenvalues and eigenvectors of a matrix.

Syntax

```
e = eig(M)
(V,D) = eig(M)
```

Description

`eig(M)` returns the vector of eigenvalues of the square matrix `M`.

`(V,D) = eig(M)` returns a diagonal matrix `D` of eigenvalues and a matrix `V` whose columns are the corresponding eigenvectors. They are such that `M∗V = V∗D`.

Examples

Eigenvalues as a vector:

```
eig([1,2;3,4])
  -0.3723
  5.3723
```

Eigenvectors, and eigenvalues as a diagonal matrix:

```
(V,D) = eig([1,2;2,1])
 V =
  0.7071 0.7071
  -0.7071 0.7071
 D =
  -1 0
  0 3
```

Checking that the result is correct:

```
[1,2;2,1] ∗ V
  -0.7071 2.1213
  0.7071 2.1213
V ∗ D
  -0.7071 2.1213
  0.7071 2.1213
```

See also

`schur`, `svd`, `det`, `roots`

expm

Exponential of a square matrix.

Syntax

```
M2 = expm(M1)
```

Description

`expm(M)` is the exponential of the square matrix `M`, which is usually different from the element-wise exponential of `M` given by `exp`.

Examples

```
expm([1,1;1,1])
  4.1945 3.1945
  3.1945 4.1945
exp([1,1;1,1])
  2.7183 2.7183
  2.7183 2.7183
```

See also

`logm`, operator `^`, `exp`

fft

Fast Fourier Transform.

Syntax

```
F = fft(f)
F = fft(f,n)
F = fft(f,n,dim)
```

Description

`fft(f)` returns the discrete Fourier transform (DFT) of the vector `f`, or the DFT's of each columns of the array `f`. With a second argument `n`, the `n` first values are used; if `n` is larger than the length of the data, zeros are added for padding. An optional argument `dim` gives the dimension along which the DFT is performed; it is 1 for calculating the DFT of the columns of `f`, 2 for its rows, and so on. `fft(f,[],dim)` specifies the dimension without resizing the array.

`fft` is based on a mixed-radix Fast Fourier Transform if the data length is non-prime. It can be very slow if the data length has large prime factors or is a prime number.

The coefficients of the DFT are given from the zero frequency to the largest frequency (one point less than the inverse of the sampling period). If the input `f` is real, its DFT has symmetries, and the first half contain all the relevant information.

Examples

```
fft(1:4)
  10 -2+2j -2 -2-2j
fft(1:4, 3)
  6 -1.5+0.866j -1.5-0.866j
```

See also

`ifft`

fft2

2-d Fast Fourier Transform.

Syntax

```
F = fft2(f)
F = fft2(f, size)
F = fft2(f, nr, nc)
F = fft2(f, n)
```

Description

`fft2(f)` returns the 2-d Discrete Fourier Transform (DFT along dimensions 1 and 2) of array `f`.

With two or three input arguments, `fft2` resizes the two first dimensions by cropping or by padding with zeros. `fft2(f,nr,nc)` resizes first dimension to `nr` rows and second dimension to `nc` columns. In `fft2(f,size)`, the new size is given as a two-element vector `[nr,nc]`. `fft2(F,n)` is equivalent to `fft2(F,n,n)`.

If the first argument is an array with more than two dimensions, `fft2` performs the 2-d DFT along dimensions 1 and 2 separately for each plane along remaining dimensions; `fftn` performs an DFT along each dimension.

See also

`ifft2`, `fft`, `fftn`

fftn

n-dimension Fast Fourier Transform.

Syntax

```
F = fftn(f)
F = fftn(f, size)
```

Description

`fftn(f)` returns the n-dimension Discrete Fourier Transform of array `f` (DFT along each dimension of `f`).

With two input arguments, `fftn(f,size)` resizes `f` by cropping or by padding `f` with zeros.

See also

`ifftn`, `fft`, `fft2`

filter

Digital filtering of data.

Syntax

```
y = filter(b,a,u)
y = filter(b,a,u,x0)
y = filter(b,a,u,x0,dim)
(y, xf) = filter(...)
```

Description

`filter(b,a,u)` filters vector u with the digital filter whose coefficients are given by polynomials b and a. The filtered data can also be an array, filtered along the first non-singleton dimension or along the dimension specified with a fifth input argument. The fourth argument, if provided and different than the empty matrix `[]`, is a matrix whose columns contain the initial state of the filter and have `max(length(a),length(b))-1` element. Each column correspond to a signal along the dimension of filtering. The result y, which has the same size as the input, can be computed with the following code if u is a vector:

```
b = b / a(1);
a = a / a(1);
if length(a) > length(b)
  b = [b, zeros(1, length(a)-length(b))];
else
  a = [a, zeros(1, length(b)-length(a))];
end
n = length(x);
for i = 1:length(u)
  y(i) = b(1) * u(i) + x(1);
  for j = 1:n-1
    x(j) = b(j + 1) * u(i) + x(j + 1) - a(j + 1) * y(i);
  end
  x(n) = b(n + 1) * u(i) - a(n + 1) * y(i);
end
```

The optional second output argument is set to the final state of the filter.

Examples

```
filter([1,2], [1,2,3], ones(1,10))
  1 1 -2 4 1 -11 22 -8 -47 121

u = [5,6,5,6,5,6,5];
p = 0.8;
filter(1-p, [1,-p], u, p*u(1))
      % low-pass with matching initial state
  5 5.2 5.16 5.328 5.2624 5.4099 5.3279
```

See also

conv, deconv, conv2

funm

Matrix function.

Syntax

```
Y = funm(X, fun)
(Y, err) = funm(X, fun)
```

Description

`funm(X,fun)` returns the matrix function of square matrix X specified by function `fun`. `fun` takes a scalar input argument and gives a scalar output. It is either specified by its name or given as an anonymous or inline function or a function reference.

With a second output argument `err`, `funm` also returns an estimate of the relative error.

Examples

```
funm([1,2;3,4], @sin)
  -0.4656  -0.1484
  -0.2226  -0.6882
X = [1,2;3,4];
funm(X, @(x) (1+x)/(2-x))
   -0.25  -0.75
  -1.125  -1.375
(eye(2)+X)/(2*eye(2)-X)
   -0.25  -0.75
  -1.125  -1.375
```

See also

expm, logm, sqrtm, schur

householder

Householder transform.

Syntax

```
(nu, beta) = householder(x)
```

Description

The householder transform is an orthogonal matrix transform which sets all the elements of a column to zero, except the first one. It is the elementary step used by QR decomposition.

The matrix transform can be written as a product by an orthogonal square matrix `P=I-beta*nu*nu'`, where `I` is the identity matrix, `beta` is a scalar, and `nu` is a column vector where `nu(1)` is 1. `householder(x)`, where x is a real or complex non-empty column vector, gives nu and `beta` such that `P*x=[y;Z]`, where y is a scalar and Z a zero column vector.

Example

```
x = [2; 5; 10];
(nu, beta) = householder(x)
  nu =
    1.0000
 0.3743
 0.7486
  beta =
    1.1761
P = eye(3) - beta * nu * nu'
  P =
    -0.1761 -0.4402 -0.8805
    -0.4402  0.8352 -0.3296
    -0.8805 -0.3296  0.3409
P * x
  ans =
   -11.3578
     0.0000
     0.0000
```

It is more efficient to avoid calculating P explicitly. Multiplication by P, either as `P*A` (to set elements to zero) or `B*P'` (to accumulate transforms), can be performed by passing `nu` and `beta` to `householderapply`:

```
householderapply(x, nu, beta)
  ans =
   -11.3578
     0.0000
     0.0000
```

See also

`householderapply`, `qr`

householderapply

Apply Householder transform.

Syntax

```
B = householderapply(A, nu, beta)
B = householderapply(A, nu, beta, 'r')
```

Description

`householderapply(A,nu,beta)` apply the Householder transform defined by column vector `nu` (where `nu(1)` is 1) and real scalar `beta`, as obtained by `householder`, to matrix `A`; i.e. it computes `A-nu*beta*nu'*A`.

`householderapply(A,nu,beta,'r')` apply the inverse Householder transform to matrix `A`; i.e. it computes `A-A*nu*beta*nu'`.

See also

`householder`

ifft

Inverse Fast Fourier Transform.

Syntax

```
f = ifft(F)
f = ifft(F, n)
f = ifft(F, n, dim)
```

Description

`ifft` returns the inverse Discrete Fourier Transform (inverse DFT). Up to the sign and a scaling factor, the inverse DFT and the DFT are the same operation: for a vector, `ifft(d) = conj(fft(d))/length(d)`. `ifft` has the same syntax as `fft`.

Examples

```
F = fft([1,2,3,4])
  F =
    10 -2+2j -2 -2-2j
ifft(F)
    1 2 3 4
```

See also

`fft`, `ifft2`, `ifftn`

ifft2

Inverse 2-d Fast Fourier Transform.

Syntax

```
f = ifft2(F)
f = ifft2(F, size)
f = ifft2(F, nr, nc)
f = ifft2(F, n)
```

Description

`ifft2` returns the inverse 2-d Discrete Fourier Transform (inverse DFT along dimensions 1 and 2).

With two or three input arguments, `ifft2` resizes the two first dimensions by cropping or by padding with zeros. `ifft2(F,nr,nc)` resizes first dimension to `nr` rows and second dimension to `nc` columns. In `ifft2(F,size)`, the new size is given as a two-element vector `[nr,nc]`. `ifft2(F,n)` is equivalent to `ifft2(F,n,n)`.

If the first argument is an array with more than two dimensions, `ifft2` performs the inverse 2-d DFT along dimensions 1 and 2 separately for each plane along remaining dimensions; `ifftn` performs an inverse DFT along each dimension.

Up to the sign and a scaling factor, the inverse 2-d DFT and the 2-d DFT are the same operation. `ifft2` has the same syntax as `fft2`.

See also

`fft2`, `ifft`, `ifftn`

ifftn

Inverse n-dimension Fast Fourier Transform.

Syntax

```
f = ifftn(F)
f = ifftn(F, size)
```

Description

`ifftn(F)` returns the inverse n-dimension Discrete Fourier Transform of array F (inverse DFT along each dimension of F).

With two input arguments, `ifftn(F,size)` resizes F by cropping or by padding F with zeros.

Up to the sign and a scaling factor, the inverse n-dimension DFT and the n-dimension DFT are the same operation. `ifftn` has the same syntax as `fftn`.

See also

`fftn`, `ifft`, `ifft2`

hess

Hessenberg reduction.

Syntax

```
(P,H) = hess(A)
H = hess(A)
```

Description

`hess(A)` reduces the square matrix A A to the upper Hessenberg form H using an orthogonal similarity transformation P∗H∗P'=A. The result H is zero below the first subdiagonal and has the same eigenvalues as A.

Example

```
(P,H)=hess([1,2,3;4,5,6;7,8,9])
 P =
   1        0       0
   0       -0.4961 -0.8682
   0       -0.8682  0.4961
 H =
   1       -3.597  -0.2481
   -8.0623 14.0462  2.8308
   0        0.8308 -4.6154e-2
P∗H∗P'
ans =
   1        2       3
   4        5       6
   7        8       9
```

See also

`lu`, `qr`, `schur`

inv

Inverse of a square matrix.

Syntax

```
M2 = inv(M1)
```

Description

`inv(M1)` returns the inverse `M2` of the square matrix `M1`, i.e. a matrix of the same size such that `M2*M1 = M1*M2 = eye(size(M1))`. `M1` must not be singular; otherwise, its elements are infinite.

To solve a set of linear of equations, the operator \ is more efficient.

Example

```
inv([1,2;3,4])
  -2 1
  1.5 -0.5
```

See also

operator `/`, operator \, `pinv`, `lu`, `rank`, `eye`

kron

Kronecker product.

Syntax

```
M = kron(A, B)
```

Description

`kron(A,B)` returns the Kronecker product of matrices `A` (size m1 by n1) and `B` (size m2 by n2), i.e. an m1*m2-by-n1*n2 matrix made of m1 by n1 submatrices which are the products of each element of `A` with `B`.

Example

```
kron([1,2;3,4],ones(2))
  1  1  2  2
  1  1  2  2
  3  3  4  4
  3  3  4  4
```

See also

`repmat`

kurtosis

Kurtosis of a set of values.

Syntax

```
k = kurtosis(A)
k = kurtosis(A, dim)
```

Description

`kurtosis(A)` gives the kurtosis of the columns of array `A` or of the row vector `A`. The dimension along which `kurtosis` proceeds may be specified with a second argument.

The kurtosis measures how much values are far away from the mean. It is 3 for a normal distribution, and positive for a distribution which has more values far away from the mean.

Example

```
kurtosis(rand(1, 10000))
  1.8055
```

See also

`mean`, `var`, `skewness`, `moment`

linprog

Linear programming.

Syntax

```
x = linprog(c, A, b)
x = linprog(c, A, b, xlb, xub)
```

Description

`linprog(c,A,b)` solves the following linear programming problem:

$$\begin{aligned} &\min cx \\ &\text{s.t. } Ax \leq b \end{aligned}$$

The optimum `x` is either finite, infinite if there is no bounded solution, or not a number if there is no feasible solution.

Additional arguments may be used to constrain `x` between lower and upper bounds. `linprog(c,A,b,xlb,xub)` solves the following linear programming problem:

$$\begin{aligned} \min & \, c\,x \\ \text{s.t. } & A x \;\leq\; b \\ & x \;\geq\; x_{\text{lb}} \\ & x \;\leq\; x_{\text{ub}} \end{aligned}$$

If xub is missing, there is no upper bound. `xlb` and xub may have less elements than `x`, or contain `-inf` or `+inf`; corresponding elements have no lower and/or upper bounds.

Examples

Maximize $3x + 2y$ subject to $x + y \leq 9$, $3x + y \leq 18$, $x \leq 7$, and $y \leq 6$:

```
c = [-3,-2];
A = [1,1; 3,1; 1,0; 0,1];
b = [9; 18; 7; 6];
x = linprog(c, A, b)
  x =
    4.5
    4.5
```

A more efficient way to solve the problem, with bounds on variables:

```
c = [-3,-2];
A = [1,1; 3,1];
b = [9; 18];
xlb = [];
xub = [7; 6];
x = linprog(c, A, b, xlb, xub)
  x =
    4.5
    4.5
```

Check that the solution is feasible and bounded:

```
all(isfinite(x))
  true
```

logm

Matrix logarithm.

Syntax

```
Y = logm(X)
(Y, err) = logm(X)
```

Description

`logm(X)` returns the matrix logarithm of `X`, the inverse of the matrix exponential. `X` must be square. The matrix logarithm does not always exist.

With a second output argument `err`, `logm` also returns an estimate of the relative error `norm(expm(logm(X))-X)/norm(X)`.

Example

```
Y = logm([1,2;3,4])
  Y =
   -0.3504 + 2.3911j  0.9294 - 1.0938j
     1.394 - 1.6406j  1.0436 + 0.7505j
expm(Y)
     1 - 5.5511e-16j  2 -7.7716e-16j
     3 - 8.3267e-16j  4
```

See also

`expm`, `sqrtm`, `funm`, `schur`, `log`

lu

LU decomposition.

Syntax

```
(L, U, P) = lu(A)
(L2, U) = lu(A)
Y = lu(A)
```

Description

With three output arguments, `lu(A)` computes the LU decomposition of matrix `A` with partial pivoting, i.e. a lower triangular matrix `L`, an upper triangular matrix `U`, and a permutation matrix `P` such that `P*A=L*U`. If `A` in an m-by-n mytrix, `L` is m-by-min(m,n), `U` is min(m,n)-by-n and `P` is m-by-m. `A` can be rank-deficient.

With two output arguments, `lu(A)` permutes the lower triangular matrix and gives `L2=P'*L`, such that `A=L2*U`.

With a single output argument, `lu` gives `Y=L+U-eye(n)`.

Example

```
X = [1,2,3;4,5,6;7,8,8];
(L,U,P) = lu(X)
L =
 1     0     0
 0.143 1     0
```

```
 0.571 0.5   1
U =
 7     8     8
 0     0.857 1.857
 0     0     0.5
P =
 0 0 1
 1 0 0
 0 1 0
P*X-L*U
ans =
 0 0 0
 0 0 0
 0 0 0
```

See also

`inv`, `qr`, `svd`

lyap

Continuous-time Lyapunov equation.

Syntax

```
X = lyap(A, B, C)
X = lyap(A, C)
```

Description

`lyap(A,B,C)` calculates the solution X of the following continuous-time Lyapunov equation:

$$AX + XB + C = 0$$

All matrices are real.

With two input arguments, `lyap(A,C)` solves the following Lyapunov equation:

$$AX + XA' + C = 0$$

Example

```
A = [3,1,2;1,3,5;6,2,1];
B = [2,7;8,3];
C = [2,1;4,5;8,9];
X = lyap(A, B, C)
  X =
    0.1635   -0.1244
   -0.2628    0.1311
   -0.7797   -0.7645
```

See also

`dlyap`, `care`

max

Maximum value of a vector or of two arguments.

Syntax

```
x = max(v)
(v,ind) = max(v)
v = max(M,[],dim)
(v,ind) = max(M,[],dim)
M3 = max(M1,M2)
```

Description

`max(v)` returns the largest number of vector `v`. NaN's are ignored. The optional second output argument is the index of the maximum in `v`; if several elements have the same maximum value, only the first one is obtained. The argument type can be double, single, or integer of any size.

`max(M)` operates on the columns of the matrix `M` and returns a row vector. `max(M,[],dim)` operates along dimension `dim` (1 for columns, 2 for rows).

`max(M1,M2)` returns a matrix whose elements are the maximum between the corresponding elements of the matrices `M1` and `M2`. `M1` and `M2` must have the same size, or be a scalar which can be compared against any matrix.

Examples

```
(mx,ix) = max([1,3,2,5,8,7])
  mx =
    8
  ix =
    5
max([1,3;5,nan], [], 2)
    3
    5
max([1,3;5,nan], 2)
    2 3
    5 2
```

See also

`min`

mean

Arithmetic mean of a vector.

Syntax

```
x = mean(v)
v = mean(M)
v = mean(M,dim)
```

Description

`mean(v)` returns the arithmetic mean of the elements of vector `v`. `mean(M)` returns a row vector whose elements are the means of the corresponding columns of matrix `M`. `mean(M,dim)` returns the mean of matrix `M` along dimension `dim`; the result is a row vector if `dim` is 1, or a column vector if `dim` is 2.

Examples

```
mean(1:5)
  7.5
mean((1:5)')
  7.5
mean([1,2,3;5,6,7])
  3 4 5
mean([1,2,3;5,6,7],1)
  3 4 5
mean([1,2,3;5,6,7],2)
  2
  6
```

See also

`cov`, `std`, `var`, `median`, `sum`, `prod`

median

Median.

Syntax

```
x = median(v)
v = median(M)
v = median(M, dim)
```

Description

`median(v)` gives the median of vector v, i.e. the value x such that half of the elements of v are smaller and half of the elements are larger. The result is `NaN` if any value is `NaN`.

`median(M)` gives a row vector which contains the median of the columns of `M`. With a second argument, `median(M,dim)` operates along dimension `dim`.

Example

```
median([1, 2, 5, 6, inf])
  5
```

See also

`mean`, `sort`

min

Minimum value of a vector or of two arguments.

Syntax

```
x = min(v)
(v,ind) = min(v)
v = min(M,[],dim)
(v,ind) = min(M,[],dim)
M3 = min(M1,M2)
```

Description

`min(v)` returns the largest number of vector v. NaN's are ignored. The optional second smallest argument is the index of the minimum in v; if several elements have the same minimum value, only the first one is obtained. The argument type can be double, single, or integer of any size.

`min(M)` operates on the columns of the matrix `M` and returns a row vector. `min(M,[],dim)` operates along dimension `dim` (1 for columns, 2 for rows).

`min(M1,M2)` returns a matrix whose elements are the minimum between the corresponding elements of the matrices `M1` and `M2`. `M1` and `M2` must have the same size, or be a scalar which can be compared against any matrix.

Examples

```
(mx,ix) = min([1,3,2,5,8,7])
  mx =
    1
  ix =
    1
min([1,3;5,nan], [], 2)
    1
    5
min([1,3;5,nan], 2)
    1 2
    2 2
```

See also

max

moment

Central moment of a set of values.

Syntax

```
m = moment(A, order)
m = moment(A, order, dim)
```

Description

`moment(A,order)` gives the central moment (moment about the mean) of the specified order of the columns of array `A` or of the row vector `A`. The dimension along which `moment` proceeds may be specified with a third argument.

Example

```
moment(randn(1, 10000), 3)
  3.011
```

See also

`mean`, `var`, `skewness`, `kurtosis`

norm

Norm of a vector or matrix.

Syntax

```
x = norm(v)
x = norm(v,kind)
x = norm(M)
x = norm(M,kind)
```

Description

With one argument, `norm` calculates the 2-norm of a vector or the induced 2-norm of a matrix. The optional second argument specifies the kind of norm.

Kind	Vector	Matrix
none or 2	`sqrt(sum(abs(v).^2))`	largest singular value (induced 2-norm)
1	`sum(abs(V))`	largest column sum of abs
inf or `'inf'`	max(abs(v))	largest row sum of abs
-inf	`min(abs(v))`	invalid
p	`sum(abs(V).^p)^(1/p)`	invalid
`'fro'`	`sqrt(sum(abs(v).^2))`	`sqrt(sum(diag(M'*M)))`

Examples

```
norm([3,4])
  5
norm([2,5;9,3])
  10.2194
norm([2,5;9,3],1)
  11
```

See also

`abs`, `hypot`, `svd`

null

Null space.

Syntax

```
Z = null(A)
Z = null(A, tol=tol)
```

Description

`null(A)` returns a matrix `Z` whose columns are an orthonormal basis for the null space of m-by-n matrix `A`. `Z` has `n - rank(A)` columns, which are the last right singular values of `A`; that is, those corresponding to

the singular values below a small tolerance. This tolerance can be specified with a named argument `tol`.

Without input argument, `null` gives the null value (the unique value of the special null type, not related to linear algebra).

Example

```
null([1,2,3;1,2,4;1,2,5])
  -0.8944
  0.4472
  8.0581e-17
```

See also

`svd`, `orth`, null (null value)

orth

Orthogonalization.

Syntax

```
Q = orth(A)
Q = orth(A, tol=tol)
```

Description

`orth(A)` returns a matrix `Q` whose columns are an orthonormal basis for the range of those of matrix `A`. `Q` has `rank(A)` columns, which are the first left singular vectors of `A` (that is, those corresponding to the largest singular values).

Orthogonalization is based on the singular-value decomposition, where only the singular values larger than some small threshold are considered. This threshold can be specified with an optional named argument.

Example

```
orth([1,2,3;1,2,4;1,2,5])
  -0.4609 0.788
  -0.5704 8.9369e-2
  -0.6798 -0.6092
```

See also

`svd`, `null`

pinv

Pseudo-inverse of a matrix.

Syntax

```
M2 = pinv(M1)
M2 = pinv(M1, tol)
M2 = pinv(M1, tol=tol)
```

Description

`pinv(M1)` returns the pseudo-inverse of matrix `M`. For a nonsingular square matrix, the pseudo-inverse is the same as the inverse. For an arbitrary matrix (possibly nonsquare), the pseudo-inverse `M2` has the following properties: `size(M2) = size(M1')`, `M1*M2*M1 = M1`, `M2*M1*M2 = M2`, and the norm of `M2` is minimum. The pseudo-inverse is based on the singular-value decomposition, where only the singular values larger than some small threshold are considered. This threshold can be specified with an optional second argument `tol` or as a named argument.

If `M1` is a full-rank matrix with more rows than columns, `pinv` returns the least-square solution `pinv(M1)*y = (M1'*M1)\M1'*y` of the over-determined system `M1*x = y`.

Examples

```
pinv([1,2;3,4])
    -2    1
     1.5 -0.5
M2 = pinv([1;2])
  M2 =
    0.2 0.4
[1;2] * M2 * [1;2]
    1
    2
M2 * [1;2] * M2
    0.2 0.4
```

See also

`inv`, `svd`

poly

Characteristic polynomial of a square matrix or polynomial coefficients based on its roots.

Syntax

```
pol = poly(M)
pol = poly(r)
```

Description

With a matrix argument, `poly(M)` returns the characteristic polynomial `det(x*eye(size(M))-M)` of the square matrix `M`. The roots of the characteristic polynomial are the eigenvalues of `M`.

With a vector argument, `poly(r)` returns the polynomial whose roots are the elements of the vector `r`. The first coefficient of the polynomial is 1. If the complex roots form conjugate pairs, the result is real.

Examples

```
poly([1,2;3,4]
  1 -5 -2
roots(poly([1,2;3,4]))
  5.3723
 -0.3723
eig([1,2;3,4])
 -0.3723
  5.3723
poly(1:3)
  1 -6 11 -6
```

See also

`roots`, `det`

polyder

Derivative of a polynomial or a polynomial product or ratio.

Syntax

```
A1 = polyder(A)
C1 = polyder(A, B)
(N1, D1) = polyder(N, D)
```

Description

`polyder(A)` returns the polynomial which is the derivative of the polynomial `A`. Both polynomials are given as vectors of coefficients, highest power first. The result is a row vector.

With a single output argument, `polyder(A,B)` returns the derivative of the product of polynomials `A` and `B`. It is equivalent to `polyder(conv(A,B))`.

With two output arguments, `(N1,D1)=polyder(N,D)` returns the derivative of the polynomial ratio `N/D` as `N1/D1`. Input and output arguments are polynomial coefficients.

Examples

Derivative of $x^3 + 2x^2 + 5x + 2$:

```
polyder([1, 2, 5, 2])
  3 4 5
```

Derivative of $(x^3 + 2x^2 + 5x + 2)/(2x + 3)$:

```
(N, D) = polyder([1, 2, 5, 2], [2, 3])
  N =
    4 13 12 11
  D =
    4 12  9
```

See also

`polyint`, `polyval`, `poly`, `addpol`, `conv`

polyint

Integral of a polynomial.

Syntax

```
pol2 = polyint(pol1)
pol2 = polyint(pol1, c)
```

Description

`polyint(pol1)` returns the polynomial which is the integral of the polynomial `pol1`, whose zero-order coefficient is 0. Both polynomials are given as vectors of coefficients, highest power first. The result is a row vector. A second input argument can be used to specify the integration constant.

Example

```
Y = polyint([1, 2, 3, 4, 5])
  Y =
    0.2   0.5   1     2     5     0
y = polyder(Y)
  y =
    1     2     3     4     5
Y = polyint([1, 2, 3, 4, 5], 10)
  Y =
    0.2   0.5   1     2     5     10
```

See also

`polyder`, `polyval`, `poly`, `addpol`, `conv`

polyval

Numeric value of a polynomial evaluated at some point.

Syntax

```
y = polyval(pol, x)
```

Description

`polyval(pol,x)` evaluates the polynomial `pol` at `x`, which can be a scalar or a matrix of arbitrary size. The polynomial is given as a vector of coefficients, highest power first. The result has the same size as `x`.

Examples

```
polyval([1,3,8], 2)
  18
polyval([1,2], 1:5)
  3 4 5 6 7
```

See also

`polyder`, `polyint`, `poly`, `addpol`, `conv`

prod

Product of the elements of a vector.

Syntax

```
x = prod(v)
v = prod(M)
v = prod(M,dim)
```

Description

`prod(v)` returns the product of the elements of vector `v`. `prod(M)` returns a row vector whose elements are the products of the corresponding columns of matrix `M`. `prod(M,dim)` returns the product of matrix `M` along dimension `dim`; the result is a row vector if `dim` is 1, or a column vector if `dim` is 2.

Examples

```
prod(1:5)
  120
prod((1:5)')
  120
prod([1,2,3;5,6,7])
```

```
  5 12 21
prod([1,2,3;5,6,7],1)
  5 12 21
prod([1,2,3;5,6,7],2)
  6
  210
```

See also

sum, mean, operator ∗

qr

QR decomposition.

Syntax

```
(Q, R, E) = qr(A)
(Q, R) = qr(A)
R = qr(A)
(Qe, Re, e) = qr(A, false)
(Qe, Re) = qr(A, false)
Re = qr(A, false)
```

Description

With three output arguments, qr(A) computes the QR decomposition of matrix A with column pivoting, i.e. a square unitary matrix Q and an upper triangular matrix R such that A∗E=Q∗R. With two output arguments, qr(A) computes the QR decomposition without pivoting, such that A=Q∗R. With a single output argument, qr gives R.

With a second input argument with the value false, if A has m rows and n columns with m>n, qr produces an m-by-n Q and an n-by-n R. Bottom rows of zeros of R, and the corresponding columns of Q, are discarded. With column pivoting, the third output argument e is a permutation vector: A(:,e)=Q∗R.

Examples

```
(Q,R) = qr([1,2;3,4;5,6])
Q =
   -0.169     0.8971   0.4082
   -0.5071    0.276   -0.8165
   -0.8452   -0.345    0.4082
R =
   -5.9161   -7.4374
         0    0.8281
         0         0
(Q,R) = qr([1,2;3,4;5,6],false)
```

```
  Q =
    0.169     0.8971
    0.5071    0.276
    0.8452   -0.345
  R =
    5.9161    7.4374
    0         0.8281
```

See also

`lu`, `schur`, `hess`, `svd`

rank

Rank of a matrix.

Syntax

```
x = rank(M)
x = rank(M, tol)
x = rank(M, tol=tol)
```

Description

`rank(M)` returns the rank of matrix `M`, i.e. the number of lines or columns linearly independent. To obtain it, the singular values are computed and the number of values significantly larger than 0 is counted. The value below which they are considered to be 0 can be specified with the optional second argument or named argument.

Examples

```
rank([1,1;0,0])
 1
rank([1,1;0,1j])
 2
```

See also

`svd`, `cond`, `pinv`, `det`

roots

Roots of a polynomial.

Syntax

```
r = roots(pol)
r = roots(M)
r = roots(M,dim)
```

Description

`roots(pol)` calculates the roots of the polynomial `pol`. The polynomial is given by the vector of its coefficients, highest power first, while the result is a column vector.

With a matrix as argument, `roots(M)` calculates the roots of the polynomials corresponding to each column of `M`. An optional second argument is used to specify in which dimension `roots` operates (1 for columns, 2 for rows). The roots of the i:th polynomial are in the i:th column of the result, whatever the value of `dim` is.

Examples

```
roots([1, 0, -1])
  1
 -1
roots([1, 0, -1]')
  1
 -1
roots([1, 1; 0, 5; -1, 6])
  1 -2
 -1 -3
roots([1, 0, -1]', 2)
  []
```

See also

`poly`, `eig`

schur

Schur factorization.

Syntax

```
(U,T) = schur(A)
T = schur(A)
(U,T) = schur(A, 'c')
T = schur(A, 'c')
```

Description

`schur(A)` computes the Schur factorization of square matrix `A`, i.e. a unitary matrix `U` and a square matrix `T` (the *Schur matrix*) such that `A=U*T*U'`. If `A` is complex, the Schur matrix is upper triangular, and its diagonal contains the eigenvalues of `A`; if `A` is real, the Schur matrix is real upper triangular, except that there may be 2-by-2 blocks on the main diagonal which correspond to the complex eigenvalues of `A`. To force a complex Schur factorization with an upper triangular matrix `T`, `schur` is given a second input argument `'c'` or `'complex'`.

Examples

Schur factorization:

```
A = [1,2;3,4];
(U,T) = schur(A)
  U =
   -0.8246   -0.5658
    0.5658   -0.8246
  T =
   -0.3723   -1
         0    5.3723
```

Since T is upper triangular, its diagonal contains the eigenvalues of A:

```
eig(A)
  ans =
   -0.3723
    5.3723
```

For a matrix with complex eigenvalues, the real Schur factorization has 2x2 blocks on its diagonal:

```
T = schur([1,0,0;0,1,2;0,-3,1])
  T =
     1     0     0
     0     1     2
     0    -3     1
T = schur([1,0,0;0,1,2;0,-3,1],'c')
  T =
     1             0             0
     0             1 + 2.4495j   1
     0             0             1 - 2.4495j
```

See also

`lu`, `hess`, `qr`, `eig`

skewness

Skewness of a set of values.

Syntax

```
s = skewness(A)
s = skewness(A, dim)
```

Description

`skewness(A)` gives the skewness of the columns of array `A` or of the row vector `A`. The dimension along which `skewness` proceeds may be specified with a second argument.

The skewness measures how asymmetric a distribution is. It is 0 for a symmetric distribution, and positive for a distribution which has more values much larger than the mean.

Example

```
skewness(randn(1, 10000).^2)
  2.6833
```

See also

`mean`, `var`, `kurtosis`, `moment`

sqrtm

Matrix square root.

Syntax

```
Y = sqrtm(X)
(Y, err) = sqrtm(X)
```

Description

`sqrtm(X)` returns the matrix square root of `X`, such that `sqrtm(X)^2=X`. `X` must be square. The matrix square root does not always exist.

With a second output argument `err`, `sqrtm` also returns an estimate of the relative error `norm(sqrtm(X)^2-X)/norm(X)`.

Example

```
Y = sqrtm([1,2;3,4])
  Y =
    0.5537 + 0.4644j   0.807 - 0.2124j
    1.2104 - 0.3186j  1.7641 + 0.1458j
Y^2
    1   2
    3   4
```

See also

`expm`, `logm`, `funm`, `schur`, `chol`, `sqrt`

std

Standard deviation.

Syntax

```
x = std(v)
x = std(v, p)
v = std(M)
v = std(M, p)
v = std(M, p, dim)
```

Description

`std(v)` gives the standard deviation of vector `v`, normalized by `length(v)-1`. With a second argument, `std(v,p)` normalizes by `length(v)-1` if `p` is false, or by `length(v)` if `p` is true.

`std(M)` gives a row vector which contains the standard deviation of the columns of `M`. With a third argument, `std(M,p,dim)` operates along dimension `dim`.

Example

```
std([1, 2, 5, 6, 10, 12])
  4.3359
```

See also

`mean`, `var`, `cov`

sum

Sum of the elements of a vector.

Syntax

```
x = sum(v)
v = sum(M)
v = sum(M,dim)
```

Description

`sum(v)` returns the sum of the elements of vector `v`. `sum(M)` returns a row vector whose elements are the sums of the corresponding columns of matrix `M`. `sum(M,dim)` returns the sum of matrix `M` along dimension `dim`; the result is a row vector if `dim` is 1, or a column vector if `dim` is 2.

Examples

```
sum(1:5)
  15
sum((1:5)')
  15
```

```
sum([1,2,3;5,6,7])
  6 8 10
sum([1,2,3;5,6,7],1)
  6 8 10
sum([1,2,3;5,6,7],2)
  6
  18
```

See also

`prod`, `mean`, operator +

svd

Singular value decomposition.

Syntax

```
s = svd(M)
(U,S,V) = svd(M)
(U,S,V) = svd(M,false)
```

Description

The singular value decomposition `(U,S,V) = svd(M)` decomposes the m-by-n matrix `M` such that `M = U*S*V'`, where `S` is an m-by-n diagonal matrix with decreasing positive diagonal elements (the singular values of `M`), `U` is an m-by-m unitary matrix, and `V` is an n-by-n unitary matrix. The number of non-zero diagonal elements of `S` (up to rounding errors) gives the rank of `M`.

When `M` is rectangular, in expression `U*S*V'`, some columns of `U` or `V` are multiplied by rows or columns of zeros in `S`, respectively. `(U,S,V) = svd(M,false)` produces `U`, `S` and `V` where these columns or rows are discarded (relationship `M = U*S*V'` still holds):

Size of `A`	**Size of** `U`	**Size of** `S`	**Size of** `V`
m by n, m <= n	m by m	m by m	n by m
m by n, m > n	m by n	n by n	n by n

`svd(M,true)` produces the same result as `svd(M)`.

With one output argument, `s = svd(M)` returns the vector of singular values `s=diag(S)`.

The singular values of `M` can also be computed with `s = sqrt(eig(M'*M))`, but `svd` is faster and more robust.

Examples

```
(U,S,V)=svd([1,2;3,4])
 U =
```

```
   0.4046 0.9145
   0.9145 -0.4046
  S =
   5.465 0
   0 0.366
  V =
   0.576 -0.8174
   0.8174 0.576
 U*S*V'
   1 2
   3 4
 svd([1,2;1,2])
   3.1623
   3.4697e-19
```

See also

`eig`, `pinv`, `rank`, `cond`, `norm`

trace

Trace of a matrix.

Syntax

```
tr = trace(M)
```

Description

`trace(M)` returns the trace of the matrix `M`, i.e. the sum of its diagonal elements.

Example

```
trace([1,2;3,4])
  5
```

See also

`norm`, `diag`

var

Variance of a set of values.

Syntax

```
s2 = var(A)
s2 = var(A, p)
s2 = var(A, p, dim)
```

Description

`var(A)` gives the variance of the columns of array `A` or of the row vector `A`. The variance is normalized with the number of observations minus 1, or by the number of observations if a second argument is true. The dimension along which `var` proceeds may be specified with a third argument.

See also

`mean`, `std`, `cov`, `kurtosis`, `skewness`, `moment`

10.22 Array Functions

arrayfun

Function evaluation for each element of an array.

Syntax

```
(B1, ...) = arrayfun(fun, A1, ...)
```

Description

`arrayfun(fun,A)` evaluates function `fun` for each element of numeric array `A`. Each evaluation must give a scalar result of numeric (or logical or char) type; results are returned as a numeric array the same size as `A`. First argument is a function reference, an inline function, or the name of a function as a string.

With more than two input arguments, `arrayfun` calls function `fun` as `feval(fun,A1(i),A2(i),...)`. All array arguments must have the same size, but their type can be different.

With two output arguments or more, `arrayfun` evaluates function `fun` with the same number of output arguments and builds a separate array for each output. Without output argument, `arrayfun` evaluates `fun` without output argument.

`arrayfun` differs from `cellfun`: all input arguments of `arrayfun` are arrays of any type (not necessarily cell arrays), and corresponding elements are provided provided to `fun`. With `map`, input arguments as well as output arguments are cell arrays.

Examples

```
arrayfun(@isempty, {1, ''; {}, ones(5)})
  F T
  T F
map(@isempty, {1, ''; {}, ones(5)})
  2x2 cell array
```

```
(m, n) = arrayfun(@size, {1, ''; {}, ones(2, 5)})
  m =
    1  0
    0  2
  n =
    1  0
    0  5
```

See also

`cellfun`, `map`, `fevalx`

cat

Array concatenation.

Syntax

```
cat(dim, A1, A2, ...)
```

Description

`cat(dim,A1,A2,...)` concatenates arrays `A1`, `A2`, etc. along dimension `dim`. Other dimensions must match. `cat` is a generalization of the comma and the semicolon inside brackets.

Examples

```
cat(2, [1,2;3,4], [5,6;7,8])
  1  2  5  6
  3  4  7  8
cat(3, [1,2;3,4], [5,6;7,8])
  2x2x2 array
    (:,:,1) =
      1  2
      3  4
    (:,:,2) =
      5  6
      7  8
```

See also

operator `[]`, operator `;`, operator `,`

cell

Cell array of empty arrays.

Syntax

```
C = cell(n)
C = cell(n1,n2,...)
C = cell([n1,n2,...])
```

Description

`cell` builds a cell array whose elements are empty arrays `[]`. The size of the cell array is specified by one integer for a square array, or several integers (either as separate arguments or in a vector) for a cell array of any size.

Example

```
cell(2, 3)
  2x3 cell array
```

See also

`zeros`, operator `{}`, `iscell`

cellfun

Function evaluation for each cell of a cell array.

Syntax

```
A = cellfun(fun, C)
A = cellfun(fun, C, ...)
A = cellfun(fun, S)
A = cellfun(fun, S, ...)
```

Description

`cellfun(fun,C)` evaluates function `fun` for each cell of cell array `C`. Each evaluation must give a scalar result of numeric, logical, or character type; results are returned as a non-cell array the same size as `C`. First argument is a function reference, an inline function, or the name of a function as a string.

With more than two input arguments, `cellfun` calls function `fun` as `feval(fun,C{i},other)`, where `C{i}` is each cell of `C` in turn, and `other` stands for the remaining arguments of `cellfun`.

The second argument can be a structure array `S` instead of a cell array. In that case, `fun` is called with `S(i)`.

`cellfun` differs from `map` in two ways: the result is a non-cell array, and remaining arguments of `cellfun` are provided directly to `fun`.

Examples

```
cellfun(@isempty, {1, ''; {}, ones(5)})
  F T
  T F
map(@isempty, {1, ''; {}, ones(5)})
  2x2 cell array
cellfun(@size, {1, ''; {}, ones(5)}, 2)
  1 0
  0 5
```

See also

`map`, `arrayfun`

diag

Creation of a diagonal matrix or extraction of the diagonal elements of a matrix.

Syntax

```
M = diag(v)
M = diag(v,k)
v = diag(M)
v = diag(M,k)
```

Description

With a vector input argument, `diag(v)` creates a square diagonal matrix whose main diagonal is given by `v`. With a second argument, the diagonal is moved by that amount in the upper right direction for positive values, and in the lower left direction for negative values.

With a matrix input argument, the main diagonal is extracted and returned as a column vector. A second argument can be used to specify another diagonal.

Examples

```
diag(1:3)
  1 0 0
  0 2 0
  0 0 3
diag(1:3,1)
  0 1 0 0
  0 0 2 0
  0 0 0 3
  0 0 0 0
M = magic(3)
M =
```

```
  8 1 6
  3 5 7
  4 9 2
diag(M)
  8
  5
  2
diag(M,1)
  1
  7
```

See also

`tril`, `triu`, `eye`, `trace`

eye

Identity matrix.

Syntax

```
M = eye(n)
M = eye(m,n)
M = eye([m,n])
M = eye(..., type)
```

Description

eye builds a matrix whose diagonal elements are 1 and other elements 0. The size of the matrix is specified by one integer for a square matrix, or two integers (either as two arguments or in a vector of two elements) for a rectangular matrix.

An additional input argument can be used to specify the type of the result. It must be the string `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` (64-bit arrays are not supported on all platforms).

Examples

```
eye(3)
  1 0 0
  0 1 0
  0 0 1
eye(2, 3)
  1 0 0
  0 1 0
eye(2, 'int8')
  2x2 int8 array
    1 0
    0 1
```

See also

`ones`, `zeros`, `diag`

fevalx

Function evaluation with array expansion.

Syntax

```
(Y1,...) = fevalx(fun,X1,...)
```

Description

`(Y1,Y2,...)=fevalx(fun,X1,X2,...)` evaluates function `fun` with input arguments X1, X2, etc. Arguments must be arrays, which are expanded if necessary along singleton dimensions so that all dimensions match. For instance, three arguments of size 3x1x2, 1x5 and 1x1 are replicated into arrays of size 3x5x2. Output arguments are assigned to Y1, Y2, etc. Function `fun` is specified either by its name as a string, by a function reference, or by an inline or anonymous function.

Example

```
fevalx(@plus, 1:5, (10:10:30)')
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
```

See also

`feval`, `meshgrid`, `repmat`, `inline`, operator @

find

Find the indices of the non-null elements of an array.

Syntax

```
ix = find(v)
[s1,s2] = find(M)
[s1,s2,x] = find(M)
... = find(..., n)
... = find(..., n, dir)
```

Description

With one output argument, `find(v)` returns a vector containing the indices of the nonzero elements of `v`. `v` can be an array of any dimension; the indices correspond to the internal storage ordering and can be used to access the elements with a single subscript.

With two output arguments, `find(M)` returns two vectors containing the subscripts (row in the first output argument, column in the second output argument) of the nonzero elements of 2-dim array `M`. To obtain subscripts for an array of higher dimension, you can convert the single output argument of `find` to subscripts with `ind2sub`.

With three output arguments, `find(M)` returns in addition the nonzero values themselves in the third output argument.

With a second input argument `n`, `find` limits the maximum number of elements found. It searches forward by default; with a third input argument `dir`, `find` gives the `n` first nonzero values if `dir` is `'first'` or `'f'`, and the `n` last nonzero values if `dir` is `'last'` or `'l'`.

Examples

```
ix = find([1.2,0;0,3.6])
ix =
  1
  4
[s1,s2] = find([1.2,0;0,3.6])
s1 =
  1
  2
s2 =
  1
  2
[s1,s2,x] = find([1.2,0;0,3.6])
s1 =
  1
  2
s2 =
  1
  2
x =
  1.2
  3.6
A = rand(3)
  A =
    0.5599    0.3074    0.5275
    0.3309    0.8077    0.3666
    0.7981    0.6424    0.6023
find(A > 0.7, 2, 'last')
  7
  5
```

See also

`nnz`, `sort`

flipdim

Flip an array along any dimension.

Syntax

```
B = flipdim(A, dim)
```

Description

`flipdim(A,dim)` gives an array which has the same size as `A`, but where indices of dimension `dim` are reversed.

Examples

```
flipdim(cat(3, [1,2;3,4], [5,6;7,8]), 3)
  2x2x2 array
    (:,:,1) =
      5  6
      7  8
  (:,:,2) =
      1  2
      3  4
```

See also

`fliplr`, `flipud`, `rot90`, `reshape`

fliplr

Flip an array or a list around its vertical axis.

Syntax

```
A2 = fliplr(A1)
list2 = fliplr(list1)
```

Description

`fliplr(A1)` gives an array `A2` which has the same size as `A1`, but where all columns are placed in reverse order.

`fliplr(list1)` gives a list `list2` which has the same length as `list1`, but where all top-level elements are placed in reverse order. Elements themselves are left unchanged.

Examples

```
fliplr([1,2;3,4])
  2 1
  4 3
fliplr({1, 'x', {1,2,3}})
  {{1,2,3}, 'x', 1}
```

See also

`flipud`, `flipdim`, `rot90`, `reshape`

flipud

Flip an array upside-down.

Syntax

```
A2 = flipud(A1)
```

Description

`flipud(A1)` gives an array `A2` which has the same size as `A1`, but where all lines are placed in reverse order.

Example

```
flipud([1,2;3,4])
  3 4
  1 2
```

See also

`fliplr`, `flipdim`, `rot90`, `reshape`

ind2sub

Conversion from single index to row/column subscripts.

Syntax

```
(i, j, ...) = ind2sub(size, ind)
```

Description

`ind2sub(size,ind)` gives the subscripts of the element which would be retrieved from an array whose size is specified by `size` by the single index `ind`. `size` must be either a scalar for square matrices or a vector of two elements or more for arrays. `ind` can be an array; the result is calculated separately for each element and has the same size.

Example

```
M = [3, 6; 8, 9];
M(3)
  8
(i, j) = ind2sub(size(M), 3)
  i =
    2
  j =
    1
M(i, j)
  8
```

See also

`sub2ind`, `size`

interp1

1D interpolation.

Syntax

```
yi = interp1(x, y, xi)
yi = interp1(x, y, xi, method)
yi = interp1(y, xi)
yi = interp1(y, xi, method)
yi = interp1(..., method, extraval)
```

Description

`interp1(x,y,xi)` interpolates data along one dimension. Input data are defined by vector `y`, where element `y(i)` corresponds to coordinates `x(i)`. Interpolation is performed at points defined in vector `xi`; the result is a vector of the same size as `xi`.

If `y` is an array, interpolation is performed along dimension 1 (i.e. along its columns), and `size(y,1)` must be equal to `length(x)`. Then if `xi` is a vector, interpolation is performed at the same points for each remaining dimensions of `y`, and the result is an array of size `[length(xi),size(y)(2:end)]`; if `xi` is an array, all sizes must match `y` except for the first one.

If `x` is missing, it defaults to `1:size(y,1)`.

The default interpolation method is piecewise linear. An additional input argument can be provided to specify it with a string (only the first character is considered):

Argument	**Meaning**
'0' or 'nearest'	nearest value
'<'	lower coordinate
'>'	higher coordinate
'1' or 'linear'	piecewise linear
'3' or 'cubic'	piecewise cubic
'p' or 'pchip'	pchip

Cubic interpolation gives continuous values and first derivatives, and null second derivatives at end points. Pchip (piecewise cubic Hermite interpolation) also gives continuous values and first derivatives, but guarantees that the interpolant stays within the limits of the data in each interval (in particular monotonicity is preserved) at the cost of larger second derivatives.

With vectors, `interp1` produces the same result as `interpn`; vector orientations do not have to match, though.

When the method is followed by a scalar number `extraval`, that value is assigned to all values outside the range defined by `x` (i.e. extrapolated values). The default is NaN.

Examples

One-dimension interpolation:

```
interp1([1, 2, 5, 8], [0.1, 0.2, 0.5, 1], [0, 2, 3, 7])
  nan   0.2000   0.3000   0.8333
interp1([1, 2, 5, 8], [0.1, 0.2, 0.5, 1], [0, 2, 3, 7], '0')
  nan   0.2000   0.2000   1.0000
```

Interpolation of multiple values:

```
t = 0:10;
y = [sin(t'), cos(t')];
tnew = 0:0.4:8;
ynew = interp1(t, y, tnew)
  ynew =
    0.0000  1.0000
    0.3366  0.8161
    ...
    0.8564  0.2143
    0.9894 -0.1455
```

See also

`interpn`

interpn

Multidimensional interpolation.

Syntax

```
Vi = interpn(x1, ..., xn, V, xi1, ..., xin)
Vi = interpn(x1, ..., xn, V, xi1, ..., xin, method)
Vi = interpn(..., method, extraval)
```

Description

`interpn(x1,...,xn,V,xi1,...,xin)` interpolates data in a space of `n` dimensions. Input data are defined by array `V`, where element `V(i,j,...)` corresponds to coordinates `x1(i)`, `x2(j)`, etc. Interpolation is performed for each coordinates defined by arrays `xi1`, `xi2`, etc., which must all have the same size; the result is an array of the same size.

Length of vectors x1, x2, ... must match the size of `V` along the corresponding dimension. Vectors x1, x2, ... must be sorted (monotonically increasing or decreasing), but they do not have to be spaced uniformly. Interpolated points outside the input volume are set to nan. Input and output data can be complex. Imaginary parts of coordinates are ignored.

The default interpolation method is multilinear. An additional input argument can be provided to specify it with a string (only the first character is considered):

Argument	**Meaning**
`'0'` or `'nearest'`	nearest value
`'<'`	lower coordinates
`'>'`	higher coordinates
`'1'` or `'linear'`	multilinear

Method `'<'` takes the sample where each coordinate has its index as large as possible, lower or equal to the interpolated value, and smaller than the last coordinate. Method `'>'` takes the sample where each coordinate has its index greater or equal to the interpolated value.

When the method is followed by a scalar number `extraval`, that value is assigned to all values outside the input volume (i.e. extrapolated values). The default is NaN.

Examples

One-dimension interpolation:

```
interpn([1, 2, 5, 8], [0.1, 0.2, 0.5, 1], [0, 2, 3, 7])
  nan   0.2000   0.3000   0.8333
interpn([1, 2, 5, 8], [0.1, 0.2, 0.5, 1], [0, 2, 3, 7], '0')
  nan   0.2000   0.2000   1.0000
```

Three-dimension interpolation:

```
D = cat(3,[0,1;2,3],[4,5;6,7]);
interpn([0,1], [0,1], [0,1], D, 0.2, 0.7, 0.5)
  3.1000
```

Image rotation (we define original coordinates between -0.5 and 0.5 in vector c and arrays X and Y, and the image as a linear gradient between 0 and 1):

```
c = -0.5:0.01:0.5;
X = repmat(c, 101, 1);
Y = X';
phi = 0.2;
Xi = cos(phi) * X - sin(phi) * Y;
Yi = sin(phi) * X + cos(phi) * Y;
D = 0.5 + X;
E = interpn(c, c, D, Xi, Yi);
E(isnan(E)) = 0.5;
```

See also

`interp1`

intersect

Set intersection.

Syntax

```
c = intersect(a, b)
(c, ia, ib) = intersect(a, b)
```

Description

`intersect(a,b)` gives the intersection of sets a and b, i.e. it gives the set of members of both sets a and b. Sets are any type of numeric, character or logical arrays, or lists or cell arrays of character strings. Multiple elements of input arguments are considered as single members; the result is always sorted and has unique elements.

The optional second and third output arguments are vectors of indices such that if `(c,ia,ib)=intersect(a,b)`, then c is `a(ia)` as well as `b(ib)`.

Example

```
a = {'a','bc','bbb','de'};
b = {'z','bc','aa','bbb'};
(c, ia, ib) = intersect(a, b)
  c =
    {'bbb','bc'}
```

```
  ia =
    3 2
  ib =
    4 2
a(ia)
  {'bbb','bc'}
b(ib)
  {'bbb','bc'}
```

Set exclusive or can also be computed as the union of a and b minus the intersection of a and b:

```
setdiff(union(a, b), intersect(a, b))
  {'a','aa','de','z'}
```

See also

`unique`, `union`, `setdiff`, `setxor`, `ismember`

inthist

Histogram of an integer array.

Syntax

```
h = inthist(A, n)
```

Description

`inthist(A,n)` computes the histogram of the elements of integer array A between 0 and n-1. A must have an integer type (int8/16/32/64 or uint8/16/32/64). The result is a row vector h of length n, where `h(i)` is the number of elements in A with value `i-1`.

Example

```
A = map2int(rand(100), 0, 1, 'uint8');
h = inthist(A, 10)
  h =
    37 31 34 34 32 35 38 36 36 32
```

See also

`hist`

ipermute

Inverse permutation of the dimensions of an array.

Syntax

```
B = ipermute(A, perm)
```

Description

`ipermute(A,perm)` returns an array with the same elements as `A`, but where dimensions are permuted according to the vector of dimensions `perm`. It performs the inverse permutation of `permute`. `perm` must contain integers from `1` to `n`; dimension `i` in `A` becomes dimension `perm(i)` in the result.

Example

```
size(ipermute(rand(3,4,5), [2,3,1]))
  5 3 4
```

See also

`permute`, `ndims`, `squeeze`

isempty

Test for empty array, list or struct.

Syntax

```
b = isempty(A)
b = isempty(list)
b = isempty(S)
```

Description

`isempty(obj)` gives true if `obj` is the empty array `[]` of any type (numeric, char, logical or cell array) or the empty struct, and false otherwise.

Examples

```
isempty([])
  true
isempty(0)
  false
isempty('')
  true
isempty({})
  true
isempty({{}})
  false
isempty(struct)
  true
```

See also

`size`, `length`

iscell

Test for cell arrays.

Syntax

```
b = iscell(X)
```

Description

`iscell(X)` gives true if X is a cell array or a list, and false otherwise.

Examples

```
iscell({1;2})
  true
iscell({1,2})
  true
islist({1;2})
  false
```

See also

`islist`

ismember

Test for set membership.

Syntax

```
b = ismember(m, s)
(b, ix) = ismember(m, s)
```

Description

`ismember(m,s)` tests if elements of array `m` are members of set `s`. The result is a logical array the same size as `m`; each element is `true` if the corresponding element of `m` is a member of `s`, or false otherwise. `m` must be a numeric array or a cell array, matching type of set `s`.

With a second output argument `ix`, `ismember` also gives the index of the corresponding element of `m` in `s`, or 0 if the element is not a member of `s`.

Example

```
s = {'a','bc','bbb','de'};
m = {'d','a','x';'de','a','z'};
(b, ix) = ismember(m, s)
  b =
    F T F
    T T F
  ix =
    0 1 0
 4 1 0
```

See also

`intersect`, `union`, `setdiff`, `setxor`

length

Length of a vector or a list.

Syntax

```
n = length(v)
n = length(list)
```

Description

`length(v)` gives the length of vector `v`. `length(A)` gives the number of elements along the largest dimension of array `A`. `length(list)` gives the number of elements in a list.

Examples

```
length(1:5)
  5
length((1:5)')
  5
length(ones(2,3))
  3
length({1, 1:6, 'abc'})
  3
length({{}})
  1
```

See also

`size`, `numel`, `end`

linspace

Sequence of linearly-spaced elements.

Syntax

```
v = linspace(x1, x2)
v = linspace(x1, x2, n)
```

Description

`linspace(x1,x2)` produces a row vector of 50 values spaced linearly from x1 to x2 inclusive. With a third argument, `linspace(x1,x2,n)` produces a row vector of n values.

Examples

```
linspace(1,10)
  1.0000 1.1837 1.3673 ... 9.8163 10.0000
linspace(1,2,6)
  1.0  1.2  1.4  1.6  1.8  2.0
```

See also

`logspace`, operator `:`

logspace

Sequence of logarithmically-spaced elements.

Syntax

```
v = logspace(x1, x2)
v = logspace(x1, x2, n)
```

Description

`logspace(x1,x2)` produces a row vector of 50 values spaced logarithmically from 10^x1 to 10^x2 inclusive. With a third argument, `logspace(x1,x2,n)` produces a row vector of n values.

Example

```
logspace(0,1)
  1.0000 1.0481 1.0985 ... 9.1030 9.5410 10.0000
```

See also

`linspace`, operator `:`

magic

Magic square.

Syntax

```
M = magic(n)
```

Description

A magic square is a square array of size n-by-n which contains each integer between 1 and n^2, and whose sum of each column and of each line is equal. `magic(n)` returns magic square of size n-by-n.

There is no 2-by-2 magic square. If the size is 2, the matrix [1,3;4,2] is returned instead.

Example

```
magic(3)
  8 1 6
  3 5 7
  4 9 2
```

See also

`zeros`, `ones`, `eye`, `rand`, `randn`

meshgrid

Arrays of X-Y coordinates.

Syntax

```
(X, Y) = meshgrid(x, y)
(X, Y) = meshgrid(x)
```

Description

`meshgrid(x,y)` produces two arrays of x and y coordinates suitable for the evaluation of a function of two variables. The input argument x is copied to the rows of the first output argument, and the input argument y is copied to the columns of the second output argument, so that both arrays have the same size. `meshgrid(x)` is equivalent to `meshgrid(x,x)`.

Example

```
(X, Y) = meshgrid(1:5, 2:4)
  X =
    1  2  3  4  5
    1  2  3  4  5
    1  2  3  4  5
  Y =
    2  2  2  2  2
```

```
     3  3  3  3  3
     4  4  4  4  4
 Z = atan2(X, Y)
   Z =
     0.4636    0.7854    0.9828    1.1071    1.1903
     0.3218    0.5880    0.7854    0.9273    1.0304
     0.2450    0.4636    0.6435    0.7854    0.8961
```

See also

`ndgrid`, `repmat`

ndgrid

Arrays of N-dimension coordinates.

Syntax

```
(X1, ..., Xn) = ndgrid(x1, ..., xn)
(X1, ..., Xn) = ndgrid(x)
```

Description

`ndgrid(x1,...,xn)` produces `n` arrays of `n` dimensions. Array `i` is obtained by reshaping input argument `i` as a vector along dimension `i` and replicating it along all other dimensions to match the length of other input vectors. All output arguments have the same size.

With one input argument, `ndgrid` reuses it to match the number of output arguments.

`(Y,X)=ndgrid(y,x)` is equivalent to `(X,Y)=meshgrid(x,y)`.

Example

```
(X1, X2) = ndgrid(1:3)
  X1 =
    1   1   1
    2   2   2
    3   3   3
  X2 =
    1   2   3
    1   2   3
    1   2   3
```

See also

`meshgrid`, `repmat`

ndims

Number of dimensions of an array.

Syntax

```
n = ndims(A)
```

Description

`ndims(A)` returns the number of dimensions of array `A`, which is at least 2. Scalars, row and column vectors, and matrices have 2 dimensions.

Examples

```
ndims(magic(3))
  2
ndims(rand(3,4,5))
  3
```

See also

`size`, `squeeze`, `permute`, `ipermute`

nnz

Number of nonzero elements.

Syntax

```
n = nnz(A)
```

Description

`nnz(A)` returns the number of nonzero elements of array `A`. Argument `A` must be a numeric, char or logical array.

Examples

```
nnz(-2:2)
  4
nnz(magic(3) > 3)
  6
```

See also

`find`

num2cell

Conversion from numeric array to cell array.

Syntax

```
C = num2cell(A)
C = num2cell(A, dims)
```

Description

`num2cell(A)` creates a cell array the same size as numeric array `A`. The value of each cell is the corresponding elements of `A`.

`num2cell(A,dims)` cuts array `A` along the dimensions *not* in `dims` and creates a cell array with the result. Dimensions of cell array are the same as dimensions of `A` for dimensions not in `dims`, and 1 for dimensions in `dims`; dimensions of cells are the same as dimensions of `A` for dimensions in `dims`, and 1 for dimensions not in `dims`.

Argument `A` can be a numeric array of any dimension and class, a logical array, or a char array.

Examples

```
num2cell([1, 2; 3, 4])
  {1, 2; 3, 4}
num2cell([1, 2; 3, 4], 1)
  {[1; 3], [2; 4]}
num2cell([1, 2; 3, 4], 2)
  {[1, 2]; [3, 4]}
```

See also

`num2list`, `permute`

numel

Number of elements of an array.

Syntax

```
n = numel(A)
```

Description

`numel(A)` gives the number of elements of array `A`. It is equivalent to `prod(size(A))`.

Examples

```
numel(1:5)
  5
numel(ones(2, 3))
  6
numel({1, 1:6; 'abc', []})
  4
numel({2, 'vwxyz'})
  2
```

See also

`size`, `length`

ones

Array of ones.

Syntax

```
A = ones(n)
A = ones(n1, n2, ...)
A = ones([n1, n2, ...])
A = ones(..., type)
```

Description

ones builds an array whose elements are 1. The size of the array is specified by one integer for a square matrix, or several integers (either as separate arguments or in a vector) for an array of any size.

An additional input argument can be used to specify the type of the result. It must be the string `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` (64-bit arrays are not supported on all platforms).

Examples

```
ones(2,3)
  1 1 1
  1 1 1
ones(2, 'int32')
  2x2 int32 array
    1 1
    1 1
```

See also

`zeros`, `eye`, `rand`, `randn`, `repmat`

permute

Permutation of the dimensions of an array.

Syntax

```
B = permute(A, perm)
```

Description

`permute(A,perm)` returns an array with the same elements as `A`, but where dimensions are permuted according to the vector of dimensions `perm`. It is a generalization of the matrix transpose operator. `perm` must contain integers from 1 to `n`; dimension `perm(i)` in `A` becomes dimension `i` in the result.

Example

```
size(permute(rand(3,4,5), [2,3,1]))
  4 5 3
```

See also

`ndims`, `squeeze`, `ipermute`, `num2cell`

rand

Uniformly-distributed random number.

Syntax

```
x = rand
A = rand(n)
A = rand(n1, n2, ...)
A = rand([n1, n2, ...])
A = rand(..., type)
rand('seed', s);
```

Description

`rand` builds a scalar pseudo-random number uniformly distributed between 0 and 1. The lower bound 0 may be reached, but the upper bound 1 is never. The default generator is based on a scalar 64-bit seed, which theoretically has a period of 2^64-2^32 numbers. This seed can be set with the arguments `rand('seed',s)`, where `s` is a scalar. `rand('seed',s)` returns the empty array [] as output argument. To discard it, the statement should be followed by a semicolon. The generator can be changed with `rng`.

`rand(n)`, `rand(n1,n2,...)` and `rand([n1,n2,...])` return an n-by-n square array or an array of arbitrary size whose elements are pseudo-random numbers uniformly distributed between 0 and 1.

An additional input argument can be used to specify the type of the result, `'double'` (default) or `'single'`. With the special value `'raw'`, `rand` returns an unscaled integer result of type double which corresponds to the uniform output of the random generator before it is mapped to the range between 0 and 1. The scaling factor can be retrieved in the field `rawmax` of the structure returned by `rng`.

Examples

```
rand
  0.2361
rand(1, 3)
  0.6679 0.8195 0.2786
rand('seed',0);
rand
  0.2361
```

See also

`randn`, `randi`, `rng`

randi

Uniformly-distributed integer random number.

Syntax

```
x = randi(nmax)
x = randi(range)
M = randi(..., n)
M = randi(..., n1, n2, ...)
M = randi(..., [n1, n2, ...])
M = randi(..., class)
```

Description

`randi(nmax)` produces a scalar pseudo-random integer number uniformly distributed between 1 and `nmax`. `randi(range)`, where `range` is a two-element vector `[nmin,nmax]`, produces a scalar pseudo-random integer number uniformly distributed between `nmin` and `nmax`.

With more numeric input arguments, `randi` produces arrays of pseudo-random integer numbers. `randi(range,n)` produces an n-by-n square array, and `randi(range,[n1,n2,...])` or `randi(range,n1,n2,...)` produces an array of the specified size.

The number class of the result can be specified with a final string argument. The default is `'double'`.

Examples

```
randi(10)
  3
randi(10, [1, 5])
  3 4 6 8 1
randi([10,15], [1, 5])
  12 14 13 10 13
randi(8, [1, 5], 'uint8')
  1x5 uint8 array
    3 4 5 7 2
```

See also

rand, randn, rng

randn

Normally-distributed random number

Syntax

```
x = randn
A = randn(n)
A = randn(n1, n2, ...)
A = randn([n1, n2, ...])
A = randn(..., type)
randn('seed', s);
```

Description

randn builds a scalar pseudo-random number chosen from a normal distribution with zero mean and unit variance. The default generator is based on a scalar 64-bit seed, which theoretically has a period of 2^64-2^32 numbers. This seed can be set with the arguments randn('seed',s), where s is a scalar. The seed is the same as the seed of rand and rng. randn('seed',s) returns the empty array [] as output argument. To discard it, the statement should be followed by a semicolon. The generator can be changed with rng.

randn(n), randn(n1,n2,...) and randn([n1,n2,...]) return an n-by-n square array or an array of arbitrary size whose elements are pseudo-random numbers chosen from a normal distribution.

An additional input argument can be used to specify the type of the result. It must be the string 'double' (default) or 'single'.

Examples

```
randn
  1.5927
randn(1, 3)
  0.7856 0.6489 -0.8141
randn('seed',0);
randn
  1.5927
```

See also

rand, randi, rng

repmat

Replicate an array.

Syntax

```
B = repmat(A, n)
B = repmat(A, m, n)
B = repmat(A, [n1,...])
```

Description

`repmat` creates an array with multiple copies of its first argument. It can be seen as an extended version of `ones`, where 1 is replaced by an arbitrary array.

With 3 input arguments, `repmat(A,m,n)` replicates array `A` `m` times vertically and `n` times horizontally. The type of the first argument (number, character, logical, cell, or structure array) is preserved.

With two input arguments, `repmat(A,n)` produces the same result as `repmat(A,n,n)`.

With a vector as second argument, the array can be replicated along more than two dimensions; `repmat(A,m,n)` produces the same result as `repmat(A,[m,n])`.

Examples

```
repmat([1,2;3,4], 1, 2)
  1 2 1 2
  3 4 3 4
repmat('abc', 3)
  abcabcabc
  abcabcabc
  abcabcabc
```

See also

`zeros`, `ones`, operator `:`, `kron`, `replist`

reshape

Rearrange the elements of an array to change its shape.

Syntax

```
A2 = reshape(A1)
A2 = reshape(A1, n1, n2, ...)
A2 = reshape(A1, [n1, n2, ...])
```

Description

`reshape(A1)` gives a column vector with all the elements of array `A1`. If `A1` is a variable, `reshape(A1)` is the same as `A1(:)`.

`reshape(A1,n1,n2,...)` or `reshape(A1,[n1,n2,...])` changes the dimensions of array `A1` so that the result has `m` rows and `n` columns.

A1 must have n1∗n2∗... elements; read row-wise, both A1 and the result have the same elements.

When dimensions are given as separate elements, one of them can be replaced with the empty array []; it is replaced by the value such that the number of elements of the result matches the size of input array.

Remark: code should not rely on the internal data layout. Array elements are currently stored row-wise, but this may change in the future. reshape will remain consistant with indexing, though; reshape(A,s)(i)==A(i) for any compatible size s.

Example

```
reshape([1,2,3;10,20,30], 3, 2)
  1  2
  3  10
  20 30
reshape(1:12, 3, [])
  1  2  3  4
  5  6  7  8
  9 10 11 12
```

See also

operator ()

rng

State of random number generator.

Syntax

```
rng(type)
rng(seed)
rng(seed, type)
rng(state)
state = rng
```

Description

Random (actually pseudo-random) number generators produce sequences of numbers whose statistics make them difficult to distinguish from true random numbers. They are used by functions rand, randi, randn and random. They are characterized by a type string and a state.

With a numeric input argument, rng(seed) sets the state based on a seed. The state is usually an array of unsigned 32-bit integer numbers. rng uses the seed to produce an internal state which is valid for the type of random number generator. The default seed is 0.

With a string input argument, `rng(type)` sets the type of the random number generator and resets the state to its initial value (default seed). The following types are recognized:

`'original'` Original generator used until LME 6.

`'mcg16807'` Multiplicative congruential generator. The state is defined by `s(i+1)=mod(a*s(i),m)` with a=7^5 and `m`=2^31-1, and the generated value is `s(i)/m`.

`'mwc'` Concatenation of two 16-bit multiply-with-carry generators. The period is about 2^60.

`'kiss'` **or** `'default'` Combination of mwc, a 3-shift register, and a congruential generator. The period is about 2^123.

With two input arguments, `rng(seed,type)` sets both the seed and the type of the random number generator.

With an output argument, `state=rng` gets the current state, which can be restored later by calling `rng(state)`. The state is a structure.

Examples

```
rng(123);
R = rand(1,2)
  R =
    0.2838    0.4196
s = rng
  s =
    type: 'original'
    state: real 2x1
    rawmax: 4294967296
R = rand
  R =
    0.5788
rng(s)
R = rand
  R =
    0.5788
```

Reference

The MWC and KISS generators are described in George Marsaglia, *Random numbers for C: The END?*, Usenet, sci.stat.math, 20 Jan 1999.

See also

`rand`, `randn`, `randi`

rot90

Array rotation.

Syntax

```
A2 = rot90(A1)
A2 = rot90(A1, k)
```

Description

`rot90(A1)` rotates array `A1` 90 degrees counter-clockwise; the top left element of `A1` becomes the bottom left element of `A2`. If `A1` is an array with more than two dimensions, each plane corresponding to the first two dimensions is rotated.

In `rot90(A1,k)`, the second argument is the number of times the array is rotated 90 degrees counter-clockwise. With `k = 2`, the array is rotated by 180 degrees; with `k = 3` or `k = -1`, the array is rotated by 90 degrees clockwise.

Examples

```
rot90([1,2,3;4,5,6])
  3 6
  2 5
  1 4
rot90([1,2,3;4,5,6], -1)
  4 1
  5 2
  6 3
rot90([1,2,3;4,5,6], -1)
  6 5 4
  3 2 1
fliplr(flipud([1,2,3;4,5,6]))
  6 5 4
  3 2 1
```

See also

`fliplr`, `flipud`, `reshape`

setdiff

Set difference.

Syntax

```
c = setdiff(a, b)
(c, ia) = setdiff(a, b)
```

Description

`setdiff(a,b)` gives the difference between sets a and b, i.e. the set of members of set a which do not belong to b. Sets are any type of numeric, character or logical arrays, or lists or cell arrays of character strings. Multiple elements of input arguments are considered as single members; the result is always sorted and has unique elements.

The optional second output argument is a vector of indices such that if `(c,ia)=setdiff(a,b)`, then c is `a(ia)`.

Example

```
a = {'a','bc','bbb','de'};
b = {'z','bc','aa','bbb'};
(c, ia) = setdiff(a, b)
  c =
    {'a','de'}
  ia =
    1 4
a(ia)
  {'a','de'}
```

See also

`unique`, `union`, `intersect`, `setxor`, `ismember`

setxor

Set exclusive or.

Syntax

```
c = setxor(a, b)
(c, ia, ib) = setxor(a, b)
```

Description

`setxor(a,b)` performs an exclusive or operation between sets a and b, i.e. it gives the set of members of sets a and b which are not members of the intersection of a and b. Sets are any type of numeric, character or logical arrays, or lists or cell arrays of character strings. Multiple elements of input arguments are considered as single members; the result is always sorted and has unique elements.

The optional second and third output arguments are vectors of indices such that if `(c,ia,ib)=setxor(a,b)`, then c is the union of `a(ia)` and `b(ib)`.

Example

```
a = {'a','bc','bbb','de'};
b = {'z','bc','aa','bbb'};
(c, ia, ib) = setxor(a, b)
  c =
    {'a','aa','de','z'}
  ia =
    1 4
  ib =
    3 1
union(a(ia),b(ib))
  {'a','aa','de','z'}
```

Set exclusive or can also be computed as the union of a and b minus the intersection of a and b:

```
setdiff(union(a, b), intersect(a, b))
  {'a','aa','de','z'}
```

See also

`unique`, `union`, `intersect`, `setdiff`, `ismember`

size

Size of an array.

Syntax

```
v = size(A)
(m, n) = size(A)
m = size(A, i)
```

Description

`size(A)` returns the number of rows and the number of elements along each dimension of array `A`, either in a row vector or as scalars if there are two output arguments or more.

`size(A,i)` gives the number of elements in array `A` along dimension `i`: `size(A,1)` gives the number of rows and `size(A,2)` the number of columns.

Examples

```
M = ones(3, 5);
size(M)
  3 5
(m, n) = size(M)
  m =
```

```
     3
   n =
     5
 size(M, 1)
   3
 size(M, 2)
   5
```

See also

`length`, `numel`, `ndims`, `end`

sort

Array sort.

Syntax

```
(A_sorted, ix) = sort(A)
(A_sorted, ix) = sort(A, dim)
(A_sorted, ix) = sort(A, dir)
(A_sorted, ix) = sort(A, dim, dir)
(list_sorted, ix) = sort(list)
(list_sorted, ix) = sort(list, dir)
```

Description

`sort(A)` sorts separately the elements of each column of array `A`, or the elements of `A` if it is a row vector. The result has the same size as `A`. Elements are sorted in ascending order, with NaNs at the end. For complex arrays, numbers are sorted by magnitude.

The optional second output argument gives the permutation array which transforms `A` into the sorted array. It can be used to reorder elements in another array or to sort the rows of a matrix with respect to one of its columns, as shown in the last example below. Order of consecutive identical elements is preserved.

If a second numeric argument `dim` is provided, the sort is performed along dimension `dim` (columns if `dim` is 1, rows if 2, etc.)

An additional argument can specify the ordering direction. It must be the string `'ascending'` (or `'a'`) for ascending order, or `'descending'` (or `'d'`) for descending order. In both cases, NaNs are moved to the end.

`sort(list)` sorts the elements of a list, which must be strings. Cell arrays are sorted like lists, not column-wise like numeric arrays. The second output argument is a row vector. The direction can be specified with a second input argument.

Examples

```
sort([3,6,2,3,9,1,2])
  1 2 2 3 3 6 9
sort([2,5,3;nan,4,2;6,1,1])
  2   1   1
  6   4   2
  nan 5   3
sort([2,5,3;nan,4,2;6,1,1], 'd')
  6   5   3
  2   4   2
  nan 1   1
sort({'def', 'abcd', 'abc'})
  {'abc', 'abcd', 'def'}
```

To sort the rows of an array after the first column, one can obtain the permutation vector by sorting the first column, and use it as subscripts on the array rows:

```
M = [2,4; 5,1; 3,9; 4,0]
  2 4
  5 1
  3 9
  4 0
(Ms, ix) = sort(M(:,1));
M(ix,:)
  2 4
  3 9
  4 0
  5 1
```

Algorithm

Shell sort.

See also

`unique`

squeeze

Suppression of singleton dimensions of an array.

Syntax

```
B = squeeze(A)
```

Description

`squeeze(A)` returns an array with the same elements as `A`, but where dimensions equal to 1 are removed. The result has at least 2 dimensions; row and column vectors keep their dimensions.

Examples

```
size(squeeze(rand(1,2,3,1,4)))
  2 3 4
size(squeeze(1:5))
  1 5
```

See also

`permute`, `ndims`

sub2ind

Conversion from row/column subscripts to single index.

Syntax

```
ind = sub2ind(size, i, j)
ind = sub2ind(size, i, j, k, ...)
```

Description

`sub2ind(size,i,j)` gives the single index which can be used to retrieve the element corresponding to the `i`:th row and the `j`:th column of an array whose size is specified by `size`. `size` must be either a scalar for square matrices or a vector of two elements or more for other arrays. If `i` and `j` are arrays, they must have the same size: the result is calculated separately for each element and has the same size.

`sub2ind` also accepts sizes and subscripts for arrays with more than 2 dimensions. The number of indices must match the length of `size`.

Example

```
M = [3, 6; 8, 9];
M(2, 1)
  8
sub2ind(size(M), 2, 1)
  7
M(3)
  8
```

See also

`ind2sub`, `size`

tril

Extraction of the lower triangular part of a matrix.

Syntax

```
L = tril(M)
L = tril(M,k)
```

Description

`tril(M)` extracts the lower triangular part of a matrix; the result is a matrix of the same size where all the elements above the main diagonal are set to zero. A second argument can be used to specify another diagonal: 0 is the main diagonal, positive values are above and negative values below.

Examples

```
M = magic(3)
M =
  8 1 6
  3 5 7
  4 9 2
tril(M)
  8 0 0
  3 5 0
  4 9 2
tril(M,1)
  8 1 0
  3 5 7
  4 9 2
```

See also

`triu`, `diag`

triu

Extraction of the upper triangular part of a matrix.

Syntax

```
U = triu(M)
U = triu(M,k)
```

Description

`tril(M)` extracts the upper triangular part of a matrix; the result is a matrix of the same size where all the elements below the main diagonal are set to zero. A second argument can be used to specify another diagonal; 0 is the main diagonal, positive values are above and negative values below.

Examples

```
M = magic(3)
M =
  8 1 6
  3 5 7
  4 9 2
triu(M)
  8 1 6
  0 5 7
  0 0 2
triu(M,1)
  0 1 6
  0 0 7
  0 0 0
```

See also

`tril`, `diag`

union

Set union.

Syntax

```
c = union(a, b)
(c, ia, ib) = union(a, b)
```

Description

`union(a,b)` gives the union of sets a and b, i.e. it gives the set of members of sets a or b or both. Sets are any type of numeric, character or logical arrays, or lists or cell arrays of character strings. Multiple elements of input arguments are considered as single members; the result is always sorted and has unique elements.

The optional second and third output arguments are vectors of indices such that if `(c,ia,ib)=union(a,b)`, then elements of c are the elements of `a(ia)` or `b(ib)`; the intersection of `a(ia)` and `b(ib)` is empty.

Example

```
a = {'a','bc','bbb','de'};
b = {'z','bc','aa','bbb'};
(c, ia, ib) = union(a, b)
  c =
    {'a','aa','bbb','bc','de','z'}
  ia =
    1 3 2 4
```

```
   ib =
     3 1
 a(ia)
   {'a','bbb','bc','de'}
 b(ib)
   {'aa','z'}
```

Set exclusive or can also be computed as the union of a and b minus the intersection of a and b:

```
 setdiff(union(a, b), intersect(a, b))
   {'a','aa','de','z'}
```

See also

`unique`, `intersect`, `setdiff`, `setxor`, `ismember`

unique

Keep unique elements.

Syntax

```
v2 = unique(v1)
list2 = unique(list1)
(b, ia, ib) = unique(a)
```

Description

With an array argument, `unique(v1)` sorts its elements and removes duplicate elements. Unless `v1` is a row vector, `v1` is considered as a column vector.

With an argument which is a list of strings, `unique(list)` sorts its elements and removes duplicate elements.

The optional second output argument is set to a vector of indices such that if `(b,ia)=unique(a)`, then `b` is `a(ia)`.

The optional third output argument is set to a vector of indices such that if `(b,ia,ib)=unique(a)`, then `a` is `b(ib)`.

Examples

```
(b,ia,ib) = unique([4,7,3,8,7,1,3])
  b =
    1 3 4 7 8
  ia =
    6 3 1 2 4
  ib =
    3 4 2 5 4 1 2
unique({'def', 'ab', 'def', 'abc'})
  {'ab', 'abc', 'def'}
```

See also

`sort`, `union`, `intersect`, `setdiff`, `setxor`, `ismember`

unwrap

Unwrap angle sequence.

Syntax

```
a2 = unwrap(a1)
a2 = unwrap(a1, tol)
A2 = unwrap(A1, tol, dim)
```

Description

`unwrap(a1)`, where `a1` is a vector of angles in radians, returns a vector `a2` of the same length, with the same values modulo 2π, starting with the same value, and where differences between consecutive values do not exceed π. It is useful for interpolation in a discrete set of angles and for plotting.

With two input arguments, `unwrap(a1,tol)` reduces the difference between two consecutive values only if it is larger (in absolute value) than `tol`. If `tol` is smaller than π, or the empty array `[]`, the default value of π is used.

With three input arguments, `unwrap(A1,tol,dim)` operates along dimension `dim`. The result is an array of the same size as `A1`. The default dimension for arrays is 1.

Example

```
unwrap([0, 2, 4, 6, 0, 2])
  0.00  2.00  4.00  6.00  6.28  8.28
```

See also

`mod`, `rem`

zeros

Zero array.

Syntax

```
A = zeros(n)
A = zeros(n1,n2,...)
A = zeros([n1,n2,...])
A = zeros(..., type)
```

Description

`zeros` builds an array whose elements are 0. The size of the array is specified by one integer for a square matrix, or several integers (either as separate arguments or in a vector) for an array of any size.

An additional input argument can be used to specify the type of the result. It must be the string `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` (64-bit arrays are not supported on all platforms).

Examples

```
zeros([2,3])
  0 0 0
  0 0 0
zeros(2)
  0 0
  0 0
zeros(1, 5, 'uint16')
  1x5 uint16 array
    0 0 0 0 0
```

See also

`ones`, `cell`, `eye`, `rand`, `randn`, `repmat`

10.23 Triangulation Functions

delaunay

2-d Delaunay triangulation.

Syntax

```
t = delaunay(x, y)
(t, e) = delaunay(x, y)
```

Description

`delaunay(x,y)` calculates the Delaunay triangulation of 2-d points given by arrays `x` and `y`. Both arrays must have the same number of values, `m`. The result is an array of three columns. Each row corresponds to a triangle; values are indices in `x` and `y`.

The second output argument, if requested, is a logical vector of size `m`-by-1; elements are true if the corresponding point in `x` and `y` belongs to the convex hull of the set of points.

The Delaunay triangulation is a net of triangles which link all the starting points in such a way that no point is included in the circumscribed circle of any other triangle. Triangles are "as equilateral" as possible.

Example

Delaunay triangulation of 20 random points:

```
x = rand(20, 1);
y = rand(20, 1);
(t, e) = delaunay(x, y);
```

With Sysquake graphical functions, points belonging to the convex hull are displayed as crosses and interior points as circles:

```
clf;
scale equal;
plot(x(e), y(e), 'x');
plot(x(~e), y(~e), 'o');
```

Array of vertex indices is modified to have closed triangles:

```
t = [t, t(:, 1)];
```

Triangles are displayed:

```
plot(x(t), y(t));
```

See also

`delaunayn`, `voronoi`

delaunayn

N-d Delaunay triangulation.

Syntax

```
t = delaunayn(x)
(t, e) = delaunayn(x)
```

Description

`delaunayn(x)` calculates the Delaunay triangulation of points given by the rows of array `x` in a space of dimension `size(x,2)`. The result is an array with one more column. Each row corresponds to a simplex; values are row indices in `x` and give the vertices of each polyhedron.

The second output argument, if requested, is a logical vector with as many elements as rows in `x`; elements are true if the corresponding point in `x` belongs to the convex hull of the set of points.

See also

`delaunay`, `tsearchn`, `voronoin`

griddata

Data interpolation in 2-d plane.

Syntax

```
vi = griddata(x, y, v, xi, yi)
vi = griddata(x, y, v, xi, yi, method)
```

Description

`griddata(x,y,v,xi,yi)` interpolates values at coordinates given by the corresponding elements of arrays `xi` and `yi` in a 2-dimension plane. Original data are defined by corresponding elements of arrays `x`, `y`, and `v` (which must have the same size), such that the value at coordinate `[x(i);y(i)]` is `v(i)`. The result is an array with the same size as `xi` and `yi` where `vi(j)` is the value interpolated at `[xi(j);yi(j)]`.

All coordinates are real (imaginary components are ignored). Values v and `vi` can be real or complex. The result for coordinates outside the convex hull defined by x and y is NaN.

`griddata` is based on Delaunay triangulation. The interpolation method used in each triangle is linear by default, or can be specified with an additional input argument, a string:

Argument	**Meaning**
`'0'` or `'nearest'`	nearest value
`'1'` or `'linear'`	linear

Example

Nearest value interpolation in 2D plane of a few values `v(x,y)`. The plane is sampled with a regular grid with `meshgrid`.

```
x = [0.2; 1.8; 0.7; 0.9; 1.6];
y = [0.2; 0.7; 1.8; 1.1; 1.7];
v = [0.1; 0.3; 0.9; 0.5; 0.4];
(xi, yi) = meshgrid(0:0.01:2);
vi = griddata(x, y, v, xi, yi, '0');
```

In Sysquake, the result can be displayed as a contour plot. For locations where the values cannot be interpolated, i.e. outside the convex hull defined by x and y, values are set to 0.

```
vi(isnan(vi)) = 0;
contour(vi, [], 20);
```

See also

`delaunay`, `tsearch`, `griddatan`, `interpn`

griddatan

Data interpolation in N-d space.

Syntax

```
vi = griddatan(x, v, xi)
vi = griddatan(x, v, xi, method)
```

Description

`griddatan(x,v,xi)` interpolates values at coordinates given by the p rows of p-by-n array `xi` in an n-dimension space. Original data are defined by `m`-by-n array `x` and `m`-by-1 column vector `v`, such that the value at coordinate `x(i,:)'` is `v(i)`. The result is a p-by-1 column vector `vi` where `vi(j)` is the value interpolated at `xi(j,:)'`.

Coordinates `x` and `xi` are real (imaginary components are ignored). Values `v` and `vi` can be real or complex. The result for coordinates outside the convex hull defined by `x` is NaN.

`griddatan` is based on Delaunay triangulation. The interpolation method used in each simplex is linear by default, or can be specified with an additional input argument, a string:

Argument	**Meaning**
`'0'` or `'nearest'`	nearest value
`'1'` or `'linear'`	linear

Example

Linear interpolation in 2D plane of a few values `v(x,y)`. The plane is sampled with a regular grid with `meshgrid`. Since `griddatan` interpolates a 1-dim array of points, the result is reshaped to match `x` and `y` (compare with the example of `griddata`).

```
x = [0.2; 1.8; 0.7; 0.9; 1.6];
y = [0.2; 0.7; 1.8; 1.1; 1.7];
v = [0.1; 0.3; 0.9; 0.5; 0.4];
(xi, yi) = meshgrid(0:0.01:2);
vi = griddatan([x,y], v, [xi(:),yi(:)], '1');
vi = reshape(vi, size(xi));
```

In Sysquake, the result can be displayed as a contour plot. For locations where the values cannot be interpolated, i.e. outside the convex hull defined by x and y, values are set to 0.

```
vi(isnan(vi)) = 0;
contour(vi, [], 20);
```

See also

`delaunayn`, `tsearchn`, `griddata`, `interpn`

tsearch

Search of points in triangles.

Syntax

```
ix = tsearch(x, y, t, xi, yi)
```

Description

`tsearch(x,y,t,xi,yi)` searches in which triangle is located each point given by the corresponding elements of arrays `xi` and `yi`. Corresponding elements of arrays `x` and `y` represent the vertices of the triangles, and rows of array `t` represent their indices in `x` and `y`; array `t` is usually the result of `delaunay`. Dimensions of `x` and `y`, and of `xi` and `yi`, must be equal. The result is an array with the same size as `xi` and `yi` where each element is the row index in `t` of the first triangle which contains the point, or NaN if the point is outside all triangles (i.e. outside the convex hull of points defined by `x` and `y` if `t` is a proper triangulation such as the one computed with `delaunay`).

Example

Search for triangles containing points [0,0] and [0,1] corresponding to Delaunay triangulation of 20 random points:

```
x = randn(20, 1);
y = randn(20, 1);
t = delaunay(x, y);
xi = [0, 0];
yi = [0, 1];
ix = tsearch(x, y, t, xi, yi);
```

See also

`tsearchn`, `delaunay`, `voronoi`, `griddata`

tsearchn

Search of points in triangulation simplices.

Syntax

```
ix = tsearchn(x, t, xi)
```

Description

`tsearchn(x,t,xi)` searches in which simplex each point given by the rows of array `xi` is located. Rows of array `x` represent the vertices of the simplices, and rows of array `t` represent their indices in `x`; array `t` is usually the result of `delaunayn`. Dimensions must match: in a space of `n` dimensions, `x` and `xi` have `n` columns, and `t` has `n+1` columns. The result is a column vector with one element for each row of `xi`, which is the row index in `t` of the first simplex which contains the point, or NaN if the point is outside all simplices (i.e. outside the convex hull of points `x` if `t` is a proper triangulation of `x` such as the one computed with `delaunayn`).

Example

Search for simplices containing points [0,0] and [0,1] corresponding to Delaunay triangulation of 20 random points:

```
x = randn(20, 2);
t = delaunayn(x);
xi = [0, 0; 0, 1];
ix = tsearchn(x, t, xi);
```

See also

`tsearch`, `delaunayn`, `voronoin`, `griddatan`

voronoi

2-d Voronoi tessalation.

Syntax

```
(v, p) = voronoi(x, y)
```

Description

`voronoi(x,y)` calculates the Voronoi tessalation of the set of 2-d points given by arrays `x` and `y`. Both arrays must have the same number of values, `m`. The first output argument `v` is an array of two columns which contains the coordinates of the vertices of the Voronoi cells, one row per vertex. The first row contains infinity and is used as a marker for unbounded Voronoi cells. The second output argument `p` is a list of vectors of row indices in `v`; each element describes the Voronoi cell corresponding to a point in `x`. In each cell, vertices are sorted counterclockwise.

Voronoi tessalation is a tessalation (a partition of the plane) such that each region is the set of points closer to one of the initial point than to any other one. Two regions are in contact if and only if their initial points are linked in the corresponding Delaunay triangulation.

Example

Voronoi tessalation of 20 random points:

```
x = rand(20, 1);
y = rand(20, 1);
(v, p) = voronoi(x, y);
```

These points are displayed as crosses with Sysquake graphical functions. The scale is fixed, because Voronoi polygons can have vertices which are far away from the points.

```
clf;
scale('equal', [0,1,0,1]);
plot(x, y, 'x');
```

Voronoi polygons are displayed in a loop, skipping unbounded polygons. The first vertex is repeated to have closed polygons. Since `plot` expects row vectors, vertex coordinates are transposed.

```
for p1 = p
  if ~any(p1 == 1)
    p1 = [p1, p1(1)];
    plot(v(p1,1)', v(p1,2)');
  end
end
```

See also

`voronoin`, `delaunay`

voronoin

N-d Voronoi tessalation.

Syntax

```
(v, p) = voronoin(x)
```

Description

`voronoin(x)` calculates the Voronoi tessalation of the set of points given by the rows of arrays `x` in a space of dimension `n=size(x,2)`. The first output argument `v` is an array of `n` columns which contains the coordinates of the vertices of the Voronoi cells, one row per vertex. The first row contains infinity and is used as a marker for unbounded Voronoi cells. The second output argument `p` is a list of vectors of row indices in `v`; each element describes the Voronoi cell corresponding to a point in `x`. In each cell, vertices are sorted by index.

See also
`voronoi`, `delaunayn`

10.24 Integer Functions

uint8 uint16 uint32 uint64 int8 int16 int32 int64

Conversion to integer types.

Syntax

```
B = uint8(A)
B = uint16(A)
B = uint32(A)
B = uint64(A)
B = int8(A)
B = int16(A)
B = int32(A)
B = int64(A)
```

Description

The functions convert a number or an array to unsigned or signed integers. The name contains the size of the integer in bits.

To avoid a conversion from double to integer, constant literal numbers should be written with a type suffix, such as `12int32`. This is the only way to specify large 64-bit numbers, because double-precision floating-point numbers have a mantissa of 56 bits.

Constant arrays of `uint8` can also be encoded in a compact way using base64 inline data.

`uint64` and `int64` are not supported on platforms with tight memory constraints.

Examples

```
uint8(3)
  3uint8
3uint8
  3uint8
uint8([50:50:400])
  1x8 uint8 array
    50 100 150 200 250  44  94 144
@/base64 MmSWyPosXpA=
   50
  100
  ...
  144
```

```
int8([50:50:400])
  1x8 int8 array
    50  100 -106  -56   -6   44   94 -112
```

The base64 data above is obtained with the following expression:

```
base64encode(uint8([50:50:400]))
```

See also

`double`, `single`, `char`, `logical`, `map2int`

intmax

Largest integer.

Syntax

```
i = intmax
i = intmax(type)
```

Description

Without input argument, `intmax` gives the largest signed 32-bit integer. `intmax(type)` gives the largest integer of the type specified by string `type`, which can be `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'int8'`, `'int16'`, `'int32'`, or `'int64'` (64-bit integers are not supported on all platforms). The result has the corresponding integer type.

Examples

```
intmax
  2147483647int32
intmax('uint16')
  65535uint16
```

See also

`intmin`, `realmax`, `flintmax`, `uint8` and related functions, `map2int`

intmin

Smallest integer.

Syntax

```
i = intmin
i = intmin(type)
```

Description

Without input argument, `intmin` gives the smallest signed 32-bit integer. `intmin(type)` gives the largest integer of the type specified by string `type`, which can be `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'int8'`, `'int16'`, `'int32'`, or `'int64'` (64-bit integers are not supported on all platforms). The result has the corresponding integer type.

Examples

```
intmin
  -2147483648int32
intmin('uint16')
  0uint16
```

See also

`intmax`, `realmin`, `uint8` and related functions, `map2int`

map2int

Mapping of a real interval to an integer type.

Syntax

```
B = map2int(A)
B = map2int(A, vmin, vmax)
B = map2int(A, vmin, vmax, type)
```

Description

`map2int(A,vmin,vmax)` converts number or array `A` to 8-bit unsigned integers. Values between `vmin` and `vmax` in `A` are mapped linearly to values 0 to 255. With a single input argument, the default input interval is 0 to 1.

`map2int(A,vmin,vmax,type)` converts `A` to the specified type, which can be any integer type given as a string: `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'int8'`, `'int16'`, `'int32'`, or `'int64'` (64-bit integers are not supported on all platforms). The input interval is mapped to its full range.

In all cases, input values outside the interval are clipped to the minimum or maximum values.

Examples

```
map2int(-0.2:0.2:1.2)
  1x5 uint8 array
    0   0  51 102 153 204 255 255
```

```
map2int([1,3,7], 0, 10, 'uint16')
  1x3 uint16 array
    6553 19660 45875
map2int([1,3,7], 0, 10, 'int16')
  1x3 int16 array
    -26214 -13107  13107
```

See also

`uint8` and related functions.

10.25 Non-Linear Numerical Functions

fminbnd

Minimum of a function.

Syntax

```
(x, y) = fminbnd(fun, x0)
(x, y) = fminbnd(fun, [xlow,xhigh])
(x, y) = fminbnd(..., options)
(x, y) = fminbnd(..., options, ...)
(x, y, didConverge) = fminbnd(...)
```

Description

`fminbnd(fun,...)` finds numerically a local minimum of function `fun`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least one input argument `x`, and it returns one output argument, also a real number. `fminbnd` finds the value `x` such that `fun(x)` is minimized.

Second argument tells where to search; it can be either a starting point or a pair of values which must bracket the minimum.

The optional third argument may contain options. It is either the empty array `[]` for default options, or the result of `optimset`.

Remaining input arguments of `fminbnd`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `fminbnd('fun',x0,[],2,5)` calls `fun` as `fun(x,2,5)` and minimizes its value with respect to `x`.

The first output argument of `fminbnd` is the value of `x` at optimum. The second output argument, if it exists, is the value of `fun(x)` at optimum. The third output argument, if it exists, is set to true if `fminbnd` has converged to an optimum, or to false if it has not; in that case, other output arguments are set to the best value obtained. With one or two output arguments, `fminbnd` throws an error if it does not converge.

Examples

Minimum of a sine near 2, displayed with 15 digits:

```
fprintf('%.15g\n', fminbnd(@sin, 2));
  4.712389014989218
```

To find the minimum of $ce^x - \sin x$ between -1 and 10 with $c = 0.1$, the expression is written as an inline function stored in variable `fun`:

```
fun = inline('c*exp(x)-sin(x)', 'x', 'c');
```

Then `fminbnd` is used, with the value of `c` passed as an additional argument:

```
x = fminbnd(fun,[-1,10],[],0.1)
  x =
    1.2239
```

With an anonymous function, this becomes

```
c = 0.1;
fun = @(x) c*exp(x)-sin(x);
x = fminbnd(fun,[-1,10])
  x =
    1.2239
```

Attempt to find the minimum of an unbounded function:

```
(x,y,didConverge) = fminbnd(@exp,10)
  x =
    -inf
  y =
    0
  didConverge =
    false
```

See also

`optimset`, `fminsearch`, `fzero`, `inline`, operator `@`

fminsearch

Minimum of a function in Rˆn.

Syntax

```
x = fminsearch(fun, x0)
x = fminsearch(..., options)
x = fminsearch(..., options, ...)
(x, y, didConverge) = fminsearch(...)
```

Description

`fminsearch(fun,x0,...)` finds numerically a local minimum of function `fun`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least one input argument `x`, a real scalar, vector or array, and it returns one output argument, a scalar real number. `fminsearch` finds the value `x` such that `fun(x)` is minimized, starting from point `x0`.

The optional third input argument may contain options. It is either the empty array `[]` for default options, or the result of `optimset`.

Remaining input arguments of `fminsearch`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `fminsearch('fun',x0,[],2,5)` calls `fun` as `fun(x,2,5)` and minimizes its value with respect to `x`.

The first output argument of `fminsearch` is the value of `x` at optimum. The second output argument, if it exists, is the value of `fun(x)` at optimum. The third output argument, if it exists, is set to true if `fminsearch` has converged to an optimum, or to false if it has not; in that case, other output arguments are set to the best value obtained. With one or two output arguments, `fminsearch` throws an error if it does not converge.

Algorithm

`fminsearch` implements the Nelder-Mead simplex method. It starts from a polyhedron centered around `x0` (the "simplex"). Then at each iteration, either vertex `x_i` with the maximum value `fun(x_i)` is moved to decrease it with a reflexion-expansion, a reflexion, or a contraction; or the simplex is shrinked around the vertex with minimum value. Iterations stop when the simplex is smaller than the tolerance, or when the maximum number of iterations or function evaluations is reached (then an error is thrown).

Examples

Minimum of a sine near 2, displayed with 15 digits:

```
fprintf('%.15g\n', fminsearch(@sin, 2));
  4.712388977408411
```

Maximum of $xe^{-x^2y^2}xy - 0.1x^2$ The function is defined as an anonymous function stored in variable `fun`:

```
fun = @(x,y) x.*exp(-(x.*y).^2).*x.*y-0.1*x.^2;
```

In Sysquake, the contour plot can be displayed with the following commands:

```
[X,Y] = meshgrid(0:0.02:3, 0:0.02:3);
contour(feval(fun, X, Y), [0,3,0,3], 0.1:0.05:0.5);
```

The maximum is obtained by minimizing the opposite of the function, rewritten to use as input a single variable in R^2:

```
mfun = @(X) -(X(1)*exp(-(X(1)*X(2))^2)*X(1)*X(2)-0.1*X(1)^2);
fminsearch(mfun, [1, 2])
  2.1444  0.3297
```

Here is another way to find this maximum, by calling `fun` from an intermediate anonymous function:

```
fminsearch(@(X) -fun(X(1),X(2)), [1, 2])
  2.1444  0.3297
```

For the same function with a constraint $x < 1$, the objective function can be modified to return $+\infty$ for inputs outside the feasible region (note that we can start on the constraint boundary, but starting from the infeasible region would probably fail):

```
fminsearch(@(X) X(1) < 1 ? -fun(X(1),X(2)) : inf, [1, 2])
  1    0.7071
```

See also

`optimset`, `fminbnd`, `lsqnonlin`, `fsolve`, `inline`, operator `@`

fsolve

Solve a system of nonlinear equations.

Syntax

```
x = fsolve(fun, x0)
x = fsolve(..., options)
x = fsolve(..., options, ...)
(x, y, didConverge) = fsolve(...)
```

Description

`fsolve(fun,x0,...)` finds numerically a zero of function `fun`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least one input argument `x`, a real scalar, vector or array, and it returns one output argument `y` whose size should match `x`. `fsolve` attempts to find the value `x` such that `fun(x)` is zero, starting from point `x0`. Depending on the existence of any solution and on the choice of `x0`, `fsolve` may fail to find a zero.

The optional third input argument may contain options. It is either the empty array `[]` for default options, or the result of `optimset`.

Remaining input arguments of `fsolve`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `fsolve(@fun,x0,[],2,5)` finds the value of `x` such that the result of `fun(x,2,5)` is zero.

The first output argument of `fsolve` is the value of `x` at zero. The second output argument, if it exists, is the value of `fun(x)` at zero; it should be a vector or array whose elements are zero, up to the tolerance, unless `fsolve` cannot find it. The third output argument, if it exists, is set to true if `fsolve` has converged to a solution, or to false if it has not; in that case, other output arguments are set to the best value obtained. With one or two output arguments, `fsolve` throws an error if it does not converge.

Algorithm

`fsolve` minimizes the sum of squares of the vector elements returned by `fun` using the Nelder-Mead simplex method of `fminsearch`.

Example

One of the zeros of `x1^2+x2^2=10`, `x2=exp(x1)`:

```
[x, y, didConverge] = fsolve(@(x) [x(1)^2+x(2)^2-10; x(2)-exp(x(1))], [0; 0]
  x =
   -3.1620
    0.0423
  y =
   -0.0000
   -0.0000
  didConverge =
    true
```

See also

`optimset`, `fminsearch`, `fzero`, `inline`, operator `@`

fzero

Zero of a function.

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,[xlow,xhigh])
x = fzero(...,options)
x = fzero(...,options,...)
```

Description

`fzero(fun,...)` finds numerically a zero of function `fun`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least one input argument `x`, and it returns one output argument, also a real number. `fzero` finds the value `x` such that `fun(x)==0`, up to some tolerance.

Second argument tells where to search; it can be either a starting point or a pair of values `xlow` and `xhigh` which must bracket the zero, such that `fun(xlow)` and `fun(xhigh)` have opposite sign.

The optional third argument may contain options. It is either the empty array `[]` for the default options, or the result of `optimset`.

Additional input arguments of `fzero` are given as additional input arguments to the function specified by `fun`. They permit to parameterize the function.

Examples

Zero of a sine near 3, displayed with 15 digits:

```
fprintf('%.15g\n', fzero(@sin, 3));
  3.141592653589793
```

To find the solution of $e^x = c + \sqrt{x}$ between 0 and 100 with $c = 10$, a function `f` whose zero gives the desired solution is written:

```
function y = f(x,c)
  y = exp(x) - c - sqrt(x);
```

Then `fsolve` is used, with the value of `c` passed as an additional argument:

```
x = fzero(@f,[0,100],[],10)
  x =
    2.4479
f(x,10)
  1.9984e-15
```

An anonymous function can be used to avoid passing `10` as an additional argument, which can be error-prone since a dummy empty option arguments has to be inserted.

```
x = fzero(@(x) f(x,10), [0,100])
  x =
    2.4479
```

See also

`optimset`, `fminsearch`, `inline`, operator `@`, `roots`

integral

Numerical integration.

Syntax

```
y = integral(fun, a, b)
y = integral(fun, a, b, options)
```

Description

`integral(fun,a,b)` integrates numerically function `fun` between a and `b`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has a single input argument and a single output argument, both scalar real or complex.

Options can be provided with named arguments. The following options are accepted:

Name	Default	Meaning
`AbsTol`	`1e-6`	maximum absolute error
`RelTol`	`1e-3`	maximum relative error
`Display`	`false`	statistics display

Example

$$\int_0^2 te^{-t}\mathrm{d}t$$

```
integral(@(t) t*exp(-t), 0, 2, AbsTol=1e-9)
  0.5940
```

See also

`sum`, `ode45`, `inline`, operator `@`

lsqcurvefit

Least-square curve fitting.

Syntax

```
param = lsqcurvefit(fun, param0, x, y)
param = lsqcurvefit(..., options)
param = lsqcurvefit(..., options, ...)
(param, r, didConverge) = lsqcurvefit(...)
```

Description

`lsqcurvefit(fun,p0,x,y,...)` finds numerically the parameters of function `fun` such that it provides the best fit for the curve defined by `x` and `y` in a least-square sense. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least two input arguments: `p`, the parameter vector, and `x`, a vector or array of input data; it returns one output argument, a vector or array the same size as `x` and `y`. Its header could be

```
function y = f(param, x)
```

`lsqcurvefit` finds the value `p` which minimizes `sum((fun(p,x)-y).^2)`, starting from parameters `p0`. All values are real.

The optional fifth input argument may contain options. It is either the empty array `[]` for default options, or the result of `optimset`.

Remaining input arguments of `lsqcurvefit`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `lsqcurvefit('fun',p0,x,y,[],2,5)` calls `fun` as `fun(p,x,2,5)` and find the (local) least-square solution with respect to `p`.

The first output argument of `lsqcurvefit` is the value of `p` at optimum. The second output argument, if it exists, is the value of the cost function at optimum. The third output argument, if it exists, is set to true if `lsqcurvefit` has converged to an optimum, or to false if it has not; in that case, other output arguments are set to the best value obtained. With one or two output arguments, `lsqcurvefit` throws an error if it does not converge.

Algorithm

Like `lsqnonlin`, `lsqcurvefit` is based on the Nelder-Mead simplex method.

Example

Find the best curve fit of `y=a*sin(b*x+c)` with respect to parameters `a`, `b` and `c`, where `x` and `y` are given (see the example of `lsqnonlin` for another way to solve the same problem).

```
% assume nominal parameter values a0=2, b0=3, c0=1
a0 = 2; b0 = 3; c0 = 1;
% reset the seed of rand and randn for reproducible results
rand('s', 0); randn('s', 0);
% create x and y, with noise
x0 = rand(1, 100);
x = x0 + 0.05 * randn(1, 100);
y = a0 * sin(b0 * x0 + c0) + 0.05 * randn(1, 100);
```

```
% find least-square curve fit, starting from 1, 1, 1
p0 = [1; 1; 1];
p_ls = lsqcurvefit(@(p, x) p(1) * sin(p(2) * x + p(3)), p0, x, y)
  p_ls =
    2.0060
    2.8504
    1.0836
```

In Sysquake, the solution can be displayed with

```
fplot(@(x) a0 * sin(b0 * x + c0), [0,1], 'r');
plot(x, y, 'o');
fplot(@(x) p_ls(1)*sin(p_ls(2)*x+p_ls(3)), [min(x), max(x)]);
legend('Nominal\nSamples\nLS fit', 'r_kok_');
```

See also

`optimset`, `lsqnonlin`, `inline`, operator `@`

lsqnonlin

Nonlinear least-square solver.

Syntax

```
x = lsqnonlin(fun, x0)
x = lsqnonlin(..., options)
x = lsqnonlin(..., options, ...)
(x, y, didConverge) = lsqnonlin(...)
```

Description

`lsqnonlin(fun,x0,...)` finds numerically the value such that the sum of squares of the output vector produced by `fun` is a local minimum. `fun` is either specified by its name or given as an anonymous or inline function or a function reference. It has at least one input argument `x`, a real scalar, vector or array, and it returns one output argument, a real vector or array. Its header could be

```
function y = f(x)
```

`lsqnonlin` finds the value `x` such that `sum(fun(x(:)).^2)` is minimized, starting from point `x0`.

The optional third input argument may contain options. It is either the empty array `[]` for default options, or the result of `optimset`.

Remaining input arguments of `lsqnonlin`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `lsqnonlin('fun',x0,[],2,5)` calls `fun` as

`fun(x,2,5)` and find the (local) least-square solution with respect to x.

The first output argument of `lsqnonlin` is the value of x at optimum. The second output argument, if it exists, is the value of `fun(x)` at optimum. The third output argument, if it exists, is set to true if `lsqnonlin` has converged to an optimum, or to false if it has not; in that case, other output arguments are set to the best value obtained. With one or two output arguments, `lsqnonlin` throws an error if it does not converge.

Algorithm

Like `fminsearch`, `lsqnonlin` is based on the Nelder-Mead simplex method.

Example

Find the least-square solution of `a*sin(b*x+c)-y` with respect to parameters a, b and c, where x and y are given (see the example of `lsqcurvefit` for another way to solve the same problem).

```
% assume nominal parameter values a0=2, b0=3, c0=1
a0 = 2; b0 = 3; c0 = 1;
% reset the seed of rand and randn for reproducible results
rand('s', 0); randn('s', 0);
% create x and y, with noise
x0 = rand(1, 100);
x = x0 + 0.05 * randn(1, 100);
y = a0 * sin(b0 * x0 + c0) + 0.05 * randn(1, 100);
% find least-square solution, starting from 1, 1, 1
p0 = [1; 1; 1];
p_ls = lsqnonlin(@(p) p(1) * sin(p(2) * x + p(3)) - y, p0)
  p_ls =
    2.0060
    2.8504
    1.0836
```

In Sysquake, the solution can be displayed with

```
fplot(@(x) a0 * sin(b0 * x + c0), [0,1], 'r');
plot(x, y, 'o');
fplot(@(x) p_ls(1)*sin(p_ls(2)*x+p_ls(3)), [min(x), max(x)]);
legend('Nominal\nSamples\nLS fit', 'r_kok_');
```

See also

`optimset`, `fminsearch`, `lsqcurvefit`, `inline`, operator `@`

ode23 ode45 ode23s

Ordinary differential equation integration.

Syntax

```
(t,y) = ode23(fun,[t0,tend],y0)
(t,y) = ode23(fun,[t0,tend],y0,options)
(t,y) = ode23(fun,[t0,tend],y0,options,...)
(t,y,te,ye,ie) = ode23(...)
(t,y) = ode45(fun,[t0,tend],y0)
(t,y) = ode23s(fun,[t0,tend],y0)
...
```

Description

`ode23(fun,[t0,tend],y0)` and `ode45(fun,[t0,tend],y0)` integrate numerically an ordinary differential equation (ODE). Both functions are based on a Runge-Kutta algorithm with adaptive time step; `ode23` is low-order and ode45 high-order. In most cases for non-stiff equations, ode45 is the best method.

`ode23s(fun,[t0,tend],y0)` integrates numerically an ordinary differential equation with a low-order algorithm suitable for stiff systems.

The function to be integrated is either specified by its name or given as an anonymous or inline function or a function reference. It should have at least two input arguments and exactly one output argument:

```
function yp = f(t,y)
```

The function calculates the derivative `yp` of the state vector `y` at time `t`.

Integration is performed over the time range specified by the second argument `[t0,tend]`, starting from the initial state `y0`. It may stop before reaching `tend` if the integration step cannot be reduced enough to obtain the required tolerance. If the function is continuous, you can try to reduce `MinStep` in the options argument (see below).

The optional fourth argument may contain options. It is either the empty array `[]` for the default options, or the result of `odeset` (the use of a vector of option values is deprecated.)

Events generated by options `Events` or `EventTime` can be obtained by three additional output arguments: `(t,y,te,ye,ie)=...` returns event times in `te`, the corresponding states in `ye` and the corresponding event identifiers in `ie`.

Additional input arguments of `ode45` are given as additional input arguments to the function specified by `fun`. They permit to parameterize the ODE.

`ode23s` needs the jacobian of the ODE. The jacobian can be passed as a constant square matrix or as a function in the `Jacobian` option; otherwise numerical approximations are computed.

Examples

Let us integrate the following ordinary differential equation (Van Der Pol equation), parameterized by μ:

$$x'' = \mu\left(1-x^2\right)x' - x$$

Let $y_1 = x$ and $y_2 = x'$; their derivatives are

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu\left(1-y_1^2\right)y_2 - y_1 \end{aligned}$$

and can be computed by the following function:

```
function yp = f(t, y, mu)
yp = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
```

The following ode45 call integrates the Van Der Pol equation from 0 to 10 with the default options, starting from $x(0) = 2$ and $x'(0) = 0$, with $\mu = 1$ (see Fig. 10.1):

```
(t, y) = ode45(@f, [0,10], [2;0], [], 1);
```

The same result can be obtained with an anonymous function:

```
mu=1;
(t, y) = ode45(@(t,y) [y(2); mu*(1-y(1)^2)*y(2)-y(1)],
  [0,10], [2;0]);
```

The `plot` command expects traces along the second dimension; consequently, the result of ode45 should be transposed.

```
plot(t', y');
```

If μ is large, the ODE is stiff. ode23s is much more efficient.

```
mu=100;
(t, y) = ode23s(@(t,y) [y(2); mu*(1-y(1)^2)*y(2)-y(1)],
  [0,300], [2;0]);
```

While ode23s can be called with the same arguments as ode23 and ode45, it is more efficient to provide a function which computes directly the jacobian of the ODE to avoid numerical approximations. The jacobian of the Van Der Pol equations can be computed by an anonymous function and passed to ode23s as a named argument:

```
mu=100;
(t, y) = ode23s(@(t,y) [y(2); mu*(1-y(1)^2)*y(2)-y(1)],
  [0,300], [2;0],
  Jacobian=@(t,y) [0, 1; -2*mu*y(1)*y(2)-1, mu*(1-y(1)^2)]);
```

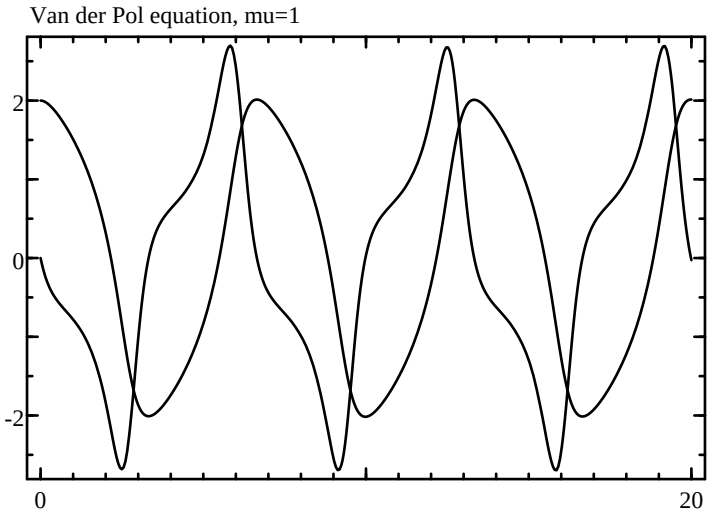

Figure 10.1 Van der Pol equation with $\mu = 1$ integrated with ode45

See also

odeset, integral, inline, operator @, expm

odeset

Options for ordinary differential equation integration.

Syntax

```
options = odeset
options = odeset(name1=value1, ...)
options = odeset(name1, value1, ...)
options = odeset(options0, name1=value1, ...)
options = odeset(options0, name1, value1, ...)
```

Description

odeset(name1,value1,...) creates the option argument used by ode23, ode45 and ode23s. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Alternatively, options can be given with named arguments. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, odeset creates a structure with all the default options. Note that ode23 etc. also interpret the lack of an option argument, or the empty array [], as a request to use the default values. Options can also be passed directly to them as named arguments.

When its first input argument is a structure, odeset adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options (empty arrays mean "automatic"):

Name	**Default**	**Meaning**
AbsTol	1e-6	maximum absolute error
Events	[] (none)	state-based event function
EventTime	[] (none)	time-based event function
InitialStep	[] (10∗MinStep)	initial time step
Jacobian	[] (undefined)	ODE jacobian
MaxStep	[] (time range/10)	maximum time step
MinStep	[] (time range/1e6)	minimum time step
NormControl	false	error control on state norm
OnEvent	[] (none)	event function
OutputFcn	[] (none)	output function
Past	false	provide past times and states
PreArg	{}	list of prepended input arguments
Refine	[] (1, 4 for ode45)	refinement factor
RelTol	1e-3	maximum relative error
Stats	false	statistics display

Jacobian

The jacobian is used only by ode23s. With an empty matrix, ode23s calculates numerical approximations. If possible, it is more efficient to provide the jacobian either as a constant square matrix if it is constant, or as a function called by ode23s at each integration step. The function is defined as

```
function J = jac(t, y)
```

The jacobian of vector function f(y) is the square matrix whose columns are the partial derivatives of f with respect to y(1), y(2) etc.

Time steps and output

Several options control how the time step is tuned during the numeric integration. Error is calculated separately on each element of y if NormControl is false, or on norm(y) if it is true; time steps are chosen so that it remains under AbsTol or RelTol times the state, whichever is larger. If this cannot be achieved, for instance if the system is stiff and requires an integration step smaller than MinStep, integration is aborted.

'Refine' specifies how many points are added to the result for each integration step. When it is larger than 1, additional points are interpolated, which is much faster than reducing MaxStep.

The output function `OutputFcn`, if defined, is called after each step. It is a function name in a string, a function reference, or an anonymous or inline function, which can be defined as

```
function stop = outfun(tn, yn)
```

where `tn` is the time of the new samples, `yn` their values, and `stop` a logical value which is false to continue integrating or true to stop. The number of new samples is given by the value of `Refine`; when multiple values are provided, `tn` is a row vector and `yn` is a matrix whose columns are the corresponding states. The output function can be used for incremental plots, for animations, or for managing large amounts of output data without storing them in variables.

Events

Events are additional time steps at controlled time, to change instantaneously the states, and to base the termination condition on the states. Time instants where events occur are either given explicitly by `EventTime`, or implicitly by `Events`. There can be multiple streams of events, which are checked independently and are identified by a positive integer for `Events`, or a negative integer for `EventTime`. For instance, for a ball which bounces between several walls, the intersection between each wall and the ball trajectory would be a different event stream.

For events which occur at regular times, `EventTime` is an n-by-two matrix: for each row, the first column gives the time step `ts`, and the second column gives the offset `to`. Non-repeating events are specified with an infinite time step `ts`. Events occur at time `t=to+k*ts`, where k is an integer.

When event time is varying, `EventTime` is a function which can be defined as

```
function eventTime = eventtimefun(t, y, ...)
```

where `t` is the current time, `y` the current state, and the ellipsis stand for additional arguments passed to `ode*`. The function returns a (column) vector whose elements are the times where the next event occurs. In both cases, each row corresponds to a different event stream.

For events which are based on the state, the value of a function which depends on the time and the states is checked; the event occurs when its sign changes. `Events` is a function which can be defined as

```
function (value, isterminal, direction) ...
          = eventsfun(t, y, ...)
```

Input arguments are the same as for `EventTime`. Output arguments are (column) vectors where each element `i` corresponds to an event

stream. An event occurs when `value(i)` crosses zero, in either direction if `direction(i)==0`, from negative to nonnegative if `direction(i)>0`, or from positive to nonpositive if `direction(i)<0`. The event terminates integration if `isterminal(i)` is true. The `Events` function is evaluated for each state obtained by integration; intermediate time steps obtained by interpolation when `Refine` is larger than `1` are not considered. When an event occurs, the integration time step is reset to the initial value, and new events are disabled during the next integration step to avoid shattering. `MaxStep` should be used if events are missed when the result of `Events` is not monotonous between events.

When an event occurs, function `OnEvent` is called if it exists. It can be defined as

```
function yn = onevent(t, y, i, ...)
```

where `i` identifies the event stream (positive for events produced by `Events` or negative for events produced by `EventTime`); and the output `yn` is the new value of the state, immediately after the event.

The primary goal of `ode*` functions is to integrate states. However, there are systems where some states are constant between events, and are changed only when an event occurs. For instance, in a relay with hysteresis, the output is constant except when the input overshoots some value. In the general case, `ni` states are integrated and `n-ni` states are kept constant between events. The total number of states `n` is given by the length of the initial state vector `y0`, and the number of integrated states `ni` is given by the size of the output of the integrated function. Function `OnEvent` can produce a vector of size `n` to replace all the states, of size `n-ni` to replace the non-integrated states, or empty to replace no state (this can be used to display results or to store them in a file, for instance).

Event times are computed after an integration step has been accepted. If an event occurs before the end of the integration step, the step is shortened; event information is stored in the output arguments of `ode*` `te`, `ie` and `ye`; and the `OnEvent` function is called. The output arguments `t` and `y` of `ode*` contain two rows with the same time and the state right before the event and right after it. The time step used for integration is not modified by events.

Additional arguments

`Past` is a logical value which, if true, specifies that the time and state values computed until now (what will eventually be the result of `ode23`, `ode45` or `ode23s`) are passed as additional input arguments to functions called during intergration. This is especially useful for delay differential equations (DDE), where the state at some time point in the

past can be interpolated from the integration results accumulated until now with `interp1`. Assuming no additional parameters or `PreArg` (see below), functions must be defined as

```
function yp = f(t,y,tpast,ypast)
function stop = outfun(tn,yn,tpast,ypast)
function eventTime = eventtimefun(t,y,tpast,ypast)
function (value, isterminal, direction) ...
          = eventsfun(t,y,tpast,ypast)
function yn = onevent(t,y,tpast,ypast,i)
function J = jac(t,y,tpast,ypast)
```

`PreArg` is a list of additional input arguments for all functions called during integration; they are placed before normal arguments. For example, if its value is `{1,'abc'}`, the integrated function is called as `fun(1,'abc',t,y)`, the output function as `outfun(1,'abc',tn,yn)`, and so on.

Examples

Default options

```
odeset
  AbsTol: 1e-6
  Events: []
  EventTime: []
  InitialStep: []
  Jacobian: []
  MaxStep: []
  MinStep: []
  NormControl: false
  OnEvent: []
  OutputFcn: []
  PreArg: {}
  Refine: []
  RelTol: 1e-3
  Stats: false
```

Options passed as named arguments

Unless options must be stored as a whole in a variable, it is often more convenient to pass them directly to the integration function as named arguments. The following calls are equivalent.

```
(t, y) = ode45(fun, tspan, y0, odeset('RelTol', 1e-4));
(t, y) = ode45(fun, tspan, y0, odeset(RelTol=1e-4));
(t, y) = ode45(fun, tspan, y0, RelTol=1e-4);
```

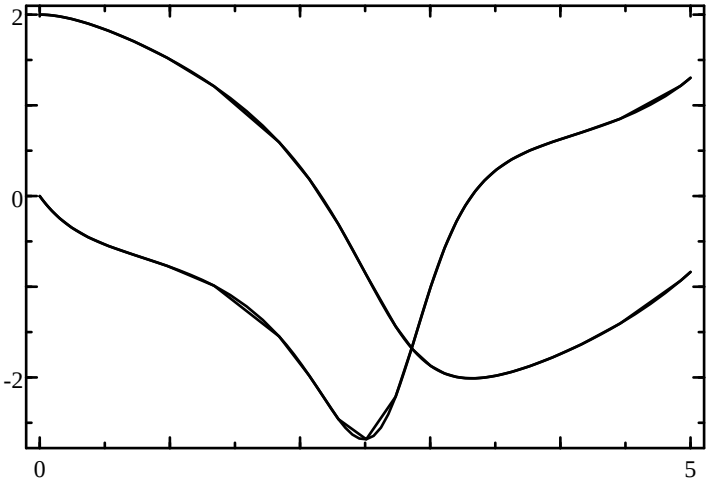

Figure 10.2 Van der Pol equation with `Refine` set to 1 and 4

Option `Refine`

ode45 is typically able to use large time steps to achieve the requested tolerance. When plotting the output, however, interpolating it with straight lines produces visual artifacts. This is why ode45 inserts 3 interpolated points for each calculated point, based on the fifth-order approximation calculated for the integration (`Refine` is 4 by default). In the following code, curves with and without interpolation are compared (see Fig. 10.2). Note that the numbers of evaluations of the function being integrated are the same.

```
mu = 1;
fun = @(t,y) [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
(t, y) = ode45(fun, [0,5], [2;0],
               Refine=1, Stats=true);
  Number of function evaluations: 289
  Successful steps: 42
  Failed steps (error too large): 6
size(y)
  43  2
(ti, yi) = ode45(fun, [0,5], [2;0],
                 Stats=true);
  Number of function evaluations: 289
  Successful steps: 42
  Failed steps (error too large): 6
size(yi)
  169  2
plot(ti', yi', 'g');
plot(t', y');
```

State-based events

For simulating a ball bouncing on the ground, an event is generated every time the ball hits the ground, and its speed is changed instantaneously. Let `y(1)` be the height of the ball above the ground, and `y(2)` its speed (SI units are used). The state-space model is

```
y' = [y(2); -9.81];
```

An event occurs when the ball hits the ground:

```
value = y(1);
isterminal = false;
direction = -1;
```

When the event occurs, a new state is computed:

```
yn = [0; -damping*y(2)];
```

To integrate this, the following functions are defined:

```
function yp = ballfun(t, y, damping)
  yp = [y(2); -9.81];
function (v, te, d) = ballevents(t, y, damping)
  v = y(1);    // event when the height becomes negative
  te = false;  // do not terminate
  d = -1;      // only for negative speeds
function yn = ballonevent(t, y, i, damping)
  yn = [0; -damping*y(2)];
```

Ball state is integrated during 5 s (see Fig. 10.3) with

```
opt = odeset(Events=@ballevents,
             OnEvent=@ballonevent);
(t, y) = ode45(@ballfun, [0, 5], [2; 0], opt, 1);
plot(t', y');
```

Time events with discontinuous function

If the function being integrated has discontinuities at known time instants, option `EventTime` can be used to insure an accurate switching time. Consider a first-order filter with input $u(t)$, where $u(t) = 0$ for $t < 1$ and $u(t) = 1$ for $t \geq 1$. The following function is defined for the state derivative:

```
function yp = filterfun(t, y)
  yp = -y + (t <= 1 ? 0 : 1);
```

A single time event is generated at $t = 1$:

```
opt = odeset(EventTime=[inf, 1]);
(t, y) = ode45(@filterfun, [0, 5], 0, opt);
plot(t', y');
```

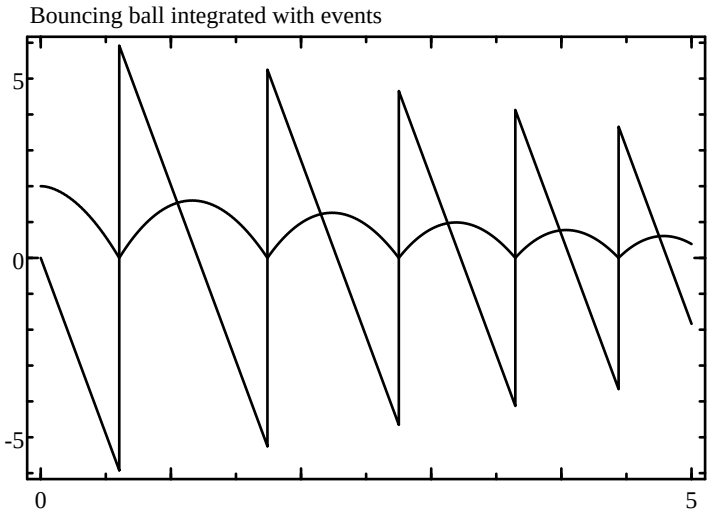

Figure 10.3 Bouncing ball integrated with events

Function `filterfun` is integrated in the normal way until $t = 1$ inclusive, with $u = 0$. This is why the conditional expression in `filterfun` is *less than or equal to* and not *less than*. Then the event occurs, and integration continues from $t = 1 + \epsilon$ with $u = 0$.

Early termination

The normal termination criterion is the final time specified in the `tspan` argument of ODE solver functions. A state-based criterion can be specified with either a state-based event `Events` or an output function `OutputFcn`. An output function can be simpler to specify, and faster because it does not attempt to reach precisely a state-based transition.

The next example integrates the free fall of an object until its height becomes negative. The state contains the height in `x(1)` and the height derivative (the speed) in `x(2)`. The objects starts at rest at an height of 10 m. The final time specified in `tspan` (100 s) is assumed to be large enough. The integration terminates as soon as any new height is negative (multiple samples are fed to `OutputFcn` at each integration step if `Refine>1`).

```
fder = @(t,x) [x(2); -9.81];
x0 = [10; 0];
(t,x) = ode45(fder, [0,100], x0, OutputFcn=@(tn,xn) any(xn(:,1)<0));
```

Non-integrated state

For the last example, we will consider a system made of an integrator and a relay with hysteresis in a loop. Let `y(1)` be the output of the

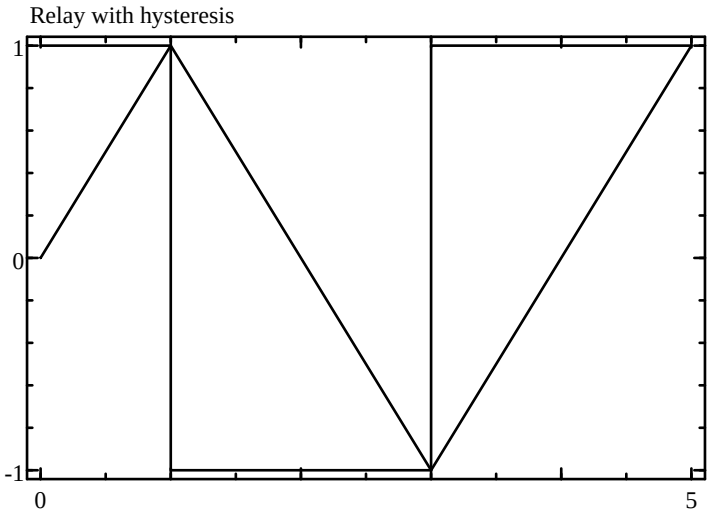

Figure 10.4 Relay with hysteresis integrated with events

integrator and `y(2)` the output of the relay. Only `y(1)` is integrated:

```
yi' = y(2);
```

An event occurs when the integrator is larger or smaller than the hysteresis:

```
value = y(1) - y(2);
isTerminal = false;
direction = sign(y(2));
```

When the event occurs, a new value is computed for the 2nd state:

```
yn = -y(2);
```

To integrate this, the following functions are defined:

```
function yp = relayfun(t, y)
  yp = y(2);
function (v, te, d) = relayevents(t, y)
  v = y(1) - y(2);
  te = false;
  d = sign(y(2));
function yn = relayonevent(t, y, i)
  yn = -y(2);
```

The initial state is `[0;1]`; 0 for the integrator, and 1 for the output of the relay. State is integrated during 5 s (see Fig. 10.4) with

```
(t, y) = ode45(@relayfun, [0, 5], [0; 1],
               Events=@relayevents, OnEvent=@relayonevent);
plot(t', y');
```

Delay differential equation

A system whose Laplace transform is $Y(s)/U(s) = e^{-ds}/(s^2 + s)$ (first order + integrator + delay d) is simulated with unit negative feedback. The reference signal is 1 for $t > 0$. First, the open-loop system is converted from transfer function to state-space, such that $x'(t) = Ax(t) + Bu(t)$ and $y(t) = Cx(t - d)$. The closed-loop state-space model is obtained by setting $u(t) = 1 - y(t)$, which gives $x'(t) = Ax(t) + BCx(t - d)$.

Delayed state is interpolated from past results with `interp1`. Note that values for $t < 0$ (extrapolated) are set to 0, and that values more recent than the last result are interpolated with the state passed to `f` for current `t`.

```
(A,B,C) = tf2ss(1,[1,1,0]);
d = 0.1;
x0 = zeros(length(A),1);
tmax = 10;
f = @(t,x,tpast,xpast) ...
     A*x+B*(1-C*interp1([tpast;t],[xpast;x.'],t-d,'1',0).');
(t,x) = ode45(f, [0,tmax], x0, Past=true);
```

Output y can be computed from the state:

```
y = C * interp1(t,x,t-d,'1',0).';
```

See also

`ode23`, `ode45`, `ode23s`, `optimset`, `interp1`

optimset

Options for minimization and zero finding.

Syntax

```
options = optimset
options = optimset(name1=value1, ...)
options = optimset(name1, value1, ...)
options = optimset(options0, name1=value1, ...)
options = optimset(options0, name1, value1, ...)
```

Description

`optimset(name1,value1,...)` creates the option argument used by `fminbnd`, `fminsearch`, `fzero`, `fsolve`, and other optimization functions. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Alternatively, options can be given with named arguments. Options which are not specified have a default value. The

result is a structure whose fields correspond to each option. Without any input argument, `optimset` creates a structure with all the default options. Note that `fminbnd`, `fminsearch`, and `fzero` also interpret the lack of an option argument, or the empty array `[]`, as a request to use the default values. Options can also be passed directly to `fminbnd` and other similar functions as named arguments.

When its first input argument is a structure, `optimset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options (empty arrays mean "automatic"):

Name	**Default**	**Meaning**
`Display`	`false`	detailed display
`MaxFunEvals`	`1000`	maximum number of evaluations
`MaxIter`	`500`	maximum number of iterations
`TolX`	`[]`	maximum relative error

The default value of `TolX` is `eps` for `fzero` and `sqrt(eps)` for `fminbnd` and `fminsearch`.

Examples

Default options:

```
optimset
  Display: false
  MaxFunEvals: 1000
  MaxIter: 500
  TolX: []
```

Display of the steps performed to find the zero of $\cos x$ between 1 and 2:

```
fzero(@cos, [1,2], optimset('Display',true))
  Checking lower bound
  Checking upper bound
  Inverse quadratic interpolation 2,1.5649,1
  Inverse quadratic interpolation 1.5649,1.571,2
  Inverse quadratic interpolation 1.571,1.5708,1.5649
  Inverse quadratic interpolation 1.5708,1.5708,1.571
  Inverse quadratic interpolation 1.5708,1.5708,1.571
ans =
  1.5708
```

See also

`fzero`, `fminbnd`, `fminsearch`, `lsqnonlin`, `lsqcurvefit`

quad

Numerical integration.

Syntax

```
y = quad(fun, a, b)
y = quad(fun, a, b, tol)
y = quad(fun, a, b, tol, trace)
y = quad(fun, a, b, tol, trace, ...)
```

Description

`quad(fun,a,b)` integrates numerically real function `fun` between a and `b`. `fun` is either specified by its name or given as an anonymous or inline function or a function reference.

The optional fourth argument is the requested relative tolerance of the result. It is either a positive real scalar number or the empty matrix (or missing argument) for the default value, which is `sqrt(eps)`. The optional fifth argument, if true or nonzero, makes quad displays information at each step.

Additional input arguments of quad are given as additional input arguments to function `fun`. They permit to parameterize the function.

Example

$$\int_0^2 te^{-t}\mathrm{d}t$$

```
quad(@(t) t*exp(-t), 0, 2)
  0.5940
```

Remark

Function `quad` is obsolete and should be replaced with `integral`, which supports named options and complex numbers.

See also

`integral`, operator `@`

10.26 String Functions

base32decode

Decode base32-encoded data.

Syntax

```
strb = base32decode(strt)
```

Description

`base32decode(strt)` decodes the contents of string `strt` which represents data encoded with base32. Characters which are not 'A'-'Z', '2'-'7', or '=' are ignored. Decoding stops at the end of the string or when '=' is reached.

See also

`base32encode`, `base64decode`

base32encode

Encode data using base32.

Syntax

```
strt = base32encode(strb)
```

Description

`base32encode(strb)` encodes the contents of string `strb` which represents binary data. The result contains only characters 'A'-'Z' and '2'-'7', and linefeed every 56 characters. It is suitable for transmission or storage on media which accept only uppercase letters and digits, without '0' or '1' easy to misinterpret as letters.

Each character of encoded data represents 5 bits of binary data; i.e. one needs eight characters for five bytes. The five bits represent 32 different values, encoded with the characters 'A' to 'Z' and '2' to '7' in this order. When the binary data have a length which is not a multiple of 5, encoded data are padded with 2, 3, 5 or 6 characters '=' to have a multiple of 8.

Base32 encoding is an Internet standard described in RFC 4648.

Example

```
s = base32encode(char(0:10))
  s =
    AAAQEAYEAUDAOCAJBI======
d = double(base32decode(s))
  d =
    0  1  2  3  4  5  6  7  8  9 10
```

See also

`base32decode`, `base64encode`

base64decode

Decode base64-encoded data.

Syntax

```
strb = base64decode(strt)
```

Description

`base64decode(strt)` decodes the contents of string `strt` which represents data encoded with base64. Characters which are not 'A'-'Z', 'a'-'z', '0'-'9', '+', '/', or '=' are ignored. Decoding stops at the end of the string or when '=' is reached.

See also

`base64encode`, `base32decode`

base64encode

Encode data using base64.

Syntax

```
strt = base64encode(strb)
```

Description

`base64encode(strb)` encodes the contents of string `strb` which represents binary data. The result contains only characters 'A'-'Z', 'a'-'z', '0'-'9', '+', '/', and '='; and linefeed every 60 characters. It is suitable for transmission or storage on media which accept only text.

Each character of encoded data represents 6 bits of binary data; i.e. one needs four characters for three bytes. The six bits represent 64 different values, encoded with the characters 'A' to 'Z', 'a' to 'z', '0' to '9', '+', and '/' in this order. When the binary data have a length which is not a multiple of 3, encoded data are padded with one or two characters '=' to have a multiple of 4.

Base64 encoding is an Internet standard described in RFC 2045.

Example

```
s = base64encode(char(0:10))
  s =
    AAECAwQFBgcICQo=
double(base64decode(s))
  0  1  2  3  4  5  6  7  8  9 10
```

See also

`base64decode`, `base32encode`

char

Convert an array to a character array (string).

Syntax

```
s = char(A)
S = char(s1, s2, ...)
```

Description

`char(A)` converts the elements of matrix `A` to characters, resulting in a string of the same size. Characters are stored in unsigned 16-bit words. The shape of `A` is preserved. Even if most functions ignore the string shape, you can force a row vector with `char(A(:).')`.

`char(s1,s2,...)` concatenates vertically the arrays given as arguments to produce a string matrix. If the strings do not have the same number of columns, blanks are added to the right.

Examples

```
char(65:70)
  ABCDEF
char([65, 66; 67, 68](:).')
  ABCD
char('ab','cde')
  ab
  cde
char('abc',['de';'fg'])
  abc
  de
  fg
```

See also

`setstr`, uint16, operator `:`, operator `.'`, `ischar`, `logical`, `double`, `single`

deblank

Remove trailing blank characters from a string.

Syntax

```
s2 = deblank(s1)
```

Description

deblank(s1) removes the trailing blank characters from string s1. Blank characters are spaces (code 32), tabulators (code 9), carriage returns (code 13), line feeds (code 10), and null characters (code 0).

Example

```
double(' \tAB  CD\r\n\0')
  32  9 65 66 32 32 67 68 13 10 0
double(deblank(' \tAB  CD\n\r\0')))
  32  9 65 66 32 32 67 68
```

See also

strtrim

hmac

HMAC authentication hash.

Syntax

```
hash = hmac(hashtype, key, data)
hash = hmac(hashtype, key, data, type=t)
```

Description

hmac(hashtype,key,data) calculates the authentication hash of data with secret key key and the method specified by hashtype: 'md5', 'sha1', 'sha224', 'sha256', 'sha384', or 'sha512'. Both arguments data and key can be strings (char arrays) which are converted to UTF-8, or int8 or uint8 arrays. The key can be up to 64 bytes; longer keys are truncated. The result is a string of hexadecimal digits whose length depends on the hash method, from 32 for HMAC-MD5 to 128 for HMAC-SHA512.

Named argument type can change the output type. It can be 'uint8' for an uint8 array of 16 or 20 bytes (raw HMAC-MD5 or HMAC-SHA1 hash result), 'hex' for its representation as a string of 32 or 40 hexadecimal digits (default), or base64 for its conversion to Base64 in a string of 24 or 28 characters.

HMAC is an Internet standard described in RFC 2104.

Examples

HMAC-MD5 of 'Authenticated message' using secret key 'secret':

```
hmac('md5', 'secret', 'Authenticated message')
  4f557b1f67bc4790e6e9568e2f458cf0
```

Same result computed explicitly, with the notations of RFC 2104: B is the block length, L is the hash length (16 for HMAC-MD5 or 20 for HMAC-SHA1), K is the key padded with zeros to have size B, and H is the hash function, defined here to produce a uint8 hash instead of an hexadecimal string like the LME functions `md5` or `sha1`.

```
B = 64;
L = 16;
H = @(a) uint8(sscanf(md5(a), '%2x')');
key = uint8('secret');
data = uint8('Authenticated message');
K = [key, zeros(1, B - length(key), 'uint8')];
hash = H([bitxor(K, 0x5cuint8), H([bitxor(K, 0x36uint8), data])]);
sprintf('%.2x', hash)
```

Simple implementation of the HOTP and TOTP password algorithms (RFC 4226 and 6238) often used for two-factor authentication, with their default parameter values. The password is assumed to be base32-encoded.

```
function n = hotp(pass, cnt)
  k = uint8(base32decode(pass));
  c = bwrite(cnt, 'uint64;b');
  // or c=bwrite([floor(c/2^32),mod(c,2^32)],'uint32;b');
  hs = hmac('sha1', k, c, type='uint8');
  ob = mod(hs(20), 16);
  dt = mod(sread(hs(ob + (1:4)), [], 'uint32;b'), 2^31);
  n = mod(dt, 1e6);
function n = totp(pass)
  t = floor(posixtime / 30);
  n = hotp(pass, t);
```

Simple implementation of the PBKDF2 key stretching algorithm (RFC 2898):

```
function dk = pbkdf2_hmac(hashtype, p, salt, c, dkLen)
  hLen = length(hmac(hashtype, '', '')) / 2;
  dk = uint8([]);
  for i = 1:ceil(dkLen / hLen)
    u = hmac(hashtype, p, [salt, bwrite(i, 'uint32;b')], type='uint8');
    f = u;
    for j = 2:c
      u = hmac(hashtype, p, u, type='uint8');
      f = bitxor(f, u);
    end
    dk = [dk, f];
  end
  dk = dk(1:dkLen);
```

Test of PBKDF2-HMAC-SHA1 with values provided in RFC 6070 (output format is switched to hexadecimal for easier comparison):

```
format int x
pbkdf2_hmac_sha1('sha1', 'password', 'salt', 4096, 20)
  0x4b  0x0 0x79  0x1 0xb7 0x65 0x48 0x9a 0xbe 0xad
  0x49 0xd9 0x26 0xf7 0x21 0xd0 0x65 0xa4 0x29 0xc1
format
```

See also

md5, sha1

ischar

Test for a string object.

Syntax

```
b = ischar(obj)
```

Description

`ischar(obj)` is true if the object `obj` is a character string, false otherwise. Strings can have more than one line.

Examples

```
ischar('abc')
  true
ischar(0)
  false
ischar([])
  false
ischar('')
  true
ischar(['abc';'def'])
  true
```

See also

isletter, isspace, isnumeric, islogical, isinteger, islist, isstruct, setstr, char

isdigit

Test for decimal digit characters.

Syntax

```
b = isdigit(s)
```

Description

For each character of string `s`, `isdigit(s)` is true if it is a digit (`'0'` to `'9'`) and false otherwise. The result is a logical array with the same size as the input argument.

Examples

```
isdigit('a123bAB12* ')
  F T T T F F F T T F F
```

See also

`isletter`, `isspace`, `lower`, `upper`, `ischar`

isletter

Test for letter characters.

Syntax

```
b = isletter(s)
```

Description

For each character of string `s`, `isletter(s)` is true if it is an ASCII letter (a-z or A-Z) and false otherwise. The result is a logical array with the same size as the input argument.

`isletter` gives false for letters outside the 7-bit ASCII range; `unicodeclass` should be used for Unicode-aware tests.

Examples

```
isletter('abAB12*')
  T T T T F F F F
```

See also

`isdigit`, `isspace`, `lower`, `upper`, `ischar`, `unicodeclass`

isspace

Test for space characters.

Syntax

```
b = isspace(s)
```

Description

For each character of string `s`, `isspace(s)` is true if it is a space, a tabulator, a carriage return or a line feed, and false otherwise. The result is a logical array with the same size as the input argument.

Example

```
isspace('a\tb c\nd')
  F T F T F T F
```

See also

`isletter`, `isdigit`, `ischar`

latex2mathml

Convert LaTeX equation to MathML.

Syntax

```
str = latex2mathml(tex)
str = latex2mathml(tex, mml1, mml2, ...)
str = latex2mathml(..., displaymath=b)
```

Description

`latex2mathml(tex)` converts LaTeX equation in string `tex` to MathML. LaTeX equations may be enclosed between dollars or double-dollars, but this is not mandatory. In string literals, backslash and tick characters must be escaped as \\ and \' respectively.

With additional arguments, which must be strings containing MathML, parameters #1, #2, ... in argument `tex` are converted to argument i+1.

The following LaTeX features are supported:

- variables (each letter is a separate variable)
- numbers (sequences of digit and dot characters)
- superscripts and subscripts, prime (single or multiple)
- braces used to group subexpressions or specify arguments with more than one token
- operators (+, -, comma, semicolon, etc.)

- control sequences for character definitions, with greek characters in lower case (\alpha, ..., \omega, \varepsilon, \vartheta, \varphi) and upper case (\Alpha, ..., \Omega), arrows (\leftarrow or \gets, \rightarrow or \to, \uparrow, \downarrow, \leftrightarrow, \updownarrow, \Leftarrow, \Rightarrow, \Uparrow, \Downarrow, \Leftrightarrow, \Updownarrow, \nwarrow, \nearrow, \searrow, \swarrow, \mapsto, \hookleftarrow, \hookrightarrow, \Longleftrightarrow, \longmapsto), and symbols (\|, \ell, \partial, \infty, \emptyset, \nabla, \perp, \angle, \triangle, \backslash, \forall, \exists, \flat, \natural, \sharp, \pm, \mp, \cdot, \times, \star, \diamond, \cap, \cup, etc.)

- \not followed by comparison operator, such as \not< or \not\approx

- control sequences for function definitions (\arccos, \arcsin, \arctan, \arg, \cos, \cosh, \cot, \coth, \csc, \deg, \det, \dim, \exp, \gcd, \hom, \inf, \injlim, \ker, \lg, \liminf, \limsup, \ln, \log, \max, \min, \Pr, \projlim, \sec, \sin, \sinh, \sup, \tan, \tanh) and custom functions with \operatorname

- accents (\hat, \check, \tilde, \acute, \grave, \dot, \ddot, \dddot, \breve, \bar, \vec, \overline, \widehat, \widetilde, \underline)

- \left and \right

- fractions with \frac or \over

- roots with \sqrt (without optional radix) or \root...\of...

- \atop, \overset, and \underset

- large operators (\bigcap, \bigcup, \bigodot, \bigoplus, \bigotimes, \bigsqcup, \biguplus, \bigvee, \bigwedge, \coprod, \prod, and \sum with implicit \limits for limits below and above the symbol; and \int, \iint, \iiint, \iiiint, \oint, and \oiint with implicit \nolimits for limits to the right of the symbol)

- \limits and \nolimits for functions and large operators

- matrices with \matrix, \pmatrix, \bmatrix, \Bmatrix, \vmatrix, \Vmatrix, \begin{array}{...}.../\end{array}; values are separated with & and rows with \cr or \\

- font selection with \rm for roman, \bf for bold face, and \mit for math italic

- color with `\color{`*c*`}` where *c* is `black`, `red`, `green`, `blue`, `cyan`, `magenta`, `yellow`, `white`, `orange`, `violet`, `purple`, `brown`, `darkgray`, `gray`, or `lightgray`
- hidden element with `\phantom`
- text with `\hbox{...}` (brace contents is taken verbatim)
- horizontal spaces with `\,` `\:` `\;` `\quad` `\qquad` and `\!`

LaTeX features not enumerated above, such as definitions and nested text and equations, are not supported.

`latex2mathml` has also features which are missing in LaTeX. Unicode is used for both LaTeX input and MathML output. Some semantics is recognized to build subexpressions which are revealed in the resulting MathML. For instance, in `x+(y+z)w`, `(y+z)` is a subpexpressions; so is `(y+z)w` with an implicit multiplication (resulting in the `<mo>&it;<mo>` MathML operator), used as the second operand of the addition. LaTeX code (like mathematical notation) is sometimes ambiguous and is not always converted to the expected MathML (e.g. `a(b+c)` is converted to a function call while the same notation could mean the product of a and b+c), but this should not have any visible effect when the MathML is typeset.

Operators can be used as freely as in LaTeX. Missing operands result in `<none/>`, as if there were an empty pair of braces `{}`. Consecutive terms are joined with implicit multiplications.

Named argument `displaymath` specifies whether the vertical space is tight, like in inline equations surrounded by text (`false`), or unconstrained, as rendered in separate lines (`true`). It affects the position of some limits. The default is `true`.

Examples

```
latex2mathml('xy^2')
  <mrow><mi>x</mi><mo>&it;</mo><msup><mi>y</mi><mn>2</mn></msup></mr
mml = latex2mathml('\\frac{x_3+5}{x_1+x_2}');
mml = latex2mathml('$\\root n \\of x$');
mml = latex2mathml('\\pmatrix{x & \\sqrt y \\cr \\sin\\phi & \\hat\\ell}');
mml = latex2mathml('\\dot x = #1', mathml([1,2;3,0], false));
mml = latex2mathml('\\lim_{x \\rightarrow 0} f(x)', displaymath=true)
mml = latex2mathml('\\lim_{x \\rightarrow 0} f(x)', displaymath=false)
```

See also

`mathml`

lower

Convert all uppercase letters to lowercase.

Syntax

```
s2 = lower(s1)
```

Description

`lower(s1)` converts all the uppercase letters of string `s1` to lowercase, according to the Unicode Character Database.

Example

```
lower('abcABC123')
  abcabc123
```

See also

`upper`, `isletter`

mathml

Conversion to MathML.

Syntax

```
str = mathml(x)
str = mathml(x, false)
str = mathml(..., Format=f, NPrec=n)
```

Description

`mathml(x)` converts its argument `x` to MathML presentation, returned as a string.

By default, the MathML top-level element is `<math>`. If the result is to be used as a MathML subelement of a larger equation, a second input argument equal to the logical value `false` can be specified to suppress `<math>`.

By default, `mathml` converts numbers like format `'%g'` of `sprintf`. Named arguments can override them: `Format` is a single letter format recognized by `sprintf` and `NPrec` is the precision (number of decimals).

Example

```
mathml(pi)
  <math>
  <mn>3.1416</mn>
  </math>
mathml(1e-6, Format='e', NPrec=2)
  <math>
  <mrow><mn>1.00</mn><mo>&CenterDot;</mo><msup><mn>10</mn><mn>-6</m
  </math>
```

See also

`mathmlpoly`, `latex2mathml`, `sprintf`

mathmlpoly

Conversion of a polynomial to MathML.

Syntax

```
str = mathmlpoly(pol)
str = mathmlpoly(pol, var)
str = mathmlpoly(..., power)
str = mathmlpoly(..., false)
str = mathmlpoly(..., Format=f, NPrec=n)
```

Description

`mathmlpoly(coef)` converts polynomial coefficients `pol` to MathML presentation, returned as a string. The polynomial is given as a vector of coefficients, with the highest power first; e.g., $x^2 + 2x - 3$ is represented by `[1,2,-3]`.

By default, the name of the variable is *x*. An optional second argument can specify another name as a string, such as `'y'`, or a MathML fragment beginning with a less-than character, such as `'<mn>3</mn>'`.

Powers can be specified explicitly with an additional argument, a vector which must have the same length as the polynomial coefficients. Negative and fractional numbers are allowed; the imaginary part, if any, is ignored.

By default, the MathML top-level element is `<math>`. If the result is to be used as a MathML subelement of a larger equation, an additional input argument (the last unnamed argument) equal to the logical value `false` can be specified to suppress `<math>`.

Named arguments `format` and `NPrec` have the same effect as with `mathml`.

Examples

Simple third-order polynomial:

```
mathmlpoly([1,2,5,3])
```

Polynomial with negative powers of variable q:

```
c = [1, 2.3, 4.5, -2];
mathmlpoly(c, 'q', -(0:numel(c)-1))
```

Rational fraction:

```
str = sprintf('<mfrac>%s%s</mfrac>',
  mathmlpoly(num, false),
  mathmlpoly(den, false));
```

See also
`mathml`

md5

Calculate MD5 digest.

Syntax

```
digest = md5(strb)
digest = md5(fd)
digest = md5(..., type=t)
```

Description

`md5(strb)` calculates the MD5 digest of `strb` which represents binary data. `strb` can be a string (only the least-significant byte of each character is considered) or an array of bytes of class `uint8` or `int8`. The result is a string of 32 hexadecimal digits. It is believed to be hard to create the input to get a given digest, or to create two inputs with the same digest.

`md5(fd)` calculates the MD5 digest of the bytes read from file descriptor `fd` until the end of the file. The file is left open.

Named argument `type` can change the output type. It can be `'uint8'` for an uint8 array of 16 bytes (raw MD5 hash result), `'hex'` for its representation as a string of 32 hexadecimal digits (default), or `base64` for its conversion to Base64 in a string of 24 characters.

MD5 digest is an Internet standard described in RFC 1321.

Examples

MD5 of the three characters 'a', 'b', and 'c':

```
md5('abc')
  900150983cd24fb0d6963f7d28e17f72
```

This can be compared to the result of the command tool `md5` found on many unix systems:

```
$ echo -n abc | md5
900150983cd24fb0d6963f7d28e17f72
```

The following statements calculate the digest of the file `'somefile'`:

```
fd = fopen('somefile');
digest = md5(fd);
fclose(fd);
```

See also

`sha1`, `hmac`

regexp regexpi

Regular expression match.

Syntax

```
(startIx, endIx, length, grExt) = regexp(str, re)
(startIx, endIx, grExt) = regexpi(str, re)
```

Description

`regexp(str,re)` matches regular expression `re` in string `str`. A regular expression is a string which contains meta-characters to match classes of characters, repetitions and alternatives, as described below.

Once a match is found, the remaining part of `str` is parsed from the end of the previous match to find more matches. The result of `regexp` is an array of start indices in `str` and an array of corresponding end indices. Empty matches have a length `endIx-startIx-1=0`.

The third output argument, if present, is set to a list whose items correspond to matches. Items are arrays of size 2-by-ng. Each row corresponds to a group, i.e. a subexpression in parentheses in the regular expression; the first column contains the index of the first character in `str` and the second column contains the index of the last character.

`regexpi` is similar to `regexp`, except that letter case is ignored.

The following regular expression elements are recognized:

Any character other than those described below Literal match.

`.` **(dot)** Any character.

`\0` Nul (0).

`\t` Tab (9).

`\n` Newline (10).

`\v` Vertical tab (11).

`\f` Form feed (12).

`\r` Carriage return (13).

`\P` **where** `P` **is one of** `\()[]{}?*+/` `P`

`\xNN` Character whose code is NN in hexadecimal.

`\uNNNN` Character whose code is NNNN in hexadecimal.

`[...]` Any of the characters in brackets. Characters can be enumarated (e.g. `[ax2]` to match a, x or 2), provided as ranges with a hyphen (e.g. `[a-c]` to match a, b or c) or any combination. Caret `^` must not appear first; closing bracket `]` must appear first; and hyphen must not be used in a way which could be interpreted as a range.

`[^...]` Any character not enumated in brackets (e.g. `[^a-z]` for any character except for lowercase letters).

`AB` Catenation of `A` and `B`.

`A|B` One of `A` or `B`. `|` has the lowest priority: `ab|c` matches ab or c.

`A?` `A` (if possible) or nothing.

`A*` As many repetitions of `A` as possible, including none.

`A+` As many repetitions of `A` as possible, at least one.

`A{n}` Exactly `n` repetitions of `A`.

`A{n,}` At least `n` repetitions of `A` (as many as possible).

`A{n,m}` Between `n` and `m` repetitions of `A` (as many as possible).

`A??` Nothing (if possible) or `A`.

`A*?` As few repetitions of `A` as possible, including none.

`A+?` As few repetitions of `A` as possible, at least one.

`A{n,}?` At least `n` repetitions of `A` (as few as possible).

`A{n,m}?` Between `n` and `m` repetitions of `A` (as few as possible).

`A?+`**,** `A*?`**,** `A++`**,** `A{...}+` Possessive repetitions: as many as possible, but once the maximum number has been found, does not try less repetitions should the remaining part of the regular expression fail to match anything.

`(A)` Group; matches subexpression `A`, which is captured for further reference as `\N`.

`(?:A)` Group without capture; just matches subexpression `A`.

`\N` **where** `N` **is a digit from 1 to 9** Character substring which was matched by the N:th group delimited by parentheses.

`^` Matches beginning of string.

`$` Matches end of string.

`\b` Beginning or end of word.

`(?=A)` Positive lookahead: succeeds if what follows matches `A` without consuming `A`.

`(?!A)` Negative lookahead: succeeds if what follows does not match `A` without consuming `A`.

`(?# comment)` Comment (ignored).

`\d` Digit (can be used inside or outside brackets).

`\D` Not a digit (can be used inside or outside brackets).

`\s` White space (can be used inside or outside brackets).

`\S` Not white space (can be used inside or outside brackets).

`\w` Alphanumeric (can be used inside or outside brackets).

`\W` Not alphanumeric (can be used inside or outside brackets).

`[:alnum:]` Same as `A-Za-z0-9` (must be used inside brackets, e.g. `[[:alnum:]]`)

`[:alpha:]` Same as `A-Za-z` (must be used inside brackets, e.g. `[[:alpha:]]`)

`[:blank:]` Same as `\x20\x09`, i.e. space or tab (must be used inside brackets, e.g. `[[:blank:]]`)

`[:cntrl:]` Same as `\0-\x1f` (must be used inside brackets, e.g. `[[:cntrl:]]`)

`[:digit:]` Same as `0-9` (must be used inside brackets, e.g. `[[:digit:]]`)

`[:graph:]` Same as `\x21-\x7e`, i.e. ASCII characters without space and control characters (must be used inside brackets, e.g. `[[:graph:]]`)

`[:lower:]` Same as `a-z` (must be used inside brackets, e.g. `[^[:lower:][:digit:]]` which is equivalent to `[^a-z0-9]`)

`[:print:]` Same as `\x20-\x7e`, i.e. ASCII characters without control characters (must be used inside brackets, e.g. `[[:print:]]`)

`[:punct:]` Same as `!"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~` (must be used inside brackets, e.g. `[[:punct:]]`)

`[:space:]` Same as `\x20\x09\x0a\x0c\x0d` or `\s` (must be used inside brackets, e.g. `[[:space:]]`)

`[:upper:]` Same as `A-Z` (must be used inside brackets, e.g. `[[:upper:]]`)

`[:word:]` Same as `[:alnum:]_` (must be used inside brackets, e.g. `[[:word:]]`)

`[:xdigit:]` Same as `0-9A-Fa-f` (must be used inside brackets, e.g. `[[:xdigit:]]`)

Quantifiers ?, ∗ and +, and their lazy and possessive versions (suffixed with ? or + respectively) have the highest priority. Priority can be changed with parentheses, e.g. `(abc)∗` or `(a|bc)d`.

Examples

Simple match without metacharacter:

```
(startIx, endIx) = regexp('Some random string', 'om')
  startIx =
     2 10
  endIx =
     3 11
```

Dot to match any character:

```
regexp('Some random string', 'S..e')
  1
```

Anchor to end of string:

```
regexp('Some random string', '..$')
  17
```

Repetition:

```
regexp('Some random string', 'r.∗m')
  6
```

By default, repetitions are greedy (as many as possible):

```
(startIx, endIx) = regexp('Some random string', '.∗m')
  startIx =
     1
  endIx =
    11
```

Lazy repetition (as few as possible):

```
(startIx, endIx) = regexp('Some random string', '.∗?m')
  startIx =
     1  4
  endIx =
     3 11
```

Possessive repetitions keep the largest number of repetitions which provides a match regardless of subsequent failures:

```
(startIx, endIx) = regexp('Some random string', '.*m ')
  startIx =
     1
  endIx =
    12
(startIx, endIx) = regexp('Some random string', '.*+m ')
  startIx =
    []
  endIx =
    []
```

Since backslash is an escape character in LME strings, it must be escaped itself:

```
(startIx, endIx) = regexp('Some random string', '\\b\\w.+?\\b')
  startIx =
    1  6 13
  endIx =
    4 11 18
```

Reference to a captured group:

```
(startIx, endIx) = regexp('xx-ab-ab', '(.+)-\\1')
  startIx =
    4
  endIx =
    8
```

Positive lookahead to find words followed by a colon without picking the colon itself:

```
(startIx, endIx) = regexp('mailto:foo@example.com', '\\b\\w+(?=:)')
  startIx =
    1
  endIx =
    6
```

Group (the extent of the whole match is ignored using placeholder output arguments ~):

```
(~, ~, grExt) = regexp('Regexp are fun', '\\b(\\w+)\\s+(\\w+)\\s+(\\w+)\\b');
grExt{1}
  1  6
  8 10
 12 14
```

Match ignoring case:

```
regexpi('Some random string', 'some')
  1
```

Case-explicit character classes are still case-significant, but character enumerations or ranges are not:

```
regexpi('Some random string', '^[[:lower:]]')
  []
regexpi('Some random string', '^[a-z]')
  1
```

See also

`strfind`, `strtok`

setstr

Conversion of an array to a string.

Syntax

```
str = setstr(A)
```

Description

`setstr(A)` converts the elements of array `A` to characters, resulting in a string of the same size. Characters are stored in unsigned 16-bit words.

Example

```
setstr(65:75)
  ABCDEFGHIJK
```

See also

`char`, `uint16`, `logical`, `double`

sha1 sha2

Calculate SHA-1 or SHA-2 digest.

Syntax

```
digest = sha1(strb)
digest = sha1(fd)
digest = sha1(..., type=t)
digest = sha2(...)
digest = sha2(..., variant=v)
```

Description

`sha1(strb)` calculates the SHA-1 digest of `strb` which represents binary data. `strb` can be a string (only the least-significant byte of each character is considered) or an array of bytes of class `uint8` or `int8`. The result is a string of 40 hexadecimal digits. It is believed to be hard to create the input to get a given digest, or to create two inputs with the same digest.

`sha1(fd)` calculates the SHA-1 digest of the bytes read from file descriptor `fd` until the end of the file. The file is left open.

Named argument `type` can change the output type. It can be `'uint8'` for an uint8 array of 20 bytes (raw SHA-1 hash result), `'hex'` for its representation as a string of 40 hexadecimal digits (default), or `base64` for its conversion to Base64 in a string of 28 characters.

SHA-1 digest is an Internet standard described in RFC 3174.

`sha2` calculates the SHA-256 digest, a 256-bit variant of the SHA-2 hash algorithm. Its arguments are the same as those of `sha1`. In addition, named argument `variant` can specify one of the supported SHA-2 variants: 224, 256 (default), 384, or 512.

Example

SHA-1 digest of the three characters `'a'`, `'b'`, and `'c'`:

```
sha1('abc')
  a9993e364706816aba3e25717850c26c9cd0d89d
```

SHA-224 digest of the empty message `''`:

```
sha2('', variant=224)
  d14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f
```

See also

`md5`, `hmac`

split

Split a string.

Syntax

```
L = split(string, separator)
```

Description

`split(string,separator)` finds substrings of `string` separated by `separator` and return them as a list. Empty substring are discarded. `sepatator` is a string of one or more characters.

Examples

```
split('abc;de;f', ';')
  {'abc', 'de', 'f'}
split('++a+++b+++','++')
  {'a', '+b', '+'}
```

See also

`strfind`

strcmp

String comparison.

Syntax

```
b = strcmp(s1, s2)
b = strcmp(s1, s2, n)
```

Description

`strcmp(s1, s2)` is true if the strings `s1` and `s2` are equal (i.e. same length and corresponding characters are equal). `strcmp(s1, s2, n)` compares the strings up to the n:th character. Note that this function does not return the same result as the `strcmp` function of the standard C library.

Examples

```
strcmp('abc','abc')
  true
strcmp('abc','def')
  false
strcmp('abc','abd',2)
  true
strcmp('abc','abd',5)
  false
```

See also

`strcmpi`, operator ===, operator ~==, operator ==, `strfind`, `strmatch`

strcmpi

String comparison with ignoring letter case.

Syntax

```
b = strcmpi(s1, s2)
b = strcmpi(s1, s2, n)
```

Description

`strcmpi` compares strings for equality, ignoring letter case. In every other respect, it behaves like `strcmp`.

Examples

```
strcmpi('abc','aBc')
  true
strcmpi('Abc','abd',2)
  true
```

See also

`strcmp`, operator ===, operator ~==, operator ==, `strfind`, `strmatch`

strfind

Find a substring in a string.

Syntax

```
pos = strfind(str, sub)
```

Description

`strfind(str,sub)` finds occurrences of string `sub` in string `str` and returns a vector of the positions of all occurrences, or the empty vector `[]` if there is none. Occurrences may overlap.

Examples

```
strfind('ababcdbaaab','ab')
  1 3 10
strfind('ababcdbaaab','ac')
  []
strfind('aaaaaa','aaa')
  1 2 3
```

See also

`find`, `strcmp`, `strrep`, `split`, `strmatch`, `strtok`

strmatch

String match.

Syntax

```
i = strmatch(str, strMatrix)
i = strmatch(str, strList)
i = strmatch(..., 'exact')
```

Description

`strmatch(str,strMatrix)` compares string `str` with each row of the character matrix `strMatrix`; it returns the index of the first row whose beginning is equal to `str`, or 0 if no match is found. Case is significant.

`strmatch(str,strList)` compares string `str` with each element of list or cell array `strList`, which must be strings.

With a third argument, which must be the string `'exact'`, `str` must match the complete row or element of the second argument, not only the beginning.

Examples

```
strmatch('abc',['axyz';'uabc';'abcd';'efgh'])
  3
strmatch('abc',['axyz';'uabc';'abcd';'efgh'],'exact')
  0
strmatch('abc',{'ABC','axyz','abcdefg','ab','abcd'})
  3
```

See also

`strcmp`, `strfind`

strrep

Replace a substring in a string.

Syntax

```
newstr = strrep(str, sub, repl)
```

Description

`strrep(str,sub,repl)` replaces all occurrences of string `sub` in string `str` with string `repl`.

Examples

```
strrep('ababcdbaaab','ab','X')
  'XXcdbaaX'
strrep('aaaaaaa','aaa','12345')
  '1234512345a'
```

See also

`strfind`, `strcmp`, `strmatch`, `strtok`

strtok

Token search in string.

Syntax

```
(token, remainder) = strtok(str)
(token, remainder) = strtok(str, separators)
```

Description

`strtok(str)` gives the first token in string `str`. A token is defined as a substring delimited by separators or by the beginning or end of the string; by default, separators are spaces, tabulators, carriage returns and line feeds. If no token is found (i.e. if `str` is empty or contains only separator characters), the result is the empty string.

The optional second output is set to what follows immediately the token, including separators. If no token is found, it is the same as `str`.

An optional second input argument contains the separators in a string.

Examples

Strings are displayed with quotes to show clearly the separators.

```
strtok(' ab cde ')
  'ab'
(t, r) = strtok(' ab cde ')
  t =
    'ab'
  r =
    ' cde '
(t, r) = strtok('2, 5, 3')
  t =
    '2'
  r =
    ', 5, 3'
```

See also

`strmatch`, `strfind`, `strtrim`

strtrim

Remove leading and trailing blank characters from a string.

Syntax

```
s2 = strtrim(s1)
```

Description

`strtrim(s1)` removes the leading and trailing blank characters from string `s1`. Blank characters are spaces (code 32), tabulators (code 9), carriage returns (code 13), line feeds (code 10), and null characters (code 0).

Example

```
double(' \tAB  CD\r\n\0')
  32  9 65 66 32 32 67 68 13 10 0
double(strtrim(' \tAB  CD\n\r\0')))
  65 66 32 32 67 68
```

See also

`deblank`, `strtok`

unicodeclass

Unicode character class.

Syntax

```
cls = unicodeclass(c)
```

Description

`unicodeclass(c)` gives the Unicode character class (General_Category property in the Unicode Character Database) of its argument `c`, which must be a single-character string. The result is one of the following two-character strings:

Class	Description	Class	Description
`'Lu'`	Letter, Uppercase	`'Pi'`	Punctuation, Initial qupte
`'Ll'`	Letter, Lowercase	`'Pf'`	Punctuation, Final Quote
`'Lt'`	Letter, Titlecase	`'Po'`	Punctuation, Other
`'Lm'`	Letter, Modifier	`'Sm'`	Symbol, Math
`'Lo'`	Letter, Other	`'Sc'`	Symbol, Currency
`'Mn'`	Mark, Non-Spcacing	`'Sk'`	Symbol, Modifier
`'Mc'`	Mark, Spacing Combining	`'So'`	Symbol, Other
`'Me'`	Mark, Enclosing	`'Zs'`	Separator, Spcace
`'Nd'`	Number, Decimal Digit	`'Zl'`	Separator, Line
`'Nl'`	Number, Letter	`'Zp'`	Separator, Paragraph
`'No'`	Number, Other	`'Cc'`	Other, Control
`'Pc'`	Punctuation, Connector	`'Cf'`	Other, Format
`'Pd'`	Punctuation, Dash	`'Cs'`	Other, Surrogate
`'Ps'`	Punctuation, Open	`'Co'`	Other, Private Use
`'Pe'`	Punctuation, Close	`'Cn'`	Other, Not Assigned

See also

`isletter`, `isdigit`, `isspace`

upper

Convert all lowercase letters to lowercase.

Syntax

```
s2 = upper(s1)
```

Description

`upper(s1)` converts all the lowercase letters of string `s1` to uppercase, according to the Unicode Character Database.

Example

```
upper('abcABC123')
  ABCABC123
```

See also

`lower`, `isletter`

utf32decode

Decode Unicode characters encoded with UTF-32.

Syntax

```
str = utf32decode(b)
```

Description

`utf32decode(b)` decodes the contents of uint32 or int32 array `b` which represents Unicode characters encoded with UTF-32 (basically, Unicode code point). The result is a standard character array with a single row, usually encoded with UTF-16. Invalid codes are ignored.

If all the codes in `b` correspond to the Basic Multilingual Plane (16-bits, and not surrogate 0xd800-0xdfff), the result is equivalent to `char(b)`.

See also

`utf32encode`, `utf8decode`

utf32encode

Encode a string of Unicode characters using UTF-32.

Syntax

```
b = utf32encode(str)
```

Description

`utf32encode(str)` encodes the contents of character array `str` using UTF-32. Each Unicode character in `str`, made of 1 or 2 UTF-16 words, corresponds to one UTF-32 code. The result is an array of unsigned 32-bit integers.

If all the characters in `str` correspond to the Basic Multilingual Plane (16-bits, and no surrogate pairs), the result is equivalent to `uint32(str)`.

Examples

```
utf32encode('abc')
  1x3 uint32 array
    97  98  99
str = utf32decode(65872uint32);
double(str)
  55296 56656
utf32encode(str)
  65872uint32
```

See also

`utf32decode`, `utf8encode`

utf8decode

Decode Unicode characters encoded with UTF-8.

Syntax

```
str = utf8decode(b)
```

Description

`utf8decode(b)` decodes the contents of uint8 or int8 array `b` which represents Unicode characters encoded with UTF-8. Each Unicode character corresponds to up to 4 bytes of UTF-8 code. The result is a standard character array with a single row; characters are usually encoded as UTF-16, with 1 or 2 words per character. Invalid codes (for example when the beginning of the decoded data does not correspond to a character boundary) are ignored.

See also

`utf8encode`, `utf32decode`

utf8encode

Encode a string of Unicode characters using UTF-8.

Syntax

```
b = utf8encode(str)
```

Description

`utf8encode(str)` encodes the contents of character array `str` using UTF-8. Each Unicode character in `str` corresponds to up to 4 bytes of UTF-8 code. The result is an array of unsigned 8-bit integers.

If the input string does not contain Unicode characters, the output is invalid.

Example

```
b = utf8encode(['abc', 200, 2000, 20000])
  b =
    1x10 uint8 array
      97  98  99 195 136 223 144 228 184 160
str = utf8decode(b);
double(str)
  97    98    99   200  2000 20000
```

See also

`utf8decode`, `utf32encode`

10.27 Quaternions

Quaternion functions support scalar and arrays of quaternions. Basic arithmetic operators and functions are overloaded to support expressions with the same syntax as for numbers and matrices.

Quaternions are numbers similar to complex numbers, but with four components instead of two. The unit imaginary parts are named i, j, and k. A quaternion can be written $w + ix + jy + kz$. The following relationships hold:

$$i^2 = j^2 = k^2 = ijk = -1$$

It follows that the product of two quaternions is not commutative; for instance, $ij = k$ but $ji = -k$.

Quaternions are convenient to represent arbitrary rotations in the 3d space. They are more compact than matrices and are easier to normalize. This makes them suitable for simulation and control of mechanical systems and vehicles, such as flight simulators and robotics.

Functions below are specific to quaternions:

Function	Purpose
`isquaternion`	test for quaternion type
`q2mat`	conversion to rotation matrix
`q2rpy`	conversion to attitude angles
`q2str`	conversion to string
`qimag`	imaginary parts
`qinv`	element-wise inverse
`qnorm`	scalar norm
`qslerp`	spherical linear interpolation
`quaternion`	quaternion creation
`rpy2q`	conversion from attitude angles

Operators below accept quaternions as arguments:

Function	Operator	Purpose
`ctranspose`	`'`	conjugate transpose
`eq`	`==`	element-wise equality
`horzcat`	`[,]`	horizontal array concatenation
`ldivide`	`.\`	left division
`ne`	`~=`	element-wise inequality
`minus`	`-`	difference
`mldivide`	`\`	matrix left division
`mrdivide`	`/`	matrix right division
`mtimes`	`*`	matrix multiplication
`plus`	`+`	addition
`rdivide`	`./`	division
`times`	`.*`	multiplication
`transpose`	`.'`	transpose
`uminus`	`-`	unary minus
`uplus`	`+`	unary plus
`vertcat`	`[;]`	vertical array concatenation

Most of these operators work as expected, like with complex scalars and matrices. Multiplication and left/right division are not commutative. Matrix operations are not supported: operators `*`, `/`, `\`, and `^` are defined as a convenience (they are equivalent to `.*`, `./`, `.\`, and `.^` respectively) and work only element-wise with scalar arguments.

Mathematical functions below accept quaternions as arguments; with arrays of quaternions, they are applied to each element separately.

Function	Purpose
`abs`	absolute value
`conj`	conjugate
`cos`	cosine
`exp`	exponential
`log`	natural logarithm
`real`	real part
`sign`	quaternion sign (normalization)
`sin`	sine
`sqrt`	square root

Functions below performs computations on arrays of quaternions.

Function	Purpose
`cumsum`	cumulative sum
`diff`	differences
`double`	conversion to array of double
`mean`	arithmetic mean
`sum`	sum

Functions below are related to array size.

Function	Purpose
`beginning`	first subscript
`cat`	array concatenation
`end`	last subscript
`flipdim`	flip array
`fliplr`	flip left-right
`flipud`	flip upside-down
`ipermute`	dimension inverse permutation
`isempty`	test for empty array
`length`	length of vector
`ndims`	number of dimensions
`numel`	number of elements
`permute`	dimension permutation
`repmat`	array replication
`reshape`	array reshaping
`rot90`	array rotation
`size`	array size
`squeeze`	remove singleton dimensions

Finally, functions below are related to output and assignment.

Function	Purpose
`disp`	display
`dumpvar`	conversion to string
`subsasgn`	assignment to subarrays or to quaternion parts
`subsref`	reference to subarrays or to quaternion parts

Function `imag` is replaced with `qimag` which gives a quaternion with

the real part set to zero, because there are three imaginary components instead of one with complex numbers.

Operators and functions which accept multiple arguments convert automatically double arrays to quaternions, ignoring the imaginary part of complex numbers.

Conversion to numeric arrays with `double` adds a dimension for the real part and the three imaginary parts. For example, converting a scalar quaternion gives a 4-by-1 double column vector and converting a 2-by-2 quaternion array gives a 2-by-2-by-4 double array. Real and imaginary components can be accessed with the field access notation: `q.w` is the real part of `q`, `q.x`, `q.y`, and `q.z` are its imaginary parts, and `q.v` is its imaginary parts as an array similar to the result of `double` but without the real part.

isquaternion

Test for a quaternion.

Syntax

```
b = isquaternion(q)
```

Description

`isquaternion(q)` is true if the input argument is a quaternion and false otherwise.

Examples

```
isquaternion(2)
  false
isquaternion(quaternion(2))
  true
```

See also

`quaternion`, `isnumeric`

q2mat

Conversion from quaternion to rotation matrix.

Syntax

```
R = q2mat(q)
```

Description

`R=q2mat(q)` gives the 3x3 orthogonal matrix `R` corresponding to the rotation given by scalar quaternion `q`. For a vector `a=[x;y;z]` and its representation as a pure quaternion `aq=quaternion(x,y,z)`, the rotation can be performed with quaternion multiplication `bq=q*aq/q` or matrix multiplication `b=R*a`.

Input argument `q` does not have to be normalized; a quaternion corresponding to a given rotation is defined up to a multiplicative factor.

Example

```
q = rpy2q(0.1, 0.3, 0.2);
R = q2mat(q)
  R =
    0.9363 -0.1688  0.3080
    0.1898  0.9810  0.0954
   -0.2955  0.0954  0.9506
aq = quaternion(1, 2, 3);
q * aq / q
  1.5228i+2.0336j+2.7469k
a = [1; 2; 3];
R * a
  1.5228
  2.4380
  2.7469
```

See also

`q2rpy`, `rpy2q`, `quaternion`

q2rpy

Conversion from quaternion to attitude angles.

Syntax

```
(pitch, roll, yaw) = q2rpy(q)
```

Description

`q2rpy(q)` gives the pitch, roll, and yaw angles corresponding to the rotation given by quaternion `q`. It is the inverse of `rpy2q`. All angles are given in radians.

If the input argument is a quaternion array, the results are arrays of the same size; conversion from quaternion to angles is performed independently on corresponding elements.

See also

`rpy2q`, `q2mat`, `quaternion`

q2str

Conversion from quaternion to string.

Syntax

```
str = q2str(q)
```

Description

`q2str(q)` converts quaternion `q` to its string representation, with the same format as `disp`.

See also

`quaternion`, `format`

qimag

Quaternion imaginary parts.

Syntax

```
b = qimag(q)
```

Description

`qimag(q)` gives the imaginary parts of quaternion `q` as a quaternion, i.e. the same quaternion where the real part is set to zero. `real(q)` gives the real part of quaternion `q` as a double number.

Example

```
q = quaternion(1,2,3,4)
  q =
    1+2i+3j+4k
real(q)
  1
qimag(q)
  2i+3j+4k
```

See also

`quaternion`

qinv

Quaternion element-wise inverse.

Syntax

```
b = qinv(q)
```

Description

`qinv(q)` gives the inverse of quaternion `q`. If its input argument is a quaternion array, the result is an quaternion array of the same size whose elements are the inverse of the corresponding elements of the input.

The inverse of a normalized quaternion is its conjugate.

Example

```
q = quaternion(0.4,0.1,0.2,0.2)
  q =
    0.4+0.1i+0.2j+0.2k
p = qinv(q)
  p =
    1.6-0.4i-0.8j-0.8k
abs(q)
  0.5
abs(p)
  2
```

See also

`quaternion`, `qnorm`, `conj`

qnorm

Quaternion scalar norm.

Syntax

```
n = qnorm(q)
```

Description

`qnorm(q)` gives the norm of quaternion `q`, i.e. the sum of squares of its components, or the square of its absolute value. If `q` is an array of quaternions, `qnorm` gives a double array of the same size where each element is the norm of the corresponding element of `q`.

See also

`quaternion`, `abs`

qslerp

Quaternion spherical linear interpolation.

Syntax

```
q = qslerp(q1, q2, t)
```

Description

`qslerp(q1,q2,t)` performs spherical linear interpolation between quaternions `q1` and `q2`. The result is on the smallest great circle arc defined by normalized `q1` and `q2` for values of real number `t` between 0 and 1.

If `q1` or `q2` is 0, the result is NaN. If they are opposite, the great circle arc going through 1, or 1i, is picked.

If input arguments are arrays of compatible size (same size or scalar), the result is a quaternion array of the same size; conversion from angles to quaternion is performed independently on corresponding elements.

Example

```
q = qslerp(1, rpy2q(0, 1, -1.5), [0, 0.33, 0.66, 1]);
(roll, pitch, yaw) = q2rpy(q)
  roll =
    0.0000  0.1843  0.2272  0.0000
  pitch =
    0.0000  0.3081  0.6636  1.0000
  yaw =
    0.0000 -0.4261 -0.8605 -1.5000
```

See also

`quaternion`, `rpy2q`, `q2rpy`

quaternion

Quaternion creation.

Syntax

```
q = quaternion
q = quaternion(w)
q = quaternion(c)
q = quaternion(x, y, z)
q = quaternion(w, x, y, z)
q = quaternion(w, v)
```

Description

With a real argument, `quaternion(x)` creates a quaternion object whose real part is `w` and imaginary parts are 0. With a complex

argument, `quaternion(c)` creates the quaternion object `real(c)+i*imag(c)`.

With four real arguments, `quaternion(w,x,y,z)` creates the quaternion object `w+i*x+j*y+k*z`.

With three real arguments, `quaternion(x,y,z)` creates the pure quaternion object `i*x+j*y+k*z`.

In all these cases, the arguments may be scalars or arrays of the same size.

With two arguments, `quaternion(w,v)` creates a quaternion object whose real part is `w` and imaginary parts is array `v`. `v` must have one more dimension than `w` for the three imaginary parts.

Without argument, `quaternion` returns the zero quaternion object.

The real or imaginary parts of a quaternion can be accessed with field access, such as `q.w`, `q.x`, `q.y`, `q.z`, and `q.v`.

Examples

```
q = quaternion(1, 2, 3, 4)
  q =
    1+2i+3j+4k
q + 5
  6+2i+3j+4k
q * q
  -28+4i+6j+8k
Q = [q, 2; 2*q, 5]
  2x2 quaternion array
Q.y
  3  0
  6  0
q = quaternion(1, [5; 3; 7])
  q =
    1+5i+3j+7k
q.v
  5
  3
  7
```

See also

`real`, `qimag`, `q2str`, `rpy2q`

rpy2q

Conversion from attitude angles to quaternion.

Syntax

```
q = rpy2q(pitch, roll, yaw)
```

Description

`rpy2q(pitch,roll,yaw)` gives the quaternion corresponding to a rotation of angle `yaw` around the z axis, followed by a rotation of angle `pitch` around the y axis, followed by a rotation of angle `roll` round the x axis. All angles are given in radians. The result is a normalized quaternion whose real part is $\cos(\vartheta/2)$ and imaginary part $\sin(\vartheta/2)\left(v_x i + v_y j + v_z k\right)$, for a rotation of ϑ around unit vector $\left[v_x\ v_y\ v_z\right]^T$. The rotation is applied to a point $[x\ y\ z]^T$ given as a pure quaternion $a = xi + yj + zk$, giving point a also as a pure quaternion; then `b=q*a/q` and `a=q\b*q`. The rotation can also be seen as changing coordinates from body to absolute, where the body's attitude is given by `pitch`, `roll` and `yaw`.

In order to have the usual meaning of pitch, roll and yaw, the x axis must be aligned with the direction of motion, the y axis with the lateral direction, and the z axis with the vertical direction, with the usual sign conventions for cross products. Two common choices are x pointing forward, y to the left, and z upward; or x forward, y to the right, and z downward.

If input arguments are arrays of compatible size (same size or scalar), the result is a quaternion array of the same size; conversion from angles to quaternion is performed independently on corresponding elements.

Example

Conversion of two vectors from aircraft coordinates (x axis forward, y axis to the left, z axis upward) to earth coordinates (x directed to the north, y to the west, z to the zenith). In aircraft coordinates, vectors are `[2;0;0]` (propeller position) and `[0;5;0]` (left wing tip). The aircraft attitude has a pitch of 10 degrees upward, i.e. -10 degrees with the choice of axis, and null roll and yaw.

```
q = rpy2q(0, -10*pi/180, 0)
  q =
    0.9962-0.0872j
q * quaternion(2, 0, 0) / q
  1.9696i+0.3473k
q * quaternion(0, 5, 0) / q
  5j
```

See also

`q2rpy`, `q2mat`, `quaternion`

10.28 List Functions

apply

Function evaluation with arguments in lists.

Syntax

```
listout = apply(fun, listin)
listout = apply(fun, listin, nargout)
listout = apply(fun, listin, na)
listout = apply(fun, listin, nargout, na)
```

Description

`listout=apply(fun,listin)` evaluates function `fun` with input arguments taken from the elements of list `listin`. Output arguments are grouped in list `listout`. Function `fun` is specified by either its name as a string, a function reference, or an anonymous or inline function.

The number of expected output arguments can be specified with an optional third input argument `nargout`. By default, the maximum number of output arguments is requested, up to 256; this limit exists to prevent functions with an unlimited number of output arguments, such as `deal`, from filling memory.

With a 4th argument `na` (or 3rd if `nargout` is not specified), named arguments can be provided as a structure.

Examples

```
apply(@size, {magic(3)}, 2)
  {3, 3}
apply(@(x,y) 2*x+3*y, {5, 10})
  {40}
```

The maximum number of output arguments of `min` is 2 (minimum value and its index):

```
apply(@min, {[8, 3, 4, 7]})
  {3, 2}
```

Two equivalent ways of calling `disp` with a named argument `fd` to specify the standard error file descriptor 2:

```
disp(123, fd=2);
apply(@disp, {123}, 0, {fd=2});
```

See also

`map`, `feval`, `inline`, operator @, `varargin`, `namedargin`, `varargout`

join

List concatenation.

Syntax

```
list = join(l1, l2, ...)
```

Description

`join(l1,l2,...)` joins elements of lists `l1`, `l2`, etc. to make a larger list.

Examples

```
join({1,'a',2:5}, {4,2}, {{'xxx'}})
  {1,'a',[2,3,4,5],4,2,{'xxx'}}
join()
  {}
```

See also

operator `,`, operator `;`, `replist`

islist

Test for a list object.

Syntax

```
b = islist(obj)
```

Description

`islist(obj)` is true if the object `obj` is a list, false otherwise.

Examples

```
islist({1, 2, 'x'})
  true
islist({})
  true
islist([])
  false
ischar('')
  false
```

See also

`isstruct`, `isnumeric`, `ischar`, `islogical`, `isempty`

list2num

Conversion from list to numeric array.

Syntax

```
A = list2num(list)
```

Description

`list2num(list)` takes the elements of `list`, which must be numbers or arrays, and concatenates them on a row (along second dimension) as if they were placed inside brackets and separated with commas. Element sizes must be compatible.

Example

```
list2num({1, 2+3j, 4:6})
  1  2+3j  4  5  6
```

See also

`num2list`, operator `[]`, operator `,`

map

Function evaluation for each element of a list

Syntax

```
(listout1,...) = map(fun, listin1, ...)
```

Description

`map(fun,listin1,...)` evaluates function `fun` successively for each corresponding elements of the remaining arguments, which must be lists or cell arrays. It returns the result(s) of the evaluation as list(s) or cell array(s) with the same size as inputs. Input lists which contain a single element are repeated to match other arguments if necessary. Function `fun` is specified by either its name as a string, a function reference, or an anonymous or inline function.

Examples

```
map('max', {[2,6,4], [7,-1], 1:100})
  {6, 7, 100}
map(@(x) x+10, {3,7,16})
  {13, 17, 26}
(nr, nc) = map(@size, {1,'abc',[4,7;3,4]})
```

```
   nr =
     {1,1,2}
   nc =
     {1,3,2}
 s = map(@size, {1,'abc',[4,7;3,4]})
   s =
     {[1,1], [1,3], [2,2]}
 map(@disp, {'hello', 'lme'})
   hello
   lme
```

Lists with single elements are expanded to match the size of other lists. The following example computes `atan2(1,2)` and `atan2(1,3)`:

```
 map(@atan2, {1}, {2,3})
   {0.4636,0.3218}
```

See also

`apply`, `cellfun`, `for`, `inline`, operator @

num2list

Conversion from array to list.

Syntax

```
 list = num2list(A)
 list = num2list(A, dim)
```

Description

`num2list(A)` creates a list with the elements of non-cell array `A`.
`num2list(A,dim)` cuts array `A` along dimension `dim` and creates a list with the result.

Examples

```
 num2list(1:5)
   {1, 2, 3, 4, 5}
 num2list([1,2;3,4])
   {1, 2, 3, 4}
 num2list([1, 2; 3, 4], 1)
   {[1, 2], [3, 4]}
 num2list([1, 2; 3, 4], 2)
   {[1; 3], [2; 4]}
```

See also

`list2num`, `num2cell`

replist

Replicate a list.

Syntax

```
listout = replist(listin, n)
```

Description

`replist(listin,n)` makes a new list by concatenating `n` copies of list `listin`.

Example

```
replist({1, 'abc'}, 3)
  {1,'abc',1,'abc',1,'abc'}
```

See also

`join`, `repmat`

10.29 Structure Functions

cell2struct

Convert a cell array to a structure array.

Syntax

```
SA = cell2struct(CA, fields)
SA = cell2struct(CA, fields, dim)
```

Description

`cell2struct(CA,fields)` converts a cell array to a structure array. The size of the result is `size(SA)(2:end)`, where `nf` is the number of fields. Field `SA(i1,i2,...).f` of the result contains cell `CA{j,i1,i2,...}`, where `f` is field `field{j}`. Argument `fields` contains the field names as strings.

With a third input argument, `cell2struct(CA,fields,dim)` picks fields of each element along dimension `dim`. The size of the result is the size of `CA` where dimension `dim` is removed.

Examples

```
SA = cell2struct({1, 'ab'; 2, 'cde'}, {'a', 'b'});
SA = cell2struct({1, 2; 'ab', 'cde'}, {'a', 'b'}, 2);
```

See also
`struct2cell`

fieldnames

List of fields of a structure.

Syntax
```
fields = fieldnames(strct)
```

Description
`fieldnames(strct)` returns the field names of structure `strct` as a list of strings.

Example
```
fieldnames({a=1, b=1:5})
  {'a', 'b'}
```

See also
`struct`, `isfield`, `orderfields`, `rmfield`

getfield

Value of a field in a structure.

Syntax
```
value = getfield(strct, name)
```

Description
`getfield(strct,name)` gets the value of field `name` in structure `strct`. It is an error if the field does not exist. `getfield(s,'f')` gives the same value as `s.f`. `getfield` is especially useful when the field name is not fixed, but is stored in a variable or is the result of an expression.

See also
operator `.`, `struct`, `setfield`, `rmfield`

isfield

Test for the existence of a field in a structure.

Syntax

```
b = isfield(strct, name)
```

Description

`isfield(strct, name)` is true if the structure `strct` has a field whose name is the string `name`, false otherwise.

Examples

```
isfield({a=1:3, x='abc'}, 'a')
  true
isfield({a=1:3, x='abc'}, 'A')
  false
```

See also

`fieldnames`, `isstruct`, `struct`

isstruct

Test for a structure object.

Syntax

```
b = isstruct(obj)
```

Description

`isstruct(obj)` is true if its argument `obj` is a structure or structure array, false otherwise.

Examples

```
isstruct({a=123})
  true
isstruct({1, 2, 'x'})
  false
a.f = 3;
isstruct(a)
  true
```

See also

`struct`, `isfield`, `isa`, `islist`, `ischar`, `isobject`, `islogical`

orderfields

Reorders the fields of a structure.

Syntax

```
strctout = orderfields(strctin)
strctout = orderfields(strctin, structref)
strctout = orderfields(strctin, names)
strctout = orderfields(strctin, perm)
(strctout, perm) = orderfields(...)
```

Description

With a single input argument, `orderfields(strctin)` reorders structure fields by sorting them by field names.

With two input arguments, `orderfields` reorders the fields of the first argument after the second argument. Second argument can be a permutation vector containing integers from 1 to `length(strctin)`, another structure with the same field names, or a list of names. In the last cases, all the fields of the structure must be present in the second argument.

The (first) output argument is a structure with the same fields and the same value as the first input argument; the only difference is the field order. An optional second output argument is set to the permutation vector.

Examples

```
s = {a=123, c=1:3, b='abcde'}
  s =
    a: 123
    c: real 1x3
    b: 'abcde'
(t, p) = orderfields(s)
  t =
    a: 123
    b: 'abcde'
    c: real 1x3
  p =
    1
    3
    2
t = orderfields(s, {'c', 'b', 'a'})
  t =
    c: real 1x3
    b: 'abcde'
    a: 123
```

See also

`struct`, `fieldnames`

rmfield

Deletion of a field in a structure.

Syntax

```
strctout = rmfield(strctin, name)
```

Description

`strctout=rmfield(strctin,name)` makes a structure `strctout` with the same fields as `strctin`, except for field named `name` which is removed. If field `name` does not exist, `strctout` is the same as `strctin`.

Example

```
x = rmfield({a=1:3, b='abc'}, 'a');
fieldnames(x)
  b
```

See also

`struct`, `setfield`, `getfield`, `orderfields`

setfield

Assignment to a field in a structure.

Syntax

```
strctout = setfield(strctin, name, value)
```

Description

`strctout=setfield(strctin,name,value)` makes a structure `strctout` with the same fields as `strctin`, except that field named `name` is added if it does not exist yet and is set to `value`. `s=setfield(s,'f',v)` has the same effect as `s.f=v`; `s=setfield(s,str,v)` has the same effect as `s.(str)=v`.

See also

operator `.`, `struct`, `getfield`, `rmfield`

struct

Creation of a structure

Syntax

```
strct = struct(field1=value1, field2=value2, ...)
strct = struct(fieldname1, value1, fieldname2, value2, ...)
strct = {field1=value1, field2=value2, ...}
```

Description

`struct` builds a new structure. With named arguments, the name of each argument is used as the field name. Otherwise, input arguments are used by pairs to create the fields; for each pair, the first argument is the field name, provided as a string, and the second one is the field value.

Instead of named arguments, a more compact notation consists in writing named values between braces. In that case, all values must be named; when no value has a name, a list is created, and mixed named and unnamed values are invalid. Fields are separated by commas; semicolons separate elements of n-by-1 struct arrays. See the documentation of braces for more details.

Examples

Three equivalent ways to create a structure with two fields a and b:

```
x = {a=1, b=2:5};
x = struct(a=1, b=2:5);
x = struct('a', 1, 'b', 2:5);
x.a
  1
x.b
  2 3 4 5
```

See also

`structarray`, `isstruct`, `isfield`, `rmfield`, `fieldnames`, operator {}

struct2cell

Convert a structure array to a cell array.

Syntax

```
CA = struct2cell(SA)
```

Description

`struct2cell(SA)` converts a structure or structure array to a cell array. The size of the result is `[nf,size(SA)]`, where `nf` is the number of fields. Cell `CA{j,i1,i2,...}` of the result contains field `SA(i1,i2,...).f`, where `f` is the j:th field.

Example

```
SA = cell2struct({1, 'ab'; 2, 'cde'}, {'a', 'b'});
CA = struct2cell(SA);
```

See also

`cell2struct`

structarray

Create a structure array.

Syntax

```
SA = structarray(field1=A1, field2=A2, ...)
SA = structarray(fieldname1, A1, fieldname2, A2, ...)
```

Description

`structarray` builds a new structure array. With named arguments, the name of each argument is used as the field name, and the value is a cell array whose elements become the corresponding values in the result. Otherwise, input arguments are used by pairs to create the fields; for each pair, the first argument is the field name, provided as a string, and the second one is the field values as a cell array.

In both cases, all cell arrays must have the same size; the resulting structure array has the same size.

Example

The following assignments produce the same result:

```
SA = structarray(a = {1,2;3,4}, b = {'a', 1:3; 'def', true});
SA = structarray('a', {1,2;3,4}, 'b', {'a', 1:3; 'def', true});
```

See also

`struct`, `cell2struct`

structmerge

Merge the fields of two structures.

Syntax

```
S = structmerge(S1, S2)
```

Description

`structmerge(S1,S2)` merges the fields of S1 and S2, producing a new structure containing the fields of both input arguments. More precisely, to build the result, `structmerge` starts with S1; each field which also exists in S2 is set to the value in S2; and finally, fields in S2 which do not exist in S1 are added.

If S1 and/or S2 are structure arrays, they must have the same size or one of them must be a simple structure (size 1x1). The result is a structure array of the same size where each element is obtained separately from the corresponding elements of S1 and S2; a simple structure argument is reused as necessary.

Examples

```
S = structmerge({a=2}, {b=3})
  S =
    a: 2
    b: 3
S = structmerge({a=1:3, b=4}, {a='AB', c=10})
  S =
    a: 'AB'
    b: 4
    c: 10
```

See also

`fieldnames`, `setfield`, `cat`

10.30 Object Functions

class

Object creation.

Syntax

```
object = class(strct, 'classname')
object = class(strct, 'classname', parent1, ...)
str = class(object)
```

Description

`class(strct,'classname')` makes an object of the specified class with the data of structure `strct`. Object fields can be accessed only from methods of that class, i.e. functions whose name is `classname::methodname`. Objects must be created by the class constructor `classname::classname`.

`class(strct,'classname',parent1,...)` makes an object of the specified class which inherits fields and methods from one or several other object(s) `parent1`, ... Parent objects are inserted as additional fields in the object, with the same name as the class. Fields of parent objects cannot be directly accessed by the new object's methods, only by the parent's methods.

`class(object)` gives the class of `object` as a string. The table below gives the name of native types.

Class	**Native type**
`double`	real or complex double scalar or array
`single`	real or complex single scalar or array
`int8/16/32/64`	8/16/32/64-bit signed integer scalar or array
`uint8/16/32/64`	8/16/32/64-bit unsigned integer scalar or array
`logical`	logical scalar or array
`char`	character or character array
`list`	list
`cell`	cell array
`struct`	scalar structure
`structarray`	structure array
`inline`	inline function
`funref`	function reference
`null`	null value

Examples

```
o1 = class({fld1=1, fld2=rand(4)}, 'c1');
o2 = class({fld3='abc'}, 'c2', o1);
class(o2)
  c2
```

See also

`struct`, `inferiorto`, `superiorto`, `isa`, `isobject`, `methods`

inferiorto

Set class precedence.

Syntax

```
inferiorto('class1', ...)
```

Description

Called in a constructor, `inferiorto('class1',...)` specifies that the class has a lower precedence than classes whose names are given as input arguments. Precedence is used when a function has object arguments of different classes: the method defined for the class with the highest precedence is called.

See also

`superiorto`, `class`

isa

Test for an object of a given class.

Syntax

```
b = isa(object,'classname')
```

Description

`isa(object,'classname')` returns true of `object` is an object of class `class`, directly or by inheritance. In addition to the class names given by `class`, the following classes are supported:

Class	Native type
`cell`	list or cell array
`numeric`	double, single or integer scalar or array
`float`	double or single scalar or array
`integer`	integer scalar or array

Example

```
isa(pi,'double')
  true
```

See also

`class`, `isobject`, `methods`

isnull

Test for a null value.

Syntax

```
b = isnull(a)
```

Description

`isnull(a)` returns true if a is the null value created with `null`, or false for any value of any other type.

See also

`class`, `null`

isobject

Test for an object.

Syntax

```
b = isobject(a)
```

Description

`object(a)` returns true if a is an object created with `class`.

See also

`class`, `isa`, `isstruct`

methods

List of methods for a class.

Syntax

```
methods classname
list = methods('classname')
```

Description

`methods classname` displays the list of methods defined for class `classname`. Inherited methods and private methods are ignored. With an output argument, `methods` gives produces a list of strings.

See also

`class`, `info`

null

Null value.

Syntax

```
obj = null
```

Description

`null` gives the only value of the null data type. It stands for the lack of any value. Null values can be tested with `isnull` or with equality or inequality operators == and ~=.

With an input argument, `null(A)` gives the null space of matrix `A`.

Examples

```
n = null
  n =
    null
isnull(n)
  true
n == null
  true
n ~= null
  false
class(n)
  null
```

See also

`isnull`, `null` (linear algebra)

superclasses

Get list of superclasses.

Syntax

```
list = superclasses(obj)
```

Description

`superclasses(obj)` gives the list of the names of parent classes (superclasses) of the object `obj`. Parent classes are specified as additional arguments to `class` when the object is constructed.

Example

```
use lti;
G = tf(1, [1, 2]);
class(G)
  tf
superclasses(G)
  {'lti'}
isa(G, 'lti')
  true
```

See also

`class`, `isa`

10.31 Logical Functions

all

Check whether all the elements are true.

Syntax

```
v = all(A)
v = all(A,dim)
b = all(v)
```

Description

`all(A)` performs a logical AND on the elements of the columns of array `A`, or the elements of a vector. If a second argument `dim` is provided, the operation is performed along that dimension.

`all` can be omitted if its result is used by `if` or `while`, because these statements consider an array to be true if all its elements are nonzero.

Examples

```
all([1,2,3] == 2)
  false
all([1,2,3] > 0)
  true
```

See also

any, operator &, `bitall`

any

Check whether any element is true.

Syntax

```
v = any(A)
v = any(A,dim)
b = any(v)
```

Description

`any(A)` performs a logical OR on the elements of the columns of array `A`, or the elements of a vector. If a second argument `dim` is provided, the operation is performed along that dimension.

Examples

```
any([1,2,3] == 2)
  true
any([1,2,3] > 5)
  false
```

See also

`all`, operator |, `bitany`

bitall

Check whether all the corresponding bits are true.

Syntax

```
v = bitall(A)
v = bitall(A,dim)
b = bitall(v)
```

Description

`bitall(A)` performs a bitwise AND on the elements of the columns of array `A`, or the elements of a vector. If a second argument `dim` is provided, the operation is performed along that dimension. `A` can be a double or an integer array. For double arrays, `bitall` uses the 32 least-significant bits.

Examples

```
bitall([5, 3])
  1
bitall([7uint8, 6uint8; 3uint8, 6uint8], 2)
  2x1 uint8 array
    6
    2
```

See also

`bitany`, `all`, `bitand`

bitand

Bitwise AND.

Syntax

```
c = bitand(a, b)
```

Description

Each bit of the result is the binary AND of the corresponding bits of the inputs.

The inputs can be scalar, arrays of the same size, or a scalar and an array. If the input arguments are of type double, so is the result, and the operation is performed on 32 bits.

Examples

```
bitand(1,3)
  1
bitand(1:6,1)
  1 0 1 0 1 0
bitand(7uint8, 1234int16)
  2int16
```

See also

`bitor`, `bitxor`, `bitall`, `bitget`

bitany

Check whether any of the corresponding bits is true.

Syntax

```
v = bitany(A)
v = bitany(A,dim)
b = bitany(v)
```

Description

`bitany(A)` performs a bitwise OR on the elements of the columns of array `A`, or the elements of a vector. If a second argument `dim` is provided, the operation is performed along that dimension. `A` can be a double or an integer array. For double arrays, `bitany` uses the 32 least-significant bits.

Examples

```
bitany([5, 3])
  7
bitany([0uint8, 6uint8; 3uint8, 6uint8], 2)
  2x1 uint8 array
    6
    7
```

See also

`bitall`, `any`, `bitor`

bitcmp

Bit complement (bitwise NOT).

Syntax

```
b = bitcmp(i)
b = bitcmp(a, n)
```

Description

`bitcmp(i)` gives the 1-complement (bitwise NOT) of the integer `i`.

`bitcmp(a,n)`, where `a` is an integer or a double, gives the 1-complement of the `n` least-significant bits. The result has the same type as `a`.

The inputs can be scalar, arrays of the same size, or a scalar and an array. If `a` is of type double, so is the result, and the operation is performed on at most 32 bits.

Examples

```
bitcmp(1,4)
  14
bitcmp(0, 1:8)
  1 3 7 15 31 63 127 255
bitcmp([0uint8, 1uint8, 255uint8])
  1x3 uint8 array
  255 254   0
```

See also

`bitxor`, operator ~

bitget

Bit extraction.

Syntax

```
b = bitget(a, n)
```

Description

`bitget(a, n)` gives the `n`:th bit of integer `a`. `a` can be an integer or a double. The result has the same type as `a`. `n`=1 corresponds to the least significant bit.

The inputs can be scalar, arrays of the same size, or a scalar and an array. If `a` is of type double, so is the result, and `n` is limited to 32.

Examples

```
bitget(123,5)
  1
bitget(7, 1:8)
  1 1 1 0 0 0 0 0
bitget(5uint8, 2)
  0uint8
```

See also

`bitset`, `bitand`, `bitshift`

bitor

Bitwise OR.

Syntax

```
c = bitor(a, b)
```

Description

The input arguments are converted to 32-bit unsigned integers; each bit of the result is the binary OR of the corresponding bits of the inputs.

The inputs can be scalar, arrays of the same size, or a scalar and an array. If the input arguments are of type double, so is the result, and the operation is performed on 32 bits.

Examples

```
bitor(1,2)
  3
bitor(1:6,1)
  1 3 3 5 5 7
bitor(7uint8, 1234int16)
  1239int16
```

See also

`bitand`, `bitxor`, `bitany`, `bitget`

bitset

Bit assignment.

Syntax

```
b = bitset(a, n)
b = bitset(a, n, v)
```

Description

`bitset(a,n)` sets the n:th bit of integer a to 1. a can be an integer or a double. The result has the same type as a. n=1 corresponds to the least significant bit. With 3 input arguments, `bitset(a,n,v)` sets the bit to 1 if v is nonzero, or clears it if v is zero.

The inputs can be scalar, arrays of the same size, or a mix of them. If a is of type double, so is the result, and n is limited to 32.

Examples

```
bitset(123,10)
  635
bitset(123, 1, 0)
  122
bitset(7uint8, 1:8)
  1x8 uint8 array
    7   7   7  15  23  39  71 135
```

See also

`bitget`, `bitand`, `bitor`, `bitxor`, `bitshift`

bitshift

Bit shift.

Syntax

```
b = bitshift(a, shift)
b = bitshift(a, shift, n)
```

Description

The first input argument is converted to a 32-bit unsigned integer, and shifted by `shift` bits, to the left if `shift` is positive or to the right if it is negative. With a third argument n, only n bits are retained.

The inputs can be scalar, arrays of the same size, or a mix of both.

Examples

```
bitshift(1,3)
  8
bitshift(8, -2:2)
  2 4 8 16 32
bitshift(15, 0:3, 4)
  15 14 12 8
```

See also

`bitget`

bitxor

Bitwise exclusive OR.

Syntax

```
c = bitxor(a, b)
```

Description

The input arguments are converted to 32-bit unsigned integers; each bit of the result is the binary exclusive OR of the corresponding bits of the inputs.

The inputs can be scalar, arrays of the same size, or a scalar and an array.

Examples

```
bitxor(1,3)
  2
bitxor(1:6,1)
  0 3 2 5 4 7
bitxor(7uint8, 1234int16)
  1237int16
```

See also

`bitcmp`, `bitand`, `bitor`, `bitget`

false

Boolean constant *false*.

Syntax

```
b = false
B = false(n)
B = false(n1, n2, ...)
B = false([n1, n2, ...])
```

Description

The boolean constant false can be used to set the value of a variable. It is equivalent to `logical(0)`. The constant 0 is equivalent in many cases; indices (to get or set the elements of an array) are an important exception.

With input arguments, `false` builds a logical array whose elements are false. The size of the array is specified by one integer for a square matrix, or several integers (either as separate arguments or in a vector) for an array of any size.

Examples

```
false
  false
islogical(false)
  true
false(2,3)
  F F F
  F F F
```

See also

`true`, `logical`, `zeros`

graycode

Conversion to Gray code.

Syntax

```
g = graycode(n)
```

Description

`graycode(n)` converts the integer number `n` to Gray code. The argument `n` can be an integer number of class double (converted to an unsigned integer) or any integer type. If it is an array, conversion is performed on each element. The result has the same type and size as the input.

Gray code is an encoding which maps each integer of `s` bits to another integer of `s` bits, such that two consecutive codes (i.e. `graycode(n)` and `graycode(n+1)` for any `n`) have only one bit which differs.

Example

```
graycode(0:7)
  0 1 3 2 6 7 5 4
```

See also

`igraycode`

igraycode

Conversion from Gray code.

Syntax

```
n = igraycode(g)
```

Description

`igraycode(n)` converts the Gray code `g` to the corresponding integer. It is the inverse of `graycode`. The argument `n` can be an integer number of class double (converted to an unsigned integer) or any integer type. If it is an array, conversion is performed on each element. The result has the same type and size as the input.

Example

```
igraycode(graycode(0:7))
  0 1 2 3 4 5 6 7
```

See also

`graycode`

islogical

Test for a boolean object.

Syntax

```
b = islogical(obj)
```

Description

`islogical(obj)` is true if `obj` is a logical value, and false otherwise. The result is always a scalar, even if `obj` is an array. Logical values are obtained with comparison operators, logical operators, test functions, and the function `logical`.

Examples

```
islogical(eye(10))
  false
islogical(~eye(10))
  true
```

See also

`logical`, `isnumeric`, `isinteger`, `ischar`

logical

Transform a number into a boolean.

Syntax

```
B = logical(A)
```

Description

`logical(x)` converts array or number `A` to logical (boolean) type. All nonzero elements of `A` are converted to true, and zero elements to false.

Logical values are stored as 0 for false or 1 for true in unsigned 8-bit integers. They differ from the `uint8` type when they are used to select the elements of an array or list.

Examples

```
a=1:3; a([1,0,1])
Index out of range
a=1:3; a(logical([1,0,1]))
  1 3
```

See also

`islogical`, `uint8`, `double`, `char`, `setstr`, operator `()`

true

Boolean constant *true*.

Syntax

```
b = true
B = true(n)
B = true(n1, n2, ...)
B = true([n1, n2, ...])
```

Description

The boolean constant `true` can be used to set the value of a variable. It is equivalent to `logical(1)`. The constant 1 is equivalent in many cases; indices (to get or set the elements of an array) are an important exception.

With input arguments, `true` builds a logical array whose elements are true. The size of the array is specified by one integer for a square matrix, or several integers (either as separate arguments or in a vector) for an array of any size.

Examples

```
true
  true
islogical(true)
  true
true(2)
  T T
  T T
```

See also

false, logical, ones

xor

Exclusive or.

Syntax

```
b3 = xor(b1,b2)
```

Description

xor(b1,b2) performs the exclusive or operation between the corresponding elements of b1 and b2. b1 and b2 must have the same size or one of them must be a scalar.

Examples

```
xor([false false true true],[false true false true])
  F T T F
xor(pi,8)
  false
```

See also

operator &, operator |

10.32 Dynamical System Functions

This section describes functions related to linear time-invariant dynamical systems.

c2dm

Continuous-to-discrete-time conversion.

Syntax

```
(numd,dend) = c2dm(numc,denc,Ts)
dend = c2dm(numc,denc,Ts)
(numd,dend) = c2dm(numc,denc,Ts,method)
dend = c2dm(numc,denc,Ts,method)
(Ad,Bd,Cd,Dd) = c2dm(Ac,Bc,Cc,Dc,Ts,method)
```

Description

`(numd,dend) = c2dm(numc,denc,Ts)` converts the continuous-time transfer function `numc/denc` to a discrete-time transfer function `numd/dend` with sampling period `Ts`. The continuous-time transfer function is given by two polynomials in s, and the discrete-time transfer function is given by two polynomials in z, all as vectors of coefficients with highest powers first.

`c2dm(numc,denc,Ts,method)` uses the specified conversion method. `method` is one of

Method	**Description**
`'zoh'` or `'z'`	zero-order hold (default)
`'foh'` or `'f'`	first-order hold
`'tustin'` or `'t'`	Tustin (bilinear transformation)
`'matched'` or `'m'`	Matched zeros, poles and gain

The input and output arguments `numc`, `denc`, `numd`, and `dend` can also be matrices; in that case, the conversion is applied separately on each row with the same sampling period `Ts`.

`c2dm(Ac,Bc,Cc,Dc,Ts,method)` performs the conversion from continuous-time state-space model (Ac,Bc,Cc,Dc) to discrete-time state-space model (Ad,Bd,Cd,Dd), defined by

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned}$$

and

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(k) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

Method `'matched'` is not supported for state-space models.

Examples

```
(numd, dend) = c2dm(1, [1, 1], 0.1)
  numd =
    0.0952
  dend =
    1 -0.9048
(numd, dend) = c2dm(1, [1, 1], 0.1, 'foh')
  numd =
    0.0484
  dend =
    1 -0.9048
(numd, dend) = c2dm(1, [1, 1], 0.1, 'tustin')
```

```
numd =
  0.0476 0.0476
dend =
  1 -0.9048
```

See also

d2cm

d2cm

Discrete-to-continuous-time conversion.

Syntax

```
(numc,denc) = d2cm(numd,dend,Ts)
denc = d2cm(numd,dend,Ts)
(numc,denc) = d2cm(numd,dend,Ts,method)
denc = d2cm(numd,dend,Ts,method)
```

Description

`(numc,denc) = d2cm(numd,dend,Ts,method)` converts the discrete-time transfer function `numd/dend` with sampling period `Ts` to a continuous-time transfer function `numc/denc`. The continuous-time transfer function is given by two polynomials in s, and the discrete-time transfer function is given by two polynomials in z, all as vectors of coefficients with highest powers first.

Method is

Method	Description
`'tustin'` or `'t'`	Tustin (bilinear transformation) (default)

The input and output arguments `numc`, `denc`, `numd`, and `dend` can also be matrices; in that case, the conversion is applied separately on each row with the same sampling period `Ts`.

`d2cm(Ad,Bd,Cd,Dd,Ts,method)` performs the conversion from discrete-time state-space model (Ad,Bd,Cd,Dd) to continuous-time state-space model (Ac,Bc,Cc,Dc), defined by

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(k) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

and

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned}$$

Example

```
(numd, dend) = c2dm(1, [1, 1], 5, 't')
  numd =
    0.7143 0.7143
  dend =
    1 0.4286
(numc, denc) = d2cm(numd, dend)
  numc =
    -3.8858e-17 1
  denc =
    1 1
```

See also

`c2dm`

dmargin

Robustness margins of a discrete-time system.

Syntax

```
(gm,psi,wc,wx) = dmargin(num,den,Ts)
(gm,psi,wc,wx) = dmargin(num,den)
```

Description

The open-loop discrete-time transfer function is given by the two polynomials `num` and `den`, with sampling period `Ts` (default value is 1). If the closed-loop system (with negative feedback) is unstable, all output arguments are set to an empty matrix. Otherwise, `dmargin` calculates the gain margins `gm`, which give the interval of gain for which the closed-loop system remains stable; the phase margin `psi`, always positive if it exists, which defines the symmetric range of phases which can be added to the open-loop system while keeping the closed-loop system stable; the critical frequency associated to the gain margins, where the open-loop frequency response intersects the real axis around -1; and the cross-over frequency associated to the phase margin, where the open-loop frequency response has a unit magnitude. If the Nyquist diagram does not cross the unit circle, `psi` and `wx` are empty.

Examples

Stable closed-loop, Nyquist inside unit circle:

```
(gm,psi,wc,wx) = dmargin(0.005,poly([0.9,0.9]))
  gm = [-2, 38]
```

```
  psi = []
  wc = [0, 0.4510]
  wx = []
```

Stable closed-loop, Nyquist crosses unit circle:

```
(gm,psi,wc,wx) = dmargin(0.05,poly([0.9,0.9]))
  gm = [-0.2, 3.8]
  psi = 0.7105
  wc = [0, 0.4510]
  wx = 0.2112
```

Unstable closed-loop:

```
(gm,psi,wc,wx) = dmargin(1,poly([0.9,0.9]))
  gm = []
  psi = []
  wc = []
  wx = []
```

Caveats

Contrary to many functions, `dmargin` cannot be used with several transfer functions simultaneously, because not all of them may correspond simultaneously to either stable or unstable closed-loop systems.

See also

`margin`

margin

Robustness margins of a continuous-time system.

Syntax

```
(gm,psi,wc,wx) = margin(num,den)
```

Description

The open-loop continuous-time transfer function is given by the two polynomials `num` and `den`. If the closed-loop system (with negative feedback) is unstable, all output arguments are set to an empty matrix. Otherwise, `margin` calculates the gain margins `gm`, which give the interval of gain for which the closed-loop system remains stable; the phase margin `psi`, always positive if it exists, which defines the symmetric range of phases which can be added to the open-loop system

while keeping the closed-loop system stable; the critical frequency associated to the gain margins, where the open-loop frequency response intersects the real axis around -1; and the cross-over frequency associated to the phase margin, where the open-loop frequency response has a unit magnitude. If the Nyquist diagram does not cross the unit circle, `psi` and `wx` are empty.

Examples

Stable closed-loop, Nyquist inside unit circle:

```
(gm,psi,wc,wx) = margin(0.5,poly([-1,-1,-1]))
  gm = [-2, 16]
  psi = []
  wc = [0, 1.7321]
  wx = []
```

Stable closed-loop, Nyquist crosses unit circle:

```
(gm,psi,wc,wx) = margin(4,poly([-1,-1,-1]))
  gm = [-0.25 2]
  psi = 0.4737
  wc = [0, 1.7321]
  wx = 1.2328
```

Unstable closed-loop:

```
(gm,psi,wc,wx) = margin(10,poly([-1,-1,-1]))
  gm = []
  psi = []
  wc = []
  wx = []
```

Caveats

Contrary to many functions, `margin` cannot be used with several transfer functions simultaneously, because not all of them may correspond simultaneously to either stable or unstable closed-loop systems.

See also

`dmargin`

movezero

Change the position of a real or complex zero in a real-coefficient polynomial.

Syntax

```
pol2 = movezero(pol1, p0, p1)
(pol2, p1actual) = movezero(pol1, p0, p1)
```

Description

`movezero` should be used in the mousedrag handle when the user drags the zero of a polynomial with real coefficients. It insures a consistent user experience.

If `p0` is a real or complex zero of the polynomial `pol1`, `movezero` computes a new polynomial `pol2`, with real coefficients, a zero at `p1`, and most other zeros unchanged. If `p0` and `p1` are real,

```
pol2 = conv(deconv(pol1, [1, -p0]), [1, -p1])
```

If `p0` and `p1` are complex and their imaginary part has the same sign,

```
pol2 = conv(deconv(pol1, poly([p0,conj(p0)])), ...
            poly([p1,conj(p1)]))
```

Otherwise, a real pole `p0` is moved to complex pole `p1` if `imag(p1) > 0` and there is another real pole in `pol0`. A complex pole `p0` can be moved to `real(p1)` if `imag(p1)*imag(p0) < 0`; in that case, `conj(p0)` is moved to `real(p0)`.

If it exists, the second output argument is set to the actual value of the displaced pole. It can be used to provide feedback to the user during the drag.

Examples

```
roots(movezero(poly([1,3,2+3j,2-3j]),1,5))
  5
  3
  2+3j
  2-3j
roots(movezero(poly([1,3,2+3j,2-3j]),1,2j))
  2j
  -2j
  2+3j
  2-3j
roots(movezero(poly([1,3,2+3j,2-3j]),2+3j,5+8j))
  1
  3
  5+8j
  5-8j
roots(movezero(poly([1,3,2+3j,2-3j]),2+3j,5-8j))
  1
  3
  5
```

```
  2
(pol, newPole) = movezero(poly([1,3,2+3j,2-3j]),2+3j,5-8j);
newPole
  5
```

See also

`roots`, `conv`, `deconv`

ss2tf

Conversion from state space to transfer function.

Syntax

```
(num,den) = ss2tf(A,B,C,D)
den = ss2tf(A,B,C,D)
(num,den) = ss2tf(A,B,C,D,iu)
den = ss2tf(A,B,C,D,iu)
```

Description

A continuous-time linear time-invariant system can be represented by the state-space model

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where $x(t)$ is the state, $u(t)$ the input, $y(t)$ the output, and $ABCD$ four constant matrices which characterize the model. If it is a single-input single-output system, it can also be represented by its transfer function `num/den`. `(num,den) = ss2tf(A,B,C,D)` converts the model from state space to transfer function. If the state-space model has multiple outputs, `num` is a matrix whose lines correspond to each output (the denominator is the same for all outputs). If the state-space model has multiple inputs, a fifth input argument is required and specifies which one to consider.

For a sampled-time model, exactly the same function can be used. The derivative is replaced by a forward shift, and the variable s of the Laplace transform is replaced by the variable z of the z transform. But as long as coefficients are concerned, the conversion is the same.

The degree of the denominator is equal to the number of states, i.e. the size of `A`. The degree of the numerator is equal to the number of states if `D` is not zero, and one less if `D` is zero.

If `D` is zero, it can be replaced by the empty matrix `[]`.

Example

Conversion from the state-space model $dx/dt = -x + u$, $y = x$ to the transfer function $Y(s)/U(s) = 1/(s+1)$:

```
(num, den) = ss2tf(-1, 1, 1, 0)
  num =
    1
  den =
    1 1
```

See also

`tf2ss`

tf2ss

Conversion from transfer function to state space.

Syntax

```
(A,B,C,D) = tf2ss(num,den)
```

Description

`tf2ss(num,den)` returns the state-space representation of the transfer function `num`/`den`, which is given as two polynomials. The transfer function must be causal, i.e. `num` must not have more columns than `den`. Systems with several outputs are specified by a `num` having one row per output; the denominator `den` must be the same for all the outputs.

`tf2ss` applies to continuous-time systems (Laplace transform) as well as to discrete-time systems (z transform or delta transform).

Example

```
(A,B,C,D) = tf2ss([2,5],[2,3,8])
  A =
    -1.5 -4
    1 0
  B =
    1
    0
  C =
    1 2.5
  D =
    0
```

See also

`ss2tf`, `zp2ss`

zp2ss

Conversion from transfer function given by zeros and poles to state space.

Syntax

```
(A,B,C,D) = zp2ss(z,p,k)
```

Description

`zp2ss(z,p,k)` returns the state-space representation of the transfer function with zeros `z`, poles `p` and gain `k` (ratio of leading coefficients of numerator and denominator in decreasing powers). The transfer function must be causal, i.e. the number of zeros must not be larger than the number of poles. `zp2ss` supports only systems with one input and one output. Complex zeros and complex poles must make complex conjugate pairs, so that the corresponding polynomials have real coefficients.

`zp2ss` applies to continuous-time systems (Laplace transform) as well as to discrete-time systems (z transform or delta transform).

Example

```
(A,B,C,D) = zp2ss([1;2],[3;4-1j;4+1j],5)
  A =
    8  -17    1
    1    0    0
    0    0    3
  B =
    0
    0
    1
  C =
   25  -75    5
  D =
    0
```

See also

`tf2ss`

10.33 Input/Output Functions

bwrite

Store data in an array of bytes.

Syntax

```
s = bwrite(data)
s = bwrite(data, precision)
```

Description

`bwrite(data)` stores the contents of the matrix `data` into an array of class `uint8`. The second parameter is the precision, whose meaning is the same as for `fread`. Its default value is 'uint8'.

Examples

```
bwrite(12345, 'uint32;l')
  1x4 uint8 array
    57  48   0   0
bwrite(12345, 'uint32;b')
  1x4 uint8 array
     0   0  48  57
```

See also

swrite, sread, fwrite, sprintf, `typecast`

clc

Clear the text window or panel.

Syntax

```
clc
clc(fd)
```

Description

`clc` (clear console) clears the contents of the command-line window or panel.

`clc(fd)` clears the contents of the window or panel associated with file descriptor `fd`.

disp

Simple display on the standard output.

Syntax

```
disp(obj)
disp(obj, fd=fd)
```

Description

`disp(obj)` displays the object `obj`. Command `format` may be used to control how numbers are formatted.

With named argument `fd`, `disp(obj,fd=fd)` writes `obj` to the file descriptor `fd`.

Example

```
disp('hello')
hello
```

See also

`format`, `fprintf`

fclose

Close a file.

Syntax

```
fclose(fd)
fclose('all')
```

Description

`fclose(fd)` closes the file descriptor `fd` which was obtained with functions such as `fopen`. Then `fd` should not be used anymore. `fclose('all')` closes all the open file descriptors.

feof

Check end-of-file status.

Syntax

```
b = feof(fd)
```

Description

`feof(fd)` is false if more data can be read from file descriptor `fd`, and true if the end of the file has been reached.

Example

Count the number of lines and characters in a file (`fopen` and `fclose` are not available in all LME applications):

```
fd = fopen('data.txt');
lines = 0;
characters = 0;
while ~feof(fd)
  str = fgets(fd);
  lines = lines + 1;
  characters = characters + length(str);
end
fclose(fd);
```

See also

`ftell`

fflush

Flush the input and output buffers.

Syntax

```
fflush(fd)
```

Description

`fflush(fd)` discards all the data in the input buffer and forces data out of the output buffer, when the device and their driver permits it. `fflush` can be useful to recover from errors.

fgetl

Reading of a single line.

Syntax

```
line = fgetl(fd)
line = fgetl(fd, n)
```

Description

A single line (of at most n characters) is read from a text file. The end of line character is discarded. Upon end of file, `fgetl` gives an empty string.

See also

`fgets`, `fscanf`

fgets

Reading of a single line.

Syntax

```
line = fgets(fd)
line = fgets(fd, n)
```

Description

A single line (of at most n characters) is read from a text file. Unless the end of file is encountered before, the end of line (always a single line feed) is preserved. Upon end of file, `fgets` gives an empty string.

See also

`fgetl`, `fscanf`

fionread

Number of bytes which can be read without blocking.

Syntax

```
n = fionread(fd)
```

Description

`fionread(fd)` returns the number of bytes which can be read without blocking. For a file, all the data until the end of the file can be read; but for a device or a network connection, `fionread` gives the number of bytes which have already been received and are stored in the read buffer.

If the number of bytes cannot be determined, `fionread` returns `-1`.

See also

`fread`

format

Default output format.

Syntax

```
format
format short
format short e
format short eng
format short g
format long
format long e
format long eng
format long g
format int
format int d
format int u
format int x
format int o
format int b
format bank
format rat
format '+'
format i
format j
format loose
format compact
```

Description

`format` changes the format used by command `disp` and for output produced with expressions which do not end with a semicolon. The following arguments are recognized:

Arguments	Meaning
(none)	fixed format with 0 or 4 digits, loose spacing
`short`	fixed format with 0 or 4 digits
`short e`	exponential format with 4 digits
`short eng`	engineering format with 4 digits
`short g`	general format with up to 4 digits
`long`	fixed format with 0 or 15 digits
`long e`	exponential format with 15 digits
`long eng`	engineering format with 15 digits
`long g`	general format with up to 15 digits
`int`	signed decimal integer
`int d`	signed decimal integer
`int u`	unsigned decimal integer
`int x`	hexadecimal integer
`int o`	octal integer
`int b`	binary integer
`bank`	fixed format with 2 digits (for currencies)
`rat`	rational approximation
`+`	'+', '-' or 'I' for nonzero, space for zero
`i`	symbol `i` to represent the imaginary unit
`j`	symbol `j` to represent the imaginary unit
`loose`	empty lines to improve readability
`compact`	no empty line

Format for numbers, for imaginary unit symbol and for spacing is set separately. Format `rat` displays rational approximations like `rat` with the default tolerance, but also displays the imaginary part if it exists. Format `'+'` displays compactly numeric and boolean arrays: positive numbers and complex numbers with a positive real part are displayed as +, negative numbers or complex numbers with a negative real part as `-`, pure imaginary nonzero numbers as `I`, and zeros as spaces.

The default format is `format short g`, `format j`, and `format compact`.

See also

`disp`, `fprintf`, `rat`

fprintf

Formatted output.

Syntax

```
n = fprintf(fd,format,a,b,...)
n = fprintf(format,a,b,...)
n = fprintf(..., fd=fd, NPrec=nPrec)
```

Description

`fprintf(format,a,b,...)` converts its arguments to a string and writes it to the standard output.

`fprintf(fd,format,a,b,...)` specifies the output file descriptor. The file descriptor can also be specified as a named argument `fd`.

In addition to `fd`, `fprintf` also accepts named argument `NPrec` for the default number of digits in floating-point numbers.

See `sprintf` for a description of the conversion process.

Example

```
fprintf('%d %.2f %.3E %g\n',1:3,pi)
1 2.00 3.000E0 3.1416
  22
```

See also

`sprintf`, `fwrite`

fread

Raw input.

Syntax

```
(a, count) = fread(fd)
(a, count) = fread(fd, size)
(a, count) = fread(fd, size, precision)
```

Description

`fread(fd)` reads signed bytes from the file descriptor `fd` until it reaches the end of file. It returns a column vector whose elements are signed bytes (between -128 and 127), and optionally in the second output argument the number of elements it has read.

`fread(fd,size)` reads the number of bytes specified by `size`. If `size` is a scalar, that many bytes are read and result in a column vector. If `size` is a vector of two elements `[m,n]`, `m*n` elements are read row by row and stored in an m-by-n matrix. If the end of the file is reached before the specified number of elements have been read, the number of rows is reduced without throwing an error. The optional second output argument always gives the number of elements in the result. If `size` is the empty array `[]`, elements are read until the end of the file; it must be specified if there is a third argument.

With a third argument, `fread(fd,size,precision)` reads integer words of 1, 2, or 4 bytes, or IEEE floating-point numbers of 4 bytes (single precision) or 8 bytes (double precision). The meaning of the string `precision` is described in the table below.

Precision	Meaning
int8	signed 8-bit integer (-128 ≤ x ≤ 127)
char	signed 8-bit integer (-128 ≤ x ≤ 127)
int16	signed 16-bit integer (-32768 ≤ x ≤ 32767)
int32	signed 32-bit integer (-2147483648 ≤ x ≤ 2147483647)
int64	signed 64-bit integer (-9.223372e18 ≤ x ≤ 9.223372e18)
uint8	unsigned 8-bit integer (0 ≤ x ≤ 255)
uchar	unsigned 8-bit integer (0 ≤ x ≤ 255)
uint16	unsigned 16-bit integer (0 ≤ x ≤ 65535)
uint32	unsigned 32-bit integer (0 ≤ x ≤ 4294967295)
uint64	unsigned 64-bit integer (0 ≤ x ≤ 18.446744e18)
single	32-bit IEEE floating-point
double	64-bit IEEE floating-point

By default, multibyte words are encoded with the least significant byte first (little endian). The characters ';b' can be appended to specify that they are encoded with the most significant byte first (big endian); for symmetry, ';l' is accepted and ignored.

By default, the output is a double array. To get an output which has the same type as what is specified by `precision`, the character `*` can be inserted at the beginning. For instance `'*uint8'` reads bytes and stores them in an array of class `uint8`, `'*int32;b'` reads signed 32-bit words and stores them in an array of class `int32` after performing byte swapping if necessary, and `'*char'` reads bytes and stores them in a character row vector (i.e. a plain string).

Precisions `'int64'` and `'uint64'` are supported only if types `int64` and `uint64` are supported.

See also

`fwrite`, `sread`

frewind

Rewind current read or write position in a file.

Syntax

```
frewind(fd)
```

Description

`frewind(fd)` sets the position in an open file where the next input/output commands will read or write data to the beginning of the file. The argument `fd` is the file descriptor returned by `fopen` or similar functions (`fopen` is not available in all LME applications).

`frewind(fd)` has the same effect as `fseek(fd,0)` or `fseek(fd,0,'b')`.

See also

`fseek`, `ftell`

fscanf

Reading of formatted numbers.

Syntax

```
r = fscanf(fd, format)
(r, count) = fscanf(fd, format)
```

Description

A single line is read from a text file, and numbers, characters and strings are decoded according to the format string. The format string follows the same rules as `sscanf`.

The optional second output argument is set to the number of elements decoded successfully (may be different than the length of the first argument if decoding strings).

Example

Read a number from a file (`fopen` and `fclose` are not available in all LME applications):

```
fd = fopen('test.txt', 'rt');
fscanf(fd, '%f')
  2.3
fclose(fd);
```

See also

sscanf

fseek

Change the current read or write position in a file.

Syntax

```
status = fseek(fd, position)
status = fseek(fd, position, mode)
```

Description

`fseek(fd,position,mode)` changes the position in an open file where the next input/output commands will read or write data. The first argument `fd` is the file descriptor returned by `fopen` or similar functions (`fopen` is not available in all LME applications). The second argument is the new position. The third argument `mode` specifies how the position is used:

Mode	Description
b	absolute position from the beginning of the file
c	relative position from the current position
e	offset from the end of the file (must be ≤ 0)

The default value is `'b'`. Only the first character is checked, so `'beginning'` is a valid alternative for `'b'`. `fseek` returns 0 if successful or -1 if the position is outside the limits of the file contents.

See also

`frewind`, `ftell`

ftell

Get the current read or write position in a file.

Syntax

```
position = ftell(fd)
```

Description

`ftell(fd)` gives the current file position associated with file descriptor `fd`. The file position is the offset (with respect to the beginning of the file) at which the next input function will read or the next output function will write. The offset is expressed in bytes. With text files, `ftell` may not always correspond to the number of characters read or written.

See also

`fseek`, `feof`

fwrite

Raw output.

Syntax

```
count = fwrite(fd, data)
count = fwrite(fd, data, precision)
```

Description

`fwrite(fd, data)` writes the contents of the matrix `data` to the output referenced by the file descriptor `fd`. The third parameter is the precision, whose meaning is the same as for `fread`. Its default value is `'uint8'`.

See also

`fread`, `swrite`, `bwrite`

redirect

Redirect or copy standard output or error to another file descriptor.

Syntax

```
redirect(fd, fdTarget)
redirect(fd, fdTarget, copy)
redirect(fd)
R = redirect(fd)
redirect
R = redirect
```

Description

`redirect(fd,fdTarget)` redirects output from file descriptor `fd` to `fdTarget`. `fd` must be 1 for standard output or 2 for standard error. If `fdTarget==fd`, the normal behavior is restored.

`redirect(fd,fdTarget,copy)` copies output to both `fd` and `fdTarget` if `copy` is true, instead of redirecting it only to `fdTarget`. If `copy` is false, the result is the same as with two input arguments.

With zero or one input argument and without output argument, `redirect` displays the current redirection for the specified file descriptor (1 or 2) or for both of them. Note that the redirection itself may alter where the result is displayed.

With an output argument, `redirect` returns a 1-by-2 row vector if the file descriptor is specified, or a 2-by-2 matrix otherwise. The first column contains the target file descriptor and the second column, 1 for copy mode and 0 for pure redirection mode.

Examples

Create a new file `diary.txt` and copy to it both standard output and error:

```
fd = fopen('diary.txt', 'w');
redirect(1, fd, true);
redirect(2, fd, true);
```

Stop copying standard output and error and close file:

```
redirect(1, 1);
redirect(2, 2);
fclose(fd);
```

Redirect standard error to standard output and get the redirection state:

```
redirect(2, 1)
redirect
  stdout (fd=1) -> fd=1
  stderr (fd=2) -> fd=1
redirect(2)
  stderr (fd=2) -> fd=1
R = redirect
  R =
    1 0
    1 0
R = redirect(2)
  R =
    1 0
```

sprintf

Formatted conversion of objects into a string.

Syntax

```
s = sprintf(format,a,b, ...)
s = sprintf(..., NPrec=nPrec)
```

Description

`sprintf` converts its arguments to a string. The first parameter is the format string. All the characters are copied verbatim to the output string, except for the control sequences which all begin with the character '%'. They have the form

```
%fn.dt
```

where `f` is zero, one or more of the following flags:

Flag	Description
-	left alignment (default is right alignment)
+	display of a + sign for positive numbers
0	zero padding instead of spaces
#	alternate format (see below)
space	sign replaced with space for positive numbers

`n` is the optional width of the field as one or more decimal digits (default is the minimum width to display the data).

`d` is the number of digits after the decimal separator for a number displayed with a fractional part (default is 4 or what is specified by named argument `NPrec`), the minimum number of displayed digits for a number displayed as an integer, or the number of characters for a string (one or more decimal digits).

`t` is a single character denoting the type of conversion. In most cases, each control sequence corresponds to an additional argument.

All elements of arrays are used sequentially as if they were provided separately; strings are used as a whole. The table below gives the valid values of `t`.

Char.	**Conversion**
`%`	single %
`d`	decimal number as an integer
`i`	same as `d`
`x`	hexadecimal number (for integers between 0 and 2^32-1)
`X`	same as `x`, with uppercase digits
`o`	octal number (for integers between 0 and 2^32-1)
`f`	fixed number of decimals (exp. notation if abs(x)>1e18)
`F`	same as `f`, with an uppercase E
`e`	scientific notation such as 1e5
`E`	scientific notation such as 1E5
`n`	engineering notation such as 100e3
`N`	engineering notation such as 100E3
`g`	decimal or scientific notation
`G`	same as `g`, with an uppercase E
`k`	same as `g`, with as few characters as possible
`K`	same as `k`, with an uppercase E
`P`	SI prefix (k=1e3, u=1e-6) or engineering notation
`c`	character
`s`	string

The # flag forces octal numbers to begin with `0`, nonzero hexadecimal numbers to begin with `0x`, and floating-point numbers to always have a decimal point even if they do not have a fractional part.

Integer formats `%d`, `%i`, `%o` and `%x` round fractional numbers to the nearest integer.

Instead of decimal digits, the width `n` and/or the precision `d` can be replaced with character `*`; then one or two additional arguments (or elements of an array) are consumed and used as the width or precision.

Examples

Numbers:

```
sprintf('%d %.2f %.2e %.2E %.2g',pi*ones(1,5))
  3 3.14 3.14e0 3.14E0 3.14
```

Compact representation with '%k':

```
sprintf('%.1k ', 0.001, 0.11, 111, 1000)
  1e-3 0.11 111 1e3
```

Width and precision:

```
sprintf('*%8.3f*%8.6s*%-8.6s*',pi,'abcdefgh','abcdefgh')
  *   3.142*  abcdef*abcdef  *
```

Repetition of format string to convert all values:

```
sprintf('%c_','a':'z')
  a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p_q_r_s_t_u_v_w_x_y_z_
```

Width and precision provided as expressions:

```
sprintf('%*.*f', 15, 7, pi)
      3.1415927
```

Zero padding for integer format:

```
sprintf('%.3d,%.3d', 12, 12345)
  012,12345
```

Default precision:

```
sprintf('%f %e', pi, pi)
  3.1416 3.1416e0
sprintf('%f %e', pi, pi, NPrec=2)
  3.14 3.14e0
```

See also

`fprintf`, `sscanf`, `swrite`

sread

Raw input from a string or an array of bytes.

Syntax

```
(a, count) = sread(str, size, precision)
(a, count) = sread(str, [], precision)
(a, count) = sread(bytes, ...)
```

Description

`sread(str)` reads data from string `str` or array of class `uint8` or `int8` the same way as `fread` reads data from a file.

Examples

```
(data, count) = sread('abc')
  data =
    97
    98
    99
  count =
    3
(data, count) = sread('abcdef',[2,2])
  data =
    97 98
    99 100
  count =
    4
(data, count) = sread('abcd',[inf,3])
  data =
    97 98 99
  count =
    3
```

See also

`swrite`, `bwrite`, `fread`, `typecast`

sscanf

Decoding of formatted numbers.

Syntax

```
r = sscanf(str, format)
(r, count) = scanf(str, format)
(r, count, nchar) = scanf(str, format)
```

Description

Numbers, characters and strings are extracted from the first argument. Exactly what is extracted is controlled by the second argument, which can contain the following elements:

Substring in format	**Meaning**
`%c`	single character
`%s`	string
`%d`	integer number in decimal
`%x`	unsigned integer number in hexadecimal
`%o`	unsigned integer number in octal
`%i`	integer number
`%f`	floating-point number
`%e`	floating-point number
`%g`	floating-point number
`%%`	%
other character	exact match

`%i` recognizes an optional sign followed by a decimal number, an hexadecimal number prefixed with `0x` or `0X`, a binary number prefixed with `0b` or `0B`, or an octal number prefixed with `0`.

The decoded elements are accumulated in the output argument, either as a column vector if the format string contains `%d`, `%o`, `%x`, `%i`, `%f`, `%e` or `%g`, or a string if the format string contains only `%c`, `%s` or literal values. If a star is inserted after the percent character, the value is decoded and discarded. A width (as one or more decimal characters) can be inserted before `s`, `d`, `x`, `o`, `i`, `f`, `e` or `g`; it limits the number of characters to be decoded. In the input string, spaces and tabulators are skipped before decoding `%s`, `%d`, `%x`, `%o`, `%i`, `%f`, `%e` or `%g`.

The format string is recycled as many times as necessary to decode the whole input string. The decoding is interrupted if a mismatch occurs.

The optional second output argument is set to the number of elements decoded successfully (may be different than the length of the first argument if decoding strings). The optional third output argument is set to the number of characters which were consumed in the input string.

Examples

```
sscanf('f(2.3)', 'f(%f)')
  2.3
sscanf('12a34x778', '%d%c')
  12
  97
  34
 120
 778
sscanf('abc def', '%s')
 abcdef
sscanf('abc def', '%c')
 abc def
sscanf('12,34','%*d,%d')
```

```
  34
sscanf('0275a0ff', '%2x')
    2
  117
  160
  255
```

See also

`sprintf`

swrite

Store data in a string.

Syntax

```
s = swrite(data)
s = swrite(data, precision)
```

Description

`swrite(data)` stores the contents of the matrix `data` into a string. The second parameter is the precision, whose meaning is the same as for `fread`. Its default value is 'uint8'.

Examples

```
swrite(65:68)
  ABCD
double(swrite([1,2], 'int16'))
  1 0 2 0
double(swrite([1,2], 'int16;b'))
  0 1 0 2
```

See also

bwrite, fwrite, sprintf, sread

10.34 File System Functions

Access to any kind of file can be useful to analyze data which come from other applications (such as experimental data) and to generate results in a form suitable for other applications (such as source code or HTML files). Functions specific to files are described in this section. Input, output, and control are done with the following generic functions:

Function	Description
`fclose`	close the file
`feof`	check end of file status
`fflush`	flush I/O buffers
`fgetl`	read a line
`fgets`	read a line
`fprintf`	write formatted data
`fread`	read data
`frewind`	reset the current I/O position
`fscanf`	read formatted data
`fseek`	change the current I/O position
`ftell`	get the current I/O position
`fwrite`	write data
`redirect`	redirect output

fopen

Open a file.

Syntax

```
fd = fopen(path)
fd = fopen(path, mode)
```

Description

`fopen` opens a file for reading and/or writing. The first argument is a path whose format depends on the platform. If it is a plain file name, the file is located in the current directory; what "current" means also depends on the operating system. The output argument, a real number, is a file descriptor which can be used by many input/output functions, such as `fread`, `fprintf`, or `dumpvar`.

The optional second input argument, a string of one or two characters, specifies the mode. It can take one of the following values:

Mode	Meaning
(none)	same as `'r'`
`'r'`	read-only, binary mode, seek to beginning
`'w'`	read/write, binary mode, create new file
`'a'`	read/write, binary mode, seek to end
`'rt'`	read-only, text mode, seek to beginning
`'wt'`	read/write, text mode, create new file
`'at'`	read/write, text mode, seek to end

In text mode, end-of-line characters LF, CR and CRLF are all converted to LF (`'\n'`) on input. On output, they are converted to the native sequence for the operating system, which is CRLF on Windows

and LF elsewhere. To force the output end-of-line to be LF irrespectively of the operating system, use 'q' instead of 't' (e.g. 'wq' to write to a file); to force it to be CRLF, use 'T' (e.g. 'aT' to append to a file).

Examples

Reading a whole text file into a string:

```
fd = fopen('file.txt', 'rt');
str = fread(fd, inf, '*char');
fclose(fd);
```

Reading a whole text file line by line:

```
fd = fopen('file.txt', 'rt');
while ~feof(fd)
  str = fgets(fd)
end
fclose(fd);
```

Writing a matrix to a CSV (comma-separated values) text file:

```
M = magic(5);
fd = fopen('file.txt', 'wt');
for i = 1:size(M, 1)
  for j = 1:size(M, 2)-1
    fprintf(fd, '%g,', M(i,j));
  end
  fprintf(fd, '%g\n', M(i,end));
end
fclose(fd);
```

Reading 5 bytes at offset 3 in a binary file, giving an 5-by-1 array of unsigned 8-bit integers:

```
fd = fopen('file.bin');
fseek(fd, 3);
data = fread(fd, 5, '*uint8');
fclose(fd);
```

See also

`fclose`

10.35 Path Manipulation Functions

fileparts

File path splitting into directory, filename and extension.

Syntax

```
(dir, filename, ext) = fileparts(path)
```

Description

`fileparts(path)` splits string `path` into the directory (initial part until the last file separator), the filename without extension (substring after the last file separator before the last dot), and the extension (substring starting from the last dot after the last file separator).

The directory is empty if `path` does not contain any file separator, and the extension is empty if the remaining substring does not contain a dot. When these three strings are concatenated, the result is always equal to the initial path.

The separator depends on the operating system: a slash on unix (including Mac OS X and Linux), and a backslash or a slash on Windows.

Examples

```
(dir, filename, ext) = fileparts('/home/tom/report.txt')
  dir =
    /home/tom/
  filename =
    report
  ext =
    .txt
(dir, filename, ext) = fileparts('/home/tom/')
  dir =
    /home/tom/
  filename =
    ''
  ext =
    ''
(dir, filename, ext) = fileparts('results.txt.back')
  dir =
    ''
  filename =
    results.txt
  ext =
    .back
```

See also

`fullfile`, `filesep`

filesep

File path separator.

Syntax

```
ch = filesep
```

Description

`filesep` gives the character used as separator between directories and files in paths. It depends on the operating system: a slash on unix (including Mac OS X and Linux), and a backslash on Windows.

See also

`fullfile`, `fileparts`

fullfile

File path construction.

Syntax

```
path = fullfile(p1, p2, ...)
```

Description

`fullfile` constructs a file path by concatenating all its string arguments, removing separators when missing. At least one input argument is required.

Examples

```
fullfile('/home/tom/', 'report.txt')
  /home/tom/report.txt
fullfile('/home/tom', 'data', 'meas1.csv')
  /home/tom/data/meas1.csv
```

See also

`fileparts`, `filesep`

10.36 XML Functions

This section describes functions which import XML data. Two separate sets of functions implement two approaches to parse XML data:

- Document Object Model (DOM): XML is loaded entirely in memory from a file (`xmlread`) or a character string (`xmlreadstring`). Additional functions permit to traverse the DOM tree and to get its structure, the element names and attributes and the text.

- Simple API for XML (SAX): XML is parsed from a file descriptor (`saxnew`) and events are generated for document start and end, element start and end, and character sequences.

With both approaches, creation and modification of the document are not possible.

DOM

Two opaque types are implemented: DOM nodes (including document, element and text nodes), and attribute lists. A document node object is created with the functions `xmlreadstring` (XML string) or `xmlread` (XML file or other input channel). Other DOM nodes and attribute lists are obtained by using DOM methods and properties.

Methods and properties of DOM node objects

Method	Description
`fieldnames`	List of property names
`getElementById`	Get a node specified by id
`getElementsByTagName`	Get a list of all descendent nodes of the given tag nam
`subsref`	Get a property value
`xmlrelease`	Release a document node

Property	Description
`attributes`	Attribute list (opaque object)
`childElementCount`	Number of element children
`childNodes`	List of child nodes
`children`	List of element child nodes
`depth`	Node depth in document tree
`documentElement`	Root element of a document node
`firstChild`	First child node
`firstElementChild`	First element child node
`lastChild`	Last child node
`lastElementChild`	Last element child node
`line`	Line number in original XML document
`nextElementSibling`	Next sibling element node
`nextSibling`	Next sibling node
`nodeName`	Node tag name, `'#document'`, or `'#text'`
`nodeValue`	Text of a text node
`offset`	Offset in original XML document
`ownerDocument`	Owner DOM document node
`parentNode`	Parent node
`previousElementSibling`	Previous sibling element node
`previousSibling`	Previous sibling node
`textContent`	Concatenated text of all descendent text nodes
`xml`	XML representation, including all children

A document node object is released with the `xmlrelease` method. Once a document node object is released, all associated node objects become invalid. Attribute lists and native LME types (strings and numbers) remain valid.

Methods and properties of DOM attribute list objects

Method	Description
`fieldnames`	List of attribute names
`length`	Number of attributes
`subsref`	Get an attribute

Properties of attribute lists are the attribute values as strings. Properties whose name is compatible with LME field name syntax can be retrieved with the dot syntax, such as `attr.id`. For names containing invalid characters, such as accented letters, or to enumerate unknown attributes, attributes can be accessed with indexing, with either parenthesis or braces. The result is a structure with two fields `name` and `value`.

SAX

XML is read from a file descriptor, typically obtained with `fopen`. The next event is retrieved with `saxnext` which returns its description in a structure.

getElementById

Get a node specified by id.

Syntax

```
node = getElementById(root, id)
```

Description

`getElementById(root,id)` gets the node which is a descendant of node `root` and whose attribute `id` matches argument `id`. It throws an error if the node is not found.

In valid XML documents, every id must be unique. If the document is invalid, the first element with the specified id is obtained.

See also

`xmlread`, `getElementsByTagName`

getElementsByTagName

Get a list of all descendent nodes of the given tag name.

Syntax

```
node = getElementsByTagName(root, name)
```

Description

`getElementsByTagName(root,name)` collects a list of all the element nodes which are direct or indirect descendants of node `root` and whose name matches argument `name`.

Examples

```
doc = xmlreal('<p>Abc <b>de</b> <i>fg <b>hijk</b></i></p>');
b = getElementsByTagName(doc, 'b')
  b =
    {DOMNode,DOMNode}
b2 = b{2}.xml
  b2 =
    <b>hijk</b>
xmlrelease(doc);
```

See also
xmlread, getElementById

saxcurrentline

Get current line number of SAX parser.

Syntax
```
n = saxcurrentline(sax)
```

Description
saxcurrentline(sax) gets the current line of the XML file parsed by the SAX parser passed as argument. It can also be used after an error.

See also
saxcurrentpos, saxnew, saxnext

saxcurrentpos

Get current position in input stream of SAX parser.

Syntax
```
n = saxcurrentpos(sax)
```

Description
saxcurrentpos(sax) gets the current position of the XML file parsed by the SAX parser passed as argument (the number of bytes consumed thus far). It can also be used after an error.

The value given by saxcurrentpos differs from the result of ftell on the file descriptor, because the SAX parser input is buffered.

See also
saxcurrentline, saxnew, saxnext

saxnew

Create a new SAX parser.

Syntax
```
sax = saxnew(fd)
sax = saxnew(fd, Trim=t, HTML=h)
```

Description

`saxnew(fd)` create a new SAX parser to parse XML from file descriptor `fd`. The parser is an opaque (non-numeric) type. Once it is not needed anymore, it should be released with the `saxrelease` function.

Named argument `Trim` (a boolean value) specifies if white spaces are trimmed around tags. The default value is `false`.

Named argument `HTML` (a boolean value) specifies HTML mode. The default value is false (XML mode). HTML mode has the following differences with respect to XML mode:

- unknown entities and less-than characters not followed by tag names are considered as plain text;
- attribute values can be missing (same as attribute names) or unquoted;
- tag and attribute names are converted to lowercase;
- text following a start `script` tag is not interpreted until the closing `script` tag (the litteral character sequence `</script>`, possibly with spaces before `>`).

This can be used for the lowest level of a rudimentary HTML parser.

Example

```
fd = fopen('data.xml');
sax = saxnew(fd);
while true
  ev = saxnext(sax);
  switch ev.event
    case 'docBegin'
      // beginning of document
    case 'docEnd'
      // end of document
      break;
    case 'elBegin'
      // beginning of element ev.tag with attr ev.attr
    case 'elEnd'
      // end of element ev.tag
    case 'elEmpty'
      // empty element ev.tag with attr ev.attr
    case 'text'
      // text element ev.text
  end
end
saxrelease(sax);
fclose(fd);
```

See also

`saxrelease`, `saxnext`, `xmlread`

saxnext

Get next SAX event.

Syntax

```
event = saxnext(sax)
```

Description

`saxnext(sax)` gets the next SAX event and returns its description in a structure. Argument `sax` is the SAX parser created with `saxnew`.

The event structure contains the following fields:

event Event type as a string: `'docBegin'`, `'docEnd'`, `'elBegin'`, `'elEnd'`, `'elEmpty'`, or `'text'`.

tag For `'elBegin'`, `'elEnd'` and `'elEmpty'`, element tag.

attr For `'elBegin'` and `'elEmpty'`, structure array containing the element attributes. Each attribute is defined by two string fields, `name` and `value`.

text For `'text'`, text string.

See also

`saxnew`, `saxcurrentline`

saxrelease

Release a SAX parser.

Syntax

```
saxrelease(sax)
```

Description

`saxrelease(sax)` releases the SAX parser `sax` created with `saxnew`.

See also

`saxnew`

xmlread

Load a DOM document object from a file descriptor.

Syntax

```
doc = xmlread(fd)
```

Description

`xmlread(fd)` loads XML to a new DOM document node object by reading a file descriptor until the end, and returns a new document node object. The file descriptor can be closed before the document node object is used. Once the document is not needed anymore, it should be released with the `xmlrelease` method.

Example

Load an XML file `'doc.xml'` (this assumes support for files with the function `fopen`).

```
fd = fopen('doc.xml');
doc = xmlread(fd);
fclose(fd);
root = doc.documentElement;
...
xmlrelease(doc);
```

See also

`xmlreadstring`, `xmlrelease`, `saxnew`

xmlreadstring

Parse an XML string into a DOM document object.

Syntax

```
doc = xmlreadstring(str)
```

Description

`xmlreadstring(str)` parses XML from a string to a new DOM document node object. Once the document is not needed anymore, it should be released with the `xmlrelease` method.

Examples

```
xml = '<a>one <b id="x">two</b> <c id="y" num="3">three</c></a>';
doc = xmlreadstring(xml)
  doc =
  DOM document
root = doc.documentElement;
root.nodeName
  ans =
    a
root.childNodes{1}.nodeValue
  ans =
    one
root.childNodes{2}.xml
  ans =
    <b id="x">two</b>
a = root.childNodes{2}.attributes
  a =
  DOM attributes (1 item)
a.id
  x
getElementById(doc,'y').xml
  <c id="y" num="3">three</c>
xmlrelease(doc);
```

See also

`xmlread`, `xmlrelease`

xmlrelease

Release a DOM document object.

Syntax

```
xmlrelease(doc)
```

Description

`xmlrelease(doc)` releases a DOM document object. All DOM node objects obtained directly or indirectly from it become invalid.

Releasing a node which is not a document has no effect.

See also

`xmlreadstring`, `xmlread`

10.37 Search Path Function

This section describes the functions used to setup the path of directories where libraries are searched.

path

Get or change path of library files.

Syntax

```
path
str = path
path(str)
path(str1, str2)
```

Description

Without arguments, `path` displays the path of all directories where library files are searched by `use` and `useifexists`.

With an output argument, `path` returns all the paths separated by semicolons.

With an input argument, `path(p)` sets the paths to the contents of string `p`, which must be a list of semicolon-separated paths.

With two input arguments, `path(p1,p2)` sets the paths to those contained in strings `p1` and `p2`. With this syntax, one of `p1` or `p2` is typically a call to `path` itself. This permits to prepend or append new paths to the existing ones.

If a path is a string which does not contain the percent character, the full path of the file is obtained by concatenating that string, the directory separator specific to the operating system (backslash on Windows and slash on macOS and Unix) unless the path already has a trailing directory separator, and the file name with its extension. If the path contains at least one percent character, the full path is obtained by replacing the following sequences:

Sequence	Replaced by
%%	%
%b	base file name without extension
%f	filename with extension
%s	extension (file suffix)

Example

Append `/usr/local/lme` to the current path:

```
path(path, '/usr/local/lme');
```

Same effect with `%f`:

```
path(path, '/usr/local/lme/%f');
```

See also

`use`, `useifexists`

10.38 Time Functions

clock

Current date and time.

Syntax

```
t = clock
```

Description

`clock` returns a 1x6 row vector, containing the year (four digits), the month, the day, the hour, the minute and the second of the current local date and time. All numbers are integers, except for the seconds which are fractional. The absolute precision is plus or minus one second with respect to the computer's clock; the relative precision is plus or minus 1 microsecond on a Macintosh, and plus or minus 1 millisecond on Windows.

Example

```
clock
  1999 3 11 15 37 34.9167
```

See also

`posixtime`, `tic`, `toc`

posixtime

Current Posix time.

Syntax

```
t = posixtime
```

Description

`posixtime` returns the Posix time, the integral number of seconds since 1 February 1970 at 00:00:00 UTC, based on the value provided by the operating system.

Example

```
posixtime
  1438164887
```

See also

`clock`

tic

Start stopwatch.

Syntax

```
tic
t0 = tic
tic(CPUTime=true)
t0 = tic(CPUTime=true)
```

Description

Without output argument, `tic` resets the stopwatch. Typically, `tic` is used once at the beginning of the block to be timed, and `toc` at the end. `toc` can be called multiple times to get intermediate times.

With an output argument, `tic` gets the current state of the stopwatch. Multiple independent time measurements can be performed by passing this value to `toc`.

By default, `tic` and `toc` are based on the real, "wall-clock" time. `tic(CPUTime=true)` is based on CPU time instead, which gives more accurate results for non-multithreaded programs. Measurements made with wall-clock time and CPU time are independent and can be mixed freely.

See also

`toc`, `clock`

toc

Elapsed time of stopwatch.

Syntax

```
elapsed_time = toc
elapsed_time = toc(t0)
elapsed_time = toc(CPUTime=true)
elapsed_time = toc(t0, CPUTime=true)
```

Description

Without input argument, `toc` gets the time elapsed since the last execution of `tic` without output argument. Typically, `toc` is used at the end of the block of statements to be timed.

With an input argument, `toc(t0)` gets the time elapsed since the execution of `t0=tic`. Multiple independent time measurements, nested or overlapping, can be performed simultaneously.

With a named argument, `toc(CPUTime=true)` or `toc(t0,CPUTime=true)` use CPU time: `toc` measures only the time spent in the LME application. Other processes do not have a large impact. For instance, typing `tic(CPUTime=true)` at the command-line prompt, waiting 5 seconds, and typing `toc(CPUTime=true)` will show a value much smaller than 5; while typing `tic` and `toc` will show the same elapsed time a chronograph would. CPU time is usually more accurate for non-multithreaded code, and wall-clock time for multi-threaded code, or measurements involving devices or network communication.

Examples

Time spent to compute the eigenvalues of a random matrix:

```
tic; x = eig(rand(200)); toc
  0.3046
```

Percentage of time spent in computing eigenvalues in a larger program:

```
eigTime = 0;
s = [];
tic(CPUTime=true);
for i = 1:100
  A = randn(20);
  eigT0 = tic(CPUTime=true);
  s1 = eig(A);
  eigTime = eigTime + toc(eigT0, CPUTime=true);
  s = [s, s1];
end
totalTime = toc(CPUTime=true);
100 * eigTime / totalTime
  78.4820
```

See also

`tic`, `clock`

10.39 Date Functions

Date functions perform date and time conversions between the calendar date and the julian date.

The *calendar date* is the date of the proleptic Gregorian calendar, i.e. the calendar used in most countries today where centennial years are not leap unless they are a multiple of 400. This calendar was introduced by Pope Gregory XIII on October 5, 1582 (Julian Calendar, the calendar used until then) which became October 15. The calendar used in this library is proleptic, which means the rule for leap years is applied back to the past, before its introduction. Negative years are permitted; the year 0 does exist.

The *julian date* is the number of days since the reference point, January 1st -4713 B.C. (Julian calendar) at noon. The fractional part corresponds to the fraction of day after noon: a fraction of 0.25, for instance, is 18:00 or 6 P.M. The julian date is used by astronomers with GMT; but using a local time zone is fine as long as an absolute time is not required.

cal2julian

Calendar to julian date conversion.

Syntax

```
jd = cal2julian(datetime)
jd = cal2julian(year, month, day)
jd = cal2julian(year, month, day, hour, minute, second)
```

Description

`cal2julian(datetime)` converts the calendar date and time to the julian date. Input arguments can be a vector of 3 components (year, month and day) or 6 components (date and hour, minute and seconds), or scalar values provided separately. The result of `clock` can be used directly.

Example

Number of days between October 4 1967 and April 18 2005:

```
cal2julian(2005, 4, 18) - cal2julian(1967, 10, 4)
  14624
```

See also

`julian2cal`, `clock`

julian2cal

Julian date to calendar conversion.

Syntax

```
datetime = julian2cal(jd)
(year, month, day, hour, minute, second) = julian2cal(jd)
```

Description

`julian2cal(jd)` converts the julian date to calendar date and time. With a single output, the result is given a a row vector of 6 values for the year, month, day, hour, minute and second; with more output arguments, values are given separately.

Example

Date 1000 days after April 18 2005:

```
julian2cal(cal2julian(2005, 4, 18) + 1000)
  2008  1  13   0   0   0
```

See also

`cal2julian`

10.40 Threads

LME threads are code fragments which are executed concurrently. This section describes threads at the level of the LME virtual machine, scheduled in a single OS-level thread. For multiple OS-level threads, see Parallel execution.

There is always one main thread; additional threads can be created and killed at any time by any thread. Their number is limited only by the available memory. A new thread is created with the function to be executed (as a function reference, a function name, or an inline function) and optional arguments.

Threads communicate together by exchanging data in global variables or by calling functions with persistent variables. Semaphores can be created to avoid reading a variable while it is being modified by another thread, or for solving other synchronization problems. Thread switching occurs between elementary operations (such as the execution of a function or an operator, or the branch implied by a conditional or iteration command). For example, the simple expression `x(end)` where x is a global variable (which gets the last element of the vector x) may not give the expected result if another thread changes the

size of x between the evaluation of end and the retrieval of the vector element. In that case, a semaphore should be locked around `x(end)` and around the modification of x.

semaphoredelete

Delete a semaphore.

Syntax

```
semaphoredelete(id)
```

Description

`semaphoredelete(id)` deletes a semaphore which was created with `semaphorenew`, ignoring its locked or unlocked state. It is an error to use its id afterwards.

See also

`semaphorenew`

semaphorelock

Lock a semaphore.

Syntax

```
semaphorelock(id)
b = semaphorelock(id)
```

Description

`semaphorelock(id)` locks the semaphore specified by `id`, so that it cannot be locked again before being unlocked with `semaphoreunlock`. Without output argument, `semaphorelock` waits until another thread unlocks it. With an output argument, it locks it and returns true if the semaphore is not already locked, and it returns immediately false otherwise as an indication of failure.

See also

`semaphorenew`, `semaphoreunlock`

semaphorenew

Create a new semaphore.

Syntax

```
id = semaphorenew
```

Description

A semaphore is a mechanism which gives to a thread the exclusive access to a resource. To request the access, the semaphore is locked. Once it is locked, no other thread can lock it until it is unlocked. An attempt to lock a semaphore while it is already locked will either wait until the semaphore is unlocked, or fail immediately.

Semaphores are typically used when data shared by two or more threads are modified in several steps (these data can be stored in a global variable or in a file).

`semaphorenew` creates a new semaphore and returns the identifier which should be used with all other semaphore-related functions.

Example

The code below creates two threads which both use a global variable `counter`. The first thread continuously increments it, while the second thread resets it to 0 every second.

```
threadId1 = threadnew(@t1);
threadId2 = threadnew(@t2);
```

Functions `t1` and `t2` are defined as

```
function t1
  global counter;
  while true
    counter = counter + 1;
  end
function t2
  global counter;
  while true
    threadsleep(1);
    counter = 0;
  end
```

The problem with the code above is that the execution of the first thread can be interrupted right after `counter+1` has been evaluated (with a result of 3742197, for instance), but before the result has been assigned to `counter`. If the second thread resets `counter` at that time, the first thread will immediately undo that by assigning 3742197 to `counter`. To avoid that, a semaphore should be used to delay resetting `counter` to after the new value is assigned in the first thread:

```
function t1
  global counter;
```

```
  while true
    semaphorelock(countersem);
    counter = counter + 1;
    semaphoreunlock(countersem);
  end
function t2
  global counter;
  while true
    threadsleep(1);
    semaphorelock(countersem);
    counter = 0;
    semaphoreunlock(countersem);
  end
```

The semaphore is created before the threads and its identifier stored in global variable `countersem`:

```
global countersem;
countersem = semaphorenew;
threadId1 = threadnew(@t1);
threadId2 = threadnew(@t2);
```

See also

`semaphorelock`, `semaphoreunlock`, `semaphoredelete`

semaphoreunlock

Unlock a semaphore.

Syntax

```
semaphoreunlock(id)
```

Description

`semaphoreunlock(id)` unlocks the semaphore specified by `id`, so that it can be locked again by any thread. If a thread was blocked by executing `semaphorelock` on the semaphore which is unlocked, it will lock the thread and resume execution. If several threads are waiting on the same semaphore, only one of them resumes execution. There is no queue for waiting threads; which one resumes execution is unspecified.

See also

`semaphorelock`

threadkill

Kill a thread.

Syntax

```
threadkill(id)
```

Description

`threadkill(id)` interrupts execution of the thread specified by its `id` and discards the data which had been are allocated for it by `threadnew`.

See also

`threadnew`

threadnew

Create a new thread.

Syntax

```
id = threadnew(fun)
id = threadnew(fun, opt)
id = threadnew(fun, opt, par1, par2, ...)
```

Description

`threadnew(fun)` creates a new thread which will execute function `fun` without argument in parallel with the current thread and all other running threads. Function `fun` can be an inline function, a reference to function, or the name of a function. `threadnew` returns a thread id which is used by `threadkill`. The thread terminates when the function does (at the end of the function body or by executing `return`) or if is interrupted with `threadkill`. The output arguments of the function cannot be retrieved. If the thread produces a result, it should transmit it via another mean, such as global variables.

`threadnew(fun,opt)` specifies options created with `threadset`. Additional arguments, if any, are provided to function `fun` as its own arguments.

See also

`threadset`, `threadkill`, `info`

threadset

Options for thread creation.

Syntax

```
options = threadset
options = threadset(name1, value1, ...)
options = threadset(options0, name1, value1, ...)
```

Description

`threadset(name1,value1,...)` creates the option argument used by `threadnew`. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, `threadset` creates a structure with all the default options. Note that `threadnew` also interprets the lack of an option argument, or the empty array `[]`, as a request to use the default values.

When its first input argument is a structure, `threadset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options:

Name	Default	Meaning
`StackSize`	`8192`	size allocated for the execution stack
`Priority`	`0`	thread priority, from -20 (lowest) to 20 (highest)
`Running`	`true`	true for a running thread, false if suspended
`Sandbox`	`false`	true for execution in a sandbox

A thread with Sandbox set to true runs with the same restrictions as code executed with `sandbox`.

See also

`threadnew`, `sandbox`

threadsleep

Wait for a specified amount of time.

Syntax

```
threadsleep(time)
threadsleep(time, true)
```

Description

`threadsleep(time)` waits at least the specified amount of time (specified in seconds with a resolution which depends on the platform), permitting other threads to run. `threadsleep(time,false)` has the same effect.

`threadsleep(time,true)` waits for the specified amount of time relatively with the end of the previous `threadsleep` of the current thread. By having `threadsleep(time,true)` in a loop, a fixed execution frequency can be achieved even if the processing time required for the other statements in the loop can change (provided that the amount of time specified for `threadsleep` is at least as large as the processing time).

See also

`threadnew`

10.41 Parallel

Parallel execution extends LME to execute multiple tasks concurrently. It takes advantage of computers with mutiple cores per microprocessor, and/or multiple microprocessors.

Parallel execution relies on the following object classes:

`task` Tasks are the basic unit of work. They consist in an LME function call with input and output arguments.

`job` Jobs are groups of tasks seen as a single unit. A job is submitted as a whole, its tasks are executed independently and the outputs are collected and made available once all of them have finished running.

`cluster` Clusters are groups of workers. Each job is associated with a single cluster.

Worker Workers are the software and hardware resource units required to execute tasks. A worker contains a complete LME environment. If a job contains more tasks than workers, workers execute multiple tasks one after the other.

Currently, workers are OS-level threads which the OS schedules on the available microprocessor cores. For tests, serial execution can be selected with `parcluster('serial')`.

Workers are internal objects not associated with an LME class.

Examples

Task defined as an anonymous function

The first example can be run from the command window of Sysquake. Function `fun` estimates pi with Monte Carlo integration, by counting the ratio of `m` uniformly-distributed random points in the unit square whose distance to the origin is smaller than 1. A job with `n` independent tasks is run, and we wait until it is completed; at the end, its state will be either `'finished'` if all tasks are successful or `'failed'` if there has been an error. The results of all tasks are collected in `rl`, converted from a list of lists to a double array `r`, and their mean and the expected standard deviation of their mean is computed. Note that the pseudo-random number generator of each worker is automatically initialized with a different seed; therefore the pseudo-random numbers used by each task are independent.

```
cluster = parcluster('local')
job = createJob(cluster)
m = 1e5;
n = 20;
fun = @(m) nnz(rand(m,1).^2+rand(m,1).^2<1)*4/m;
for i = 1:n
  createTask(job, fun, 1, {m});
end
submit(job);
wait(job, 'finished');
jobState = job.State
rl = fetchOutputs(job);
delete(job);
r = list2num([rl{:}]);
piApproxMean = mean(r)
piApproxStd = std(r) / sqrt(length(r))
```

User functions

Workers run in separate LME instances. To give tasks access to user-defined functions, two mechanisms are provided:

Automatic library import By default, worker LME instances are initialized with the same libraries which have been explicitly imported with `use` in the context which calls `submit`, as well as the library containing the function calling `submit` itself. This means that you can use your functions transparently.

In the next example, the task function is passed as a function reference, not as an anonymous function as in the first example above. It produces four output arguments, the number of points in unit disks centered around the four unit square corners.

```
function (r00, r01, r10, r11) = task(m)
  x = rand(m, 1);
  y = rand(m, 1);
  r00 = nnz(x.^2 + y.^2 < 1) * 4 / m;
  r01 = nnz((1 - x).^2 + y.^2 < 1) * 4 / m;
  r10 = nnz(x.^2 + (1 - y).^2 < 1) * 4 / m;
  r11 = nnz((1 - x).^2 + (1 - y).^2 < 1) * 4 / m;
```

This function can be stored in a library or in the function block of an SQ file (*not* copied/pasted as a whole directly in the command window of Sysquake). The following statements are very similar to the first example. They define the task as a function reference instead of an anonymous function, and use the arithmetic mean of the four outputs (`rl` is a list whose elements contain the output arguments of each task, as a list of four values).

```
cluster = parcluster('local');
job = createJob(cluster);
m = 1e5;
n = 20;
for i = 1:n
  createTask(job, @task, 4, {m});
end
submit(job);
wait(job, 'finished');
jobState = job.State
rl = fetchOutputs(job);
delete(job);
r = list2num(map(@mean, [rl{:}]));
piApprox = mean(r)
piApproxStd = std(r) / sqrt(length(r))
```

To prevent workers from being initialized with the libraries of the calling LME context, the job property `AutoUseLib` should be set to `false`.

Worker startup commands Arbitrary commands can be run by each worker when a job is submitted. Libraries can be specified there, as well as other settings such as format, global variables etc.

The following example shows how the format of floating-point numbers is changed.

```
cluster = parcluster('local');
job = createJob(cluster);
job.Startup = 'format long';
task = createTask(job, @() disp(pi), 0, {});
task.CaptureDiary = true;
submit(job);
```

```
    wait(job, 'finished');
    taskDiary = task.Diary
```

Diary

By default, text output to stdout and stderr produced by tasks is discarded. To get it, the `CaptureDiary` property of task objects should be set to true; once the task is completed, either in `'finished'` or `'failed'` state, the task `Diary` property contains all the output to stdout and stderr (file descriptor 1 or 2) as a string.

```
cluster = parcluster('local');
job = createJob(cluster);
for i = 1:3
  task = createTask(job, @() disp(rand), 0, {});
  task.CaptureDiary = true;
end
submit(job);
wait(job, 'finished');
map(@(task) disp(task.Diary), job.Tasks)
```

Sysquake background processing

In Sysquake, parallel execution can be used to carry out heavy computations without adversary effect on the responsiveness of the user interface. The following SQ file shows how the idle handler is used to supervise parallel jobs, using and updating SQ variables shared with a figure. The parallel job computes an approximation of pi with a task function passed by reference. The idle handler submits it continuously with the number of tasks n specified with a slider in the figure.

```
variable n = 20
variable piEst = 0
variable piStd = 0
variable job = null

idle (piEst, piStd, job) = idle(piEst, piStd, job, n)

figure "Parallel Test"
  draw drawFig(n, piEst, piStd)
  mousedrag 1 n = round(_x1)

functions
{@

function drawFig(n, piEst, piStd)
  settabs('Nb tasks 9999\t');
  slider(sprintf('Nb tasks %d', n), n, [1, 500], 'l', id=1);
  settabs('Estimation std dev:  \t');
  text(sprintf('Estimation of pi:\t%.6f', piEst));
```

```
  text(sprintf('Estimation std dev:\t%.6f', piStd));

function p = taskFun(m)
  x = rand(m, 1);
  y = rand(m, 1);
  p = nnz(x.^2 + y.^2 < 1) * 4 / m;

function (piEst, piStd, job) = idle(piEst, piStd, job, n)
  if job ~== null
    switch job.State
    case 'finished'
      rl = fetchOutputs(job);
      r = list2num([rl{:}]);
      piEst = mean(r);
      piStd = std(r) / sqrt(length(r));
    case 'failed'
      piEst = nan;
      piStd = nan;
    otherwise
      // still running, don't update figure
      cancel(false);
    end
    delete(job);
  end

  m = 1e5;
  cluster = parcluster('local');
  job = createJob(cluster);
  for i = 1:n
    createTask(job, @taskFun, 1, {m});
  end
  submit(job);

@}
```

batch

Create a job with a single task.

Syntax

```
job = batch(fun, nargout, arginList)
```

See also

`createJob`, `createTask`

cancel

Cancel a job.

Syntax

```
cancel(job)
```

Description

`cancel(job)` cancels a job: pending and running tasks are brought to the failed state (running tasks are interrupted). Tasks which are already finished are left unchanged.

See also

`createJob`, `submit`

createJob

Create a new job.

Syntax

```
job = createJob(cluster)
```

Description

`createJob(cluster)` creates a new job to be run on the specified cluster. It returns the job object. The next steps are typically to add tasks with `createTask`, to execute the job with `submit`, to fetch results with `fetchOutputs`, and to release resources allocated for the job with `delete`.

Objects of class `job` have the following properties:

`AutoUseLib` `true` to let workers use the same libraries as those imported in the current context of the function which calls `submit`, including the library of the function itself (default)

`ElapsedTime` wall-clock time spent from call to `submit` to the completion of the last task

`ID` job id (integer number)

`Parent` cluster object the job is associated with

`Startup` code executed at startup by each worker

`State` current job state as a string

`Tasks` list of tasks belonging to the job

Properties `AutoUseLib` and `Startup` can be set; all other properties are read-only.

Example

Create a job with the cluster `'local'`:

```
cluster = parcluster('local');
job = createJob(cluster);
```

See also

`parcluster`, `createTask`

createTask

Create a new task.

Syntax

```
task = createTask(job, fun, nargout, arginList)
```

Description

`createTask(job,fun,nargout,arginList)` creates a new task for the specified job. A single job can contain as many jobs as required. The task is specified by the last three arguments:

`fun` function, as a function reference or an anonymous or inline function

`nargout` number of output arguments produced by `fun` (non-negative integer)

`arginList` list of input arguments

`createTask` returns a task object, which can often be ignored because a list of all the tasks created for a job can be retrieved with function `findTask` or job property `Tasks`. Task objects have the following properties:

`CaptureDiary` `true` to capture task output in `Diary` property

`Diary` task output as a string (empty if `CaptureDiary` is `false`)

`ErrorIdentifier` error identifier if state is `'failed'`

`ErrorMessage` error message if state is `'failed'`

`Function` function

`ID` task id (integer number)

`InputArguments` list of input arguments

`NumOutputArguments` number of output arguments

`OutputArguments` list of output arguments if state is `'finished'`

`Parent` job object the task belongs to

`State` current task state as a string

Property `CaptureDiary` can be set; all other properties are read-only.

Examples

Add 10 tasks which compute the mean of n=1e5 pseudo-random values:

```
for i = 1:10
  createTask(job, @(n) mean(rand(1,n)), 1, {1e5});
end
```

Run a job with 1000 tasks which compute the (local) minima of function `y=fun(X)`, where X is a vector of length 10, y is a scalar, and the starting points are chosen randomly in the unit hypercube. Function `fminsearch` is asked two output arguments, X and y. All results are collected in 10-by-1000 array Xa and 1-by-1000 array ya. The index of the minimum value of ya gives an approximation of the global minimum `Xopt` and `yopt`.

```
cluster = parcluster('local');
job = createJob(cluster);
for i = 1:1000
  createTask(job, @fminsearch, 2, {@fun, rand(10,1)});
end
submit(job);
wait(job, 'finished');
rl = fetchOutputs(job);
delete(job);
r = list2num([rl{:}]);
Xa = [map(@(taskOutputs) taskOutputs{1}, rl){:}];
ya = [map(@(taskOutputs) taskOutputs{2}, rl){:}];
(yopt, ixopt) = min(ya);
Xopt = Xa(:, ixopt);
```

See also

`createJob`

delete

Delete a job or a task.

Syntax

```
delete(job)
delete(task)
```

`delete(job)` deletes the specified job and all its tasks. It should be called once the job is completed to release its resources.

`delete(task)` deletes the specified task. It is superfluous if `delete(job)` is called.

See also

`createJob`, `createTask`

fetchOutputs

Fetch output of all tasks in a job.

Syntax

```
outputList = fetchOutputs(job)
```

Description

`fetchOutputs(job)` gets all the task outputs. Its value is a list where each elements corresponds to a task. For tasks whose state is `'finished'`, it is itself a list of `nargout` values, where `nargout` is the number of task output arguments specified by `createTask`. For tasks whose state is not `'finished'`, it is an empty list.

Usually, `fetchOutput` should be called when the job is completed, i.e. when its state is `'finished'` or `'failed'`. This can be achieved by calling `wait(job,'finished')` or by checking the job `State` property.

See also

`createTask`, `createJob`, `wait`

findTask

Find tasks in a job.

Syntax

```
tasks = findTask(job)
(pendingTasks, runningTasks, completedTasks) = findTask(job)
```

Description

With one output argument, `findTask(job)` gets a list of all the tasks defined for the specified job. It gives the same result as the job property `Tasks`.

With three output arguments, `findTask(job)` gets separate lists for the tasks whose state is `'pending'` or `'queued'`, `'running'`, and `'finished'` or `'failed'`.

See also

`createJob`

parcluster

Get a cluster.

Syntax

```
cluster = parcluster
cluster = parcluster(name)
```

Description

`parcluster(name)` gives the cluster whose name is specified as a string argument. The following names are valid:

`'local'` **(default)** Workers running in threads dispatched by the operating system on the cores of the local computer. Each worker runs in a separate LME context, which is reset at the beginning of each job submission. To make Monte-Carlo processes easier, the pseudo-random generator of each worker is initialized with a different seed.

`'serial'` Serial execution of each task in the base LME environment. Function `submit` returns only once all the tasks have been completed. Output is displayed immediately and not stored in the task's diary, regardless of task property `CaptureDiary`.

Without input argument, `parcluster` gives the default cluster, which can be changed with `pardefaultcluster`.

Objects of class `cluster` have the following properties:

`Jobs` list of jobs

`MaxNumWorkers` maximum number of workers

`Name` cluster name

`NumWorkers` number of workers; should be set typically to the number of CPU times the number of cores per CPU times the number of hardware threads per core (hyperthreading or multithreading)

`LMEMemory` **(`'local'` only)** amount of memory in bytes allocated to LME by each worker, between 256KB and 2047MB (default: 32MB)

Properties `NumWorkers` and `LMEMemory` can be set; all other properties are read-only.

See also

`pardefaultcluster`, `createJob`

pardefaultcluster

Get or set default cluster.

Syntax

```
cluster = pardefaultcluster
pardefaultcluster(cluster)
pardefaultcluster(name)
```

Description

Without input argument, `pardefaultcluster` gives the current default cluster, i.e. the cluster returned by `parcluster` when no name is given.

With an input argument, `pardefaultcluster(cluster)` sets `cluster` as the default cluster. With a string argument, `pardefaultcluster(name)` is equivalent to `pardefaultcluster(parcluster(name))`.

See also

`parcluster`

submit

Submit a job for execution.

Syntax

```
submit(job)
```

Description

`submit(job)` submits the specified job so that its tasks are executed on the cluster `job` belongs to. It returns immediately.

A job can be submitted only once.

Example

Submit a job, wait until all tasks have completed, and retrieve its results.

```
submit(job);
wait(job, 'finished');
results = fetchOutputs(job);
```

See also

`createJob`, `wait`, `cancel`, `fetchOutputs`

wait

Wait until a job state has changed.

Syntax

```
wait(job)
wait(job, state)
```

Description

With one input argument, `wait(job)` waits until the job state has changed. If the job is already in state `'finished'` or `'failed'`, it returns immediately.

With two input arguments, `wait(job,state)` waits until the job state has reached the state specified in the second argument. States `'finished'` and `'failed'` are considered to be equivalent.

Examples

Submit a job and wait until it is completed.

```
submit(job);
wait(job, 'finished');
```

After a job has been submitted, instead of waiting until it is completed with `wait`, the percentage of completed tasks can be obtained with `findTask`. The following lines can be repeated.

```
(p, r, c) = findTask(job);
p100Completed = 100 * length(c) / (length(p) + length(r) + length(c))
```

After a job is completed, if it has failed, i.e. if at least one task has thrown an error, get the error identifier and message of the first failed task and throw the error.

```
submit(job);
wait(job, 'finished');
errId = '';
if job.State === 'failed'
  for task = job.Tasks
    if task.State === 'failed'
      errId = task.ErrorIdentifier;
      errMsg = task.ErrorMessage;
      break;
    end
  end
else
  outputs = fetchOutputs(job);
end
delete(job);
if errId
  error(errId, errMsg);
end
```

See also

`submit`, `createJob`

10.42 Sysquake Graphics

The main goal of Sysquake is the interactive manipulation of graphics. Hence, graphical functions play an important role in SQ files. There are low-level commands for basic shapes as well as high-level commands for more specialized plots:

Low-level commands Low-level commands add simple shapes such as lines, marks, polygons, circles and images. With them, you can display virtually everything you want. Arguments of these commands are such that it is very easy to work globally with matrices without computing each value sequentially in a loop.

High-level commands High-level commands perform some computation of their own to process their arguments before displaying the result. This has two benefits: first, the code is simpler, more compact, and faster to develop. Second, command execution is faster, because the additional processing is not interpreted by LME, but implemented as native machine code. The information related to interactive manipulation is often easier to use, too. Most of these functions are related to automatic control and signal processing.

Commands which display data in a figure can be used only in the draw handlers or from the command line interface. Conversely, commands which change the number of subplots or the subplots themselves cannot be used in the draw handlers.

Here is the list of these two groups of commands:

Reserved for draw handlers and command-line interface

2D low-level drawing commands

```
activeregion    colormap    pcolor
area            contour     plot
bar             fplot       polar
barh            image       quiver
circle          line        text
```

2D high-level drawing commands

```
bodemag       dsigma     nyquist
bodephase     dstep      nyquist
dbodemag      erlocus    plotroots
dbodephase    hgrid      rlocus
dimpulse      hstep      sgrid
dinitial      impulse    sigma
dlsim         initial    step
dnichols      lsim       zgrid
dnyquist      ngrid
```

Note that some of these functions can be used in non-draw handlers when the result is retrieved in output arguments and not displayed.

Scaling and labels

```
altscale    plotoption      scaleoverview
label       scale           ticks
legend      scalefactor     title
```

3D

```
contour3    plot3       surf
line3       plotpoly
mesh        sensor3
```

3D scaling and lighting

```
camdolly    camroll      daspect
camorbit    camtarget    lightangle
campan      camup        material
campos      camva
camproj     camzoom
```

Controls

```
button       pushbutton    slider
popupmenu    settabs       textfield
```

Cannot be used in draw handlers

```
clf             scalesync      subplotprops
defaultstyle    subplot        subplotparam
figurestyle     subplots       subplotspring
redraw          subplotpos     subplotsync
```

Commands from both groups can be typed in the command line interface. For example, to plot the step response of the continuous-time system whose Laplace transform is $1/(s^3 + 2s^2 + 3s + 4)$, type

```
> clf
> step(1, [1,2,3,4])
```

The first command, only valid from the command line interface (or indirectly in a function called from the command line interface), clears the figure window (necessary if there was already something displayed); the second command integrates the system over a suitable range with a unit entry and null initial conditions.

10.43 Remarks on graphics

Many functions which produce the display of graphical data accept two optional arguments: one to specify the style of lines and symbols, and one to identify the graphical element for interactive manipulation.

Style

The style defines the color, the line dash pattern (for continuous traces) or the shape (for discrete points) of the data.

There are two different ways to specify the style. The first one, described below, is with a single string. The second one, introduced with Sysquake 5, is with an option structure built with `plotset` or directly with named arguments; it is more verbose, hence easier to understand, and gives access to more settings, such as line width or marker colors.

The possible values in a style string are given below. Note that the color is ignored on some output devices (such as black and white printers) and the dash pattern is used only on high-resolution devices (such as printers or EPS output). The color code is lowercase for thin lines and uppercase for thicker lines on devices which support it.

Color	**String**
black	`k`
blue	`b`
green	`g`
cyan	`c`
red	`r`
magenta	`m`
yellow	`y`
white	`w`
RGB	`h(rrggbb)`
RGB	`h(rgb)`

Dash Pattern	**String**
solid	_ (underscore)
dashed	-
dotted	:
dash-dot	!

Shape	**String**
none (invisible)	(space)
point	`.`
circle	o
cross	x
plus	+
star	*
triangle up	^
triangle down	v
square	[
diamond	<

Miscellaneous	**String**
stairs	s
stems	t
fill	f
arrow at end	a
arrows at beginning and end	A

Color `'h(rrggbb)'` specifies a color by its red, green, and blue components; each of them is given by two hexadecimal digits from 00 (minimum brightness) to ff (maximum brightness). Color `'h(rgb)'` specifies each component with a single hexadecimal digit. For example, `'h(339933)'` and `'h(393)'` both specify the same greenish gray. Like for other colors, an uppercase 'H' means that the line is thick.

Style `'s'` (stairs) is supported only by the `plot`, `dimpulse`, `dstep`, `dlsim`, and `dinitial` functions. It is equivalent to a zero-order hold, i.e. two points are linked with a horizontal segment followed by a vertical segment.

Style `'t'` (stems) draws for each value a circle like `'o'` and a vertical line which connects it to the origin (in 2D plots, y=0 for linear scale or y=1 for logarithmic scale; in 3D plots, z=0). In polar plots, stems connects points to x=y=0.

Style `'f'` (fill) fills the shape instead of drawing its contour. Exactly how the shape is filled depends on the underlying graphics architecture; if the contour intersects itself, there may be holes.

Style `'a'` adds an arrow at the end of lines drawn by `plot`, and style `'A'` adds arrows to the beginning and the end. The arrow size depends only on the default character size, neither on the line length nor on the plot scale. Its color and thickness are the same as the line's.

Many graphical commands accept data for more than one line. If the style string contains several sequences of styles, the first line borrows its style from the first sequence, the second line, from the second sequence, and so on. If there are not enough styles, they are recycled. A sequence is one or two style specifications, one of them for the color and the other one for the dash pattern or the symbol shape, in any order. Sequences of two specifications are used if possible. Commas may be used to remove ambiguity. Here are some examples:

```
plot([0,1;0,1;0,1],[1,1;2,2;3,3],'k-r!')
```

The first line (from (0,1) to (1,1)) is black and dashed, the second line (from (0,2) to (1,2)) is red and dash-dot, and the third line (from (0,3) to (1,3)) is black and dashed again.

```
plot([0,1;0,1;0,1],[1,1;2,2;3,3],'rbk')
```

The first line is red, the second line is blue, and the third line is black.

```
plot([0,1;0,1;0,1],[1,1;2,2;3,3],'-br')
```

The first and third lines are blue and dashed, and the second line is red and solid.

```
plot([0,1;0,1;0,1],[1,1;2,2;3,3],':,H(cccccc)')
```

The first and third lines are dotted, and the second line is gray, solid, and thick.

Graphic ID

The second optional argument is the graphic ID. It has two purposes. First, it specifies that the graphic element can be manipulated by the user. When the user clicks in a figure, Sysquake scans all the curves which have a non-negative graphic ID (the default value of all commands is -1, making the graphical object impossible to grasp) and sets `_z0`, `_x0`, `_y0`, `_id`, and `_ix` such that they correspond to the nearest element if it is close enough to the mouse coordinates. Second, the argument `_id` is set to the ID value so that the mousedown, mousedrag, and mouseup handlers can identify the different objects the user can manipulate.

In applications without live interactivity, such as Sysquake Remote, the graphic ID argument is accepted for compatibility reasons, but ignored.

Scale

Before any figure can be drawn on the screen, the scale (or equivalently the portion of the plane which is represented on the screen) must be determined. The scale depends on the kind of graphics, and consequently is specified in the `draw` handler, but can be changed by the user with the zoom and shift commands. What the user specifies has always the priority. If he or she has not specified a new scale, the `scale` command found in the `draw` handler is used:

```
scale([xMin,xMax,yMin,yMax])
```

If `scale` is not used, or if some of the limits are NaN (not an number), a default scale is given by the plot commands themselves. If used, the scale command should always be executed before any plot command, because several of them use the scale to calculate traces only over the visible range or to adjust the density of the calculated points of the traces.

If you need to know the limits of the displayed area in your draw handler, use `scale` to get them right after setting the default scale, so that you take into account the zoom and shift specified by the user:

```
scale(optString, [defXMin, defXMax, defYMin, defYMax]);
sc = scale;
xMin = sc(1);
xMax = sc(2);
yMin = sc(3);
yMax = sc(4);
```

Grids

In addition to the scale ticks displayed along the bounding frame, grids can be added to give visual clues and make easier the interpretation of graphics. X and Y grids are vertical or horizontal lines displayed in the figure background. They can be switched on and off by the user in the Grid menu, or switched on by programs with the `plotoption` command (they are set off by default). In the example below, both X and Y grids are switched on:

```
plotoption xgrid
plotoption ygrid
plot(rand(1,10));
```

Commands which display grids for special kind of graphics are also available:

Command	**Intended use**
`hgrid`	`nyquist`, `dnyquist`
`ngrid`	`nichols`, `dnichols`
`sgrid`	`plotroots`, `rlocus` (continuous-time)
`zgrid`	`plotroots`, `rlocus` (discrete-time)

They can be used without argument, to let the user choose the level of details: *none* means the command does not produce any output; *basic* is the default value and gives a simple, non-obstructive hint (a single line or a circle); and *full* gives more details. To change by program the default level of details (basic), `plotoption` is used. In the example below, the grid for the complex plane of the *z* transform is displayed with full details. Once the figure is displayed, the user is free to reduce the level of details with the Grid menu.

```
scale('equal', [-2,2,-2,2]);
zgrid;
plotoption fullgrid;
plotroots([1,-1.5,0.8]);
```

10.44 Base Graphical Functions

activeregion

Region associated with an ID.

Syntax

```
activeregion(xmin, xmax, ymin, ymax, id)
activeregion(X, Y, id)
```

Description

The command `activeregion` defines invisible regions with an ID for interactive manipulations in Sysquake. Contrary to most other graphical objects, a hit is detected when the mouse is inside the region, not close like with points and lines.

`activeregion(xmin,xmax,ymin,ymax,id)` defines a rectangular shape.

`activeregion(X,Y,id)` defines a polygonal shape. The start and end points do not have to be the same; the shape is closed automatically.

Example

Rectangular button. If an ID was given to `plot` without `activeregion`, a hit would be detected when the mouse is close to any of the four corners; with `activeregion`, a hit is detected when the mouse is inside the rectangle.

```
plot([50, 70, 70, 50, 50], [10, 10, 30, 30, 10]);
activeregion(50, 70, 10, 30, id=1);
```

See also

`plot`, `image`

altscale

Alternative y scale for 2D plots.

Syntax

```
altscale(b)
```

Description

`altscale(b)` selects an alternative y scale whose axis and labels are displayed on the right of the rectangular frame of 2D plots. Its input argument is a logical value which is true to select the alternative scale and false to revert to the primary scale.

Example

```
bar(1:5, rand(1, 5));
altscale(true);
plot(1:5, 3 * rand(1,5), 'R');
label('', 'y1', 'y2');
legend('y1\ny2', 'bfR');
```

See also

`scale`, `label`

area

Area plot.

Syntax

```
area(y)
area(x, y)
area(x, y, y0)
area(..., style)
area(..., style, id)
```

Description

With column vector arguments, `area(x,y)` displays the area between the horizontal axis `y=0` and the points given by `x` and `y`. When the second argument is an array with as many rows as elements in `x`, `area(x,Y)` displays the contribution of each column of `Y`, summed along each row. When both the first and second arguments are arrays of the same size, `area(X,Y)` displays independent area plots for corresponding columns of `X` and `Y` without summation.

With a single argument, `area(y)` takes integers 1, 2, ..., n for the horizontal coordinates.

With a third argument, `area(x,y,y0)` displays the area between the horizontal line `y=y0` and values defined by `y`.

The optional arguments `style` and `id` have their usual meaning. `area` uses default colors when argument `style` is missing.

Examples

Red area defined by points (1,2), (2,3), (3,1), and (5,2) above `y=0`; on top of it, blue area defined by points (1,2+1), (2,3+2) etc.

```
area([1;2;3;5],[2,1;3,2;1,5;2,1], 0, 'rb');
```

Two separate areas above `y=0.2` defined by points (1,2), (2,3), (3,1), (5,2); and (6,1), (7,2), (8,5), and (9,1).

```
area([1,6;2,7;3,8;5,9],[2,1;3,2;1,5;2,1], 0.2, 'rb');
```

See also

`plot`, `bar`, `hbar`

bar

Vertical bar plot.

Syntax

```
bar(y)
bar(x, y)
bar(x, y, w)
bar(..., kind)
bar(..., kind, style)
bar(......, id)
```

Description

`bar(x,y)` plots the columns of `y` as vertical bars centered around the corresponding value in `x`. If `x` is not specified, its default value is `1:size(y,2)`.

`bar(x,y,w)`, where `w` is scalar, specifies the relative width of each bar with respect to the horizontal distance between the bars; with values smaller than 1, bars are separated with a gap, while with values larger than 1, bars overlap. If `w` is a vector of two components `[w1,w2]`, `w1` corresponds to the relative width of each bar in a group (columns of `y`), and `w2` to the relative width of each group. Default values, used if `w` is missing or is the empty matrix `[]`, is 0.8 for both `w1` and `w2`.

`bar(...,kind)`, where `kind` is a string, specifies the kind of bar plot. The following values are recognized:

`'grouped'`	Columns of `y` are grouped horizontally (default)
`'stacked'`	Columns of `y` are stacked vertically
`'interval'`	Bars defined with min and max val.

With `'interval'`, intervals are defined by two consecutive rows of `y`, which must have an even number of rows.

The optional arguments `style` and `id` have their usual meaning. `bar` uses default colors when argument `style` is missing.

Examples

Simple bar plot (see Fig. 10.5):

```
bar([2,4,3,6;3,5,4,1]);
```

Stacked bar plot:

```
bar(1:4, magic(4), [], 'stacked');
```

Interval plot:

```
bar(1:4, [2,4,3,1;5,6,4,6], [], 'interval');
```

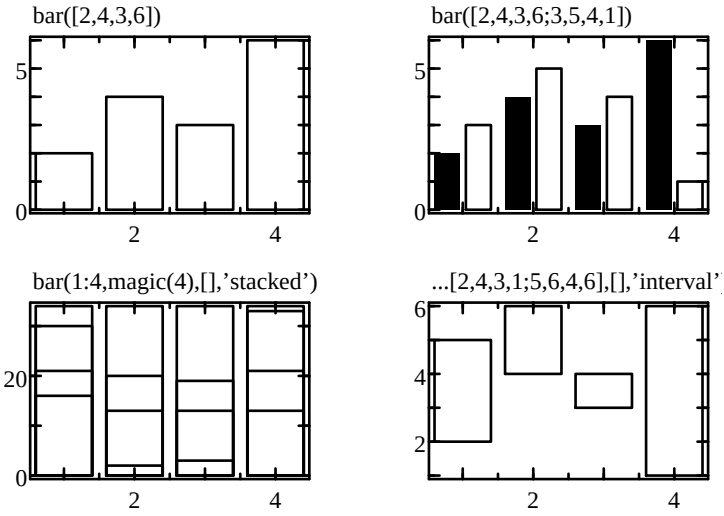

Figure 10.5 Example of `bar` with different options

See also

`barh`, `plot`

barh

Horizontal bar plot.

Syntax

```
barh(x)
barh(y, x)
barh(y, x, w)
barh(..., kind)
barh(..., kind, style)
barh(..., id)
```

Description

`barh` plots a bar plot with horizontal bars. Please see `bar` for a description of its behavior and arguments.

Examples

Simple horizontal bar plot:

```
barh([2,4,3,6;3,5,4,1]);
```

Stacked horizontal bar plot:

```
barh(1:4, magic(4), [], 'stacked');
```

Horizontal interval plot:

```
barh(1:4, [2,4,3,1;5,6,4,6], [], 'interval');
```

See also

`bar`, `plot`

circle

Add circles to the figure.

Syntax

```
circle(x,y,r)
circle(x,y,r,style)
circle(x,y,r,style,id)
```

Description

`circle(x,y,r)` draws a circle of radius `r` centered at (x,y). The arguments can be vectors to display several circles. Their dimensions must match; scalar numbers are repeated if necessary. The optional fourth and fifth arguments are the style and object ID (cf. their description above).

In mouse handlers, `_x0` and `_y0` correspond to the projection of the mouse click onto the circle; `_nb` is the index of the circle in `x`, `y` and `r`, and `_ix` is empty.

Circles are displayed as circles only if the scales along the x and y axes are the same, and linear. With different linear scales, circles are displayed as ellipses. With logarithmic scales, they are not displayed.

Examples

```
circle(1, 2, 5, 'r', 1);
circle(zeros(10,1), zeros(10, 1), 1:10);
```

See also

`plot`, `line`

colormap

Current colormap from scalar to RGB.

Syntax

```
colormap(clut)
clut = colormap
```

Description

Command `colormap(clut)` changes the color mapping from scalar values to RGB values used by commands such as `pcolor`, `image` and `surf`.

Colormaps are arrays of size n-by-3. Each row corresponds to a color; the first column is the intensity of red from 0 (no red component) to 1 (maximum intensity), the second column the intensity of green, and the third column the intensity of blue. Input values are mapped uniformly to one of the discrete color entries, 0 to the first row and 1 to the last row.

With an input argument, `colormap(clut)` sets the colormap to `clut`. With an output argument, `colormap` returns the current colormap.

See also

`pcolor`, `image`

contour

Level curves.

Syntax

```
contour(z)
contour(z, [xmin, xmax, ymin, ymax])
contour(z, [xmin, xmax, ymin, ymax], levels)
contour(z, [xmin, xmax, ymin, ymax], levels, style)
```

Description

`contour(z)` plots seven contour lines corresponding to the surface whose samples at equidistant points `1:size(z,2)` in the x direction and `1:size(z,1)` on the y direction are given by `z`. Contour lines are at equidistant levels. With a second non-empty argument `[xmin, xmax, ymin, ymax]`, the samples are at equidistant points between `xmin` and `xmax` in the x direction and between `ymin` and `ymax` in the y direction.

The optional third argument `levels`, if non-empty, gives the number of contour lines if it is a scalar or the levels themselves if it is a vector.

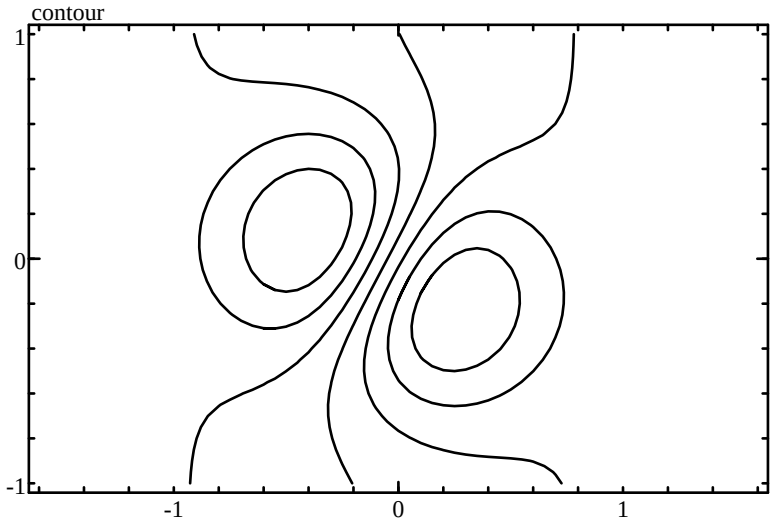

Figure 10.6 Example of `contour`

The optional fourth argument is the style of each line, from the minimum to the maximum level (styles are recycled if necessary). The default style is `'kbrmgcy'`.

When the style is `f` for a filled region, the corresponding level is filled on the side with a lower value of `z`. If the style argument is the single character `'f'`, all levels are filled with the default colors. Regions with a value of `z` smaller than the lowest level are left transparent; an explicit lower level should be specified to fill the whole rectangle.

Examples

A function is evaluated over a grid of two variables `x` and `y`, and is drawn with `contour` (see Fig. 10.6):

```
(x, y) = meshgrid(-2 + (0:40) / 10);
z = exp(-((x-0.2).^2+(y+0.3).^2)) ...
      - exp(-((x+0.5).^2+(y-0.1).^2)) + 0.1 * x;
scale equal;
contour(z, [-1,1,-1,1]);
```

Filled contours:

```
u = -2 + (0:80) / 20;
x = repmat(u, 81, 1);
y = x';
z = exp(-((x-0.2).^2+(y+0.3).^2)) ...
    - exp(-((x+0.5).^2+(y-0.1).^2)) ...
    + 0.1 * x ...
```

```
    + 0.5 * sin(y);
levels = -1:0.2:1;
scale equal;
contour(z, [-1,1,-1,1], levels, 'f');
```

See also

`image`, `quiver`

figurestyle

Figure style.

Syntax

```
figurestyle(name, style)
style = figurestyle(name)
```

Description

`figurestyle` sets or gets the style of figures. The same settings apply to all subplots; settings for specific subplots are changed with `subplotstyle`. Styles are set or got separately for each feature of the graphics (plot background, drawing, title, etc.). They are specified with the same structures as `plotset` or `fontset` (except for `'plotmargin'`), or with the corresponding named arguments.

The first argument, `name`, is the name of the style feature:

Name	Type	Feature
`'controlbg'`	`plotset`	Control background
`'controlfont'`	`fontset`	Font for controls
`'draw'`	`plotset`	Default line or mark plots
`'figfont'`	`fontset`	Font for text in figure
`'frame'`	`plotset`	Plot or subplot frame and ticks
`'grid'`	`plotset`	Special grids such as `hgrid`
`'hilight'`	`plotset`	Hilighted subplot frame for interactive figures
`'tickfont'`	`fontset`	Font for tick labels
`'labelfont'`	`fontset`	Font for axis labels
`'legend'`	`plotset`	Legend box (frame and background)
`'legendfont'`	`fontset`	Font for legend text
`'plotbg'`	`plotset`	Plot or subplot background
`'plotmargin'`		Plot margin size
`'scaleoverview'`	`plotset`	Scale overview rectangle
`'titlefont'`	`fontset`	Font used for plot or subplot titles
`'winbg'`	`plotset`	Background around plots or subplots
`'xygrid'`	`plotset`	Rectangular grid (or polar in polar plots)

`figurestyle(name,style)` changes the specified style. The style can be specified with a style structure, like what is returned by

`plotset` or `fontset`, or with named arguments. Settings which are not specified keep their default values.

With a single argument, `figurestyle(name)` returned the current specified style.

The value for `'plotmargin'` is a structure which describes the margin width around plots or subplots. It contains the following fields:

Name	**Value**
`Left`	left margin in multiple of a digit width
`Right`	right margin in multiple of a digit width
`Top`	top margin in multiple of line height
`Bottom`	bottom margin in multiple of line height
`CenteredLabelWidth`	see below
`CenteredLabelHeight`	see below
`FixedControlVPos`	see below

The fonts the widths are based on are the title font for `Top`, and the label font for the other fields. When an alternative y scale is used with `altscale`, the width of the right margin is based on `Left` instead of `Right`.

If field `CenteredLabelWidth` is larger than 0, it specifies the width of an additional margin (in multiple of a digit width) where the label of the Y axis is displayed, centered vertically. If field `CenteredLabelHeight` is larger than 0, it specifies the height of an additional bottom margin (in multiple of a line height) where the label of the X axis is displayed, centered horizontally. The default location of axis labels is at the end of the tick labels.

If field `FixedControlVPos` is `false`, controls (buttons, sliders etc.) are centered vertically in the subplot content area, or can be scrolled vertically by the user if they exceed the available space. If it is `true`, controls are aligned at the top and cannot be scrolled.

Considered as a whole, styles should be chosen such that they provide enough contrast to make all features visible. In particular, the font color should be changed when a dark background is selected. Some combinations, such as red on green, are difficult to distinguish for color-blind persons.

In Sysquake, `figurestyle` should not be used in figure draw handlers, because it applies to all subplots. It should typically be placed in init or menu handlers. To change the default figure styles which are used in all figures unless they are overridden by `figurestyle`, `defaultstyle` should be called instead.

Example

Blue appearance with different dark shades for the backgrounds, and large fonts.

```
figurestyle('winbg', FillColor='#002');
figurestyle('plotbg', FillColor='#005');
```

```
figurestyle('legend', FillColor='#00a');
figurestyle('draw', Size=18, LineWidth=4, Color='#88f');
figurestyle('grid', Size=18, LineWidth=2, Color='#66c');
figurestyle('xygrid', LineWidth=2, Color='#66c');
figurestyle('frame', LineWidth=3, Color='#44f');
figurestyle('figfont', Size=20, Color='white');
figurestyle('controlfont', Size=20, Color='white');
figurestyle('legendfont', Size=20, Color='white');
figurestyle('titlefont', Size=32, Bold=true, Color='white');
figurestyle('tickfont', Size=18, Color='white');
figurestyle('labelfont', Size=18, Color='white');
```

See also

`subplotstyle`, `plotset`, `plotfont`, `plotoption`

fontset

Options for fonts.

Syntax

```
options = fontset
options = fontset(name1=value1, ...)
options = fontset(name1, value1, ...)
options = fontset(options0, name1, value1, ...)
```

Description

`fontset(name1,value1,...)` creates the font description used by `text`. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Alternatively, options can be given with named arguments. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, `fontset` creates a structure with all the default options. Options can also be passed directly to `text` or `math` as named arguments.

When its first input argument is a structure, `fontset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options (empty arrays mean "automatic"):

Name	Default	Meaning
`Font`	`''`	font name
`Size`	`10`	character size in points
`Bold`	`false`	true for bold font
`Italic`	`false`	true for italic font
`Underline`	`false`	true for underline characters
`Color`	`[0,0,0]`	text color

The default font is used if the font name is not recognized. The color is specified as an empty array (black), a scalar (gray) or a 3-element vector (RGB) of class `double` (0=black, 1=maximum brightness) or `uint8` (0=black, 255=maximum brightness).

Examples

Default font:

```
fontset
  Font: ''
  Size: 10
  Bold: false
  Italic: false
  Underline: false
  Color: real 1x3
```

Named argument directly in `text`:

```
text(0, 0, 'Text', Font='Times', Italic=true, Bold=true)
```

See also

`text`

fplot

Function plot.

Syntax

```
fplot(fun)
fplot(fun, limits)
fplot(fun, limits, style)
fplot(fun, limits, style, id)
fplot(fun, limits, style, id, p1, p2, ...)
```

Description

Command `fplot(fun,limits)` plots function `fun`, specified by its name as a string, a function reference, or an inline or anonymous function. The function is plotted for `x` between `limit(1)` and `limit(2)`; the default limits are `[-5,5]`.

The optional third and fourth arguments are the same as for all graphical commands.

Remaining input arguments of `fplot`, if any, are given as additional input arguments to function `fun`. They permit to parameterize the function. For example `fplot('fun',[0,10],'',-1,2,5)` calls `fun` as `y=fun(x,2,5)` and displays its value for `x` between 0 and 10.

Examples

Plot a sine:

```
fplot(@sin);
```

Plot $(x+0.3)^2+a\exp -3x^2$ in red for $x \in [-2,3]$ with $a=7.2$ and an identifier of 1:

```
fun = inline(...
  'function y=f(x,a); y=(x+0.3)^2+a*exp(-3*x^2);');
fplot(fun, [-2,3], 'r', 1, 7.2);
```

Same plot with an anonymous function:

```
a = 7.2;
fplot(@(x) (x+0.3)^2+a*exp(-3*x^2), [-2,3], 'r', 1);
```

See also

`plot`, `inline`, operator `@`

image

Raster RGB or grayscale image.

Syntax

```
image(gray)
image(red, green, blue)
image(rgb)
image(..., [xmin, xmax, ymin, ymax])
image(..., mode)
image(..., id)
```

Description

`image` displays a raster image (an image defined by a rectangular array of patches of colors called *pixels*). The raster image can be either grayscale or color. A grayscale image is defined by a double matrix of pixel values in the range 0 (black) to 1 (white), by a uint8 matrix in the range 0 (black) to 255 (white), or by a uint16 matrix in the range 0 (black) to 65535 (white). A color image is defined by three matrices of equal size, corresponding to the red, green, and blue components, or by an array with three planes along the 3rd dimension. Each component is defined between 0 (black) to 1 (maximum intensity) with double values, between 0 (black) to 255 (maximum intensity) with uint8 values, or between 0 (black) and 65535 (maximum intensity) with uint16 values. If a colormap has been defined, grayscale image rendering uses it.

The position is defined by the the minimum and maximum coordinates along the horizontal and vertical axes. The raster image is scaled to fit. The first line of the matrix or matrices is displayed at the top. The position can be specified by an argument `[xmin,xmax,ymin,ymax]`; by default, it is `[0,size(im,2),0,size(im,1)]` where `im` stands for the image array or one of its RGB components.

If `mode` is 'e', the raster image is scaled down such that each pixel has the same size; otherwise, the specified position is filled with the raster image. You should use `'e'` when you want a better quality, but do not add other elements in the figure (such as marks or lines) and do not have interaction with the mouse.

Pixels on the screen are interpolated using the bilinear method if `mode` is `'1'`, and the bicubic method if `mode` is `'3'`.

Examples

Two ways to display a table of 10-by-10 random color cells (see Fig. 10.7):

```
image(rand(10), rand(10), rand(10));
image(rand(10, 10, 3));
```

A ramp of gray shades:

```
image(uint8(0:255));
```

Operator : and function `meshgrid` can be used to create the *x* and *y* coordinates used to display a function `z(x,y)` as an image.

```
(X, Y) = meshgrid(-pi:0.1:pi);
Z = cos(X.^2 + Y.^2).^2;
image(Z, [-1,1,-1,1], '3');
```

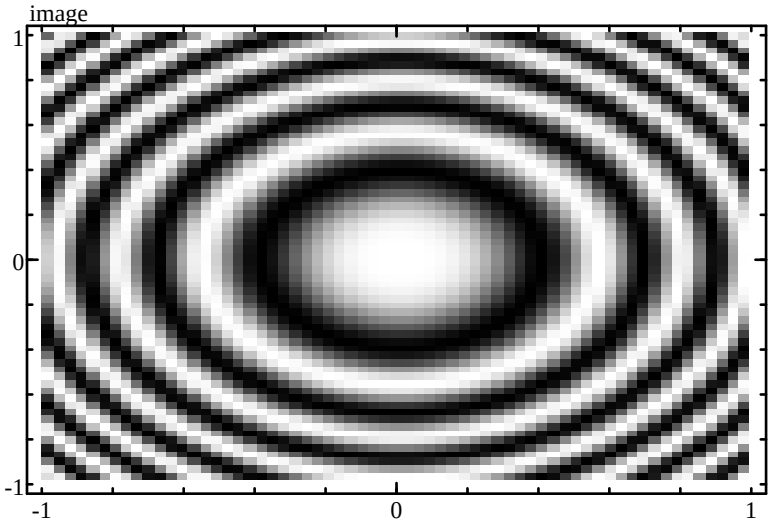

Figure 10.7 Example of `image`

See also

`contour`, `quiver`, `colormap`, `pcolor`

label

Plot labels.

Syntax

```
label(label_x)
label(label_x, label_y)
label(label_x, label_y, label_y2)
```

Description

`label(label_x, label_y)` displays labels for the x and y axes. Its arguments are strings. The label for the y axis may be omitted.

When an alternative y scale is used with `altscale`, its label can be specified with a third argument.

For a dB scale, an additional label `[dB]` is automatically displayed below the text specified by `label_y`; it is not displayed if there is no `label_y` (or an empty `label_y`). If `label_y` is a single-space string, it is replaced by `[dB]` for a dB scale (i.e. `[dB]` is aligned correctly with the top of the figure).

With `plotoption math`, labels can contain MathML or LaTeX.

Examples

```
step(1,[1,2,3,4]);
label('t [s]', 'y [m]');
```

With literal strings, the command syntax may be more convenient:

```
label Re Im;
```

dB scale with only a `[dB]` label:

```
scale logdb;
bodemag(1, [1, 2, 3]);
label('', ' ');
```

See also

`text`, `legend`, `title`, `ticks`, `altscale`, `plotoption`

legend

Plot legend.

Syntax

```
legend(str)
legend(str, style)
```

Description

`legend(str,style)` displays legends for styles defined in string `style`. In string `str`, legends are separated by linefeed characters `\n`. Legends are displayed at the top right corner of the figure in a frame. All styles are permitted: symbols, lines, and filling. They are recycled if more legends are defined in `str`. If `str` is empty, no legend is displayed.

With a single input argument, `legend(str)` uses the default style `'k'`.

With `plotoption math`, legend lines in first argument can contain MathML or LaTeX.

Example

Legend for two traces (see Fig. 10.8).

```
plot(1:20, [rand(1,20); randn(1,20)], '_x');
legend('Uniform random\nNormal random', '_x');
```

See also

`label`, `ticks`, `title`, `plotoption`

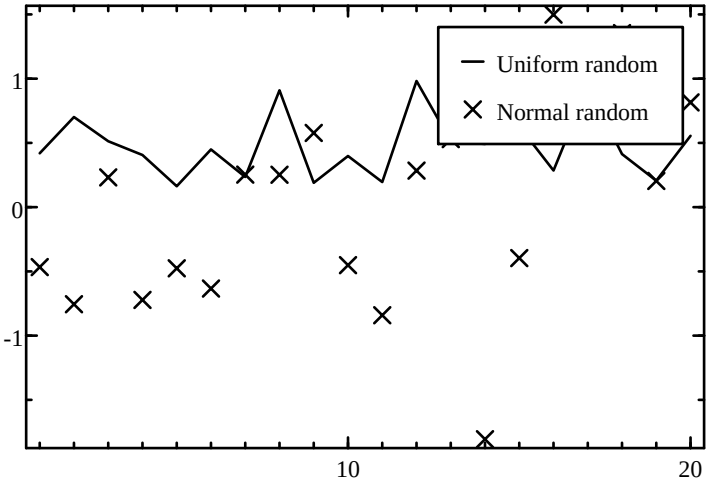

Figure 10.8 Example of `legend`

line

Plot lines.

Syntax

```
line(A, b)
line(V, P0)
line(..., style)
line(..., style, id)
```

Description

`line` displays one or several straight line(s). Each line is defined by an implicit equation or an explicit, parametric equation.

Implicit equation: Lines are defined by equations of the form $a_1x + a_2y = b$. The first argument of `line` is a matrix which contains the coefficients a_1 in the first column and a_2 in the second column. The second argument is a column vector which contains the coefficients b.

Explicit equations: Lines are defined by equations of the form $P = P_0 + \lambda V$ where P_0 is a point of the line, V a vector which defines its direction, and λ a real parameter. The first argument of `line` is a matrix which contains the coefficients v_x in the first column and v_y in the second column. The second argument is a matrix which contains the coefficients x_0 in the first column and y_0 in the second column.

In both cases, each row corresponds to a different line. If one of the arguments has one row and the other has several (or none), the same row is duplicated to match the other size.

In figures with a logarithmic scale, only horizontal and vertical lines are allowed.

The optional third and fourth arguments are the same as for all graphical commands.

In mouse handlers, `_x0` and `_y0` correspond to the projection of the mouse position onto the line; `_nb` is the index of the line in `A` and `b`, and `_ix` is empty.

Examples

Vertical line at x=5:

```
line([1,0],5)
```

Draggable horizontal blue lines at y=2 and y=3:

```
line([0,1], [2;3], 'b', 1)
```

The same lines with named arguments:

```
line([0,1], [2;3], Color='blue', id=1)
```

See also

`plot`, `circle`

math

Display MathML or LaTeX in a figure.

Syntax

```
math(x, y, string)
math(x, y, string, justification)
math(..., font)
math(..., id=id)
```

Description

With three arguments, `math(x,y,string)` renders a string as MathML or LaTeX, centered at the specified position. The third argument is assumed to be MathML unless it starts with a dollar character; in that case, it is converted to MathML as if it was processed by `latex2mathml`.

An optional fourth argument specifies how the MathML equation should be aligned with respect to the position `(x,y)`. It is a string of one or two characters from the following set:

Char.	Alignment
c	Center (may be omitted)
l	Left
r	Right
t	Top
b	Bottom

For instance, `'l'` means that the MathML equation is displayed to the right of the given position and is centered vertically, and `'rt'`, that the equation is to the bottom left of the given position.

An optional trailing argument specifies the font. It is a structure which is typically created with `fontset`; but only the base font size is used. Alternatively, the base font size can be specified with a named argument.

An ID can be specified with a named argument (not with a normal, unnamed argument).

The following MathML elements are supported: `math`, `merror`, `mfenced`, `mfrac`, `mi`, `mn`, `mo`, `mpadded`, `mphantom`, `mroot`, `mrow`, `msqrt`, `mspace`, `msub`, `msubsup`, `msup`, `mtable`, `mtd`, `mtext`, `mtr`.

Examples

```
math(0, 0, mathml([1,pi,1e30]));
math(0, 0, mathml(1e-6, Format='e', NPrec=2), Size=20);
math(0, 0, '$\\rho=\\sqrt{x^2+y^2}$');
```

See also

`text`, `mathml`, `latex2mathml`, `fontset`

pcolor

Pseudocolor plot.

Syntax

```
pcolor(C)
pcolor(X, Y, C)
pcolor(..., style)
pcolor(..., style, id)
```

Description

Command `pcolor(C)` displays a pseudocolor plot, i.e. a rectangular array where the color of each cell corresponds to the value of elements of 2-D array `C`. These values are real numbers between 0 and 1. The color used by `pcolor` depends on the current color map; the default is a grayscale from black (0) to white (1).

`pcolor(X,Y,C)` displays the plot on a grid whose vertex coordinates are given by arrays X and Y. Arrays X, Y and C must all have the same size.

With an additional string input argument, `pcolor(...,style)` specifies the style of the lines drawn between the cells.

The following argument, if it exists, is the ID used for interactivity. During interactive manipulation, the index obtained with `_ix` corresponds to the corner of the patch under the mouse with the smallest index.

Example

```
use colormaps;
n = 11;
(x, y) = meshgrid(1:n);
phi = pi/8;
X = x*cos(phi)-y*sin(phi);
Y = x*sin(phi)+y*cos(phi);
C = magic(n)/n^2;
pcolor(X, Y, C, 'k');
colormap(blue2yellow2redcm);
plotoption noframe;
```

See also

`plot`, `colormap`, `image`

plot

Generic plot.

Syntax

```
plot(y)
plot(x, y)
plot(..., style)
plot(..., style, id)
```

Description

The command `plot` displays graphical data in the current figure. The data are given as two vectors of coordinates x and y. If x is omitted, its default value is `1:size(y,2)`. Depending on the style, the points are displayed as individual marks or are linked with lines. The stairs style (`'s'`) can be used to link two successive points with a horizontal line followed by a vertical line. If x and y are matrices, each row is considered as a separate line or set of marks; if only one of them is a matrix, the other one, a row or column vector, is replicated to match the size of the other argument.

The optional fourth argument is an identification number which is used for interactive manipulation. It should be equal or larger than 1. If present and a mousedown, mousedrag and/or mouseup handler exists, the position of the mouse where the click occurs is mapped to the closest graphical element which has been displayed with an ID; for the command `plot`, the closest point is considered (lines linking the points are ignored). If such a point is found at a small distance, the built-in variables `_x0`, `_y0`, and `_z0` are set to the position of the point before it is moved; the variable `_id` is set to the ID as defined by the command `plot`; the variable `_nb` is set to the number of the row, and the variable `_ix` is set to the index of the column of the matrix x and y.

Examples

Sine between 0 and 2π:

```
x = 2 * pi * (0:100) * 0.01;
y = sin(x);
plot(x, y);
```

Ten random crosses:

```
plot(rand(1,10), rand(1,10), 'x');
```

Two traces with different styles:

```
plot(rand(2, 10),
     {Color='red', LineWidth=2;
      Marker='[]', MarkerFaceColor='navy', LineStyle='-'});
```

A complete SQ file for displaying a red triangle whose corners can be moved interactively on Sysquake:

```
variables x, y     // x and y are 1-by-3 vectors
init (x,y) = init  // init handler
figure "Triangle"
  draw drawTri(x, y)
  mousedrag (x, y) = dragTri(x, y, _ix, _x1, _y1)
functions
{@
function (x,y) = init
  x = [-1,1,0];
  y = [-1,-1,2];
  subplots('Triangle');
function drawTri(x,y)
  scale('equal', [-3, 3, -3, 3]);
  plot(x, y, FillColor='red', id=1);
function (x, y) = dragTri(x, y, ix, x1, y1)
  if isempty(ix)
```

```
    cancel;  // not a click over a point
  end
  x(ix) = x1;
  y(ix) = y1;
@}
```

See also

`fplot`, `line`, `circle`

plotoption

Set plot options.

Syntax

```
plotoption(str1, str2, ...)
plotoption opt1 opt2 ...
```

Description

`plotoption` sets the initial value of the plot options the user can change. Its arguments, character strings, can each take one of the following values.

`'frame'` Rectangular frame with tick marks and a white background around the plot (default).

`'noframe'` No frame, no tickmarks, no white background.

`'label'` Subplot name above the frame (default).

`'nolabel'` No subplot name.

`'legend'` Legend (if it has been set with `legend`).

`'nolegend'` Hidden legend.

`'trlegend'` Legend in top right corner (default).

`'tllegend'` Legend in top left corner.

`'brlegend'` Legend in bottom right corner.

`'bllegend'` Legend in bottom left corner.

`'margin'` Margin for title and labels (default).

`'nomargin'` No margin.

`'math'` MathML or LaTeX rendering in `title`, `label`, `legend`, and controls like `button` and `slider`. The string (or substring in `legend`) is parsed as MathML if the first character is `'<'`, or as LaTeX if it is `'$'`. Otherwise, it is displayed as if it was the text content of an <mtext> element, to guarantee that there is no font mismatch with mathematical expressions.

`'nomath'` No math (default).

`'xticks'` Ticks and labels for the x axis.

`'noxticks'` No ticks and labels for the x axis.

`'yticks'` Ticks and labels for the y axis.

`'noyticks'` No ticks and labels for the y axis.

`'xyticks'` Ticks and labels for the x and y axes (default).

`'noxyticks'` No ticks and labels for the x and y axes.

`'xgrid'` Grid of vertical lines for the x axis.

`'noxgrid'` No grid for the x axis.

`'ygrid'` Grid of horizontal lines for the y axis.

`'noygrid'` No grid for the y axis.

`'xygrid'` Grid of vertical and horizontal lines for the x and y axes.

`'noxygrid'` No grid for the x and y axes (default).

`'grid'` Normal details for grids displayed by `sgrid`, `zgrid`, etc. (default).

`'nogrid'` Removal of grids displayed by `sgrid`, `zgrid`, etc.

`'fullgrid'` More details for grids displayed by `sgrid`, `zgrid`, etc.

`'fill3d'` In 3D graphics, zoom in so that the bounding box fills the figure.

Examples

Display of a photographic image without frame:

```
plotoption noframe;
image(photo);
```

Math in a title:

```
plotoption math;
title '$\\hbox{Solution of}\\;\\dot{x}=f(x,t)$';
```

See also

`figurestyle`, `scale`, `legend`

plotset

Options for plot style.

Syntax

```
options = plotset
options = plotset(name1, value1, ...)
options = plotset(options0, name1, value1, ...)
```

Description

`plotset(name1,value1,...)` creates the style option argument used by functions which display graphics, such as `plot` and `line`. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, `plotset` creates a structure with the default style.

When its first input argument is a structure, `plotset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options:

Name	Default	Meaning
`ArrowEnd`	`false`	arrow at end
`ArrowStart`	`false`	arrow at start
`Color`	`[]`	line color
`Fill`	`false`	fill
`FillColor`	`[]`	filling color
`LineStyle`	`''`	line style
`LineWidth`	`[]`	line width
`Marker`	`''`	marker style
`MarkerEdgeColor`	`[]`	marker edge color
`MarkerFaceColor`	`[]`	marker face (filling) color
`Stairs`	`false`	stairs
`Stems`	`false`	stems

Colors are specified by value or by name. An empty array means the default color (usually black for lines and marker edge, none for filling, and white for marker face). A scalar number represents a shade of gray, an array of 3 numbers an RGB color. An additional element (last element of array of 2 or 4 numbers) represents the alpha component (transparency) where 0 is completely transparent; it is ignored on some platforms. Color type can be `uint8` from 0 to 255, `uint16` from

0 to 65535, or `single` or `double` from 0 to 1. In all cases, 0 represents black and the largest value, the maximum brightness.

Color names can be one of the following:

Name	**Value as** `uint8`
'black'	[0,0,0]
'blue'	[0,0,255]
'green'	[0,128,0]
'cyan'	[0,255,255]
'red'	[255,0,0]
'magenta'	[255,0,255]
'yellow'	[255,255,0]
'white'	[255,255,255]
'aqua'	[0,255,255]
'darkgray'	[169,169,169]
'darkgrey'	[169,169,169]
'darkgreen'	[0,64,0]
'fuchsia'	[255,0,255]
'gray'	[128,128,128]
'grey'	[128,128,128]
'lime'	[0,255,0]
'maroon'	[128,0,0]
'navy'	[0,0,128]
'olive'	[128,128,0]
'orange'	[255,165,0]
'purple'	[128,0,128]
'silver'	[192,192,192]
'teal'	[0,128,128]

Option `LineStyle` is an empty string for the default line style (solid line unless `FillColor` is set), or one of the following one-character strings:

Dash Pattern	`LineStyle`
solid	'_' (underscore)
dashed	'-' (hyphen)
dotted	':'
dash-dot	'!'
hidden	' ' (space)

Option `Marker` is an empty string for the default symbol (usually no symbol, or a cross for `plotroots`), or one of the following strings:

Marker Shape	`Marker`
none	`' '` (space)
point	`'.'`
circle	`'o'`
cross	`'x'`
plus	`'+'`
star	`'*'`
triangle up	`'^'`
triangle down	`'v'`
triangle left	`'<'`
triangle right	`'>'`
square	`'[]'` or `'['`
diamond	`'<>'`

An explicit `Fill=true` is usefull only for filling with the default color or colors, with functions such `contour`. Otherwise, specifying a filling color with `FillColor` implies `Fill=true`.

When `Stems` is true, a marker is drawn for each point and is linked with a vertical line which connects it to the origin (in 2D plots, y=0 for linear scale or y=1 for logarithmic scale; in 3D plots, z=0). In polar plots, stems connects points to x=y=0.

Functions which support multiple styles, such as `plot` where each trace can have a different style, accept a structure array or a list of structures. If there are less elements in the style array or list than there are traces to plot, styles are recycled, restarting from the first one. If there are too many, superfluous styles are ignored.

When `Stairs` is true, for functions wihich support it, points are connected with a horizontal line followed by a vertical line.

Examples

Default options:

```
plotset
  ArrowEnd: false
  ArrowStart: false
  Color: []
  FillColor: []
  LineStyle: ''
  LineWidth: []
  Marker: ''
  MarkerEdgeColor: []
  MarkerFaceColor: []
```

Plot of 5 random lines defined by 10 points each, odd and even ones with different styles:

```
data = rand(5, 10);
styleOdd = plotset(ArrowStart=true,
```

```
  LineWidth=2,
  Color='red',
  Size=0);
styleEven = plotset(ArrowEnd=true,
  LineWidth=2,
  Size=10,
  Color='blue',
  MarkerEdgeColor='black',
  MarkerFaceColor='yellow');
plot(data, {styleOdd, styleEven});
```

Multiple styles can also be built directly as a structure array, without `plotset`; missing fields take their default values.

```
styles = {
  LineWidth=2, Color='red';
  LineStyle='-', Color='blue'
};
plot(data, styles);
```

See also

`plot`, `plotoption`, `figurestyle`

polar

Polar plot.

Syntax

```
polar(theta, rho)
polar(..., style)
polar(..., style, id)
```

Description

Command `polar` displays graphical data in the current figure with polar coordinates. The data are given as two vectors of coordinates `theta` (in radians) and `rho`. Depending on the style, the points are displayed as individual marks or are linked with lines. If x and y are matrices, each row is considered as a separate line or set of marks; if only one of them is a matrix, the other one, a vector, is reused for each line.

Automatic scaling is performed the same way as for cartesian plots after polar coordinates have been converted. The figure axes, ticks and grids are specific to polar plots. Polar plots can be mixed with other graphical commands based on cartesian coordinates such as `plot`, `line` and `circle`.

Example

```
theta = 0:0.01:20*pi;
rho = exp(0.1 * theta) .* sin(5 * theta);
polar(theta, rho);
```

See also

`plot`

quiver

Quiver plot.

Syntax

```
quiver(x, y, u, v)
quiver(u, v)
quiver(..., scale)
quiver(..., style)
```

Description

`quiver(x,y,u,v)` displays vectors (u,v) starting at (x,y). If the four arguments are matrices of the same size, an arrow is drawn for each corresponding element. If x and y are vectors, they are repeated: x is transposed to a row vector if necessary and repeated to match the number of rows of u and v; and y is transposed to a column vector if necessary and repeated to match their number of columns. The absolute size of arrows is scaled with the average step of the grid given by x and y, so that they do not overlap if the grid is uniform.

If x and y are missing, their default values are `[1,2,...,m]` and `[1,2,...,n]` respectively, where `m` and n are the number of rows and columns of u and v.

With a 5th (or 3rd) argument, `quiver(...,scale)` multiplies the arrow lengths by the scalar number `scale`. If `scale` is zero, arrows are not scaled at all: u and v give directly the absolute value of the vectors.

With a 6th (or 4th) string argument, `quiver(...,style)` uses the specified style to draw the arrows.

Example

Force field; complex numbers are used to simplify computation.

```
scale equal;
z = fevalx(@plus, -5:0.5:5, 1j*(-5:0.5:5)');
z0 = 0.2+0.3j;
f = 1+20*sign(z-z0)./(max(abs(z-z0).^2,3));
```

```
x = real(z);
y = imag(z);
u = real(f);
v = imag(f);
quiver(x, y, u, v);
```

See also

`plot`, `image`, `contour`

scale

Set the scale.

Syntax

```
scale([xmin,xmax,ymin,ymax])
scale([xmin,xmax])
scale([xmin,xmax,ymin,ymax,zmin,zmax])
scale(features)
scale(features, usersettablefeatures)
scale(features, [xmin,xmax,ymin,ymax])
scale(features, usersettablefeatures, [xmin,xmax,ymin,ymax])
sc = scale
(sc, type) = scale
```

Description

Without output argument, the `scale` command, which should be placed before any other graphical command, sets the scale and scale options. The last parameter contains the limits of the plot, either for both x and y axes or only for the x axis in 2D graphics, or for x, y and z axes for 3D graphics. The limits are used only if the user has not changed them by zooming.

The first parameter(s) specify some properties of the scale, and which one can be changed by the user. There are two ways to specify them: with a string or with one or two integer numbers. The recommended way is with a string. The list below enumerates the possible values.

`'equal'` Same linear scale for x and y axes. Typically used for representation of the complex plane, such as the roots of a polynomial or a Nyquist diagram. For 3D graphics, same effect as `daspect([1,1,1])`.

`'pixel'` Pixel (unit) linear scale for x and y axes. Used for diagrams which cannot be scaled, such as block diagrams, Venn diagrams, or special use interface. The y axis is always oriented upward.

`'lock'` See below.

`'linlin'` Linear scale for both axes.

`'linlog'` Linear scale for the x axis, and logarithmic scale for the y axis.

`'loglin'` Logarithmic scale for the x axis, and linear scale for the y axis.

`'loglog'` Logarithmic scale for both axes.

`'lindb'` Linear scale for the x axis, and dB scale for the y axis.

`'logdb'` Logarithmic scale for the x axis, and dB scale for the y axis.

`'lindb/logdb'` Linear scale for the x axis, and dB scale for the y axis. The user can choose a logarithmic scale for the x axis, and a logarithmic or linear scale for the y axis.

`'loglog/set'` Logarithmic scale for the x and y axes, without possibility for the user to change them.

The last-but-one setting shows how to enable the options the user can choose in Sysquake. The setting and the enabled options are separated by a dash; if a simple setting is specified, the enabled options are assumed to be the same. Enabling dB always permits the user to choose a logarithmic or linear scale, and enabling a logarithmic scale always permits to choose a linear scale. The `'equal'` option cannot be combined with anything else. Changing the options in subsequent redraws is ignored, because options are under the user control.

The last setting ending with `/set` shows how to force options without letting the user override them. In this case, options can be changed during redraws. SQ files with customs ways to change the kind of scale must use this method.

When the properties are specified with one or two integer numbers, each bit corresponds to a property. Only the properties in bold in the table below can be set by the user, whatever the setting is.

Bit	Meaning
0	**log x**
2	tick on x axis
3	grid for x axis
4	labels on x axis
6	**log y**
7	**dB y**
8	tick on y axis
9	grid for y axis
10	labels on y axis
12	same scale on both axes
13	minimum grid
14	maximum grid

`scale lock` locks the scale as if the user had done it by hand. It fixes only the initial value; the user may change it back afterwards.

The scale is usually limited to a range of 1e-6 for linear scales and a ratio of 1e-6 for logarithmic scales. This avoids numeric problems, such as when a logarithmic scale is chosen and the data contain the value 0. In some rare cases, a large scale may be required. The `'lock'` option is used to push the limits from 1e-6 to 1e-24 for both linear and logarithmic scales. A second argument must be provided:

```
scale('lock', [xmin,xmax,ymin,ymax]);
```

The command must be used in a draw handler (or from the command line interface). To add other options, use a separate `scale` command:

```
scale logdb;
scale('lock', [1e-5, 1e8, 1e-9, 1e9]);
```

The scale is locked, and the user may not unlock it. In the example above, note also that a single string argument can be written without quote and parenthesis if it contains only letters and digits.

With output arguments, `scale` returns the current scale as a vector `[xmin,xmax,ymin,ymax]`. If the scale is not fixed, the vector is empty. If only the horizontal scale is set, the vector is `[xmin,xmax]`. During a mouse drag, both the horizontal and vertical scales are fixed. The values returned by `scale` reflect the zoom chosen by the user. They can be used to limit the computation of data displayed by `plot` to the visible area. The optional second output argument `type` tells whether a linear or a logarithmic scale is set for axis x and y; it is a string such as `'linlin'` or `'loglin'`.

Examples

Here are some suggestions for the most usual graphics:

Time response	(default linlin is fine)
Bode mag	`scale logdb`
Bode phase	`scale loglin`
D bode mag	`scale('lindb/logdb',[0,pi/Ts])`
D bode phase	`scale('linlin/loglin',[0,pi/Ts])`
Poles	`scale equal`
D poles	`scale('equal',[-1,1,-1,1])`
Nyquist	`scale('equal',[-1.5,1.5,-1.5,1.5])`
Nichols	`scale lindb`

Use of `scale` to display a sine in the visible x range:

```
scale([0,10]);  % default x range between 0 and 10
sc = scale;     % maybe changed by the user (1x2 or 1x4)
xmin = sc(1);
xmax = sc(2);
x = xmin + (xmax - xmin) * (0:0.01:1);
                % 101 values between xmin and xmax
y = sin(x);
plot(x, y);
```

See also

`plotoption`, `scalefactor`

scalefactor

Change the scale displayed in axis ticks and labels.

Syntax

```
scalefactor(f)
f = scalefactor
```

Description

`scalefactor(f)` sets the factor used to display the ticks and the labels. Its argument `f` can be a vector of two or three real positive numbers to set separately the x, y, and z axes, or a real positive scalar to set the same factor for all axes. `scalefactor([fx,fy])` is equivalent to `scalefactor([fx,fy,1])`. The normal factor value is 1, so that the ticks correspond to the graphical contents. With a different factor, the contents are displayed with the same scaling, but the ticks and labels are changed as if the graphical data had been scaled by the factor. For instance, you can plot data in radians (the standard angle unit in LME) and display ticks and labels in degrees by using a factor of `180/pi`.

With an output argument, `scalefactor` gives the current factors as a 2-elements vector.

Example

Display the sine with a scale in degrees:

```
phi = 0:0.01:2*pi;
plot(phi, sin(phi));
scalefactor([180/pi, 1]);
```

See also

`scale`, `plotoption`

scaleoverview

Set the scale overview rectangle.

Syntax

```
scaleoverview([xmin,xmax,ymin,ymax])
scaleoverview([xmin,xmax,ymin,ymax],'xy')
scaleoverview([xmin,xmax],'x')
scaleoverview([ymin,ymax],'y')
```

Description

`scaleoverview` sets the limits of a rectangular region used to provide an overview of the scale used in another plot. Typically, the same data are displayed in two subplots: one with a large, fixed displayed area (set with `scale`) with a smaller scale overview rectangle set with `scaleoverview`, and one with a smaller displayed area (set with `scale`) which matches the limits set with `scaleoverview` in the first plot. In Sysquake, scale synchronization is used to keep both subplots synchronized when the user zooms or drags the data in the second subplot or manipulates directly the scale overview rectangle.

By default, limits on axis x and y are provided. A second argument can specify which axis has limits: `'xy'` (default), `'x'` or `'y'` (then the first argument is an array of two elements).

See also

`scale`

subplotstyle

Subplot style.

Syntax

```
subplotstyle(name, style)
style = subplotstyle(name)
```

Description

`subplotstyle` sets or gets the style of the current subplot. In Sysquake's SQ files, it should be used in draw handlers. It has the same arguments as `figurestyle`, which handles the settings globally for all subplots, or the default settings when both `figurestyle` and `subplotstyle` are used.

Example

```
subplot 211;
subplotstyle('plotbg', FillColor='yellow');
subplotstyle('frame', LineWidth=2);
step(1, 1:3);
subplot 212;
subplotstyle('plotbg', FillColor='orange');
step(1, 1:4);
```

See also

`figurestyle`, `plotset`, `plotfont`, `plotoption`

text

Display text in a figure.

Syntax

```
text(x, y, string)
text(x, y, string, justification)
text(..., font)
text(..., id=id)
```

Description

With three arguments, `text(x,y,string)` displays a string centered at the specified position. An optional fourth argument specifies how the string should be aligned with respect to the position `(x,y)`. It is a string of one or two characters from the following set:

Char.	Alignment
c	Center (may be omitted)
l	Left
r	Right
t	Top
b	Bottom

For instance, `'l'` means that the string is displayed to the right of the given position and is centered vertically, and `'rt'`, that the string is to the bottom left of the given position.

An optional trailing argument specifies the font, size, type face, and color to use. It is a structure which is typically created with `fontset`. Alternatively, named arguments can be used directly, without `fontset`.

An ID can be specified with a named argument (not with a normal, unnamed argument).

Examples

A line is drawn between (-1,-1) and (1,1) with labels at both ends.

```
plot([-1,1], [-1,1]);
text(-1,-1, 'p1', 'tr');
text(1, 1, 'p2', 'bl');
```

Text with font specification:

```
font = fontset(Font='Times',
  Bold=true,
  Size=18,
  Color=[1,0,0]);
text(1.1, 4.2, 'Abc', font);
```

Same font with named arguments:

```
text(1.1, 4.2, 'Abc', font,
  Font='Times',
  Bold=true,
  Size=18,
  Color=[1,0,0]);
```

See also

`label`, `fontset`, `sprintf`

tickformat

Subplot tick format.

Syntax

```
tickformat(axis, format)
```

Description

`tickformat(axis,format)` specifies the format to be used for tick labels. The first argument, `axis`, specified which axis is affected: it is 1 or `'x'` for the first axis (horizontal) or 2 or `'y'` for the second axis (vertical, on the left or the right depending on the last call to

`altscale` if any). Second argument, `format`, is a string similar to the first argument of `sprintf`. Only numeric formats are supported (`%d`, `%e`, `%f`, `%g`, `%h`, `%i`, `%k`, `%n`, `%o`, `%P`, `%x`), with their options, width and precision. Double `%` gives a single `%`, and other characters are used literally.

The format is not used if ticks are specified with function `ticks`.

Examples

General format with a unit for the x axis, and fixed format with two fractional digits for the y axis:

```
step(1, [1, 2, 3, 4]);
tickformat('x', '%g h');
tickformat('y', '%.2f');
```

See also

`ticks`, `sprintf`

ticks

Subplot ticks and tick labels.

Syntax

```
ticks(axis, majorTicks)
ticks(axis, majorTicks, minorTicks)
ticks(axis, majorTicks, minorTicks, tickLabels)
```

Description

`ticks` replaces default ticks (small scale marks along the plot frame and their labels outside the frame) with custom ones.

`ticks(axis,majorTicks)` specifies the value of major ticks (large ones). The first argument, `axis`, specified which axis is affected: it is 1 or `'x'` for the first axis (horizontal) or 2 or `'y'` for the second axis (vertical, on the left or the right depending on the last call to `altscale` if any). Second argument, `majorTicks`, is an array of values where ticks are displayed; the same scaling as the one applied to the plot contents is used.

With a third argument, `ticks(axis,majorTicks,minorTicks)` also displays minor ticks (smaller ones, typically used with a finer spacing) specified by array `minorTicks`.

With a fourth argument, `ticks(axis,majorTicks,minorTicks,labels)` displays labels at the position of major ticks. Labels are given as a string of linefeed-separated substrings, such as `'one\ntwo`. If more values are

specified for major ticks than for labels, labels are reused, starting from the first one. Superfluous labels are ignored. If no minor tick is displayed, argument `minorTicks` is optional.

Values out of range are not displayed. If axis labels are specified with function `label`, tick labels which would overlap are not displayed.

3D plots have always default ticks.

Examples

Bar plot where bars correspond to months:

```
bar([2,4,3,6]);
ticks('x', 1:4, 'Jan\nFeb\nMar\nApr');
```

Tick labels with units and axis label:

```
scale([0, 10]);
step(1, [1, 2, 3, 4]);
majorTicks = 0:2:10;
minorTicks = 0:0.5:10;
labels = sprintf('%g s\n', majorTicks);
ticks('x', majorTicks, minorTicks, labels);
ticks('y', 0:0.1:1);
label Time;
```

Plot without any tick and label:

```
step(1, [1, 2, 3, 4]);
ticks('x', []);
ticks('y', []);
```

See also

`label`, `tickformat`, `legend`, `title`, `text`, `sprintf`

title

Subplot title.

Syntax

```
title(string)
```

Description

`title(string)` sets or changes the title of the current subplot.

With `plotoption math`, the title can contain MathML or LaTeX.

See also

`label`, `legend`, `ticks`, `text`, `sprintf`, `plotoption`

10.45 3D Graphics

Three-dimension graphic commands enable the representation of objects defined in three dimensions x, y and z on the two-dimension screen. The transform from the 3D space to the screen is performed as if there were a virtual camera in the 3D space with a given position, orientation, and angle of view (related to the focal length in a real camera).

Projection

The projection is defined by the following parameters:

Target point The target point is a 3D vector which defines the position where the camera is oriented to.

Projection kind Two kinds of projections are supported: orthographic and perspective.

View point The view point is a 3D vector which defines the position of the camera. For orthographic projection, it defines a direction independent from the target position; for perspective projection, it defines a position, and the view orientation is defined by the vector from view point to target point.

Up vector The up vector is a 3D vector which fixes the orientation of the camera around the view direction. The projection is such that the up vector is in a plane which is vertical in the 2D projection. Changing it makes the projection rotate around the image of the target.

View angle The view angle defines the part of the 3D space which is projected onto the image window in perspective projections. It is zero in orthographic mode.

All of these parameters can be set automatically. Here is how the whole projection and scaling process is performed:

- Scale data separately along each direction according to `daspect`
- Find bounding box of all displayed data, or use limits set with `scale`
- Find radius of circumscribed sphere of bounding box
- If the target point is automatic, set it to the center of the bounding box; otherwise, use position set with `camtarget`

- If the view point is automatic, set it to direction `[-3;-2;1]` at infinity in orthographic mode, or in that direction with respect to the target point at a distance such that the view angle of the circumscribed sphere is about 6 degrees; otherwise, use position set with `campos`
- If the up vector is automatic, set it to `[0,0,1]` (vertical, pointing upward); otherwise, use position set with `camup`
- Compute the corresponding homogeneous matrix transform
- Set the base scaling factor so that the circumscribed sphere fits the display area
- Apply an additional zoom factor which depends on `camva` and `camzoom`

Surface shading

Surface and mesh colors add information to the image, helping the viewer in interpreting it. Colors specified by the style argument also accepted by 2D graphical commands are used unchanged. Colors specified by a single-component value, RGB colors, or implicit, are processed differently whether `lightangle` and/or `material` have been executed, or not. In the first case, colors depend directly on the colors specified or the default value; in the second case, the Blinn-Phong reflection model is used with flat shading. In both cases, single-color values are mapped to colors using the current color map (set with `colormap`). Commands which accept a color argument are `mesh`, `surf`, and `plotpoly`.

Direct colors

If neither `lightangle` nor `material` has been executed, colors depend only on the `color` argument provided with `x`, `y`, and `z` coordinates. If the this argument is missing, color is obtained by mapping linearly the `z` coordinates to the full range of the current color map.

Blinn-Phong reflection model

In the Blinn-Phong reflexion model, the color of a surface depends on the intrinsic object color, the surface reflexion properties, and the relative positions of the surface, the viewer, and light sources.

camdolly

Move view position and target.

Syntax

```
camdolly(d)
```

Description

`camdolly(d)` translates the camera by 3x1 or 1x3 vector `d`, moving the target and the view point by the same amount.

See also

`campan`, `camorbit`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`, `camzoom`

camorbit

Camera orbit around target.

Syntax

```
camorbit(dphi, dtheta)
```

Description

`camorbit(dphi,dtheta)` rotates the camera around the target point by angle `dphi` around the up vector, and by angle `dtheta` around the vector pointing to the right of the projection plane. Both angles are given in radians. A positive value of `dphi` makes the camera move to the right, and a positive value of `dtheta` makes the camera move down.

See also

`camdolly`, `campan`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`, `camzoom`

campan

Tilt and pan camera.

Syntax

```
campan(dphi, dtheta)
```

Description

`campan(dphi,dtheta)` pans the camera by angle `dphi` and tilts it by angle `dtheta`. Both angles are in radians. More precisely, the target point is changed so that the vector from view point to target is rotated by angle `dphi` around the up vector, then by angle `dtheta` around a "right" vector (a vector which is horizontal in view coordinates).

See also

`camdolly`, `camorbit`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`, `camzoom`

campos

Camera position.

Syntax

```
campos(p)
campos auto
campos manual
p = campos
```

Description

`campos(p)` sets the view position to `p`. `p` is a 3D vector.

`campos auto` sets the view position to automatic mode, so that it follows the target. `campos manual` sets the view position to manual mode.

With an output argument, `campos` gives the current view position.

See also

`camdolly`, `camorbit`, `campan`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`, `camzoom`

camproj

Projection kind.

Syntax

```
camproj(str)
str = camproj
```

Description

`camproj(str)` sets the projection mode; string `str` can be either `'orthographic'` (or `'o'`) for a parallel projection, or `'perspective'` (or `'p'`) for a projection with a view point at a finite distance.

With an output argument, `camproj` gives the current projection mode.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camroll`, `camtarget`, `camup`, `camva`, `camzoom`

camroll

Camera roll around view direction.

Syntax

```
camroll(dalpha)
```

Description

`camroll(dalpha)` rotates the up vector by angle `dalpha` around the vector from view position to target. `dalpha` is given in radians. A positive value makes the scene rotate counterclockwise.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camproj`, `camtarget`, `camup`, `camva`, `camzoom`

camtarget

Target position.

Syntax

```
camtarget(p)
camtarget auto
camtarget manual
p = camtarget
```

Description

`camtarget(p)` sets the target to `p`. `p` is a 3D vector.

`camtarget auto` sets the target to automatic mode, so that it follows the center of the objects which are drawn. `camtarget manual` sets the target to manual mode.

With an output argument, `camtarget` gives the current target.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camproj`, `camroll`, `camup`, `camva`, `camzoom`

camup

Up vector.

Syntax

```
camup(p)
camup auto
camup manual
p = camup
```

Description

`camup(p)` sets the up vector to `p`. `p` is a 3D vector.

`camup auto` sets the up vector to `[0,0,1]`. `camup manual` does nothing.

With an output argument, `camup` gives the current up vector.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camproj`, `camroll`, `camtarget`, `camva`, `camzoom`

camva

View angle.

Syntax

```
camva(va)
va = camva
```

Description

`camva(va)` sets the view angle to `va`, which is expressed in degrees. The projection mode is set to `'perspective'`. The scale is adjusted so that the graphics have about the same size.

With an output argument, `camva` gives the view angle in degrees, which is 0 for an orthographic projection.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camzoom`

camzoom

Zoom in or out.

Syntax

```
camzoom(f)
```

Description

`camzoom(f)` scales the projection by a factor `f`. The image grows if `f` is larger than one, and shrinks if it is smaller.

See also

`camdolly`, `camorbit`, `campan`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`

contour3

Level curves in 3D space.

Syntax

```
contour3(z)
contour3(z, [xmin, xmax, ymin, ymax])
contour3(z, [xmin, xmax, ymin, ymax], levels)
contour3(z, [xmin, xmax, ymin, ymax], levels, style)
```

Description

`contour3(z)` plots in 3D space seven contour lines corresponding to the surface whose samples at equidistant points `1:size(z,2)` in the x direction and `1:size(z,1)` on the y direction are given by `z`. Contour lines are at equidistant levels. With a second non-empty argument `[xmin, xmax, ymin, ymax]`, the samples are at equidistant points between `xmin` and `xmax` in the x direction and between `ymin` and `ymax` in the y direction.

The optional third argument `levels`, if non-empty, gives the number of contour lines if it is a scalar or the levels themselves if it is a vector.

The optional fourth argument is the style of each line, from the minimum to the maximum level (styles are recycled if necessary). The default style is `'kbrmgcy'`.

See also

`contour`, `mesh`, `surf`

daspect

Scale ratios along x, y and z axis.

Syntax

```
daspect([rx,ry,rz])
daspect([])
R = daspect
```

Description

`daspect(R)` specifies the scale ratios along x, y and z axis. Argument `R` is a vector of 3 elements `rx`, `ry` and `rz`. Coordinates in the 3D space are divided by `rx` along the x axis, and so on, before the projection is performed. For example, a box whose size is `[2;5;3]` would be displayed as a cube with `daspect([2;5;3])`.

`daspect([])` sets the scale ratios so that the bounding box of 3D elements is displayed as a cube.

With an output argument, `R=daspect` gives the current scale ratios as a vector of 3 elements.

See also

`scale`

lightangle

Set light sources in 3D world.

Syntax

```
lightangle
lightangle(az, el)
```

Description

`lightangle(az,el)` set lighting source(s) at infinity, with asimuth `az` and elevation `el`, both in radians. With missing input argument, the default azimuth is 4 and the default elevation is 1. If `az` and `el` are vectors, they must have the same size (except if one of them is a scalar, then it is replicated as needed); `lightangle` sets multiple light sources.

See also

`material`

line3

Plot straight lines in 3D space.

Syntax

```
line3(A, b)
line3(V, P0)
line3(A, b, style)
line3(A, b, style, id)
```

Description

`line3` displays one or several straight line(s) in the 3D space. Each line is defined by two implicit equations or one explicit, parametric equation.

Implicit equation: Lines are defined by two equations of the form $a_1x + a_2y + a_3z = b$. The first argument of `line3` is a matrix which contains the coefficients a_1 in the first column, a_2 in the second column, and a_3 in the third column; two rows define a different line. The second argument is a column vector which contains the coefficients b. If one of these arguments has two rows and the other has several pairs, the same rows are reused multiple times.

Explicit equations: Lines are defined by equations of the form $P = P_0 + \lambda V$ where P_0 is a point of the line, V a vector which defines its direction, and λ a real parameter. The first argument of `line3` is a matrix which contains the coefficients v_x in the first column, v_y in the second column and v_z in the third column. The second argument is a matrix which contains the coefficients x_0 in the first column, y_0 in the second column and z_0 in the third column.

The optional third and fourth arguments are the same as for all graphical commands.

Example

Implicit or parametric forms of a vertical line at x=5, y=6:

```
line3([1,0,0;0,1,0], [5;6])
line3([0, 0, 1], [5, 6, 0])
```

See also

`plot3`, `line`

material

Surface reflexion properties.

Syntax

```
material(p)
```

Description

`material(p)` sets the reflexion properties of the Blinn-Phong model of following surfaces drawn with `surf` and `plotpoly`. Argument `p` is a scalar or a vector of two real values between 0 and 1. The first or only element, ka, is the weight of ambiant light; the second element, kd, is the weight of diffuse light reflected from all light sources.

See also
`lightangle`

mesh

Plot a mesh in 3D space.

Syntax

```
mesh(x, y, z)
mesh(z)
mesh(x, y, z, color)
mesh(z, color)
mesh(..., kind)
mesh(..., kind, style)
mesh(..., kind, style, id)
```

Description

`mesh(x,y,z)` plots a mesh defined by 2-D arrays `x`, `y` and `z`. Arguments `x` and `y` must have the same size as `z` or be vectors of `size(z,2)` and `size(z,1)` elements, respectively. If `x` and `y` are missing, their default values are coordinates from 1 to `size(z,2)` along `x` axis and from 1 to `size(z,1)` along `y` axis. Color is obtained by mapping the full range of `z` values to the color map.

`mesh(x,y,z,color)` maps values of array `color` to the color map. `color` must have the same size as `z` and contain values between 0 and 1, which are mapped to the color map.

`mesh(...,kind)` specifies which side of the mesh is visible. `kind` is a string of 1 or 2 characters: `'f'` if the front side is visible (the side where increasing `y` are on the left of increasing `x` coordinates), and `'b'` if the back side is visible. Default `''` is equivalent to `'fb'`.

`mesh(...,style)` specifies the line or symbol style of the mesh. The default `''` is to map `z` or `color` values to the color map.

`mesh(...,id)` specifies the ID used for interactivity in Sysquake.

Example

```
(X, Y) = meshgrid([-2:0.2:2]);
Z = X.*exp(-X.^2-Y.^2);
mesh(X, Y, Z);
```

See also
`plot3`, `surf`, `plotpoly`

plot3

Generic 3D plot.

Syntax

```
plot3(x, y, z)
plot3(x, y, z, style)
plot3(x, y, z, style, id)
```

Description

The command `plot3` displays 3D graphical data in the current figure. The data are given as three vectors of coordinates x, y and z. Depending on the style, the points are displayed as individual marks or are linked with lines.

If x, y and z are matrices, each row is considered as a separate line or set of marks; row or column vectors are replicated to match the size of matrix arguments if required.

`plot3(...,id)` specifies the ID used for interactivity in Sysquake.

Example

Chaotic attractor of the Shimizu-Morioka system:

```
(t,x) = ode45(@(t,x) [x(2); (1-x(3))*x(1)-0.75*x(2); x(1)^2-0.45*x(3)],
 [0,300], [1;1;1]);
plot3(x(:,1)', x(:,2)', x(:,3)', 'r');
label x y z;
campos([-1.5; -1.4; 3.1]);
```

See also

`line3`, `plotpoly`, `plot`

plotpoly

Plot polygons in 3D space.

Syntax

```
plotpoly(x, y, z, ind)
plotpoly(x, y, z, 'strip')
plotpoly(x, y, z, 'fan')
plotpoly(x, y, z, color, ind)
plotpoly(x, y, z, color, 'strip')
plotpoly(x, y, z, color, 'fan')
plotpoly(..., vis)
plotpoly(..., vis, style)
plotpoly(..., vis, style, id)
```

Description

`plotpoly(x,y,z,ind)` plots polygons whose vertices are given by vectors `x`, `y` and `z`. Rows of argument `ind` contain the indices of each polygon in arrays `x`, `y`, and `z`. Vertices can be shared by several polygons. Color of each polygon is mapped linearly from the `z` coordinate of the center of gravity of its vertices to the color map. Each polygon can be concave, but must be planar and must not self-intersect (different polygons may intersect).

`plotpoly(x,y,z,'strip')` plots a strip of triangles. Triangles are made of three consecutive vertices; their indices could be defined by the following array `ind_strip`:

```
ind_strip = ...
[ 1 2 3
  3 2 4
  3 4 5
  5 4 6
  5 6 7
  etc. ];
```

Ordering is such that triangles on the same side of the strip have the same orientation.

`plotpoly(x,y,z,'fan')` plots triangles which share the first vertex and form a fan. Their indices could be defined by the following array `ind_fan`:

```
ind_fan = ...
[ 1 2 3
  1 3 4
  1 4 5
  etc. ];
```

`plotpoly(x,y,z,color,...)` uses `color` instead of `z` to set the filling color of each polygon. `color` is always a real double array (or scalar) whose elements are between 0 and 1. How it is interpreted depends on its size:

- A scalar defines the color of all polygons; it is mapped to the color map.
- A vector of three elements defines the RGB color of all polygons (row vector if there are 3 vertices to avoid ambiguity).
- A vector with as many elements as `x`, `y` and `z` defines the color of each vertex (column vector if there are 3 vertices to avoid ambiguity). Polygons have the mean value of all their vertices, which is mapped to the color map.

– An array with as many columns as elements in x, y and z defines the RGB color of each vertex. Polygons have the mean value of all their vertices.

`plotpoly(...,vis)` uses string `vis` to specify which side of the surface is visible: `'f'` for front only, `'b'` for back only, or `'fb'` or `'bf'` for both sides. The front side is defined as the one where vertices have an anticlockwise orientation. The default is `'f'`.

`plotpoly(...,vis,style)` uses string `style` to specify the style of edges.

`plotpoly(...,id)` specifies the ID used for interactivity in Sysquake.

See also

`plot3`, `surf`

sensor3

Make graphical element sensitivive to 3D interactive displacement.

Syntax

```
sensor3(type, param, id)
sensor3(type, param, typeAlt, paramAlt, id)
```

Description

`sensor3(type,param,id)` specifies how a 3D element can be dragged interactively. Contrary to 2D graphics where the mapping between the mouse cursor and the graphical coordinates depends on two separate scaling factors, manipulation in 3D space must use a surface as an additional constraint. `sensor3` specifies this surface for a graphical object whose ID is the same as argument `id`.

The constraint surface is specified with string `type` and numeric array `param`. It always contains the selected point. For instance, if the user clicks the second point of `plot3([1,2],[5,3],[2,4],'',1)` and `sensor3` defines a horizontal plane, the move lies in horizontal plane z=4. In addition to position `_p1`, parameters specific to the constraint surface are provided in special variable `_q`, a vector of two elements.

`type = 'plane'` The constraint surface is the plane defined by the selected point `_p0` and two vectors `[vx1;vy1;vz1]` and `[vx2;vy2;vz2]` given in argument `param = [vx1,vy1,vz1; vx2,vy2,vz2]`. During the drag, `_q` contains the coefficients of these two vectors, such that `_p1 = _p0+_q'*param'`.

`type = 'sphere'` The constraint surface is a sphere whose center is defined by a point `param = [px,py,pz]`. Its R is such that the surface contains the selected point `_p0`. During the drag, `_q` contains the spherical coordinates `phi` and `theta`, such that `_p1 = param' + R * [cos(q_(1))*cos(q_(2)); sin(q_(1))*cos(q_(2)); sin(q_(2))]`.

With five input arguments, `sensor3(type,param,typeAlt,paramAlt,id)` specifies an alternative constraint surface used when the modifier key is held down.

Examples

(simple XY plane...)

(phi/theta without modifier, R with modifier with plane and ignored 2nd param)

See also

`plot3`, `mesh`, `plotpoly`, `surf`

surf

Plot a surface defined by a grid in 3D space.

Syntax

```
surf(x, y, z)
surf(z)
surf(x, y, z, color)
surf(z, color)
surf(..., vis)
surf(..., vis, style)
surf(..., vis, style, id)
```

Description

`surf(x,y,z)` plots a surface defined by 2-D arrays `x`, `y` and `z`. Arguments `x` and `y` must have the same size as `z` or be vectors of `size(z,2)` and `size(z,1)` elements, respectively. If x and y are missing, their default values are coordinates from 1 to `size(z,2)` along x axis and from 1 to `size(z,1)` along y axis. Color of each surface cell is obtained by mapping the average z values to the color map.

`surf(x,y,z,color)` maps values of array `color` to the color map. `color` must have the same size as `z` and contain values between 0 and 1.

`surf(...,vis)` specifies which side of the surface is visible. `vis` is a string of 1 or 2 characters: `'f'` if the front side is visible (the side where increasing `y` are on the left of increasing `x` coordinates), and `'b'` if the back side is visible. Default `''` is equivalent to `'fb'`.

`surf(...,style)` specifies the line or symbol style of the mesh between surface cells, or the fill style of the surface. The default `''` is to map `z` or `color` values to the color map for the surface cells and not to draw cell bounds.

`mesh(...,id)` specifies the ID used for interactivity in Sysquake.

Example

```
(X, Y) = meshgrid([-2:0.2:2]);
Z = X.*exp(-X.^2-Y.^2);
surf(X, Y, Z, 'k');
```

See also

`plot3`, `mesh`, `plotpoly`

10.46 Graphics for Dynamical Systems

Graphical commands described in this section are related to automatic control. They display the time responses and frequency responses of linear time-invariant systems defined by transfer functions or state-space models in continuous time (Laplace transform) or discrete time (z transform).

Some of these functions can return results in output arguments instead of displaying them. These values depend not only on the input arguments, but also on the current scale of the figure. For instance, the set of frequencies where the response of the system is evaluated for the Nyquist diagram is optimized in the visible area. Option Range of `responseset` can be used when this behavior is not suitable, such as for phase portraits using `lsim`. Output can be used for uncommon display purposes such as special styles, labels, or export. Evaluation or simulation functions not related to graphics, like `polyval`, `ode45` or `filter`, are better suited to other usages.

bodemag

Magnitude Bode diagram of a continuous-time system.

Syntax

```
bodemag(numc, denc)
bodemag(numc, denc, w)
```

```
bodemag(numc, denc, opt)
bodemag(numc, denc, w, opt)
bodemag(Ac, Bc, Cc, Dc)
bodemag(Ac, Bc, Cc, Dc, w)
bodemag(Ac, Bc, Cc, Dc, opt)
bodemag(Ac, Bc, Cc, Dc, w, opt)
bodemag(..., style)
bodemag(..., style, id)
(mag, w) = bodemag(...)
```

Description

`bodemag(numc,denc)` plots the magnitude of the frequency response of the continuous-time transfer function `numc/denc`. The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`bodemag(Ac,Bc,Cc,Dc)` plots the magnitude of the frequency response $Y(j\omega)/U(j\omega)$ of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$\begin{aligned} j\omega X(j\omega) &= A_c X(j\omega) + B_c U(j\omega) \\ Y(j\omega) &= C_c X(j\omega) + D_c U(j\omega) \end{aligned}$$

With output arguments, `bodemag` gives the magnitude and the frequency as column vectors. No display is produced.

Examples

Green plot for $\left|1/(s^3+2s^2+3s+4)\right|$ with $s=j\omega$ (see Fig. 10.9):

```
bodemag(1, [1, 2, 3, 4], 'g');
```

The same plot, between $\omega=0$ and $\omega=10$:

```
scale([0,10]);
bodemag(1, [1, 2, 3, 4], 'g');
```

See also

`bodephase`, `dbodemag`, `sigma`, `responseset`, `plotset`

bodephase

Phase Bode diagram for a continuous-time system.

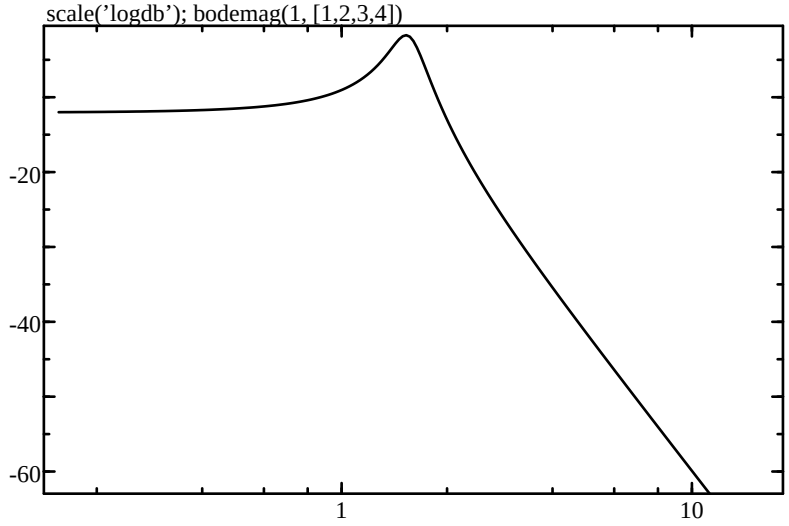

Figure 10.9 `scale('logdb'); bodemag(1, [1,2,3,4])`

Syntax

```
bodephase(numc, denc)
bodephase(numc, denc, w)
bodephase(numc, denc, opt)
bodephase(numc, denc, w, opt)
bodephase(Ac, Bc, Cc, Dc)
bodephase(Ac, Bc, Cc, Dc, w)
bodephase(Ac, Bc, Cc, Dc, opt)
bodephase(Ac, Bc, Cc, Dc, w, opt)
bodephase(..., style)
bodephase(..., style, id)
(phase, w) = bodephase(...)
```

Description

`bodephase(numc,denc)` plots the phase of the frequency response of the continuous-time transfer function `numc/denc`. The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options (such as time delay) can be provided in a structure `opt` created with `responseset`; fields `Delay` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

`bodephase(Ac,Bc,Cc,Dc)` plots the phase of the frequency response $Y(j\omega)/U(j\omega)$ of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$j\omega X(j\omega) \quad = \quad A_c X(j\omega) + B_c U(j\omega)$$

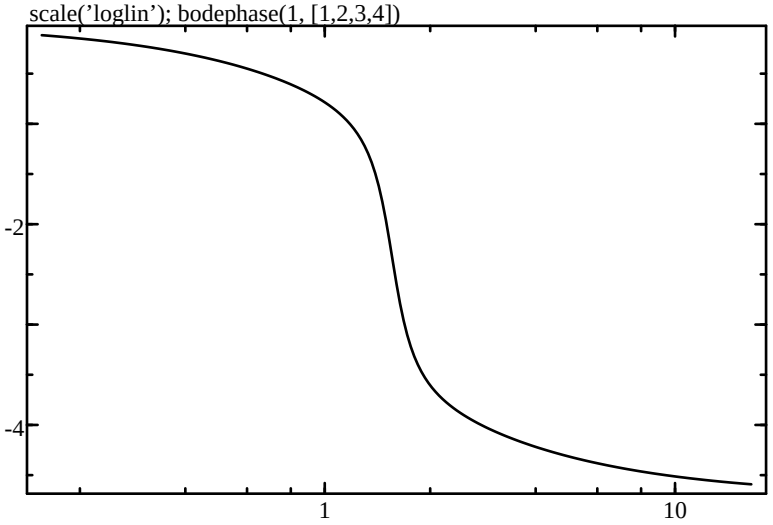

Figure 10.10 `scale('loglin'); bodephase(1, [1,2,3,4])`

$$Y(j\omega) \quad = \quad C_c X(j\omega) + D_c U(j\omega)$$

With output arguments, bodephase gives the phase and the frequency as column vectors. No display is produced.

Example

Green plot for αrg(1/($s^3 + 2s^2 + 3s + 4$)), with $s = j\omega$ (see Fig. 10.10):

```
bodephase(1, [1, 2, 3, 4], 'g');
```

See also

`bodemag`, `dbodephase`, `responseset`, `plotset`

dbodemag

Magnitude Bode diagram for a discrete-time system.

Syntax

```
dbodemag(numd, dend, Ts)
dbodemag(numd, dend, Ts, w)
dbodemag(numd, dend, Ts, opt)
dbodemag(numd, dend, Ts, w, opt)
dbodemag(Ad, Bd, Cd, Dd, Ts)
dbodemag(Ad, Bd, Cd, Dd, Ts, w)
dbodemag(Ad, Bd, Cd, Dd, Ts, opt)
```

```
dbodemag(Ad, Bd, Cd, Dd, Ts, w, opt)
dbodemag(..., style)
dbodemag(..., style, id)
(mag, w) = dbodemag(...)
```

Description

`dbodemag(numd,dend,Ts)` plots the magnitude of the frequency response of the discrete-time transfer function `numd/dend` with sampling period `Ts`. The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`dbodemag(Ad,Bd,Cd,Dd,Ts)` plots the magnitude of the frequency response $Y(j\omega)/U(j\omega)$ of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} zX(z) &= A_dX(z) + B_dU(z) \\ Y(z) &= C_dX(z) + D_dU(z) \end{aligned}$$

where $z = e^{j\omega T_s}$.

With output arguments, dbodemag gives the magnitude and the frequency as column vectors. No display is produced.

Example

```
dbodemag(1,poly([0.9,0.7+0.6j,0.7-0.6j]),1);
```

See also

`bodemag`, `dbodephase`, `dsigma`, `responseset`, `plotset`

dbodephase

Phase Bode diagram for a discrete-time system.

Syntax

```
dbodephase(numd, dend, Ts)
dbodephase(numd, dend, Ts, w)
dbodephase(numd, dend, Ts, opt)
dbodephase(numd, dend, Ts, w, opt)
dbodephase(Ad, Bd, Cd, Dd, Ts)
dbodephase(Ad, Bd, Cd, Dd, Ts, w)
dbodephase(Ad, Bd, Cd, Dd, Ts, opt)
dbodephase(Ad, Bd, Cd, Dd, Ts, w, opt)
dbodephase(..., style)
dbodephase(..., style, id)
(phase, w) = dbodephase(...)
```

Description

`dbodemag(numd,dend,Ts)` plots the phase of the frequency response of the discrete-time transfer function `numd/dend` with sampling period `Ts`. The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`dbodephase(Ad,Bd,Cd,Dd,Ts)` plots the phase of the frequency response $Y(j\omega)/U(j\omega)$ of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} zX(z) &= A_dX(z) + B_dU(z) \\ Y(z) &= C_dX(z) + D_dU(z) \end{aligned}$$

where $z = e^{j\omega T_s}$.

With output arguments, dbodephase gives the phase and the frequency as column vectors. No display is produced.

Example

```
dbodephase(1,poly([0.9,0.7+0.6j,0.7-0.6j]),1);
```

See also

`bodephase`, `dbodemag`, `responseset`, `plotset`

dimpulse

Impulse response plot of a discrete-time linear system.

Syntax

```
dimpulse(numd, dend, Ts)
dimpulse(numd, dend, Ts, opt)
dimpulse(Ad, Bd, Cd, Dd, Ts)
dimpulse(Ad, Bd, Cd, Dd, Ts, opt)
dimpulse(..., style)
dimpulse(..., style, id)
(y, t) = dimpulse(...)
```

Description

`dimpulse(numd,dend,Ts)` plots the impulse response of the discrete-time transfer function `numd/dend` with sampling period `Ts`.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`dimpulse(Ad,Bd,Cd,Dd,Ts)` plots the impulse response of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(t) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

where u(k) is a unit discrete impulse. The state-space model must have a scalar input, and may have a scalar or vector output.

With output arguments, `dimpulse` gives the output and the time as column vectors. No display is produced.

Example

```
dimpulse(1, poly([0.9,0.7+0.6j,0.7-0.6j]), 1, 'r');
```

See also

`impulse`, `dstep`, `dlsim`, `dinitial`, `responseset`, `plotset`

dinitial

Time response plot of a discrete-time linear state-space model with initial conditions.

Syntax

```
dinitial(Ad, Bd, Cd, Dd, Ts, x0)
dinitial(Ad, Cd, Ts, x0)
dinitial(..., opt)
dinitial(..., style)
dinitial(..., style, id)
(y, t) = dinitial(...)
```

Description

`dinitial(Ad,Bd,Cd,Dd,Ts,x0)` plots the output(s) of the discrete-time state-space model (Ad,Bd,Cd,Dd) with null input and initial state `x0`. The model is defined as

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(t) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

where u(k) is null. Sampling period is `Ts`. The state-space model may have a scalar or vector output.

Since there is no system input, matrices `Bd` and `Dd` are not used. They can be omitted.

The simulation time range can be provided in a structure `opt` created with `responseset`. It is a vector of two elements, the start time and the end time. Such an explicit time range is required when the response is not displayed in a plot where the x axis represents the time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `dinitial` gives the output and the time as column vectors. No display is produced.

See also

`initial`, `dimpulse`, `responseset`, `plotset`

dlsim

Time response plot of a discrete-time linear system with arbitrary input.

Syntax

```
dlsim(numd, dend, u, Ts)
dlsim(Ad, Bd, Cd, Dd, u, Ts)
dlsim(Ad, Bd, Cd, Dd, u, Ts, x0)
dlsim(..., opt)
dlsim(..., style)
dlsim(..., style, id)
dlsim(..., opt, style)
dlsim(..., opt, style, id)
(y, t) = dlsim(...)
```

Description

`dlsim(numd,dend,u,Ts)` plots the time response of the discrete-time transfer function `numd/dend` with sampling period `Ts`. The input is given in real vector `u`, where the element `i` corresponds to time `(i-1)*Ts`. Input samples before 0 and after `length(u)-1` are 0.

`dlsim(Ad,Bd,Cd,Dd,u,Ts)` plots the time response of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(t) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

where the system input at time sample k is `u(k,:)'`. For single-input systems, u can also be a row vector.

`dlsim(Ad,Bd,Cd,Dd,u,Ts,x0)` starts with initial state `x0` at time `t=0`. The length of `x0` must match the number of states. The default initial state is the zero vector.

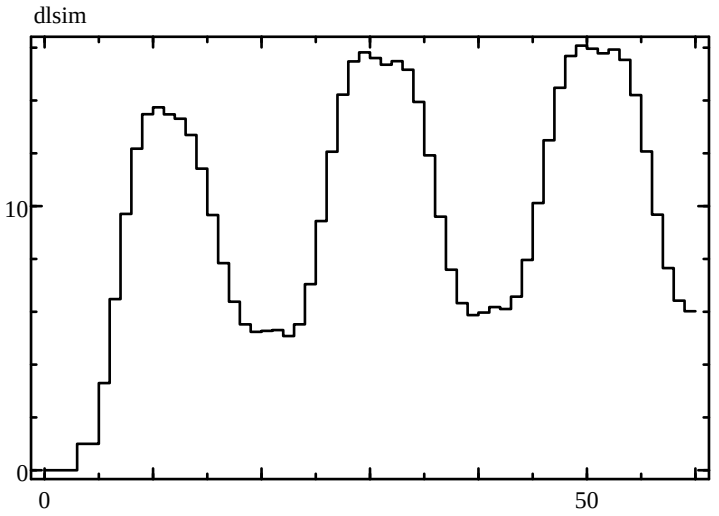

Figure 10.11 `dlsim(1, poly([0.9,0.7+0.6j,0.7-0.6j]), u)`

The simulation time range can be provided in a structure `opt` created with `responseset`. It is a vector of two elements, the start time and the end time. Such an explicit time range is required when the response is not displayed in a plot where the x axis represents the time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `dlsim` gives the output and the time as column vectors (or an array for the output of a multiple-output state-space model, where each row represents a sample). No display is produced.

Example

Simulation of a third-order system with a rectangular input (see Fig. 10.11):

```
u = repmat([ones(1,10), zeros(1,10)], 1, 3);
dlsim(1, poly([0.9,0.7+0.6j,0.7-0.6j]), u, 1, 'rs');
```

See also

`dstep`, `dimpulse`, `dinitial`, `lsim`, `responseset`, `plotset`

dnichols

Nichols diagram of a discrete-time system.

Syntax

```
dnichols(numd, dend)
dnichols(numd, dend, w)
dnichols(numd, dend, opt)
dnichols(numd, dend, w, opt)
dnichols(..., style)
dnichols(..., style, id)
w = dnichols(...)
(mag, phase) = dnichols(...)
(mag, phase, w) = dnichols(...)
```

Description

`dnichols(numd,dend)` displays the Nichols diagram of the discrete-time transfer function given by polynomials `numd` and `dend`. In discrete time, the Nichols diagram is the locus of the complex values of the transfer function evaluated at $e^{j\omega}$, where ω is a real number between 0 and π inclusive, displayed in the phase-magnitude plane. Usually, the magnitude is displayed with a logarithmic or dB scale; use `scale('lindb')` or `scale('linlog/lindb')` before `dnichols`.

The range of frequencies is selected automatically between 0 and π or can be specified in an optional argument `w`, a vector of normalized frequencies.

Further options can be provided in a structure `opt` created with `responseset`; fields `NegFreq` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

With output arguments, `dnichols` gives the magnitude and phase of the frequency response and the frequency as column vectors. No display is produced.

In Sysquake, when the mouse is over a Nichols diagram, in addition to the magnitude and phase which can be retrieved with `_y0` and `_x0`, the normalized frequency is obtained in `_q`.

Example

```
scale('lindb');
ngrid;
dnichols(3, poly([0.9,0.7+0.6j,0.7-0.6j]))
```

See also

`nichols`, `ngrid`, `dnyquist`, `responseset`, `plotset`

dnyquist

Nyquist diagram of a discrete-time system.

Syntax

```
dnyquist(numd, dend)
dnyquist(numd, dend, w)
dnyquist(numd, dend, opt)
dnyquist(numd, dend, w, opt)
dnyquist(..., style)
dnyquist(..., style, id)
w = dnyquist(...)
(re, im) = dnyquist(...)
(re, im, w) = dnyquist(...)
```

Description

The Nyquist diagram of the discrete-time transfer function given by polynomials `numd` and `dend` is displayed in the complex plane. In discrete time, the Nyquist diagram is the locus of the complex values of the transfer function evaluated at $e^{j\omega}$, where ω is a real number between 0 and π inclusive (other definitions include the range between π and 2π, which gives a symmetric diagram with respect to the real axis).

The range of frequencies is selected automatically between 0 and π or can be specified in an optional argument `w`, a vector of normalized frequencies.

Further options can be provided in a structure `opt` created with `responseset`; fields `NegFreq` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

With output arguments, `dnichols` gives the real and imaginary parts of the frequency response and the frequency as column vectors. No display is produced.

In Sysquake, when the mouse is over a Nyquist diagram, in addition to the complex value which can be retrieved with `_z0` or `_x0` and `_y0`, the normalized frequency is obtained in `_q`.

Example

Nyquist diagram with the same scale along both x and y axis and a Hall chart grid (reduced to a horizontal line) (see Fig. 10.12)

```
scale equal;
hgrid;
dnyquist(3, poly([0.9,0.7+0.6j,0.7-0.6j]))
```

See also

`nyquist`, `hgrid`, `dnichols`, `responseset`, `plotset`

dsigma

Singular value plot for a discrete-time state-space model.

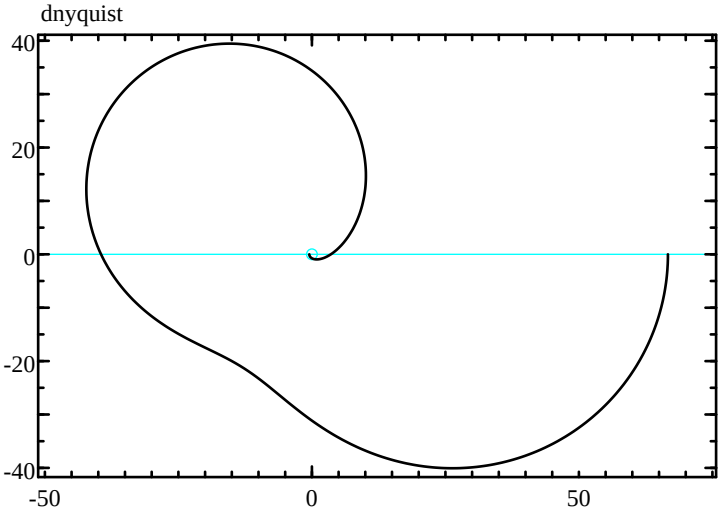

Figure 10.12 `dnyquist(3, poly([0.9,0.7+0.6j,0.7-0.6j]))`

Syntax

```
dsigma(Ad, Bd, Cd, Dd, Ts)
dsigma(Ad, Bd, Cd, Dd, Ts, w)
dsigma(Ad, Bd, Cd, Dd, Ts, opt)
dsigma(Ad, Bd, Cd, Dd, Ts, w, opt)
dsigma(..., style)
dsigma(..., style, id)
(sv, w) = dsigma(...)
```

Description

`dsigma(Ad,Bd,Cd,Dd,Ts)` plots the singular values of the frequency response of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} zX(z) &= A_dX(z) + B_dU(z) \\ Y(z) &= C_dX(z) + D_dU(z) \end{aligned}$$

where $z = e^{j\omega T_s}$ and Ts is the sampling period.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`dsigma` is the equivalent of `dbodemag` for multiple-input systems. For single-input systems, it produces the same plot.

The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

With output arguments, `dsigma` gives the singular values and the frequency as column vectors. No display is produced.

See also

dbodemag, dbodephase, sigma, responseset, plotset

dstep

Step response plot of a discrete-time linear system.

Syntax

```
dstep(numd, dend, Ts)
dstep(numd, dend, Ts, opt)
dstep(Ad, Bd, Cd, Dd, Ts)
dstep(Ad, Bd, Cd, Dd, Ts, opt)
dstep(..., style)
dstep(..., style, id)
(y, t) = dstep(...)
```

Description

dstep(numd,dend,Ts) plots the step response of the discrete-time transfer function numd/dend with sampling period Ts.

Further options can be provided in a structure opt created with responseset; field Range is utilized. The optional arguments style and id have their usual meaning.

dstep(Ad,Bd,Cd,Dd,Ts) plots the step response of the discrete-time state-space model (Ad,Bd,Cd,Dd) defined as

$$\begin{aligned} x(k+1) &= A_d x(k) + B_d u(t) \\ y(k) &= C_d x(k) + D_d u(k) \end{aligned}$$

where u(k) is a unit step. The state-space model must have a scalar input, and may have a scalar or vector output.

With output arguments, dstep gives the output and the time as column vectors. No display is produced.

Examples

Step response of a discrete-time third-order system (see Fig. 10.13):

```
dstep(1, poly([0.9,0.7+0.6j,0.7-0.6j]), 1, 'g');
```

Step response of a state-space model with two outputs, and a style argument which is a struct array of two elements to specify two different styles:

```
A = [-0.3,0.1;-0.8,-0.4];
B = [2;3];
C = [1,3;2,1];
D = [2;1];
style = {Color='navy',LineWidth=3; Color='red',LineStyle='-'};
step(A, B, C, D, style);
```

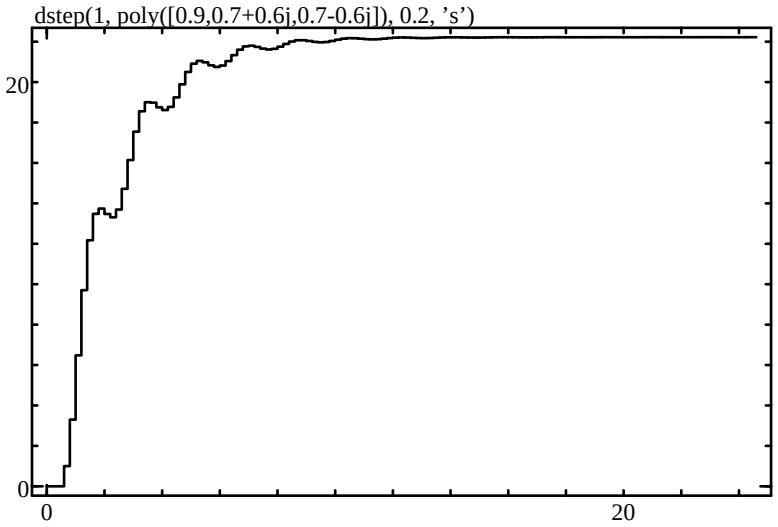

Figure 10.13 `dstep(1,poly([.9,.7+.6j,.7-.6j]),0.2,'s')`

See also

`dimpulse`, `dlsim`, `step`, `hstep`, `responseset`, `plotset`

erlocus

Root locus of a polynomial with coefficients bounded by an ellipsoid.

Syntax

```
erlocus(C0, P)
erlocus(C0, P, sizes, colors)
```

Description

`erlocus` displays the set of the roots of all the polynomial whose coefficients are bounded by an ellipsoid defined by `C0` and `P`. The polynomials are defined as `C0 + [0,dC]`, where `dC*inv(P)*dC' < 1`.

If `sizes` and `colors` are provided, `sizes` must be a vector of n values and `colors` an n-by-3 matrix whose columns correspond respectively to the red, green, and blue components. The locus corresponding to `dC*inv(P)*dC' < sizes(i)^2` is displayed with `colors(i,:)`. The vector `sizes` must be sorted from the smallest to the largest ellipsoid. The default values are `sizes = [0.1;0.5;1;2]` and `colors = [0,0,0;0,0,1;0.4,0.4,1;0.8,0.8,0.8]` (i.e. black, dark blue, light blue, and light gray).

Warning: depending on the size of the figure (in pixels) and the speed of the computer, the computation may be slow (several seconds). The number of sizes does not have a big impact.

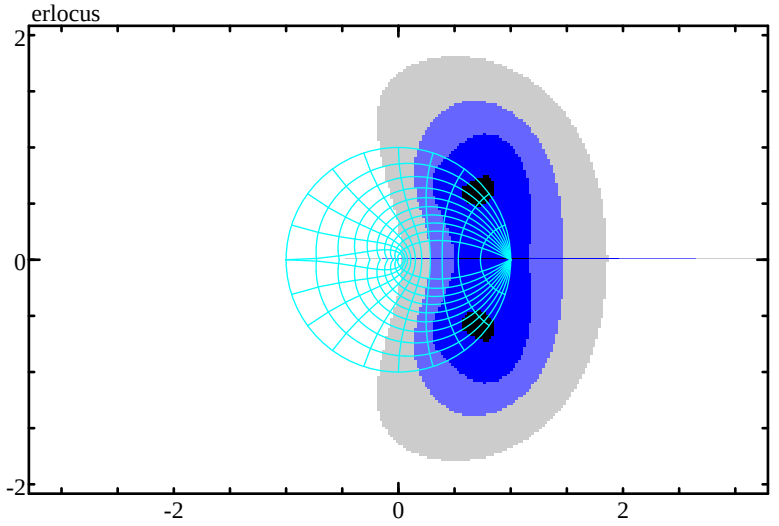

Figure 10.14 `erlocus(poly([.8,.7+.6j,.7-.6j]), eye(3))`

Example

Roots of the polynomial $(z-0.8)(z-0.7-0.6j)(z-0.7+0.6j)$, where the coefficients, in R^3, have an uncertainty bounded by a unit sphere (see Fig. 10.14).

```
scale('equal', [-2,2,-2,2]);
erlocus(poly([0.8, 0.7+0.6j, 0.7-0.6j]), eye(3));
zgrid;
```

See also

`plotroots`, `rlocus`

hgrid

Hall chart grid.

Syntax

```
hgrid
hgrid(style)
```

Description

`hgrid` plots a Hall chart in the complex plane of the Nyquist diagram. The Hall chart represents circles which correspond to the same magnitude or phase of the closed-loop frequency response. The optional argument specifies the style.

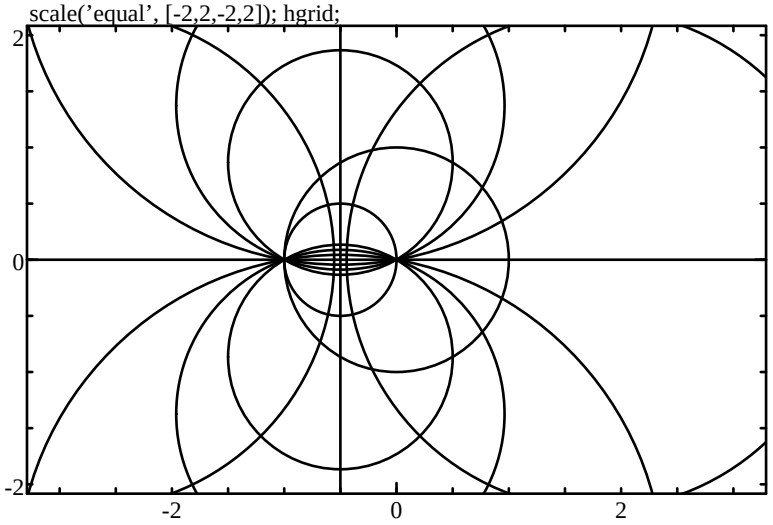

Figure 10.15 Result of `hgrid` when the whole grid is displayed.

The whole grid is displayed only if the user selects it in the Grid menu, or after the command `plotoption fullgrid`. By default, only the unit circle and the real axis are displayed. The whole grid is made of the circles corresponding to a closed-loop magnitude of plus or minus 0, 2, 4, 6, 10, and 20 dB; and to a closed-loop phase of plus or minus 0, 10, 20, 30, 45, 60, and 75 degrees.

Example

Hall chart grid with a Nyquist diagram (see Fig. 10.15):

```
scale('equal', [-1.5, 1.5, -1.5, 1.5]);
hgrid;
nyquist(20, poly([-1,-2+1j,-2-1j]))
```

See also

`ngrid`, `nyquist`, `plotset`, `plotoption`

hstep

Step response plot of a discrete-time transfer function followed by a continuous-time transfer function.

Syntax

```
hstep(numd, dend, Ts, numc, denc)
hstep(numd, dend, Ts, numc, denc, style)
hstep(numd, dend, Ts, numc, denc, style, id)
```

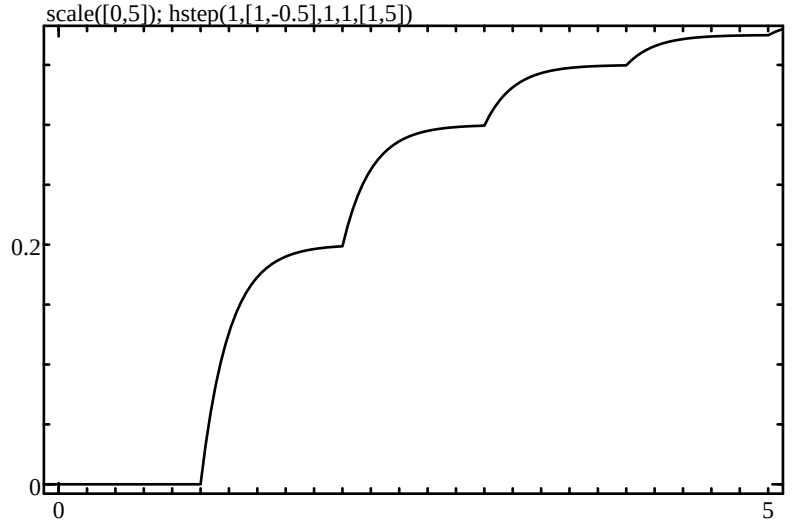

Figure 10.16 `scale([0,5]); hstep(1,[1,-0.5],1,1,[1,5])`

Description

A step is filtered first by numd/dend, a discrete-time transfer function with sampling period Ts; the resulting signal is converted to continuous-time with a zero-order hold, and filtered by the continuous-time transfer function numc/denc.

Most discrete-time controllers are used with a zero-order hold and a continuous-time system. `hstep` can display the simulated output of the system when a step is applied somewhere in the loop, e.g. as a reference signal or a disturbance. The transfer function numd/dend should correspond to the transfer function between the step and the system input; the transfer function numc/denc should be the model of the system.

Note that the simulation is performed in open loop. If an unstable system is stabilized with a discrete-time feedback controller, all closed-loop transfer functions are stable; however, the simulation with `hstep`, which uses the unstable model of the system, may diverge if it is run over a long enough time period, because of round-off errors. But in most cases, this is not a problem.

Example

Exact simulation of the output of a continuous-time system whose input comes from a zero-order hold converter (see Fig. 10.16):

```
% unstable system continuous-time transfer function
num = 1;
den = [1, -1];
```

```
% sampling at Ts = 1 (too slow, only for illustration)
Ts = 1;
[numd, dend] = c2dm(num, den, Ts);
% stabilizing proportional controller
kp = 1.5;
% transfer function between ref. signal and input
b = conv(kp, dend);
a = addpol(conv(kp, numd), dend);
% continuous-time output for a ref. signal step
scale([0,10]);
hstep(b, a, Ts, num, den);
% discrete-time output (exact)
dstep(conv(b, numd), conv(a, dend), Ts, 'o');
```

See also

`step`, `dstep`, `plotset`

impulse

Impulse response plot of a continuous-time linear system.

Syntax

```
impulse(numc, denc)
impulse(numc, denc, opt)
impulse(Ac, Bc, Cc, Dc)
impulse(Ac, Bc, Cc, Dc, opt)
impulse(..., style)
impulse(..., style, id)
(y, t) = impulse(...)
```

Description

`impulse(numc,denc)` plots the impulse response of the continuous-time transfer function `numc/denc`.

Further options can be provided in a structure `opt` created with `responseset`; fields `Delay` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

`impulse(Ac,Bc,Cc,Dc)` plots the impulse response of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t)\\ y(t) &= C_c x(t) + D_c u(t)\end{aligned}$$

where u is a Dirac impulse. The state-space model must have a scalar input, and may have a scalar or vector output.

With output arguments, `impulse` gives the output and the time as column vectors. No display is produced.

Example

```
impulse(1, 1:4, 'm');
```

See also

`dimpulse`, `step`, `lsim`, `initial`, `responseset`, `plotset`

initial

Time response plot for a continuous-time state-space model with initial conditions.

Syntax

```
initial(Ac, Bc, Cc, Dc, x0)
initial(Ac, Cc, x0)
initial(Ac, Bc, Cc, Dc, x0, opt)
initial(..., style)
initial(..., style, id)
(y, t) = initial(...)
```

Description

`initial(Ac,Bc,Cc,Dc,x0)` plots the output(s) of the continuous-time state-space model (Ac,Bc,Cc,Dc) with null input and initial state `x0`. The model is defined as

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned}$$

where u(t) is null. The state-space model may have a scalar or vector output.

Since there is no system input, matrices `Bd` and `Dd` are not used. They can be omitted.

The simulation time range can be provided in a structure `opt` created with `responseset`. It is a vector of two elements, the start time and the end time. Such an explicit time range is required when the response is not displayed in a plot where the x axis represents the time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `initial` gives the output and the time as column vectors. No display is produced.

Example

Response of a continuous-time system whose initial state is `[5;3]` (see Fig. 10.17):

```
initial([-0.3,0.1;-0.8,-0.4],[2;3],[1,3;2,1],[2;1],[5;3])
```

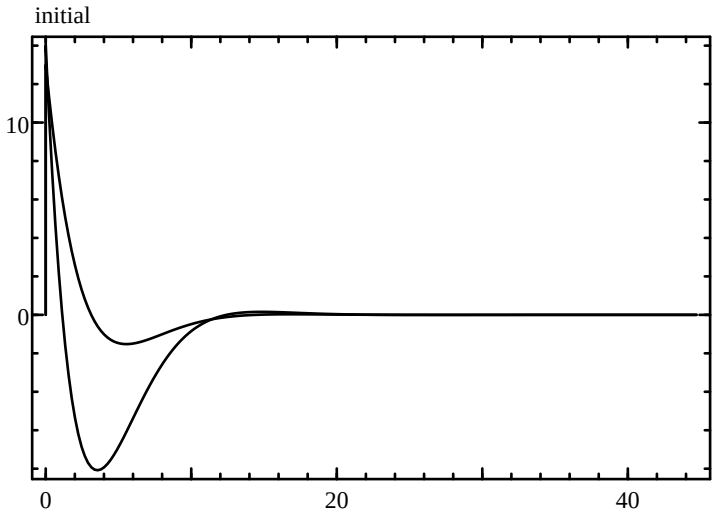

Figure 10.17 Example of `initial`

See also

`dinitial`, `impulse`, `responseset`, `plotset`

lsim

Time response plot of a continuous-time linear system with piece-wise linear input.

Syntax

```
lsim(numc, denc, u, t)
lsim(numc, denc, u, t, opt)
lsim(Ac, Bc, Cc, Dc, u, t)
lsim(Ac, Bc, Cc, Dc, u, t, opt)
lsim(Ac, Bc, Cc, Dc, u, t, x0)
lsim(Ac, Bc, Cc, Dc, u, t, x0, opt)
lsim(..., style)
lsim(..., style, id)
(y, t) = lsim(...)
```

Description

`lsim(numc,denc,u,t)` plots the time response of the continuous-time transfer function `numd/dend`. The input is piece-wise linear; it is defined by points in real vectors `t` and `u`, which must have the same length. Input before `t(1)` and after `t(end)` is 0. The input used for the simulation is interpolated to have a smooth response.

`lsim(Ac,Bc,Cc,Dc,u,t)` plots the time response of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned}$$

where the system input at time sample `t(i)` is `u(i,:)'`. For single-input systems, u can also be a row vector.

`lsim(Ac,Bc,Cc,Dc,u,t,x0)` starts with initial state `x0` at time `t=0`. The length of `x0` must match the number of states. The default initial state is the zero vector.

Options can be provided in a structure `opt` created with `responseset`:

`'Range'` The range is a vector of two elements, the start time and the end time. Such an explicit time range is required when the response is not displayed in a plot where the x axis represents the time.

`'tOnly'` When `opt.tOnly` is `true`, `lsim` produces output only at the time instants defined in `t`. The logical value `false` gives the default interpolated values.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `lsim` gives the output and the time as column vectors (or an array for the output of a multiple-output state-space model, where each row represents a sample). No display is produced.

Example

Response of continuous-time system given by its transfer function with an input defined by linear segments (see Fig. 10.18):

```
t = [0, 10, 20, 30, 50];
u = [1, 1, 0, 1, 1];
lsim(1, [1, 2, 3, 4], u, t, 'b');
```

See also

`step`, `impulse`, `initial`, `dlsim`, `responseset`, `plotset`

ngrid

Nichols chart grid.

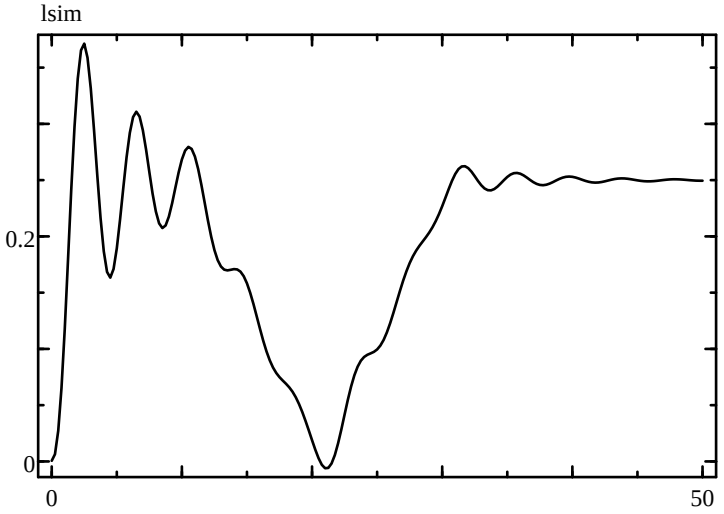

Figure 10.18 `lsim(1, [1,2,3,4], u, t)`

Syntax

```
ngrid
ngrid(mag)
ngrid(..., style)
```

Description

`ngrid` plots a Nichols chart in the complex plane of the Nichols diagram (see Fig. 10.19). The Nichols chart is a set of lines which correspond to the same magnitude of the closed-loop frequency response. The style can be specified with an input argument.

The whole grid is displayed only if the user selects it in the Grid menu, or after the command `plotoption fullgrid`. By default, only the lines corresponding to unit magnitude and to a phase equal to $-\pi(1+2k)$, with integer k, are displayed. The whole grid is made of the lines corresponding to a closed-loop magnitude of -12, -6, -3, 0, 3, 6 and 12 dB.

The closed-loop magnitude can be specified with an input argument, a scalar or an array of positive real values. If the style is also specified, it must follow the magnitue.

Examples

Plain Nichols chart grid for a Nichols diagram:

```
ngrid;
nichols(7, 1:3);
```

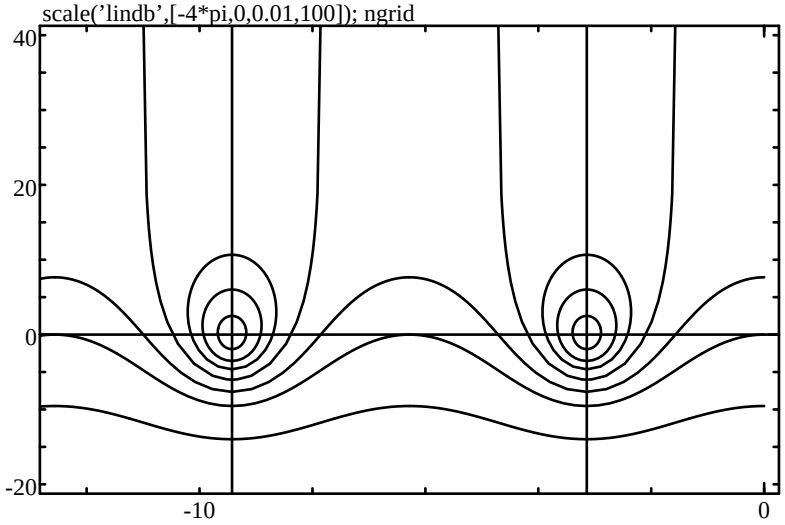

Figure 10.19 Result of `ngrid` in dB when the whole grid is displayed.

Finer Nichols chart with dashed lines:

```
ngrid(logspace(-2, 1, 20), LineStyle='-');
```

See also

`hgrid`, `nichols`, `plotset`, `plotoption`

nichols

Nichols diagram of a continuous-time system.

Syntax

```
nichols(numc, denc)
nichols(numc, denc, w)
nichols(numc, denc, opt)
nichols(numc, denc, w, opt)
nichols(..., style)
nichols(..., style, id)
w = nichols(...)
(mag, phase) = nichols(...)
(mag, phase, w) = nichols(...)
```

Description

`nichols(numc,denc)` displays the Nichols diagram of the continuous-time transfer function given by polynomials `numc` and `denc`. In con-

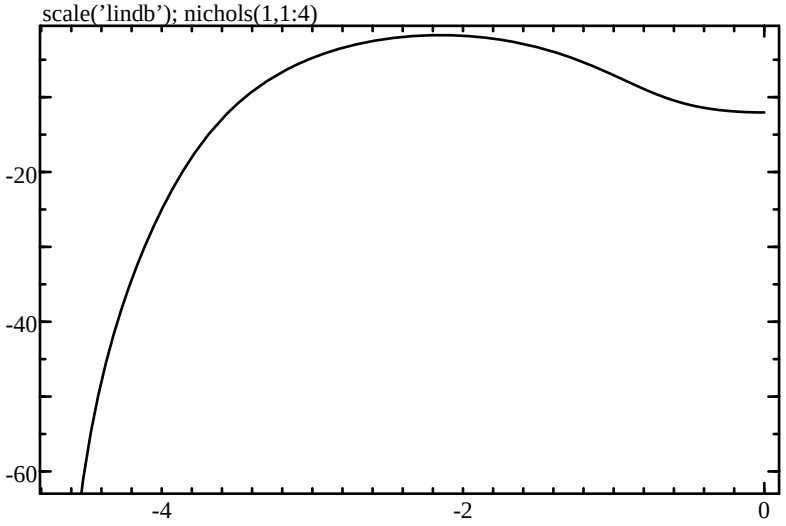

Figure 10.20 `scale('lindb'); nichols(1,1:4)`

tinuous time, the Nichols diagram is the locus of the complex values of the transfer function evaluated at $j\omega$, where ω is real positive, displayed in the phase-magnitude plane. Usually, the magnitude is displayed with a logarithmic or dB scale; use `scale('lindb')` or `scale('linlog/lindb')` before `nichols`.

The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; fields `Delay`, `NegFreq` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

With output arguments, `nichols` gives the phase and magnitude of the frequency response and the frequency as column vectors. No display is produced.

In Sysquake, when the mouse is over a Nichols diagram, in addition to the magnitude and phase which can be retrieved with `_y0` and `_x0`, the frequency is obtained in `_q`.

Examples

Nichols diagram of a third-order system (see Fig. 10.20):

```
scale('lindb');
ngrid;
nichols(20,poly([-1,-2+1j,-2-1j]));
```

Same plot with angles in degrees:

```
scale('lindb');
scalefactor([180/pi, 1]);
```

```
ngrid;
nichols(20,poly([-1,-2+1j,-2-1j]));
```

See also

`dnichols`, `ngrid`, `nyquist`, `responseset`, `plotset`, `scalefactor`

nyquist

Nyquist diagram of a continuous-time system.

Syntax

```
nyquist(numc, denc)
nyquist(numc, denc, w)
nyquist(numc, denc, opt)
nyquist(numc, denc, w, opt)
nyquist(..., style)
nyquist(..., style, id)
w = nyquist(...)
(re, im) = nyquist(...)
(re, im, w) = nyquist(...)
```

Description

The Nyquist diagram of the continuous-time transfer function given by polynomials `numc` and `denc` is displayed in the complex plane. In continuous time, the Nyquist diagram is the locus of the complex values of the transfer function evaluated at $j\omega$, where ω is real positive (other definitions include the real negative values, which gives a symmetric diagram with respect to the real axis).

The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; fields `Delay`, `NegFreq` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

With output arguments, `nyquist` gives the real and imaginary parts of the frequency response and the frequency as column vectors. No display is produced.

In Sysquake, when the mouse is over a Nyquist diagram, in addition to the complex value which can be retrieved with `_z0` or `_x0` and `_y0`, the frequency is obtained in `_q`.

Example

Nyquist diagram of a third-order system (see Fig. 10.21):

```
scale equal;
hgrid;
nyquist(20, poly([-1,-2+1j,-2-1j]))
```

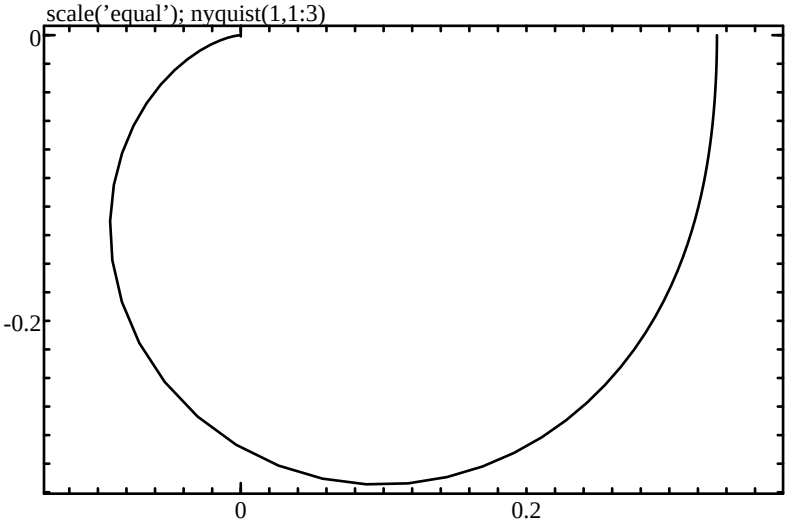

Figure 10.21 `scale equal; nyquist(1,[1,2,3])`

See also

`dnyquist`, `hgrid`, `nichols`, `responseset`, `plotset`

plotroots

Roots plot.

Syntax

```
plotroots(pol)
plotroots(pol, style)
plotroots(pol, style, id)
```

Description

`plotroots(pol)` displays the roots of the polynomial `pol` in the complex plane. If this argument is a matrix, each line corresponds to a different polynomial. The default style is crosses; it can be changed with a second argument, or with named arguments.

Example

```
den = [1, 2, 3, 4];
num = [1, 2];
scale equal;
plotroots(den, 'x');
plotroots(num, 'o');
```

See also

`rlocus`, `erlocus`, `sgrid`, `zgrid`, `plotset`, `movezero`

responseset

Options for frequency responses.

Syntax

```
options = responseset
options = responseset(name1, value1, ...)
options = responseset(options0, name1, value1, ...)
```

Description

`responseset(name1,value1,...)` creates the option argument used by functions which display frequency and time responses, such as `nyquist` and `step`. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, `responseset` creates a structure with all the default options. Note that functions such as `nyquist` and `step` also interpret the lack of an option argument as a request to use the default values. Contrary to other functions which accept options in structures, such as ode45, empty array `[]` cannot be used (it would be interpreted incorrectly as a numeric argument).

When its first input argument is a structure, `responseset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options:

Name	**Default**	**Meaning**
`Delay`	`0`	time delay
`NegFreq`	`false`	negative frequencies
`Offset`	`0`	offset
`Range`	`[]`	time or frequency range
`tOnly`	`false`	samples for specified time only (`lsim`)

Option `Delay` is used only by continuous-time frequency-response and time-response functions; for frequency responses, it subtracts a phase of `delay*w`, where `w` is the angular frequency. Option `Offset` adds a a value to the step or impulse response.

Option `NegFreq` is used in Nyquist and Nichols diagrams, continuous-time or discrete-time; when true, the response is computed for negative frequencies instead of positive frequencies. Option Range should take into account the sampling period for discrete-time commands where it is specified.

Examples

Default options:

```
responseset
  Delay: 0
  NegFreq: false
```

Nyquist diagram of $e^{-s}/(s+1)$:

```
nyquist(1, [1,1], responseset('Delay', 1));
```

Complete Nyquist diagram of $1/(s^3+2s^2+2s+1)$ with dashed line for negative frequencies:

```
nyquist(2, [1,2,2,1]);
nyquist(2, [1,2,2,1], responseset('NegFreq',true), '-');
```

See also

`bodemag`, `bodephase`, `dbodemag`, `dbodephase`, `dlsim`, `dnichols`, `dnyquist`, `dsigma`, `impulse`, `lsim`, `nichols`, `nyquist`, `sigma`, `step`

rlocus

Root locus.

Syntax

```
rlocus(num, den)
rlocus(num, den, style)
rlocus(num, den, style, id)
branches = rlocus(num, den)
```

Description

The root locus is the locus of the roots of the denominator of the closed-loop transfer function (characteristic polynomial) of the system whose open-loop transfer function is num/den when the gain is between 0 and $+\infty$ inclusive. The characteristic polynomial is $\mathrm{num} + k \cdot \mathrm{den}$, with $k \geq 0$. `rlocus` requires a system with real coefficients, causal or not. Note that the rlocus is defined the same way in the domain of the Laplace transform, the z transform, and the delta transform. The root locus is made of `length(den)-1` branches which start from each pole and end to each zero or to a real or complex point at infinity. The locus is symmetric with respect to the real axis, because the coefficients of the characteristic polynomial are real. By definition, closed-loop poles for the current gain (i.e. the roots of num+den) are on the root locus, and move on it when the gain change. `rlocus` plots

only the root locus, *not* the particular values of the roots for the current gain, a null gain or an infinite gain. If necessary, these values should be plotted with `plotroots`.

The part of the root locus which is calculated and drawn depends on the scale. If no scale has been set before explicitly with `scale` or implicitly with `plotroots` or `plot`, the default scale is set such that the zeros of `num` and `den` are visible.

With an output argument, `rlocus` gives the list of root locus branches, i.e. a list of row vectors which contain the roots. Different branches do not always have the same numbers of values, because `rlocus` adapts the gain steps for each branch. Parts of the root locus outside the visible area of the complex plane, as defined by the current scale, have enough points to avoid any interference in the visible area when they are displayed with `plot`. The gains corresponding to roots are not available directly; they can be computed as `real(polyval(den,r)/polyval(num,r))` for root `r`.

As with other plots, the `id` is used for interactive manipulation. Manipulating a root locus means changing the gain of the controller, which keeps the locus at the same place but makes the closed-loop poles move on it. Other changes are done by dragging the open-loop poles and zeros, which are plotted by `plotroots`. To change the gain, you must also plot the current closed-loop poles with the `plotroots` function and use the same ID, so that the initial click identifies the nearest closed-loop pole and the mouse drag makes Sysquake use the root locus to calculate the change of gain, which can be retrieved in `_q` (see the example below).

Examples

Root locus of $(s^2+3s+2)/(s^3+2s^2+3s+4)$ with open-loop poles and zeros added with `plotroots` (see Fig. 10.22):

```
num = [1, 3, 2];
den = [1, 2, 3, 4];
scale('equal', [-4,1,-2,2]);
sgrid;
rlocus(num, den);
plotroots(num, 'o');
plotroots(den, 'x');
```

The second example shows how `rlocus` can be used interactively in Sysquake.

```
figure "Root Locus"
  draw myPlotRLocus(num, den);
  mousedrag num = myDragRLocus(num, _q);

function
```

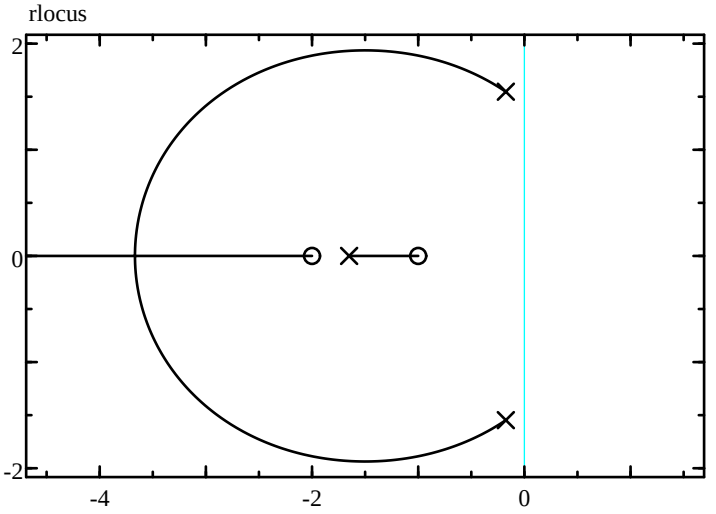

Figure 10.22 Example of `rlocus`

```
{@
function myPlotRLocus(num, den)
  scale('equal', [-3, 1, -2, 2]);
  rlocus(num, den, '', 1);
  plotroots(addpol(num, den), '^', 1);

function num = myDragRLocus(num, q)
  if isempty(q)
    cancel;
  else
    num = q * num;
  end
@}
```

Caveat

The Laguerre algorithm is used for fast evaluation (`roots` and `plotroots` are based on `eig` and have a better accuracy, but their evaluation for a single polynomial is typically 10 times slower). The price to pay is a suboptimal precision for multiple roots and/or high-order polynomials.

See also

`plotroots`, `plotset`, `erlocus`, `sgrid`, `zgrid`

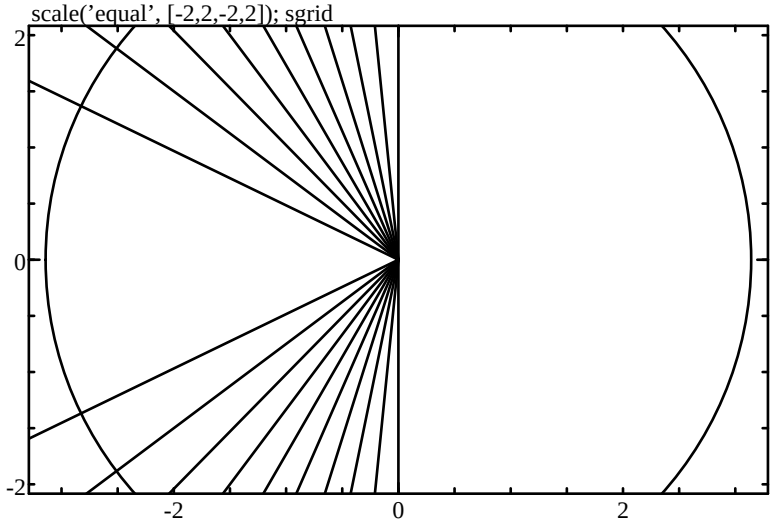

Figure 10.23 Result of `sgrid` when the whole grid is displayed.

sgrid

Relative damping and natural frequency grid for the poles of a continuous-time system.

Syntax

```
sgrid
sgrid(damping, freq)
sgrid(..., style)
```

Description

With no numeric argument, `sgrid` plots a grid of lines with constant relative damping and natural frequencies in the complex plane of s (see Fig. 10.23).

The whole grid is displayed only if the user selects it in the Grid menu, or after the command `plotoption fullgrid`. By default, only the imaginary axis (the stability limit for the poles of the Laplace transform) is displayed.

With one or two numeric arguments, `sgrid` plots only the lines for the specified values of damping and natural frequency. Let p and $\overline{p}$ be the complex conjugate roots of the polynomial $s^2 + 2\omega\zeta s + \omega^2$, where ω is the natural frequency and $\zeta < 1$ the damping. The locus of roots with a constant damping ζ is generated by $|\mathrm{Im}\, p| = \sqrt{1-\zeta^2}\mathrm{Re}\, p$ with $\mathrm{Re}\, p < 0$. The locus of roots with a constant natural frequency ω is a circle of radius ω.

The `style` argument has its usual meaning.

Example

Typical use for poles or zeros displayed in the s plane:

```
scale equal;
sgrid;
plotroots(pol);
```

See also

`zgrid`, `plotroots`, `hgrid`, `ngrid`, `plotset`, `plotoption`

sigma

Singular value plot for a continuous-time state-space model.

Syntax

```
sigma(Ac, Bc, Cc, Dc)
sigma(Ac, Bc, Cc, Dc, w)
sigma(Ac, Bc, Cc, Dc, opt)
sigma(Ac, Bc, Cc, Dc, w, opt)
sigma(..., style)
sigma(..., style, id)
(sv, w) = sigma(...)
```

Description

`sigma(Ac,Bc,Cc,Dc)` plots the singular values of the frequency response of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$\begin{aligned} j\omega X(j\omega) &= A_c X(j\omega) + B_c U(j\omega) \\ Y(j\omega) &= C_c X(j\omega) + D_c U(j\omega) \end{aligned}$$

The range of frequencies is selected automatically or can be specified in an optional argument `w`, a vector of frequencies.

Further options can be provided in a structure `opt` created with `responseset`; field Range is utilized. The optional arguments `style` and `id` have their usual meaning.

`sigma` is the equivalent of `bodemag` for multiple-input systems. For single-input systems, it produces the same plot.

With output arguments, `sigma` gives the singular values and the frequency as column vectors. No display is produced.

See also

`bodemag`, `bodephase`, `dsigma`, `responseset`, `plotset`

step

Step response plot of a continuous-time linear system.

Syntax

```
step(numc, denc)
step(numc, denc, opt)
step(Ac, Bc, Cc, Dc)
step(Ac, Bc, Cc, Dc, opt)
step(..., style)
step(..., style, id)
(y, t) = step(...)
```

Description

`step(numc,denc)` plots the step response of the continuous-time transfer function `numc/denc`.

Further options can be provided in a structure `opt` created with `responseset`; fields `Delay` and `Range` are utilized. The optional arguments `style` and `id` have their usual meaning.

`step(Ac,Bc,Cc,Dc)` plots the step response of the continuous-time state-space model (Ac,Bc,Cc,Dc) defined as

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= C_c x(t) + D_c u(t) \end{aligned}$$

where u is a unit step. The state-space model must have a scalar input, and may have a scalar or vector output.

With output arguments, `step` gives the output and the time as column vectors. No display is produced.

Example

Step response of the continuous-time system $1/(s^3+2s^2+3s+4)$ (see Fig. 10.24):

```
step(1, 1:4, 'b');
```

See also

`impulse`, `lsim`, `dstep`, `hstep`, `responseset`, `plotset`

zgrid

Relative damping and natural frequency grid for the poles of a discrete-time system.

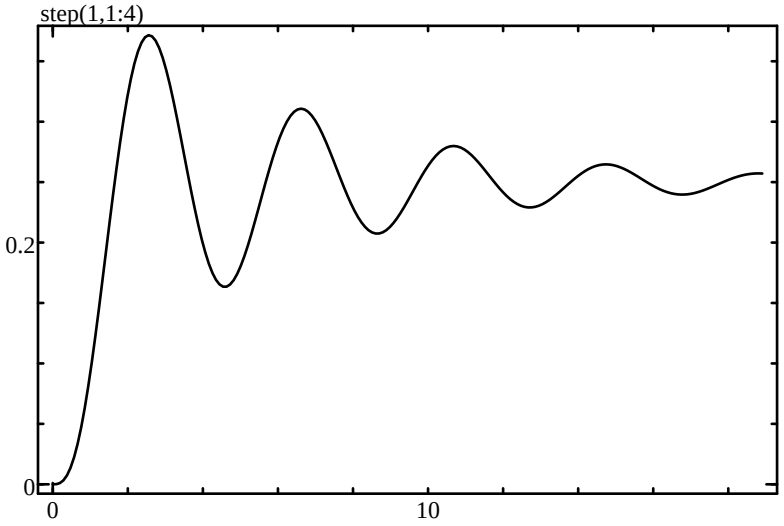

Figure 10.24 `step(1, [1,2,3,4])`

Syntax

```
zgrid
zgrid(damping, freq)
zgrid(..., style)
```

Description

With no numeric argument, `zgrid` plots a grid of lines with constant relative damping and natural frequencies in the complex plane of *z* (see Fig. 10.25).

The whole grid is displayed only if the user selects it in the Grid menu, or after the command `plotoption fullgrid`. By default, only the unit circle (the stability limit for the poles of the *z* transform) is displayed.

With one or two numeric arguments, `zgrid` plots only the lines for the specified values of damping and natural frequency. The damping ζ and the natural frequency ω are defined the same way as for the `sgrid` function, with the mapping $z = e^s$ (a normalized sampling frequency is assumed). With a damping ζ, the line z and its complex conjugate $\overline{z}$ are generated by $z = e^{(-1+j\sqrt{1-\zeta^2}/\zeta)u}$, with $0 \le u \le u_{\max}$ and $u_{\max}$ chosen such that the line has a positive imaginary part. With a natural frequency ω (typically in the range 0 for a null frequency to π for the Nyquist frequency), the line is generated by $e^{\omega e^{jv}}$, where v is such that the curve is inside the unit circle.

The `style` argument has its usual meaning.

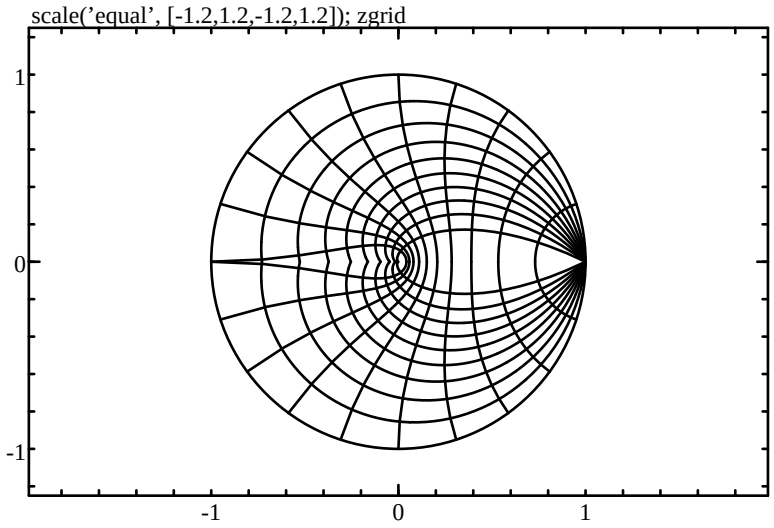

Figure 10.25 Result of `zgrid` when the whole grid is displayed.

Example

Typical use for poles or zeros displayed in the z plane:

```
scale('equal',[-1.2,1.2,-1.2,1.2]);
zgrid;
plotroots(pol);
```

See also

`sgrid`, `plotroots`, `hgrid`, `ngrid`, `plotset`, `plotoption`

10.47 Sysquake Graphical Functions

Functions

button

Button control.

Syntax

```
button(label, b, 'checkmark', style, id)
button(label, n, 'radiobutton', style, id)
```

Description

`button(label,b,'checkmark',style,id)` displays one or several checkmark controls. Checkmark controls are graphical elements whose state can be set on or off individually. The `label` argument contains several fields separated by tabulator characters (`\t`); the first field is displayed aligned to the left, and each subsequent field is displayed to the right of a checkmark button. The number of tabulators fixes the number of checkmarks. Each bit of the `b` argument corresponds to the state of one checkmark; the least significant bit corresponds to the leftmost checkmark. The `style` argument, if present and not empty, is a string which specifies the color(s) of the controls, and the `id` argument is the object id (cf. their description above).

Several rows of controls can be displayed by a single `button` command. Labels of each row are separated by newline characters (`\n`), and the state `b` becomes a column vector.

`button(label,n,'radiobutton',style,id)` displays one or several radio buttons. Radio buttons are similar to checkmarks, but are mutually exclusive. Their state is given by the `n` argument, which is the number of the active radio button on that row. If `n` is smaller than 1 or larger than the number of radio buttons, none is displayed as active.

You can display one or several rows of buttons in a single subplot. You can mix them with text and sliders (commands `text` and `slider`), but not with other graphics.

With `plotoption math`, label fields can contain MathML or LaTeX.

Usually, figure which contain buttons are associated with a mouse-drag handler. The new state is provided in `_x1` and corresponds to the value of the second argument of `button`.

Examples

A single checkmark button:

```
settabs('Style:XX\t');
button('Style:\tbold', isBold, 'checkmark', '', 1);
```

Two rows of checkmark buttons:

```
settabs('Style:XX\t\bboldXXX\t');
button(['Style:\tbold\titalic\n',...
        'Border:\ttop\tbottom\tleft\tright'], ...
          [isBold+2*isItalic; borders], 'checkmark', '', 1);
```

Mutually-exclusive radio buttons on three lines:

```
settabs('Radio:XX\t\b1XX\t');
button('Radio:\t1\t2\t3\n\t4\t5\t6\n\t7\t8\t9', ...
          [x;x-3;x-6], 'radiobutton', '', 1);
```

See also

`settabs`, `pushbutton`, `popupmenu`, `slider`, `textfield`, `text`, `plotoption`

clf

Clear the figure window.

Syntax

```
clf
```

Description

`clf` erases all the subplots, sets only one plot, and resets its scale. It can only be used from the command-line interface (directly or in a function), not in SQ files where figures are erased automatically before being updated.

If graphical commands are issued at the command line interface without `clf`, but after an SQ file has been loaded, their behavior is undefined.

drawnow

Immediate drawing of the plots.

Syntax

```
drawnow
drawnow(id)
```

Description

`drawnow` makes Sysquake immediately draws the result of graphical commands which were executed before. It should be used only from the command-line interface, or in functions called from the command line interface. Graphics generated for interactive subplots are buffered to provide optimal performances. `drawnow` may be useful for basic animations or for benchmarking Sysquake.

Without input argument, `drawnow` draws the contents of the current figure. An input argument can be used to specify the figure window identifier.

Example

```
tic;for i=1:100;clf;step(1,1:4);drawnow;end;toc
  2.2212
```

See also

`clf`

currentfigure

Get or set current figure window.

Syntax

```
id = currentfigure
currentfigure(id)
```

Description

Without input argument, `currentfigure` gets the identifier of the current figure window.

With an input argument, `currentfigure(id)` sets the figure window whose identifier is `id` as the current figure window. Contrary to `figure`, it does not bring it to the front.

`currentfigure` is not implemented in versions of Sysquake without support for multiple figure windows.

See also

`figure`

defaultstyle

Default figure style.

Syntax

```
defaultstyle(name, style)
style = defaultstyle(name)
```

Description

`defaultstyle` sets or gets the default style of figures. It has the same arguments as `figurestyle`. Default style settings are used in new figures, unless they are overridden by `figurestyle`.

See also

`figurestyle`

figure

Create or switch figure window.

Syntax

```
id = figure
figure(id)
```

Description

Without input argument, `figure` creates a new figure window which becomes the current figure window, i.e. the default target of graphical commands not in the context of SQ file draw handlers. It returns an integer number used as an identifier.

With an input argument, `figure(id)` sets the figure window whose identifier is `id` as the current figure window and brings it to the front.

`figure` is not implemented in versions of Sysquake without support for multiple figure windows.

Example

```
id1 = figure;
plot(rand(10));
id2 = figure;
fplot(@sin, [-5, 5]);
figure(id1);
plot(rand(10), 'r');
```

See also

`currentfigure`, `figuretitle`

figuretitle

Set the title of current figure window.

Syntax

```
figuretitle(str)
```

Description

Command `figuretitle(str)` sets the title of the current figure window to string `str`.

`figuretitle` is not implemented in versions of Sysquake without support for multiple figure windows.

See also

`figure`

popupmenu

Pop-up menu.

Syntax

```
popupmenu(label, entries, selection)
popupmenu(label, entries, selection, style)
popupmenu(label, entries, selection, style, id)
```

Description

`popupmenu(label,entries,selection)` displays a pop-up menu. Pop-up menu are simple menus which permits to select an entry among several alternatives. Their normal appearance displays the selected entry in a rectangular shape similar to a push button. When clicked, a menu is displayed at the same location, and the user can move the mouse with the button held down, select another entry, and release the mouse button.

Argument `label` contains a string which is displayed on the left. Argument `entries` contains the entries as a list of strings. Argument `selected` is the current selection, as an integer between 1 and the number of entries. The `style` argument, if present and not empty, is a string which specifies the color of the buttons, and the `id` argument is the object id (cf. their description above).

With `plotoption math`, label can contain MathML or LaTeX.

Usually, figures which contain pop-up menus and buttons are associated with a mousedrag handler only. Clicking a pop-up menu calls the mousedrag handler once, when the mouse button is released. The button is identified with `_id`; the index of the new selected entry is contained in `_x1`.

Example

A simple menu:

```
popupmenu('Color:', {'Red', 'Blue', 'Green'}, 1, '', 1);
```

The corresponding mousedrag handler:

```
function colorMenuHandler(_id, _x1)
  switch _id
    case 1
      fprintf('New color index: %d\n', _x1);
    otherwise
      cancel;
  end
```

See also

`settabs`, `button`, `pushbutton`, `slider`, `textfield`, `text`, `plotoption`

pushbutton

Push button control.

Syntax

```
pushbutton(label, style, id)
```

Description

`pushbutton(label,style,id)` displays one or several push buttons. Push buttons are simple buttons which triggers an action when they are pushed and released. They do not maintain any state, unlike checkmarks and radio buttons created with `button`. The `label` argument contains several fields separated by tabulator characters (`\t`); the first field is displayed aligned to the left, and each subsequent field is displayed as a separate rectangular button. The number of tabulators fixes the number of buttons. The `style` argument, if present and not empty, is a string which specifies the color(s) of the buttons, and the `id` argument is the object id (cf. their description above).

You can display one or several rows of buttons in a single subplot. You can mix them with text, other kinds of buttons and sliders, but not with other graphics.

Usually, figures which contain buttons are associated with a mousedrag handler only. Clicking a button calls the mousedrag handler once, when the mouse button is released. The button is identified with `_id` and `_ix`; no value can be retrieved.

Examples

A single button aligned to the left (a tabulator is placed to the left with `settabs`):

```
settabs('\t');
pushbutton('\tCalculate', '', 1);
```

Two buttons on the same row, green and red:

```
settabs(10);
pushbutton('Motion:\tStart\tStop', 'gr', 1);
```

The corresponding mousedrag handler:

```
function motionButtonHandler(_ix)
  switch _ix
    case 1
      fprintf('Start\n');
    case 2
      fprintf('Stop\n');
    otherwise
      cancel;
  end
```

See also

`settabs`, `button`, `popupmenu`, `slider`, `textfield`, `text`

redraw

Force the display of new graphics.

Syntax

```
redraw
```

Description

`redraw` forces Sysquake to immediately update the display of all subplots. It can only be used from the command-line interface (directly or in a function). It is useful to update the graphics of an SQ script after the variables have been changed manually from the command-line interface, or for SQ files and SQ scripts whose graphics may change at each evaluation because of random data or data imported from the outside.

scalesync

Set scale synchronization between subplots.

Syntax

```
scalesync(s)
scalesync(s, a)
```

Description

`scalesync(a)` sets scale synchronization between the subplots whose indices are given in array `a`. Subplot are numbered from the top-left one, row by row, starting from 1. When subplots have their scale synchronized, zooming or dragging one of the subplots is applied to all the members of the group.

With two arguments, `scalesync(s,a)` synchronizes subplots along the specified axes: X axis if a is `'x'`, Y axis if a is `'y'`, or both X and Y axes if a is `'xy'`.

`scalesync` is typically used in the init handler after a call to `subplots`.

In subplots where a scale overview rectangle has been defined with `scaleoverview`, it is the scale area which is synchronized and not the figure itself, as a source when the user drags or resizes the area or as a target when the user zooms or drags the subplot it is synchronized with.

Examples

Synchronize the frequency scale of the magnitude and phase subplots:

```
subplots('Step\tBode Magnitude\nNyquist\tBode Phase');
scalesync([2, 4], 'x');
```

SQ file illustrating scale overview:

```
init subplots('Simple Plot\tScale Overview')
init scalesync([1, 2])

figure "Simple Plot"
 draw plotSinc(false)

figure "Scale Overview"
 draw plotSinc(true)

functions
{@

function plotSinc(isOverview)
 if isOverview
 scaleoverview([-1, 4, -0.25, 0.2]);
 else
 scale([-1, 4, -0.25, 0.2]);
 end
 fplot(@sinc, [-10, 10]);
@}
```

See also

`subplots`, `subplotsync`, `scaleoverview`

settabs

Set the vertical alignment of text, buttons and sliders.

Syntax

```
settabs(str)
settabs(vec)
```

Description

`settabs` sets tabulator marks which are used by the commands `text`, `slider`, and `button` which follow. Its argument is either a string of runs of characters separated by the tabulator character (`\t`), or a vector of one or more integers representing the number of 'x' characters in runs. Each run is measured and correspond to the width of a column of text, sliders, and buttons. The argument of `settabs` is not

displayed. It should typically match the widest string, plus some margin, to avoid overlapping columns. Right after a tabulator character, the backspace character (\b) represents the width of a checkmark or radio button. The trailing tabulator may be omitted.

If more columns are required by `text`, `slider`, or `button` than what is defined by `settabs`, the last column width is replicated.

Examples

Alignment of text:

```
settabs('Results:xx\t999.99 \t');
text(sprintf('Results:\t%.2f\t%.2f', 2.435, 5.243));
text(sprintf('Log:\t%.2f\t%.2f', log(2.435), log(5.243)));
```

Alignment of radio buttons:

```
settabs('Choice:XXX\t\boneXX\t');
button('Choice:\tone\ttwo\tthree', 2, 'radiobutton', '', 1);
```

Two ways to set one large column followed by multiple small columns:

```
settabs('xxxxxxxxxx\txxxxx\t');
settabs([10,5]);
```

See also

`text`, `button`, `popupmenu`, `slider`, `textfield`

slider

Slider control for interactive manipulation of a scalar.

Syntax

```
slider(label, x, [min,max])
slider(label, x, [min,max], scale)
slider(label, x, [min,max], scale, style)
slider(label, x, [min,max], scale, style, id)
```

Description

The two main ways to manipulate variables in Sysquake consist in moving an element of a figure or in entering new values in a dialog box. The command `slider` provides an additional mean. It displays a set of sliders in a subplot, i.e. user-interface objects with a cursor you can drag to change continuously a scalar value. Like for any other user manipulation in a subplot, the other subplots are updated continuously if a mousedrag handler is provided.

You can display one or several sliders in a single subplot, and multiple thumbs (control element) per slider. You can mix them with text and buttons (commands `text` and `button`), but not with other graphics.

The `label` argument is a string which contains the labels for each slider, separated by a linefeed character (\n). Its width can be set with the `settabs` command. Argument `x` is the current value of the slider. It has one row per slider, and one column per thumb per slider. The rows of the third argument, an array of two columns, contain the minimum and maximum values corresponding to each slider when they are dragged to the left or right, respectively; it can be a 1-by-2 vector if the minimum and maximum values are the same for all the sliders. Argument `scale` is a string made of characters '-' for a linear scale, and 'l' or 'L' for a logarithmic scale; each character corresponds to a slider. If the string is empty, all sliders are linear; if the string contains one character, it applies to all sliders.

The `style` argument has its usual meaning; but only the color is used. Each color corresponds to a thumb. The corresponding thumbs of each slider have the same color. The `scale` argument may be omitted if the style is a structure or a named argument.

With `plotoption math`, label fields can contain MathML or LaTeX.

Sliders are either read-only if the `id` argument is missing, or interactive if it is provided. In the mousedown, mousedrag, and mouseup handlers, each slider is identified by `_nb` and each thumb by `_ix`. The `_x`, `_x0`, and `_x1` parameters can be used to obtain the value corresponding to the position of the click in the slider, the initial value, and the current value. Note that contrary to other plot commands where `_x` is always defined, `_x` is non-empty only if the click is made inside a slider; this is so because the scale can be different for each slider.

During a mouse drag, the range of the manipulated slider is locked by Sysquake, even if you change it in the draw handler. Thus, you can specify a range relative to the current value:

```
slider('x', x, [x-5, x+5], '', '', 1)
```

Examples

Two sliders are displayed. The first one, for variable weight, is linear and has a fixed range between 0 and 1; the second slider, for variable alpha, has a logarithmic scale and a dynamic range between 20 % and 500 %.

```
variable weight, alpha
figure "Sliders"
  draw drawSliders(weight, alpha)
  mousedrag (weight, alpha) = dragSliders(weight, alpha, _nb, _x1)
functions
{@
```

```
  function drawSliders(w1, w2)
    slider('Weight (0-1):\nAlpha:', ...
           [w1; w2], [0,1; w2*[0.2,5]], ...
           '-L', 'kr', 1);
  function (w1, w2) = dragSliders(w1, w2, nb, x1)
    if isempty(nb)
      cancel;
    end
    switch nb
      case 1
        w1 = x1;
      case 2
        w2 = x1;
    end
@}
```

A slider with two thumbs, blue and red. An empty placeholder argument is used for the scale so that the style string is interpreted correctly.

```
slider('Values', [2, 3], [0, 10], '', 'br', 1);
```

The same slider with a structure array for the colors and a named argument for the id:

```
slider('Values', [2, 3], [0, 10], {Color='blue'; Color='red'}, id=1);
```

Four sliders in two rows, where the layout is set in the first argument (labels separated by \t correspond to sliders in the same row). The same limits, defined in a row vector, are used for the four sliders.

```
settabs([5,20,5,20]);
slider('A\tB\nC\tD', [2;5;3;9], [0,10], id=5);
```

See also

`settabs`, `textfield`, `button`, `pushbutton`, `popupmenu`, `text`, `plotoption`

subplot

Manage subplots.

Syntax

```
subplot(m, n, i)
subplot(mni)
subplot mni
```

Description

The `subplot` function specifies the layout of subplots and where the graphical output produced by the commands which follow will be displayed. It can be used from the command-line interface (directly or indirectly in functions) or in SQ scripts. SQ files rely on a different mechanism, where each subplot is defined with a different draw handler and may be displayed on demand.

`subplot(m,n,i)`, `subplot(mni)` with a three-digits number, or `subplot mni` all set an m-by-n grid layout and select the i:th subplot, with `i` between 1 for the top-left subplot and `m*n` for the bottom-right subplot. `subplot` should be executed before each subplot with the same values of `m` and `n`.

Example

Four subplots in a 2-by-2 layout:

```
subplot 221
title('Top-left');
step(1,[1,2,3,4]);
subplot 222
title('Top-right');
plot(rand(10));
subplot 223
title('Bottom-left');
scale equal
nyquist(1,1:4);
subplot 224
title('Last one');
contour(randn(10));
```

See also

`subplots`, `title`

subplotparam

Get or restore the parameter of subplots.

Syntax

```
subplotparam({p1, p2, ...})
p = subplotparam
```

Description

The `subplotparam` function handles the parameter of each subplot. The subplot parameter is the value of `_param`, which is specific to each subplot and can contain any kind of data.

`subplotparam({p1,p2,...})` sets the parameter of each subplot. Each element corresponds to a subplot.

Without input argument, `subplotparam` returns the list of parameters of all subplots.

`subplotparam` complements `subplots`, `subplotprops`, and `subplotpos` by getting or restoring the subplots set by the user. It is typically used in the input and output handlers. It may also be used in the init, menu or key handlers. For restoring the settings, it must be placed after subplots.

See also

`subplots`, `subplotprops`, `subplotpos`, `scale`

subplotpos

Get or restore the position of subplots.

Syntax

```
subplotpos([left1,right1,top1,bottom1;...])
P = subplotpos
```

Description

The `subplotpos` function handles the position of subplots in Free Position mode, which can be enabled in the View menu.

`subplotpos(P)` sets the position of each subplot in the Figure window. Unit is typically 4 pixels, equal to the grid used to snap figures when you move or resize them with the mouse; the origin is at the top left corner of the window. `subplotpos(P)` enables Free Position mode. Each line corresponds to a subplot.

Without input argument, `subplotpos` returns the current position of all subplots if Free Position mode is enabled, or the empty array `[]` otherwise.

`subplotpos` complements `subplots` and `subplotprops` by getting or restoring the subplot positions set by the user. It is typically used in the input and output handlers. It may also be used in init, menu, or key handlers. For restoring the settings, it must be placed after subplots.

Example

```
x = subplotpos
  x =
    0 30.3125 0 20.625
    30.3125 60.625 0 20.625
    0 30.3125 20.625 41.25
    30.3125 60.625 20.625 41.25
subplotpos([0,29,0,20; 31,60,0,20;0,60,21,25])
```

See also

`subplots`, `subplotspring`, `subplotsize`, `subplotprops`, `subplotparam`, `scale`

subplotprops

Get or restore the properties of subplots.

Syntax

```
P = subplotprops
(P, P3)  = subplotprops
subplotprops(P)
subplotprops(P, P3)
```

Description

`subplotprops` complements `subplots` and `subplotpos` by getting or restoring the display options set by the user. These options include the kind of scale (linear, logarithmic or dB, equal for both axis or not, grids, etc.), the scale itself, set with the Zoom, Zoom X, and Drag modes, and the scale overview rectangle if any. Each line corresponds to a subplot. The second argument, if it exists, contains the options for 3D graphics.

`subplotprops` is typically used in the input and output handlers. It may also be used in init, menu, or key handlers. For restoring the settings, it must be placed after subplots.

Example

```
subplotprops
  1 0.1 10 1 100
  0 -1 1 -1 1
  0 -2 0 -1 1
  1 0 10 0 0
subplotprops([1,0.1,10,1,100;0,-1,1,-1,1;0,-2,0,-1,1;1,0,10,0,0])
```

See also

`subplots`, `subplotpos`, `subplotparam`, `scale`

subplots

Get or restore the number and kind of subplots.

Syntax

```
s = subplots
subplots(s)
```

Description

The subplots can be seen as a matrix of figures. Each figure is identified by the name given after the `figure` keyword in the SQ file; an empty name corresponds to an empty subplot. The `subplots` function uses a single string to identify all the subplots. The names of subplots in a row are separated by the tabulator character `'\t'`; rows are separated by the linefeed character `'\n'`. These characters play the same role as respectively the comma and the semicolon in a numeric matrix. However, rows do not have to have the same length; the row with the more subplots determines the size of the subplot array.

In Free Position mode, subplots are specified as a one-dimension array: names are separated by the linefeed character `'\n'`. The position of each subplot is specified with `subplotpos`.

The `subplots` command can be used either with no input and one output argument to retrieve the subplots currently displayed, or with one input and no output to set the number of subplots and their contents.

The most common use of `subplots` is in init handlers to set the initial set of figures, and in menu handlers to switch easily to preconfigured layouts.

With an input argument, `subplots` must not be executed in the draw, mousedown, mousedrag, mousedragcont, mouseover, mouseout, mousescroll, dragin, dragout, or fighandler handlers (the subplots must not be changed during a drag). However, it can be used in a mouseup or mousedoubleclick handler to emulate a button click which could change subplots.

Example

Set the layout to two rows of two subplots, with figures `"Step"` and `"Nyquist"` at top, and `"Poles"` and `"Bode"` at bottom:

```
subplots('Step\tNyquist\nPoles\tBode');
```

See also

`scalesync`, `subplotprops`, `subplotpos`, `subplotparam`

subplotsize

Window size assumed for subplot placement.

Syntax

```
subplotsize(width, height)
subplotsize([width, height])
S = subplotsize
```

Description

`subplotpos` sets the position and size of subplots, and `subplotspring` how they are resized and moved when the window is resized. Hence the initial window size has an effect on the subplot placement for a given size of the window. Function `subplotsize` specifies the window size assumed for `subplotpos`. If the actual window size is different, subplots are resized using the values provided with `subplotspring`.

Important: since `subplotsize` resizes all subplots using the values set by `subplotpos` and `subplotspring`, it must be executed *after* both of them.

`subplotsize(width,height)` or `subplotsize(S)`, where `S` is an array of two elements `[width,height`, sets the assumed size to `width` and `height`, with the same units as those used by `subplotpos`. Both numbers must be positive on all platforms.

Without input argument, `subplotsize` returns the current window size.

See also

`subplotpos`, `subplotspring`

subplotspring

Get or restore the resizing properties of subplots.

Syntax

```
subplotspring([sl1,sw1,sr1,st1,sh1,sb1;...])
S = subplotspring
```

Description

The `subplotspring` function handles how the position of subplots in Free Position mode is adjusted when the window is resized.

In Free Position mode, the automatic adjustment properties of each subplot is specified independently by six numbers in the range [0,1]: three for horizontal size and position, three for vertical size and position. These numbers are used when the window is resized, not when the position of subplots is changed manually or with `subplotpos`. Subplots are resized as if they were made of three springs along the horizontal axis, and three springs along the vertical axis. `subplotspring` sets or retrieves the relative spring stiffness. For instance, if the spring corresponding to the subplot width is infinitely stiff and the springs corresponding to the left and to the right have the same values, resizing the window will keep the subplot width and move it such that spaces at left and right grow or shrink proportionally.

Actual values are $e/(1+e)$, where $e = k/l$, $k = \Delta F/\Delta l$ is the spring constant, l is the length, and F is the force. Changing the window size preserves the force equilibrium. A value of 0 is infinitely flexible (i.e. other springs will keep their length for any window size change), and a value of 1 is infinitely stiff (i.e. the corresponding length will be preserved).

The following triplets are worth mentioning:

`0.5,0.5,0.5` subplot size and position proportional to the window size

`1,1,0` fixed size and position with respect to the left or top of the window

`1,0,1` fixed space on the left and on the right of, or above and below, the subplot

`0,1,1` fixed size and position with respect to the right or bottom of the window

`subplotspring(S)` sets the spring values of each subplot in the Figure window. Each row of array S corresponds to a subplot, in the same order as they are enumerated by function `subplots`. In each row, values correspond to the left, width, right, top, height, and bottom springs.

Without input argument, `subplotspring` returns the current spring values of all subplots if Free Position mode is enabled, or the empty array `[]` otherwise.

`subplotspring` complements `subplots` and `subplotpos`. It should be followed by `subplotsize` so that subplots are immediately resized if the window size does not match the size assumed by `subplotpos`. It is typically used in the input and output handlers; it may also be used in init, menu, or key handlers.

See also

`subplots`, `subplotpos`, `subplotsize`, `subplotprops`, `subplotparam`, `scale`

subplotsync

Set scale synchronization between all subplots.

Syntax

```
S = subplotsync
subplotsync(S)
```

Description

Without input argument, `subplotsync(S)` returns an n-by-2 array which defines if and how subplot scales are synchronized. When there is no synchronization at all, S is the empty array `[]`. Otherwise, it has as many rows as there are subplots. First row correspond to first subplot (top-left for a grid array), second row to second subplot of first row, and so on. For each row, the first element is 0 for no scale synchronization, 1 for synchronization of X axis, 2 for Y axis, and 3 for both X and Y axes; and the second element is an integer different for each group of synchronized subplots.

`subplotsync(S)` sets scale synchronization globally for all subplots. While `scalesync` synchronize the scale of a group of subplots together, `subplotsync` defines the synchronization for all subplots in a single call. Typically, `subplotsync(S)` should be used to restore the synchronization state obtained with `S=subplotsync`, while `scalesync` is simpler and clearer when writing an init handler.

`subplotprops` is typically used in the input and output handlers. It may also be used in init, menu, or key handlers. For restoring the settings, it must be placed after subplots.

See also

`scalesync`, `subplots`, `subplotpos`, `subplotparam`

text

Display formatted text in a figure.

Syntax

```
text(string)
text(string, font)
```

Description

With a single argument, `text` displays its string argument in the current subplot figure. It can be mixed with sliders and buttons, but no other graphics should be displayed. The string is split in rows and columns; columns are separated by the tab character (`'\t'`), and lines by the linefeed character (`'\n'`). The width of the columns can be specified with `settabs`.

A second argument specifies the type face and color to use. It is a structure which is typically created with fontset. The font and size are ignored. Alternatively, named arguments can be used directly, without fontset.

With `plotoption math`, label segments (defined by tab and linefeed characters) can contain MathML or LaTeX.

With three arguments or more, `text` displays a string at the specified position in graphics.

Examples

Two lines and two columns are displayed, with labels in the first column and numeric values in the second column. Note the use of `sprintf` to format the numbers.

```
text(sprintf('One\t%.2f\nPi\t%.f', 1, pi))
```

Red bold text:

```
text('Results', Bold=true, Color='red');
```

See also

`text` (in graphics), `settabs`, `textfield`, `slider`, `button`, `sprintf`, `plotoption`

textfield

Text field.

Syntax

```
textfield(label, format, value)
textfield(label, format, value, style)
textfield(label, format, value, style, id)
```

Description

`textfield(label,format,value)` displays a text field. Text fields are rectangular areas which display a numeric or string value and permit the user to edit it with the keyboard. Argument `label` contains a string which is displayed on the left. Argument `value`, a scalar number or a string, is the contents of the text field; it is formatted after argument `format` and aligned to the right (number) or left (string) of the text field. `format` is a string similar to the first argument of `sprintf`, with a single number-formatting or string sequence; it begins with a percent sign, an optional width, an optional dot and precision, and a single letter which must be one of `fFgGeEkKdoxX` for numbers, or `s` for string. Format `o` interprets input in octal and formats `x` and `X` in hexadecimal. The width limits the maximum number of characters (actually the number of bytes for the UTF-8 representation of the string), but is extended to accept the representation of `value`. It is useful mainly with strings.

The `style` argument, if present and not empty, is a string which specifies the color of the text field. The `id` argument is the object id (cf. their description above).

With `plotoption math`, label can contain MathML or LaTeX.

Usually, figures which contain text fields are associated with a mousedrag handler only. Editing a text field calls the mousedrag handler once, when the user types the Return key or clicks in any figure. The text field is identified with `_id`. For numbers, the new value is contained in `_x1`, with full precision (argument `format` is used only to format the value before displaying it, not to decode it). For strings, the new value is contained in `_str1`.

A single `textfield` command can display text fields for multiple numbers: in that case, argument `value` is a vector or array of values, argument `label` contains the labels displayed in front of each value as a single string where labels are separated with tabulators (`\t`) for values on the same line or line feeds (`\n`) for values on different lines, and arguments `format` and `style` can contain multiple format and style specifications. The layout is controlled by the tabulators set with `settabs`: the first label left-aligned in the first column, the first value in the second column, the second label in the third column if it follows a tabulator chacter, and so on. In the mousedrag handler, each value is identified with `_nb`.

Examples

A simple numeric text field:

```
T = 21.3;
textfield('Temperature:', '%.1f', T, '', 1);
```

The corresponding mousedrag handler:

```
function T = temperatureHandler(_id, _x1)
  switch _id
    case 1
      T = _x1;
      fprintf('New temperature: %g\n', T);
    otherwise
      cancel;
  end
```

Four text fields with different formats and styles:

```
v = [1, 2.3, pi/2, -1.7];
settabs('First value: \t 0000000\t Second value: \t0000000');
labels = 'First value:\t Second value:\nAlpha:\t Beta:';
formats = '%.1g%.1g%.3f%.3f';
styles = 'rgbk';
textfield(labels, formats, v, styles, 1);
```

A string text field for a string of up to 10 bytes, whose id is specified with a named argument:

```
str = 'Hello';
textfield('String:', '%10s', str, id = 1);
```

The corresponding mousedrag handler, with id and new value read directly in the function instead of being passed as arguments:

```
function str = stringHandler()
  switch _id
    case 1
      str = _str1;
      fprintf('New string: %s\n', str);
    otherwise
      cancel;
  end
```

See also

`settabs`, `slider`, `text`, `button`, `pushbutton`, `popupmenu`, `sprintf`, `plotoption`

10.48 Dialog Functions

Dialog functions display a modal window to request an immediate answer from the user. There is a generic function for messages or input, and two specialized ones to choose a filename by browsing the file system: `getfile` to select an existing file, and `putfile` to give a name for a new file in a specific directory.

dialog

Display a dialog box.

Syntax

```
dialog(str)
ok = dialog(str)
(x1,x2,...) = dialog(str,x10,x20,...)
(ok,x1,x2,...) = dialog(str,x10,x20,...)
... = dialog(opt, ...)
dialog(opt1=value1,...)
ok = dialog(opt1=value1,...)
... = dialog(x10,x20,...,opt1=value1,...)
```

Description

`dialog(str)` displays the string `str` in a dialog box with a button labeled OK, and waits until the user clicks the button.

With an output argument, `ok=dialog(str)` displays the string `str` in a dialog box with two buttons, labeled OK and Cancel, and waits until the user clicks one of them; the result is true if the user clicks OK and false if he clicks Cancel.

With more than one input argument and the same number of input and output arguments, `(ok,x1,x2,...)=dialog(str,x10,x20,...)` displays in a dialog box the string `str`, a text field with the value of the remaining input arguments separated by commas, and two buttons labeled OK and Cancel. The user may edit the values in the text field. If he clicks OK, the first output argument is set to true, and the remaining arguments are set to the new value of the expressions in the text field. Should the user click the Cancel button, `ok` is set to false and all the other outputs are set to an empty matrix. With one output argument less, (x1,x2,...)=dialog(str,x10,x20,...) returns the new values if the user clicks OK; otherwise, it throws a Cancel error, which may simplify menu handlers.

Instead of a string, the first input argument of `dialog` can be a structure of options created by `dialogset`. In addition to the prompt message, options permit to provide a title for the dialog window and to limit the precision of double numbers.

Alternatively, all options can be provided as named arguments. Named option arguments may not be mixed with an initial message string or an initial option structure; the unnamed arguments are all considered to be the initial values to be edited in a text field.

The syntax permitted for the expressions typed in the dialog box is a small subset of LME's. In addition to scalars, complex numbers (entered as `2+3j` without the multiplication operator), arrays and strings, are authorized the addition and subtraction operators, the negation, the transpose and complex transpose, the matrix construction functions `zeros`, `ones`, and `eye`, and the range operator `:`. Functions `struct`, `class` and `inline` and operator `@` are also permitted to create structures, objects, inline and anonymous functions respectively. If the expression typed by the user does not satisfy these rules, or if the number of comma-separated values is not equal to the number of expected arguments, the entry is rejected and the dialog box is displayed again. The user can always click the Cancel button to close the dialog box, whatever is typed in the entry field.

Examples

Simple alert box:

```
dialog(sprintf(['You cannot have more zeros than poles ',...
                '(currently %d)'], np));
```

Dialog box with OK and Cancel buttons:

```
if dialog('Do you really want to reset the weights?')
    w = [];
end
```

Dialog with options:

```
opt = dialogset(Title='System',
    Prompt='Transfer function of the system',
    NPrec=4);
num = 1/3;
den = [1/3, 1/7];
(num, den) = dialog(opt, num, den);
```

The same dialog with options provided directly to `dialog` as named arguments:

```
(num, den) = dialog(num, den,
    Title='System',
    Prompt='Transfer function of the system',
    NPrec=4);
```

Two equivalent menu handlers:

```
function (num, den) = menuHandler1(num, den)
  (ok, num, den) = dialog('Numerator, denominator', num, den);
  if ~ok
      cancel;
  end
function (num, den) = menuHandler2(num, den)
  (num, den) = dialog('Numerator, denominator', num, den);
```

Caveats

Some simplified versions of Sysquake may not implement `dialog`. In this case, `dialog` does not display any dialog box and always returns false and empty values.

See also

`dialogset`, `cancel`

dialogset

Options for dialog.

Syntax

```
options = dialogset
options = dialogset(name1, value1, ...)
options = dialogset(options0, name1, value1, ...)
```

Description

`dialogset(name1,value1,...)` creates the option argument used by `dialog`. Options are specified with name/value pairs, where the name is a string which must match exactly the names in the table below. Case is significant. Options which are not specified have a default value. The result is a structure whose fields correspond to each option. Without any input argument, `dialogset` creates a structure with all the default options.

When its first input argument is a structure, `dialogset` adds or changes fields which correspond to the name/value pairs which follow.

Here is the list of permissible options (empty arrays mean "automatic"):

Name	**Default**	**Meaning**
`Title`	`''`	title of the dialog
`Prompt`	`''`	prompt (message)
`NPrec`	15	maximum number of digits for double numbers
`OKIsDefault`	`true`	Return key is a shortcut for OK
`SingleLine`	`false`	single-line representation
`NoChangeIsCancel`	`false`	no change means Cancel

Option `NPrec` is used only to display the initial value in the dialog edit field. The full precision entered is used when decoding output values.

Option `SingleLine` can be used to display values without line breaks in the edit field (for example, matrix rows are separated with semicolons instead of line breaks).

If option `NoChangeIsCancel` is true and the edit field has not been changed at all by the user, the dialog is reported to be cancelled.

Examples

Default options:

```
dialogset
  Title: ''
  Prompt: ''
  NPrec: 15
```

See also

`dialog`

getfile

Display a dialog box for choosing a file.

Syntax

```
path = getfile
path = getfile(prompt)
path = getfile(prompt, mimetypes)
```

Description

Without input argument, `getfile` displays a dialog box where the user may choose an existing file. It gives the full path of the selected file if the user clicks the OK button, or an empty string if he clicks the Cancel button. `getfile(prompt)` displays the string `prompt` in the file dialog box. `getfile(prompt,mimetypes)` displays only the files corresponding to the MIME types enumerated in the string `mimetypes`. Different MIME types are separated by semicolons.

Note that the actual functionality may be limited by the implementation of the standard file dialog of the windowing system. For instance, the prompt may be ignored. Versions of Sysquake without low-level access return an empty string without displaying any dialog.

Example

```
path = getfile('Image file:', 'image/tiff;image/jpeg;image/png')
  /Users/sysquake/sunset.jpg
```

See also

`putfile`

putfile

Display a dialog box to enter a filename at a specific location.

Syntax

```
path = putfile
path = putfile(prompt)
path = putfile(prompt, defaultfilename)
```

Description

Without input argument, `putfile` displays a dialog box where the user may enter the name of a new file at a specific location. It gives the full path of the file if the user clicks the OK button, or an empty string if he clicks the Cancel button. `putfile(prompt)` displays the string `prompt` in the file dialog box. `putfile(prompt,defaultfilename)` proposes the string `defaultfilename` as default file name.

Note that the actual functionality may be limited by the implementation of the standard file dialog of the windowing system. For instance, the prompt may be ignored. Versions of Sysquake without low-level access return an empty string without displaying any dialog.

Example

```
path = putfile('Save as HTML:', 'report.html')
  D:\WORK\REP02.HTM
```

See also

`getfile`

10.49 Sysquake Miscellaneous Functions

cancel

Cancel an operation.

Syntax

```
cancel
cancel(false)
```

Description

In a handler, it is often useful to cancel the whole operation. Avoiding changing any variable is not enough, because it would leave a new set of variables which would make the Undo command not behave as expected. The `cancel` command tells Sysquake to cancel completely the operation, be it a menu handler or the sequence of mousedown, mousedrag and mouseup handlers. `cancel` throws an error; hence its effect will be caught if it occurs in a `try` block.

In the middle of a mousedrag operation, it may happen that the mouse cursor in over an invalid region, but the drag should not be canceled. `cancel(false)` cancels the current execution of the mousedrag or mousedragcont handler, keeping the current value of the output variables.

In a mouseover handler or idle handler, `cancel(false)` prevents the figures to be updated with execution of `draw` handlers.

Example

```
if ~dialog('Do you really want to make the system unstable?')
  cancel;
end
closedLoopRoot = 2;
```

See also

`try`, `error`

hasfeature

Check if a feature is available.

Syntax

```
b = hasfeature(str)
```

Description

`hasfeature(str)` returns true if Sysquake supports the feature whose name is given is string `str`, and false otherwise (even if the feature does not exist in any version). Currently, the following features are supported in some versions of Sysquake:

Feature name	Description
`fileio`	low-level file I/O (`fopen` etc.)
`lapack`	Lapack-based linear algebra functions
`xml`	XML DOM functions

efopen

Open a file embedded in the SQ file.

Syntax

```
fd = efopen(efblockname)
fd = efopen(efblockname, encoding)
```

Description

`efopen(efblockname)` gives a file descriptor to read the contents of a block of type `embeddedfile` in the current SQ file. The file descriptor can be used exactly as if it was obtained with `fopen` for a real file in text mode, with functions like `fgets`, `fgetl`, `fscanf`, `fread`, `fseek`, and `feof`. Function `fclose` must be used to release the file descriptor.

`efopen(efblockname,encoding)` specifies one of two possible encodings for the contents of the block: `'text'` for text (default value), or `'base64'` for base64. Base64 is used to represent binary data as text. Each character of encoded data represents 6 bits of binary data; i.e. one needs four characters for three bytes. The six bits represent 64 different values, encoded with the characters 'A' to 'Z', 'a' to 'z', '0' to '9', '+', and '/' in this order. When the binary data have a length

which is not a multiple of 3, encoded data are padded with one or two characters '=' to have a multiple of 4. The encoded data is usually split in multiple lines of about 60 characters. The decoder ignores characters not used for encoding.

With a base64 encoding, input functions have the same effect as if an nonencoded file had been opened; i.e. the encoded data are decoded on the fly.

Base64 encoding is an Internet standard described in RFC 1521.

Example

To convert a file to base64 in order to embed it in an SQ file, you can use function base64encode as follow:

```
fd = fopen(getfile);
while ~feof(fd)
  fprintf('%s\n',base64encode(char(fread(fd,45))));
end
fclose(fd);
```

If the file is too large to let you easily copy the result from the command-line interface to the SQ file, you can store it in an intermediate file:

```
fd = fopen(getfile('Input'));
out = fopen(putfile('Output'), 'wt');
while ~feof(fd)
  fprintf(out,'%s\n',base64encode(char(fread(fd,45))));
end
fclose(fd);
```

See also

`fopen`, `fclose`, `base64encode`, `base64decode`

idlestate

Control the state of idle processing.

Syntax

```
idlestate(b)
b = idlestate
```

Description

`idlestate(b)` enables the periodic execution of the idle handler if b is true, or disables it otherwise.

With an output argument, `idlestate` gives the current state of idle handler execution.

progress

Computation progress.

Syntax

```
progress(r)
```

Description

When a Sysquake handler executes slowly, typically for more than 1 second, Sysquake displays a hint that it is working and the user should wait. The exact appearance of the hint depends on the platform: it can be graphical or textual.

With multiple calls to `progress(r)`, a handler can indicate to Sysquake the ratio of work completed thus far, from `r=0` (start) to `r=1` (end). Depending on the platform, Sysquake uses the value to display the progress hint as a progress bar or a percentage. The hint is always hidden automatically at the end of the handler execution.

`progress` should be called in handlers expected to take some time to execute, such as menu, import or export handlers. But it does not harm to call it in fast handlers or often, because its own execution is quick and the display update rate remains under the control of Sysquake.

Example

```
function doComputation
  n = 1e3;
  for i = 1:n
    doComputationStep(i);
    progress(i / n);
  end
```

quit

Quit Sysquake.

Syntax

```
quit
```

Description

`quit` quits Sysquake. It has the same effect as choosing Quit or Exit in the File menu.

textoutputopen

Create a new text window for output.

Syntax

```
fd = textoutputopen(title)
fd = textoutputopen(title, markup)
```

Description

Function `textoutputopen(title)` creates a new text window with whose title is the string `title`. It returns a file descriptor which can be used with all output functions, such as `fprintf`. Text output is accumulated into a buffer which is displayed in the window.

With a second input argument, `textoutputopen(title,markup)` creates a new text window for plain text if `markup` is false, or text with markup if `markup` is true. The markup is the same as what is accepted on file descriptor 4.

The text window can be closed with `fclose(fd)`. The contents of the buffer can be reset with `clc(fd)`. Depending on the application and the platform, the

See also

`fprintf`, `fclose`, `clc`

Example

Create a window for text with markup and write some text:

```
fd = textoutputopen('Example', true);
fprintf(fd, '=Example=\n');
fprintf(fd, 'This is a paragraph.\n\n');
fprintf(fd, 'Here is a list:\n* Alpha\n* Beta\n');
```

Close window:

```
fclose(fd);
```

sqcurrentlanguage

Get current language.

Syntax

```
(lang, code) = sqcurrentlanguage
sqcurrentlanguage(lang)
sqcurrentlanguage(code)
```

Description

`sqcurrentlanguage` retrieve the current language chosen by the user for the user interface. It returns up to two output arguments: the first one if the language and the second one is the language code, as defined with `beginlanguage` in the SQ file.

With a string input argument, `sqcurrentlanguage` changes the current language to the one specified by language name or code.

sqfilepath

Get path of SQ file.

Syntax

```
str = sqfilepath
```

Description

`sqfilepath` gives the path of the current SQ file. If the path is not available (e.g. if the SQ file was not loaded from a file), `sqfilepath` returns an empty string.

Example

Get the path of a data file stored in the same directory as the SQ file:

```
pathDir = fileparts(sqfilepath);
pathDataFile = fullfile(pathDir, 'data.txt');
```

See also

`fileparts`, `fullfile`

Chapter 11

Libraries

Libraries are collections of functions which complement the set of built-in functions and operators of LME, the programming language of Sysquake. To use them, type (or add in the functions block of the SQ files which rely on them) a `use` command, such as

```
use stdlib
```

`bench` `bench` implements a benchmark which can be used to compare the performance of LME on different platforms.

`bitfield` `bitfield` implements constructors and methods for bit fields (binary numbers). Standard operators are redefined to enable the use of & and | for bitwise operations, and subscripts for bit extraction and assignment.

`colormaps` `colormaps` defines functions which create color maps for command `colormap`.

`constants` `constants` defines physical constants in SI units.

`date` `date` implements functions for date and time manipulation and conversion to and from strings.

`filter` `filter` implements functions for the design of analog and digital filters.

`lti` `lti` implements constructors and methods for Linear Time-Invariant models, whcih may represent dynamical systems as continuous-time or discrete-time state-space models or transfer functions. With them, you can use standard operator notations such as + or *, array building operators such as `[A,B;C,D]`, connection functions such as `parallel` or `feedback`, and much more.

`lti_filter` `lti_filter` implements functions for the design of analog and digital filters given as lti objects.

`lti_gr` `lti_gr`, loaded automatically with `lti`, defines methods which provide for `lti` objects the same functionality as the native graphical functions of Sysquake for dynamical systems, such as `bodemag` for the magnitude of the Bode diagram or `step` for the step response.

`polyhedra` `polyhedra` implements functions which create solid shapes with polygonal faces in 3D.

`polynom` `polynom` implements constructors and methods for polynomial and rational functions. With them, you can use standard operator notations such as + or *.

`probdist` `probdist` defines classes for probability distributions.

`sigenc` `sigenc` implements functions related to signal encoding to and decoding from a digital representation.

`solids` `solids` implements functions which create solid shapes in 3D. Solids are generated with parametric equations and displayed with `surf`.

`stat` `stat` provides more advanced statistical functions.

`stdlib` `stdlib` is the standard library of general-purpose functions for LME. Functions span from array creation and manipulation to coordinates transform and basic statistics.

`wav` `wav` implements functions for reading and writing WAV files, or encoding and decoding data encoded as WAV in memory.

11.1 stdlib

`stdlib` is a library which extends the native LME functions in the following areas:

- creation of matrices: `blkdiag`, `compan`, `hankel`, `toeplitz`
- geometry: `subspace`
- functions on integers: `primes`
- statistics: `corrcoef`, `perms`
- data processing: `circshift`, `cumtrapz`, `fftshift`, `filter2`, `hist`, `ifftshift`, `polyfit`, `polyvalm`, `trapz`

- other: `isreal`, `sortrows`

The following statement makes available functions defined in `stdlib`:

```
use stdlib
```

Functions

circshift

Shift the elements of a matrix in a circular way.

Syntax

```
use stdlib
B = circshift(A, shift_vert)
B = circshift(A, [shift_vert, shift_hor])
```

Description

`circshift(A,sv)` shifts the rows of matrix `A` downward by `sv` rows. The `sv` bottom rows of the input matrix become the `sv` top rows of the output matrix. `sv` may be negative to go the other way around.

`circshift(A,[sv,sh])` shifts the rows of matrix `A` downward by `sv` rows, and its columns to the right by `sh` columns. The `sv` bottom rows of the input matrix become the `sv` top rows of the output matrix, and the `sh` rightmost columns become the `sh` leftmost columns.

See also

`rot90`, `fliplr`, `flipud`

blkdiag

Block-diagonal matrix.

Syntax

```
use stdlib
X = blkdiag(B1, B2, ...)
```

Description

`blkdiag(B1,B2,...)` creates a block-diagonal matrix with matrix blocks `B1`, `B2`, etc. Its input arguments do not need to be square.

Example

```
use stdlib
blkdiag([1,2;3,4], 5)
  1 2 0
  3 4 0
  0 0 5
blkdiag([1,2], [3;4])
  1 2 0
  0 0 3
  0 0 4
```

See also

`diag`

compan

Companion matrix.

Syntax

```
use stdlib
X = compan(pol)
```

Description

`compan(pol)` gives the companion matrix of polynomial `pol`, a square matrix whose eigenvalues are the roots of `pol`.

Example

```
use stdlib
compan([2,3,4,5])
  -1.5  -2.0  -2.5
   1.0   0.0   0.0
   0.0   1.0   0.0
```

See also

`poly`, `eig`

corrcoef

Correlation coefficients.

Syntax

```
use stdlib
S = corrcoef(X)
S = corrcoef(X1, X2)
```

Description

`corrcoef(X)` calculates the correlation coefficients of the columns of the m-by-n matrix X. The result is a square n-by-n matrix whose diagonal is 1.

`corrcoef(X1,X2)` calculates the correlation coefficients of X1 and X2 and returns a 2-by-2 matrix. It is equivalent to `corrcoef([X1(:),X2(:)])`.

Example

```
use stdlib
corrcoef([1, 3; 2, 5; 4, 4; 7, 10])
  1       0.8915
  0.8915  1
corrcoef(1:5, 5:-1:1)
  1  -1
 -1   1
```

See also

`cov`

cumtrapz

Cumulative numeric integration with trapezoidal approximation.

Syntax

```
use stdlib
S = cumtrapz(Y)
S = cumtrapz(X, Y)
S = cumtrapz(X, Y, dim)
```

Description

`cumtrapz(Y)` calculates an approximation of the cumulative integral of a function given by the samples in Y with unit intervals. The trapezoidal approximation is used. If Y is neither a row nor a column vector, integration is performed along its columns. The result has the same size as Y. The first value(s) is (are) 0.

`cumtrapz(X,Y)` specifies the location of the samples. A third argument may be used to specify along which dimension the integration is performed.

Example

```
use stdlib
cumtrapz([2, 3, 5])
  0     2.5   6.5
cumtrapz([1, 2, 5], [2, 3, 5])
  0     2.5  14.5
```

See also

`cumsum`, `trapz`

fftshift

Shift DC frequency of FFT from beginning to center of spectrum.

Syntax

```
use stdlib
Y = fftshift(X)
```

Description

`fftshift(X)` shifts halves of vector (1-d) or matrix (2-d) X to move the DC component to the center. It should be used after `fft` or `fft2`.

See also

`fft`, `ifftshift`

filter2

Digital 2-d filtering of data.

Syntax

```
use stdlib
Y = filter2(F, X)
Y = filter2(F, X, shape)
```

Description

`filter2(F,X)` filters matrix X with kernel F with a 2-d correlation. The result has the same size as X.

An optional third argument is passed to `conv2` to specify another method to handle the borders.

`filter2` and `conv2` have three differences: arguments F and X are permuted, filtering is performed with a correlation instead of a convolution (i.e. the kernel is rotated by 180 degrees), and the default method for handling the borders is `'same'` instead of `'full'`.

See also

`filter`, `conv2`

hankel

Hankel matrix.

Syntax

```
use stdlib
X = hankel(c, r)
```

Description

hankel(c,r) creates a Hankel matrix whose first column contains the elements of vector c and whose last row contains the elements of vector r. A Hankel matrix is a matrix whose antidiagonals have the same value. In case of conflict, the first element of r is ignored. The default value of r is a zero vector the same length as c.

Example

```
use stdlib
hankel(1:3, 3:8)
  1  2  3  4  5  6
  2  3  4  5  6  7
  3  4  5  6  7  8
```

See also

toeplitz, diag

hist

Histogram.

Syntax

```
use stdlib
(N, X) = hist(Y)
(N, X) = hist(Y, m)
(N, X) = hist(Y, m, dim)
N = hist(Y, X)
N = hist(Y, X, dim)
```

Description

hist(Y) gives the number of elements of vector Y in 10 equally-spaced intervals. A second input argument may be used to specify the number of intervals. The center of the intervals may be obtained in a second output argument.

If Y is an array, histograms are computed along the dimension specified by a third argument or the first non-singleton dimension; the result N has the same size except along that dimension.

When the second argument is a vector, it specifies the centers of the intervals.

Example

```
use stdlib
(N, X) = hist(logspace(0,1), 5)
  N =
    45    21    14    11     9
  X =
     1.9   3.7   5.5   7.3   9.1
```

ifftshift

Shift DC frequency of FFT from center to beginning of spectrum.

Syntax

```
use stdlib
Y = ifftshift(X)
```

Description

`ifftshift(X)` shifts halves of vector (1-d) or matrix (2-d) X to move the DC component from the center. It should be used before `ifft` or `ifft2`. It reverses the effect of `fftshift`.

See also

`ifft`, `fftshift`

isreal

Test for a real number.

Syntax

```
use stdlib
b = isreal(x)
```

Description

`isreal(x)` is true if x is a real scalar or a matrix whose entries are all real.

Examples

```
use stdlib
isreal([2,5])
  true
isreal([2,3+2j])
  false
isreal(exp(pi*1j))
  true
```

See also

`isnumeric`, `isfloat`, `isscalar`

perms

Array of permutations.

Syntax

```
use stdlib
M = perms(v)
```

Description

`perm(v)` gives an array whose rows are all the possible permutations of vector v.

Example

```
use stdlib
perms(1:3)
  3  2  1
  3  1  2
  2  3  1
  1  3  2
  2  1  3
  1  2  3
```

See also

`sort`

polyfit

Polynomial fit.

Syntax

```
use stdlib
pol = polyfit(x, y, n)
```

Description

`polyfit(x,y,n)` calculates the polynomial (given as a vector of descending power coefficients) of order n which best fits the points given by vectors x and y. The least-square algorithm is used.

Example

```
use stdlib
pol = polyfit(1:5, [2, 1, 4, 5, 2], 3)
  pol =
    -0.6667  5.5714 -12.7619  9.8000
polyval(pol, 1:5)
    1.9429  1.2286  3.6571  5.2286  1.9429
```

polyvalm

Value of a polynomial with square matrix argument.

Syntax

```
use stdlib
Y = polyvalm(pol, X)
```

Description

`polyvalm(pol,X)` evaluates the polynomial given by the coefficients `pol` (in descending power order) with a square matrix argument.

Example

```
use stdlib
polyvalm([1,2,8],[2,1;0,1])
  16  5
   0 11
```

See also

`polyval`

primes

List of primes.

Syntax

```
use stdlib
v = primes(n)
```

Description

`primes(n)` gives a row vector which contains the primes up to `n`.

Example

```
use stdlib
primes(20)
  2  3  5  7 11 13 17 19
```

sortrows

Sort matrix rows.

Syntax

```
use stdlib
(S, index) = sortrows(M)
(S, index) = sortrows(M, sel)
(S, index) = sortrows(M, sel, dim)
```

Description

`sortrows(M)` sort the rows of matrix `M`. The sort order is based on the first column of `M`, then on the second one for rows with the same value in the first column, and so on.

`sortrows(M,sel)` use the columns specified in `sel` for comparing the rows of `M`. A third argument `dim` can be used to specify the dimension of the sort: 1 for sorting the rows, or 2 for sorting the columns.

The second output argument of `sortrows` gives the new order of the rows or columns as a vector of indices.

Example

```
use stdlib
sortrows([3, 1, 2; 2, 2, 1; 2, 1, 2])
  2  1  2
  2  2  1
  3  1  2
```

See also

`sort`

subspace

Angle between two subspaces.

Syntax

```
use stdlib
theta = subspace(A, B)
```

Description

`subspace(A,B)` gives the angle between the two subspaces spanned by the columns of `A` and `B`.

Examples

Angle between two vectors in Rˆ2:

```
use stdlib
a = [3; 2];
b = [1; 5];
subspace(a, b)
  0.7854
```

Angle between the vector `[1;1;1]` and the plane spanned by `[2;5;3]` and `[7;1;0]` in Rˆ3:

```
subspace([1;1;1], [2,7;5,1;3,0])
  0.2226
```

toeplitz

Toeplitz matrix.

Syntax

```
use stdlib
X = toeplitz(c, r)
X = toeplitz(c)
```

Description

`toeplitz(c,r)` creates a Toeplitz matrix whose first column contains the elements of vector `c` and whose first row contains the elements of vector `r`. A Toeplitz matrix is a matrix whose diagonals have the same value. In case of conflict, the first element of `r` is ignored. With one argument, `toeplitz` gives a symmetric square matrix.

Example

```
use stdlib
toeplitz(1:3, 1:5)
  1  2  3  4  5
  2  1  2  3  4
  3  2  1  2  3
```

See also

`hankel`, `diag`

trapz

Numeric integration with trapezoidal approximation.

Syntax

```
use stdlib
s = trapz(Y)
s = trapz(X, Y)
s = trapz(X, Y, dim)
```

Description

`trapz(Y)` calculates an approximation of the integral of a function given by the samples in Y with unit intervals. The trapezoidal approximation is used. If Y is an array, integration is performed along the first non-singleton dimension.

`trapz(X,Y)` specifies the location of the samples. A third argument may be used to specify along which dimension the integration is performed.

Example

```
use stdlib
trapz([2, 3, 5])
  6.5
trapz([1, 2, 5], [2, 3, 5])
  14.5
```

See also

`sum`, `cumtrapz`

11.2 stat

`stat` is a library which adds to LME advanced statistical functions.

The following statement makes available functions defined in `stat`:

```
use stat
```

Functions

bootstrp

Bootstrap estimate.

Syntax

```
use stat
(stats, samples) = bootstrp(n, fun, D1, ...)
```

Description

`bootstrp(n,fun,D)` picks random observations from the rows of matrix (or column vector) `D` to form `n` sets which have all the same size as `D`; then it applies function `fun` (a function name or reference or an inline function) to each set and returns the results in the columns of stats. Up to three different set of data can be provided.

`bootstrp` gives an idea of the robustness of the estimate with respect to the choice of the observations.

Example

```
use stat
D = rand(1000, 1);
bootstrp(5, @std, D)
    0.2938
    0.2878
    0.2793
    0.2859
    0.2844
```

geomean

Geometric mean of a set of values.

Syntax

```
use stat
m = geomean(A)
m = geomean(A, dim)
```

Description

`geomean(A)` gives the geometric mean of the columns of array `A` or of the row vector `A`. The dimension along which `geomean` proceeds may be specified with a second argument.

The geometric mean of vector v of length n is defined as $(\prod_i v_i)^{1/n}$.

Example

```
use stat
geomean(1:10)
  4.5287
mean(1:10)
  5.5
exp(mean(log(1:10)))
  4.5287
```

See also

`harmmean`, `mean`

harmmean

Harmonic mean of a set of values.

Syntax

```
use stat
m = harmmean(A)
m = harmmean(A, dim)
```

Description

`harmmean(A)` gives the harmonic mean of the columns of array `A` or of the row vector `A`. The dimension along which `harmmean` proceeds may be specified with a second argument.

The inverse of the harmonic mean is the arithmetic mean of the inverse of the observations.

Example

```
use stat
harmmean(1:10)
  3.4142
mean(1:10)
  5.5
```

See also

`geomean`, `mean`

iqr

Interquartile range.

Syntax

```
use stat
m = iqr(A)
m = iqr(A, dim)
```

Description

`iqr(A)` gives the interquartile range of the columns of array `A` or of the row vector `A`. The dimension along which `iqr` proceeds may be specified with a second argument.

The interquartile range is the difference between the 75th percentile and the 25th percentile.

Example

```
use stat
iqr(rand(1,1000))
  0.5158
```

See also

`trimmean`, `prctile`

mad

Mean absolute deviation.

Syntax

```
use stat
m = mad(A)
m = mad(A, dim)
```

Description

`mad(A)` gives the mean absolute deviation of the columns of array `A` or of the row vector `A`. The dimension along which `mad` proceeds may be specified with a second argument.

The mean absolute deviation is the mean of the absolute value of the deviation between each observation and the arithmetic mean.

Example

```
use stat
mad(rand(1,1000))
  0.2446
```

See also

`trimmean`, `mean`, `iqr`

nancorrcoef

Correlation coefficients after discarding NaNs.

Syntax

```
use stat
S = nancorrcoef(X)
S = nancorrcoef(X1, X2)
```

Description

`nancorrcoef(X)` calculates the correlation coefficients of the columns of the m-by-n matrix X. NaN values are ignored. The result is a square n-by-n matrix whose diagonal is 1.

`nancorrcoef(X1,X2)` calculates the correlation coefficients of X1 and X2 and returns a 2-by-2 matrix, ignoring NaN values. It is equivalent to `nancorrcoef([X1(:),X2(:)])`.

See also

`nanmean`, `nanstd`, `nancov`, `corrcoef`

nancov

Covariance after discarding NaNs.

Syntax

```
use stat
M = nancov(data)
M = nancov(data, 0)
M = nancov(data, 1)
```

Description

`nancov(data)` returns the best unbiased estimate m-by-m covariance matrix of the n-by-m matrix `data` for a normal distribution. NaN values are ignored. Each row of `data` is an observation where n quantities were measured. `nancov(data,0)` is the same as `nancov(data)`.

`nancov(data,1)` returns the m-by-m covariance matrix of the n-by-m matrix `data` which contains the whole population; NaN values are ignored.

See also

`nanmean`, `nanstd`, `nancorrcoef`, `cov`

nanmean

Mean after discarding NaNs.

Syntax

```
use stat
y = nanmean(A)
y = nanmean(A, dim)
```

Description

`nanmean(v)` returns the arithmetic mean of the elements of vector `v`. `nanmean(A)` returns a row vector whose elements are the means of the corresponding columns of array `A`. `nanmean(A,dim)` returns the mean of array `A` along dimension `dim`; the result is a row vector if `dim` is 1, or a column vector if `dim` is 2. In all cases, NaN values are ignored.

Examples

```
use stat
nanmean([1,2,nan;nan,6,7])
  1 4 7
nanmean([1,2,nan;nan,6,7],2)
  1.5
  6.5
nanmean([nan,nan])
  nan
```

See also

`nanmedian`, `nanstd`, `mean`

nanmedian

Median after discarding NaNs.

Syntax

```
use stat
y = nanmedian(A)
y = nanmedian(A, dim)
```

Description

`nanmedian(v)` gives the median of vector `v`, i.e. the value `x` such that half of the elements of `v` are smaller and half of the elements are larger. NaN values are ignored.

`nanmedian(A)` gives a row vector which contains the median of the columns of `A`. With a second argument, `nanmedian(A,dim)` operates along dimension `dim`.

See also

`nanmean`, `median`

nanstd

Standard deviation after discarding NaNs.

Syntax

```
use stat
y = nanstd(A)
y = nanstd(A, p)
y = nanstd(A, p, dim)
```

Description

nanstd(v) returns the standard deviation of vector v with NaN values ignored, normalized by one less than the number of non-NaN values. With a second argument, nanstd(v,p) normalizes by one less than the number of non-NaN values if p is true, or by the number of non-NaN values if p is false.

nanstd(M) gives a row vector which contains the standard deviation of the columns of M. With a third argument, nanstd(M,p,dim) operates along dimension dim. In all cases, NaN values are ignored.

Example

```
use stat
nanstd([1,2,nan;nan,6,7;10,11,12])
  6.3640 4.5092 3.5355
```

See also

nanmedian, nanstd, mean

nansum

Sum after discarding NaNs.

Syntax

```
use stat
y = nansum(A)
y = nansum(A, dim)
```

Description

nansum(v) returns the sum of the elements of vector v. NaN values are ignored. nansum(A) returns a row vector whose elements are the sums of the corresponding columns of array A. nansum(A,dim) returns the sum of array A along dimension dim; the result is a row vector if dim is 1, or a column vector if dim is 2.

See also

nanmean, sum

pdist

Pairwise distance between observations.

Syntax

```
use stat
d = pdist(M)
d = pdist(M, metric)
d = pdist(M, metric, p)
```

Description

`pdist` calculates the distance between pairs of rows of the observation matrix `M`. The result is a column vector which contains the distances between rows `i` and `j` with `i<j`. It can be reshaped into a square matrix with `squareform`.

By default, the metric used to calculate the distance is the euclidean distance; but it can be specified with a second argument:

Metric	Description
`'euclid'`	euclidean distance
`'seuclid'`	standardized euclidean distance
`'mahal'`	Mahalanobis distance
`'cityblock'`	sum of absolute values
`'minkowski'`	Minkowski metric with parameter `p`

The standardized euclidean distance is the euclidean distance after each column of `M` has been divided by its standard deviation. The Minkowski metric is based on the p-norm of vector differences.

Examples

```
use stat
pdist((1:3)')
  1 2 1
squareform(pdist((1:3)'))
  0 1 2
  1 0 1
  2 1 0
squareform(pdist([1,2,6; 3,1,7;6,1,2]))
  0         2.4495    6.4807
  2.4495    0         5.831
  6.4807    5.831     0
```

See also

`squareform`

prctile

Percentile.

Syntax

```
use stat
m = prctile(A, prc)
m = prctile(A, prc, dim)
```

Description

`prctile(A,prc)` gives the smallest values larger than `prc` percent of the elements of each column of array `A` or of the row vector `A`. The dimension along which `prctile` proceeds may be specified with a third argument.

Example

```
prctile(rand(1,1000),90)
  0.8966
```

See also

`trimmean`, `iqr`

range

Data range.

Syntax

```
use stat
m = range(A)
m = range(A, dim)
```

Description

`range(A)` gives the differences between the maximum and minimum values of the columns of array `A` or of the row vector `A`. The dimension along which `range` proceeds may be specified with a second argument.

Example

```
range(rand(1,100))
  0.9602
```

See also

`iqr`

squareform

Reshape a vector of pairwise distances into a square matrix.

Syntax

```
use stat
D = squareform(d)
```

Description

`squareform(d)` reshapes `d`, which should be the output of `pdist`, into a symmetric square matrix `D`, so that the distance between observations `i` and `j` is `D(i,j)`.

See also

`pdist`

trimmean

Trimmed mean of a set of values.

Syntax

```
use stat
m = trimmean(A, prc)
m = trimmean(A, prc, dim)
```

Description

`trimmean(A,prc)` gives the arithmetic mean of the columns of array `A` or of the row vector `A` once `prc/2` percent of the values have been removed from each end. The dimension along which `trimmean` proceeds may be specified with a third argument.

`trimmean` is less sensitive to outliers than the regular arithmetic mean.

See also

`prctile`, `geomean`, `median`, `mean`

zscore

Z score (normalized deviation).

Syntax

```
use stat
Y = zscore(X)
Y = zscore(X, dim)
```

Description

zscore(X) normalizes the columns of array X or the row vector X by subtracting their mean and dividing by their standard deviation. The dimension along which zscore proceeds may be specified with a second argument.

11.3 probdist

probdist is a library which adds to LME classes related to probability distributions. They provide an alternative interface to the algorithms in functions pdf, cdf, icdf and random. In addition, they provide methods to compute their mean, their median, their variance and their standard deviation when an explicit formula is known.

Probability distribution objects, which bundle both the distribution type and parameters, should be created with function makedist.

The following statement makes available classes defined in probdist:

```
use probdist
```

Functions

distribution::cdf

Cumulative distribution function for a distribution.

Syntax

```
s = cdf(pd, x)
```

Description

cdf(pd,x) calculates the integral of a probability density function from -infinity to x. The distribution is specified by the distribution object pd, typically created by makedist.

Example

```
use probdist
pd = makedist('normal', mu=1, sigma=0.5);
x = linspace(-1, 3);
p = pdf(pd, x);
c = cdf(pd, x);
plot(x, p, '-');
plot(x, c);
```

See also

`distribution::pdf`, `distribution::icdf`, `distribution::random`, `makedist`, `cdf`

distribution::icdf

Inverse cumulative distribution function for a distribution.

Syntax

```
x = icdf(pd, p)
```

Description

`icdf(pd,p)` calculates the value of `x` such that `cdf(pd,x)` is `p`. The distribution is specified by the distribution object `pd`, typically created by `makedist`.

`icdf` is defined for distributions `beta`, `chi2`, `gamma`, `lognormal`, `normal`, `student`, and `uniform`.

Example

```
use probdist
pd = makedist('student', nu=3);
p = cdf(pd, 4)
  p =
    0.9860
x = icdf(pd, p)
  x =
    4.0000
```

See also

`distribution::cdf`, `distribution::pdf`, `distribution::random`, `makedist`, `icdf`

makedist

Make a distribution object.

Syntax

```
use probdist
pd = makedist(name, param1=value1, ...)
```

Description

makedist(name) creates a distribution object with the default parameters. Parameters can be specified with named arguments. The result is an object whose class is a subclass of distribution.

Here is a list of distributions with the default parameter values.

Name	Default parameters	Class
'beta'	a=1,b=1	betaDistribution
'chi'	nu=1	chiDistribution
'chi2'	nu=1	chi2Distribution
'exp'	mu=1	exponentialDistribution
'logn'	mu=1,sigma=1	lognormalDistribution
'nakagami'	mu=1,omega=1	nakagamiDistribution
'norm'	mu=0,sigma=1	normalDistribution
'rayl'	b=1	rayleighDistribution
't'	nu=1	studentDistribution
'unif'	Lower=0,Upper=1	uniformDistribution
'weib'	a=1,b=1	weibullDistribution

Example

```
use probstat
pd = makedist('chi2', nu=3)
pd =
  Chi2 distribution
m_th = mean(pd)
m_th =
  3
m_data = mean(random(pd, [1, 10000]))
m_data =
  3.0027
```

distribution::mean

Mean of a distribution.

Syntax

```
m = mean(pd)
```

Description

mean(pd) gives the arithmetic mean of a distribution.

Example

```
use probdist
pd = makedist('normal', mu=3, sigma=2);
mean(pd)
  3
```

See also

distribution::var, distribution::sdev, distribution::median, makedist, mean

distribution::median

Median of a distribution.

Syntax

```
m = median(pd)
```

Description

median(pd) gives the arithmetic median of a distribution, or NaN if it cannot be computed.

Example

```
use probdist
pd = makedist('exp', mu=2);
median(pd)
  3
```

See also

distribution::var, distribution::sdev, distribution::median, makedist, median

distribution::pdf

Probability density function of a distribution.

Syntax

```
s = pdf(pd, x)
```

Description

pdf(pd,x) gives the probability of a distribution. The distribution is specified by the distribution object pd, typically created by makedist.

Example

```
use probdist
pd = makedist('lognormal', mu=2, sigma=1.5);
x = logspace(-2,1);
p = pdf(pd, x);
plot(x, p);
```

See also

distribution::cdf, distribution::icdf, distribution::random, makedist, pdf

distribution::random

Random generator for a distribution.

Syntax

```
x = random(pd)
x = random(pd, size)
```

Description

random(pd) calculates a pseudo-random number whose distribution function is specified by the distribution object pd, typically created by makedist.

Additional input arguments specify the size of the result, either as a vector (or a single scalar for a square matrix) or as scalar values. The result is an array of the specified size where each value is an independent pseudo-random variable. The default size is 1 (scalar).

Example

```
use probdist
pd = makedist('exp');
dataSize = [10, 100];
data = random(pd, dataSize);
```

See also

distribution::pdf, makedist, random

distribution::std

Standard deviation of a distribution.

Syntax

```
s = std(pd)
```

Description

std(pd) gives the standard deviation of a distribution.

Example

```
use probdist
pd = makedist('lognormal', mu=2, sigma=1.5);
std(pd)
  66.3080
std(random(pd,[1,100000]))
  68.0868
```

See also

`distribution::var`, `distribution::mean`, `distribution::median`, `makedist`, `std`

distribution::var

Variance of a distribution.

Syntax

```
s2 = var(pd)
```

Description

`var(pd)` gives the variance of a distribution.

Example

```
use probdist
pd = makedist('uniform', Lower=2, Upper=10);
var(pd)
  5.3333
var(random(pd,[1,100000]))
  5.3148
```

See also

`distribution::mean`, `distribution::sdev`, `distribution::median`, `makedist`, `var`

11.4 polynom

Library `polynom` implements the constructors and methods of two classes: `polynom` for polynomials, and `ratfun` for rational functions. Basic arithmetic operators and functions are overloaded to support expressions with the same syntax as for numbers and matrices.

The following statement makes available functions defined in `polynom`:

```
use polynom
```

Methods for conversion to MathML are defined in library polynom_mathml. Both libraries can be loaded with a single statement:

```
use polynom, polynom_mathml
```

Functions

polynom::polynom

Polynom object constructor.

Syntax

```
use polynom
p = polynom
p = polynom(coef)
```

Description

polynom(coef) creates a polynom object initialized with the coefficients in vector coef, given in descending powers of the variable. Without argument, polynom returns a polynom object initialized to 0.

The following operators and functions may be used with polynom arguments, with results analog to the corresponding functions of LME. Function roots ignores leading zero coefficients.

Op.	Function	Op.	Function
-	minus	+	plus
^	mpower		rem
\	mldivide		roots
/	mrdivide	-	uminus
*	mtimes	+	uplus

Examples

```
use polynom
p = polynom([3,0,1,-4,2])
  p =
    3x^4+x^2-4x+2
q = 3 * p^2 + 8
  q =
    27x^8+18x^6-72x^5+39x^4-24x^3+60x^2-48x+20
```

See also

`polynom::disp`, `polynom::double`, `polynom::subst`, `polynom::diff`, `polynom::int`, `polynom::inline`, `polynom::feval`, `ratfun::ratfun`

polynom::disp

Display a polynom object.

Syntax

```
use polynom
disp(p)
```

Description

`disp(p)` displays polynomial `p`. It is also executed implicitly when LME displays the polynom result of an expression which does not end with a semicolon.

Example

```
use polynom
p = polynom([3,0,1,-4,2])
  p =
    3x^4+x^2-4x+2
```

See also

`polynom::polynom`, `disp`

polynom::double

Convert a polynom object to a vector of coefficients.

Syntax

```
use polynom
coef = double(p)
```

Description

`double(p)` converts polynomial `p` to a row vector of descending-power coefficients.

Example

```
use polynom
p = polynom([3,0,1,-4,2]);
double(p)
  3 0 1 -4 2
```

See also

`polynom::polynom`

polynom::subst

Substitute the variable of a polynom object with another polynomial.

Syntax

```
use polynom
subst(a, b)
```

Description

`subst(a,b)` substitutes the variable of polynom a with polynom b.

Example

```
use polynom
p = polynom([1,2,3])
  p =
    xˆ2+3x+9
q = polynom([2,0])
  q =
    2x
r = subst(p, q)
  r =
    4xˆ2+6x+9
```

See also

`polynom::polynom`, `polynom::feval`

polynom::diff

Polynom derivative.

Syntax

```
use polynom
diff(p)
```

Description

`diff(p)` differentiates polynomial p.

Example

```
use polynom
p = polynom([3,0,1,-4,2]);
q = diff(p)
  q =
    12xˆ3+2x-4
```

See also

`polynom::polynom`, `polynom::int`, `polyder`

polynom::int

Polynom integral.

Syntax

```
use polynom
int(p)
```

Description

`int(p)` integrates polynomial `p`.

Example

```
use polynom
p = polynom([3,0,1,-4,2]);
q = int(p)
  q =
    0.6xˆ5+0.3333xˆ3-2xˆ2+2x
```

See also

`polynom::polynom`, `polynom::diff`, `polyint`

polynom::inline

Conversion from polynom object to inline function.

Syntax

```
use polynom
fun = inline(p)
```

Description

`inline(p)` converts polynomial `p` to an inline function which can then be used with functions such as `feval` and ode45.

Example

```
use polynom
p = polynom([3,0,1,-4,2]);
fun = inline(p)
  fun =
    <inline function>
dumpvar('fun', fun);
  fun = inline('function y=f(x);y=polyval([3,0,1,-4,2],x);');
```

See also

polynom::polynom, polynom::feval, ode45

polynom::feval

Evaluate a polynom object.

Syntax

```
use polynom
y = feval(p, x)
```

Description

feval(p,x) evaluates polynomial p for the value of x. If x is a vector or a matrix, the evaluation is performed separately on each element and the result has the same size as x.

Example

```
use polynom
p = polynom([3,0,1,-4,2]);
y = feval(p, 1:5)
  y =
    2    46   242   770  1882
```

See also

polynom::polynom, polynom::inline, feval

polynom::mathml

Conversion to MathML.

Syntax

```
use polynom, polynom_mathml
str = mathml(p)
str = mathml(p, false)
```

Description

`mathml(p)` converts its argument `p` to MathML presentation, returned as a string.

By default, the MathML top-level element is `<math>`. If the result is to be used as a MathML subelement of a larger equation, a last input argument equal to the logical value `false` can be specified to suppress `<math>`.

Example

```
use polynom, polynom_mathml
p = polynom([3,0,1,-4,2]);
m = mathml(p);
math(0, 0, m);
```

See also

`mathmlpoly`, `mathml`

ratfun::ratfun

Ratfun object constructor.

Syntax

```
use polynom
r = ratfun
r = ratfun(coefnum)
r = ratfun(coefnum, coefden)
```

Description

`ratfun(coefnum,coefden)` creates a ratfun object initialized with the coefficients in vectors `coefnum` and `coefden`, given in descending powers of the variable. Without argument, `ratfun` returns a ratfun object initialized to 0. If omitted, `coefden` defaults to 1.

The following operators and functions may be used with ratfun arguments, with results analog to the corresponding functions of LME.

Op.	Function	Op.	Function
	inv	*	mtimes
-	minus	+	plus
\	mldivide	-	uminus
^	mpower	+	uplus
/	mrdivide		

Example

```
use polynom
r = ratfun([3,0,1,-4,2], [2,5,0,1])
  r =
    (3x^4+x^2-4x+2)/(2x^3+5x^2+1)
```

See also

`ratfun::disp`, `ratfun::inline`, `ratfun::feval`, `polynom::polynom`

ratfun::disp

Display a ratfun object.

Syntax

```
use polynom
disp(r)
```

Description

`disp(r)` displays rational function `r`. It is also executed implicitly when LME displays the ratfun result of an expression which does not end with a semicolon.

See also

`ratfun::ratfun`, `disp`

ratfun::num

Get the numerator of a ratfun as a vector of coefficients.

Syntax

```
use polynom
coef = num(r)
```

Description

`num(r)` gets the numerator of `r` as a row vector of descending-power coefficients.

See also

`ratfun::den`, `ratfun::ratfun`

ratfun::den

Get the denominator of a ratfun as a vector of coefficients.

Syntax

```
use polynom
coef = den(a)
```

Description

`den(a)` gets the denominator of a as a row vector of descending-power coefficients.

See also

`ratfun::num`, `ratfun::ratfun`

ratfun::diff

Ratfun derivative.

Syntax

```
use polynom
diff(r)
```

Description

`diff(r)` differentiates ratfun `r`.

Example

```
use polynom
r = ratfun([1,3,0,1],[2,5]);
q = diff(r)
  q =
    (4x^3+21x^2+30x-2)/(4x^2+20x+25)
```

See also

`ratfun::ratfun`

ratfun::inline

Conversion from ratfun to inline function.

Syntax

```
use polynom
fun = inline(r)
```

Description

inline(r) converts ratfun r to an inline function which can then be used with functions such as feval and ode45.

See also

ratfun::ratfun, ratfun::feval, ode45

ratfun::feval

Evaluate a ratfun object.

Syntax

```
use polynom
y = feval(r, x)
```

Description

feval(r,x) evaluates ratfun r for the value of x. If x is a vector or a matrix, the evaluation is performed separately on each element and the result has the same size as x.

Example

```
use polynom
r = ratfun([1,3,0,1],[2,5]);
y = feval(r, 1:5)
  y =
    0.7143    2.3333    5.0000    8.6923   13.4000
```

See also

ratfun::ratfun, ratfun::inline, feval

ratfun::mathml

Conversion to MathML.

Syntax

```
use polynom, polynom_mathml
str = mathml(r)
str = mathml(r, false)
```

Description

`mathml(r)` converts its argument `r` to MathML presentation, returned as a string.

By default, the MathML top-level element is `<math>`. If the result is to be used as a MathML subelement of a larger equation, a last input argument equal to the logical value `false` can be specified to suppress `<math>`.

Example

```
use polynom, polynom_mathml
r = ratfun([1,3,0,1],[2,5]);
m = mathml(r);
math(0, 0, m);
```

See also

`mathml`

11.5 ratio

Library `ratio` implements the constructors and methods of class `ratio` for rational numbers. It is based on long integers, so that the precision is limited only by available memory. Basic arithmetic operators and functions are overloaded to support expressions with the same syntax as for numbers.

The following statement makes available functions defined in `ratio`:

```
use ratio
```

Functions

ratio::ratio

Ratio object constructor.

Syntax

```
use ratio
r = ratio
r = ratio(n)
r = ratio(num, den)
r = ratio(r)
```

Description

`ratio(num, den)` creates a rational fraction object whose value is `num/den`. Arguments `num` and `den` may be double integer numbers or longint. Common factors are canceled out. With one numeric input argument, `ratio(n)` creates a rational fraction whose denominator is 1. Without input argument, `ratio` creates a rational number whose value is 0.

With one input argument which is already a `ratio` object, `ratio` returns it without change.

The following operators and functions may be used with `ratio` objects, with results analog to the corresponding functions of LME.

Op.	Function	Op.	Function
==	eq	\	mldivide
>=	ge	^	mpower
>	gt	/	mrdivide
	inv	*	mtimes
<=	le	~=	ne
<	lt	+	plus
	max	-	uminus
	min	+	uplus
-	minus		

Examples

```
use ratio
r = ratio(2, 3)
  r =
    2/3
q = 5 * r - 1
  q =
    7/3
```

See also

`ratio::disp`, `ratio::double`, `ratio::char`

ratio::char

Display a ratio object.

Syntax

```
use ratio
char(r)
```

Description

`char(r)` converts ratio `r` to a character string.

See also

`ratio::ratio`, `ratio::disp`, `char`

ratio::disp

Display a ratio object.

Syntax

```
use ratio
disp(r)
```

Description

`disp(r)` displays ratio `r` with the same format as `char`. It is also executed implicitly when LME displays the ratio result of an expression which does not end with a semicolon.

See also

`ratio::ratio`, `ratio::char`, `disp`

ratio::double

Convert a ratio object to a floating-point number.

Syntax

```
use ratio
x = double(r)
```

Description

`double(r)` converts ratio `r` to a floating-point number of class double.

Example

```
use ratio
r = ratio(2, 3);
double(r)
  0.6666
```

See also

`ratio::ratio`

11.6 bitfield

Library `bitfield` implements the constructor and methods of class `bitfield` for bit fields (binary numbers). Basic arithmetic operators and functions are overloaded to support expressions with the same syntax as for numbers and matrices. Contrary to integer numbers, bitfield objects have a length (between 1 and 32) and are displayed in binary.

The following statement makes available functions defined in `bitfield`:

```
use bitfield
```

Functions

bitfield::beginning

First bit position in a bitfield.

Syntax

```
use bitfield
a(...beginning...)
```

Description

In the index expression of a bitfield, `beginning` is the position of the least-significant bit, i.e. 0.

See also

`bitfield::bitfield`, `bitfield::end`

bitfield::bitfield

Bitfield object constructor.

Syntax

```
use bitfield
a = bitfield
a = bitfield(n)
a = bitfield(n, wordlength)
```

Description

`bitfield(n,wordlength)` creates a bitfield object initialized with the `wordlength` least significant bits of the nonnegative integer number `n`. The default value of `wordlength` is 32 if `n` is a double, an int32 or a uint32 number; 16 is `n` is an int16 or uint16 number; or 8 if `n` is an int8 or uint8 number. Without argument, `bitfield` gives a bit field of 32 bits 0. Like any integer number in LME, `n` may be written in base 2, 8, 10, or 16: `0b1100`, `014`, 12, and `0xc` all represent the same number.

The following operators and functions may be used with bitfield arguments, with results analog to the corresponding functions of LME. Logical functions operate bitwise.

Op.	Function	Op.	Function
`&`	`and`	`~`	`not`
`==`	`eq`	`|`	`or`
`-`	`minus`	`+`	`plus`
`\`	`mldivide`	`-`	`uminus`
`/`	`mrdivide`	`+`	`uplus`
`*`	`mtimes`		`xor`
`~=`	`ne`		

Indexes into bit fields are non-negative integers: 0 represents the least-significant bit, and wordlength-1 the most-significant bit. Unlike arrays, bits are not selected with logical arrays, but with other bit fields where ones represent the bits to be selected; for example `a(0b1011)` selects bits 0, 1 and 3. This is consistent with the way `bitfield::find` is defined.

Examples

```
use bitfield
a = bitfield(123, 16)
  a =
    0b0000000001111011
b = ~a
  b =
    0b1111111110000100
b = a * 5
  b =
    0b0000001001100111
```

See also

`bitfield::disp`, `bitfield::double`

bitfield::disp

Display a bitfield object.

Syntax

```
use bitfield
disp(a)
```

Description

`disp(a)` displays bitfield a. It is also executed implicitly when LME displays the bitfield result of an expression which does not end with a semicolon.

See also

`bitfield::bitfield`, `disp`

bitfield::double

Convert a bitfield object to a double number.

Syntax

```
use bitfield
n = double(a)
```

Description

`double(a)` converts bitfield a to double number.

Example

```
use bitfield
a = bitfield(123, 16);
double(a)
  123
```

See also

`bitfield::bitfield`

bitfield::end

Last bit position in a bitfield.

Syntax

```
use bitfield
a(...end...)
```

Description

In the index expression of a bitfield, end is the position of the most-significant bit, i.e. 1 less than the word length.

See also

`bitfield::bitfield`, `bitfield::beginning`

bitfield::find

Find the ones in a bitfield.

Syntax

```
use bitfield
ix = find(a)
```

Description

`find(a)` finds the bits equal to 1 in bitfield a. The result is a vector of bit positions in ascending order; the least-significant bit is number 0.

Example

```
use bitfield
a = bitfield(123, 16)
  a =
    0b0000000001111011
ix = find(a)
  ix =
     0 1 3 4 5 6
```

See also

`bitfield::bitfield`, `find`

bitfield::int8 bitfield::int16 bitfield::int32

Convert a bitfield object to a signed integer number, with sign extension.

Syntax

```
use bitfield
n = int8(a)
n = int16(a)
n = int32(a)
```

Description

`int8(a)`, `int16(a)`, and `int32(a)` convert bitfield a to an int8, int16, or int32 number respectively. If a has less bits than the target integer and the most significant bit of a is 1, sign extension is performed; i.e. the most significant bits of the result are set to 1, so that it is negative. If a has more bits than the target integer, most significant bits are ignored.

Example

```
use bitfield
a = bitfield(9, 4);
  a =
    0x1001
i = int8(a)
  i =
    210
b = bitfield(i)
  b =
    0b11111001
```

See also

`uint8`, `uint16`, `uint32`, `bitfield::int8`, `bitfield::int16`, `bitfield::int32`, `bitfield::double`, `bitfield::bitfield`

bitfield::length

Word length of a bitfield.

Syntax

```
use bitfield
wordlength = length(a)
```

Description

`length(a)` gives the number of bits of bitfield a.

Example

```
use bitfield
a = bitfield(123, 16);
length(a)
  16
```

See also

`bitfield::bitfield`, `length`

bitfield::sign

Get the sign of a bitfield.

Syntax

```
use bitfield
s = sign(a)
```

Description

`sign(a)` gets the sign of bitfield a. The result is -1 if the most-significant bit of a is 1, 0 if all bits of a are 0, or 1 otherwise.

Example

```
use bitfield
a = bitfield(5, 3)
  a =
    0b101
sign(a)
  -1
```

See also

`bitfield::bitfield`, `sign`

bitfield::uint8 bitfield::uint16 bitfield::uint32

Convert a bitfield object to an unsigned integer number.

Syntax

```
use bitfield
n = uint8(a)
n = uint16(a)
n = uint32(a)
```

Description

`uint8(a)`, `uint16(a)`, and `uint32(a)` convert bitfield a to a uint8, uint16, or uint32 number respectively. If a has more bits than the target integer, most significant bits are ignored.

Example

```
use bitfield
a = bitfield(1234, 16);
uint8(a)
  210
```

See also

`uint8`, `uint16`, `uint32`, `bitfield::int8`, `bitfield::int16`, `bitfield::int32`, `bitfield::double`, `bitfield::bitfield`

11.7 filter

`filter` is a library which adds to LME functions for designing analog (continuous-time) and digital (discrete-time) linear filters.

The following statement makes available functions defined in `filter`:

```
use filter
```

This library provides three kinds of functions:

- `besselap`, `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`, which compute the zeros, poles and gain of the prototype of analog low-pass filter with a cutoff frequency of 1 rad/s. They correspond respectively to Bessel, Butterworth, Chebyshev type 1, Chebyshev type 2, and elliptic filters.

- `besself`, `butter`, `cheby1`, `cheby2`, and `ellip`, which provide a higher-level interface to design filters of these different types. In addition to the filter parameters (degree, bandpass and bandstop ripples), one can specify the kind of filter (lowpass, highpass, bandpass or bandstop) and the cutoff frequency or frequencies. The result can be an analog or a digital filter, given as a rational transfer function or as zeros, poles and gain.

- `lp2lp`, `lp2hp`, `lp2bp`, and `lp2bs`, which convert analog lowpass filters respectively to lowpass, highpass, bandpass, and bandstop with specified cutoff frequency or frequencies.

Transfer functions are expressed as the coefficient vectors of their numerator `num` and denominator `den` in decreasing powers of `s` (Laplace transform for analog filters) or `z` (z transform for digital filters); or as the zeros `z`, poles `p`, and gain `k`.

Functions

besselap

Bessel analog filter prototype.

Syntax

```
use filter
(z, p, k) = besselap(n)
```

Description

`besselap(n)` calculates the zeros, the poles, and the gain of a Bessel analog filter of degree `n` with a cutoff angular frequency of 1 rad/s.

See also

`besself`, `buttap`, `cheb1ap`, `cheb2ap`, `ellipap`

besself

Bessel filter.

Syntax

```
use filter
(z, p, k) = besself(n, w0)
(num, den) = besself(n, w0)
(...) = besself(n, [wl, wh])
(...) = besself(n, w0, 'high')
(...) = besself(n, [wl, wh], 'stop')
(...) = besself(..., 's')
```

Description

`besself` calculates a Bessel filter. The result is given as zeros, poles and gain if there are three output arguments, or as numerator and denominator coefficient vectors if there are two output arguments.

`besself(n,w0)`, where `w0` is a scalar, gives a digital lowpass filter of order `n` with a cutoff frequency of `w0` relatively to half the sampling frequency.

`besself(n,[wl,wh])`, where the second input argument is a vector of two numbers, gives a digital bandpass filter of order `2*n` with passband between `wl` and `wh` relatively to half the sampling frequency.

`besself(n,w0,'high')` gives a digital highpass filter of order `n` with a cutoff frequency of `w0` relatively to half the sampling frequency.

`besself(n,[wl,wh],'stop')`, where the second input argument is a vector of two numbers, gives a digital bandstop filter of order `2*n` with stopband between `wl` and `wh` relatively to half the sampling frequency.

With an additional input argument which is the string `'s'`, `besself` gives an analog Bessel filter. Frequencies are given in rad/s.

See also

`besselap`, `butter`, `cheby1`, `cheby2`, `ellip`

bilinear

Analog-to-digital conversion with bilinear transformation.

Syntax

```
use filter
(zd, pd, kd) = bilinear(zc, pc, kc, fs)
(numd, dend) = bilinear(numc, denc, fs)
```

Description

`bilinear(zc,pc,kc,fs)` converts the analog (continuous-time) transfer function given by its zeros `zc`, poles `pc`, and gain `kc` to a digital (discrete-time) transfer function given by its zeros, poles, and gain in the domain of the forward-shift operator q. The sampling frequency is `fs`. Conversion is performed with the bilinear transormation $z_d = (1 + z_c/2f_s)/(1 - z_c/2f_s)$. If the analog transfer function has less zeros than poles, additional digital zeros are added at -1 to avoid a delay.

With three input arguments, `bilinear(numc,denc,fs)` uses the coefficients of the numerators and denominators instead of their zeros, poles and gain.

buttap

Butterworth analog filter prototype.

Syntax

```
use filter
(z, p, k) = buttap(n)
```

Description

`buttap(n)` calculates the zeros, the poles, and the gain of a Butterworth analog filter of degree `n` with a cutoff angular frequency of 1 rad/s.

See also

`butter`, `besselap`, `cheb1ap`, `cheb2ap`, `ellipap`

butter

Butterworth filter.

Syntax

```
use filter
(z, p, k) = butter(n, w0)
(num, den) = butter(n, w0)
(...) = butter(n, [wl, wh])
(...) = butter(n, w0, 'high')
(...) = butter(n, [wl, wh], 'stop')
(...) = butter(..., 's')
```

Description

`butter` calculates a Butterworth filter. The result is given as zeros, poles and gain if there are three output arguments, or as numerator and denominator coefficient vectors if there are two output arguments.

`butter(n,w0)`, where `w0` is a scalar, gives a nth-order digital lowpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency.

`butter(n,[wl,wh])`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandpass filter with passband between `wl` and `wh` relatively to half the sampling frequency.

`butter(n,w0,'high')` gives a nth-order digital highpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency.

`butter(n,[wl,wh],'stop')`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandstop filter with stopband between `wl` and `wh` relatively to half the sampling frequency.

With an additional input argument which is the string `'s'`, `butter` gives an analog Butterworth filter. Frequencies are given in rad/s.

See also

`buttap`, `besself`, `cheby1`, `cheby2`, `ellip`

cheb1ap

Chebyshev type 1 analog filter prototype.

Syntax

```
use filter
(z, p, k) = cheb1ap(n, rp)
```

Description

`cheb1ap(n,rp)` calculates the zeros, the poles, and the gain of a Chebyshev type 1 analog filter of degree `n` with a cutoff angular frequency of 1 rad/s. Ripples in the passband have a peak-to-peak magnitude of `rp` dB, i.e. the peak-to-peak ratio is `10^(rp/20)`.

See also

`cheby1`, `cheb2ap`, `ellipap`, `besselap`, `buttap`

cheb2ap

Chebyshev type 2 analog filter prototype.

Syntax

```
use filter
(z, p, k) = cheb2ap(n, rs)
```

Description

`cheb2ap(n,rs)` calculates the zeros, the poles, and the gain of a Chebyshev type 2 analog filter of degree `n` with a cutoff angular frequency of 1 rad/s. Ripples in the stopband have a peak-to-peak magnitude of `rs` dB, i.e. the peak-to-peak ratio is `10^(rs/20)`.

See also

`cheby1`, `cheb1ap`, `ellipap`, `besselap`, `buttap`

cheby1

Chebyshev type 1 filter.

Syntax

```
use filter
(z, p, k) = cheby1(n, rp, w0)
(num, den) = cheby1(n, rp, w0)
(...) = cheby1(n, rp, [wl, wh])
(...) = cheby1(n, rp, w0, 'high')
(...) = cheby1(n, rp, [wl, wh], 'stop')
(...) = cheby1(..., 's')
```

Description

cheby1 calculates a Chebyshev type 1 filter. The result is given as zeros, poles and gain if there are three output arguments, or as numerator and denominator coefficient vectors if there are two output arguments.

`cheby1(n,rp,w0)`, where `w0` is a scalar, gives a nth-order digital lowpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency. Ripples in the passband have a peak-to-peak magnitude of `rp` dB, i.e. the peak-to-peak ratio is `10^(rp/20)`.

`cheby1(n,rp,[wl,wh])`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandpass filter with passband between `wl` and `wh` relatively to half the sampling frequency.

`cheby1(n,rp,w0,'high')` gives a nth-order digital highpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency.

`cheby1(n,rp,[wl,wh],'stop')`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandstop filter with stopband between `wl` and `wh` relatively to half the sampling frequency.

With an additional input argument which is the string `'s'`, cheby1 gives an analog Chebyshev type 1 filter. Frequencies are given in rad/s.

See also

`cheb1ap`, `besself`, `butter`, `cheby2`, `ellip`

cheby2

Chebyshev type 2 filter.

Syntax

```
use filter
(z, p, k) = cheby2(n, rs, w0)
(num, den) = cheby2(n, rs, w0)
(...) = cheby2(n, rs, [wl, wh])
(...) = cheby2(n, rs, w0, 'high')
(...) = cheby2(n, rs, [wl, wh], 'stop')
(...) = cheby2(..., 's')
```

Description

cheby2 calculates a Chebyshev type 2 filter. The result is given as zeros, poles and gain if there are three output arguments, or as numerator and denominator coefficient vectors if there are two output arguments.

`cheby2(n,rs,w0)`, where `w0` is a scalar, gives a nth-order digital lowpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency. Ripples in the stopband have a peak-to-peak magnitude of `rs` dB, i.e. the peak-to-peak ratio is `10^(rs/20)`.

`cheby2(n,rs,[wl,wh])`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandpass filter with passband between `wl` and `wh` relatively to half the sampling frequency.

`cheby2(n,rs,w0,'high')` gives a nth-order digital highpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency.

`cheby2(n,rs,[wl,wh],'stop')`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandstop filter with stopband between `wl` and `wh` relatively to half the sampling frequency.

With an additional input argument which is the string `'s'`, cheby2 gives an analog Chebyshev type 2 filter. Frequencies are given in rad/s.

See also

`cheb2ap`, `besself`, `butter`, `cheby1`, `ellip`

ellip

Elliptic filter.

Syntax

```
use filter
(z, p, k) = ellip(n, rp, rs, w0)
(num, den) = ellip(n, rp, rs, w0)
(...) = ellip(n, rp, rs, [wl, wh])
(...) = ellip(n, rp, rs, w0, 'high')
(...) = ellip(n, rp, rs, [wl, wh], 'stop')
(...) = ellip(..., 's')
```

Description

`ellip` calculates a elliptic filter, or Cauer filter. The result is given as zeros, poles and gain if there are three output arguments, or as numerator and denominator coefficient vectors if there are two output arguments.

`ellip(n,rp,rs,w0)`, where `w0` is a scalar, gives a nth-order digital lowpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency. Ripples have a peak-to-peak magnitude of `rp` dB in the passband and of `rs` dB in the stopband (peak-to-peak ratios are respectively `10^(rp/20)` and `10^(rs/20)`).

`ellip(n,rp,rs,[wl,wh])`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandpass filter with passband between `wl` and `wh` relatively to half the sampling frequency.

`ellip(n,rp,rs,w0,'high')` gives a nth-order digital highpass filter with a cutoff frequency of `w0` relatively to half the sampling frequency.

`ellip(n,rp,rs,[wl,wh],'stop')`, where the second input argument is a vector of two numbers, gives a 2nth-order digital bandstop filter with stopband between `wl` and `wh` relatively to half the sampling frequency.

With an additional input argument which is the string `'s'`, `ellip` gives an analog elliptic filter. Frequencies are given in rad/s.

See also

`ellipap`, `besself`, `butter`, `cheby1`, `cheby2`

ellipap

Elliptic analog filter prototype.

Syntax

```
use filter
(z, p, k) = ellipap(n, rp, rs)
```

Description

`ellipap(n,rp,rs)` calculates the zeros, the poles, and the gain of an elliptic analog filter of degree `n` with a cutoff angular frequency of 1 rad/s. Ripples have a peak-to-peak magnitude of `rp` dB in the passband and of `rs` dB in the stopband (peak-to-peak ratios are respectively `10^(rp/20)` and `10^(rs/20)`).

See also

`ellip`, `cheb1ap`, `cheb1ap`, `besselap`, `buttap`

lp2bp

Lowpass prototype to bandpass filter conversion.

Syntax

```
use filter
(z, p, k) = lp2bp(z0, p0, k0, wc, ww)
(num, den) = lp2bp(num0, den0, wc, ww)
```

Description

`lp2bp` convert a lowpass analog filter prototype (with unit angular frequency) to a bandpass analog filter with the specified center angular frequency `w0` and bandwidth `ww`. `lp2bp(z0,p0,k0,wc,ww)` converts a filter given by its zeros, poles, and gain; `lp2bp(num0,den0,wc,ww)` converts a filter given by its numerator and denominator coefficients in decreasing powers of s.

The new filter $F(s)$ is

$$F(s) = F_0\left(\frac{s^2 + \omega_c^2 - \omega_w^2/4}{\omega_w s}\right)$$

where $F_0(s)$ is the filter prototype. The filter order is doubled.

See also

`lp2lp`, `lp2hp`, `lp2bs`

lp2bs

Lowpass prototype to bandstop filter conversion.

Syntax

```
use filter
(z, p, k) = lp2bs(z0, p0, k0, wc, ww)
(num, den) = lp2bs(num0, den0, wc, ww)
```

Description

`lp2bs` convert a lowpass analog filter prototype (with unit angular frequency) to a bandstop analog filter with the specified center angular frequency `w0` and bandwidth `ww`. `lp2bs(z0,p0,k0,wc,ww)` converts a filter given by its zeros, poles, and gain; `lp2bs(num0,den0,wc,ww)` converts a filter given by its numerator and denominator coefficients in decreasing powers of s.

The new filter $F(s)$ is

$$F(s) = F_0\left(\frac{\omega_w s}{s^2 + \omega_c^2 - \omega_w^2/4}\right)$$

where $F_0(s)$ is the filter prototype. The filter order is doubled.

See also

`lp2lp`, `lp2hp`, `lp2bp`

lp2hp

Lowpass prototype to highpass filter conversion.

Syntax

```
use filter
(z, p, k) = lp2hp(z0, p0, k0, w0)
(num, den) = lp2hp(num0, den0, w0)
```

Description

`lp2hp` convert a lowpass analog filter prototype (with unit angular frequency) to a highpass analog filter with the specified cutoff angular frequency `w0`. `lp2hp(z0,p0,k0,w0)` converts a filter given by its zeros, poles, and gain; `lp2hp(num0,den0,w0)` converts a filter given by its numerator and denominator coefficients in decreasing powers of s.

The new filter $F(s)$ is

$$F(s) = F_0(\frac{1}{\omega_0 s})$$

where $F_0(s)$ is the filter prototype.

See also

`lp2lp`, `lp2bp`, `lp2bs`

lp2lp

Lowpass prototype to lowpass filter conversion.

Syntax

```
use filter
(z, p, k) = lp2lp(z0, p0, k0, w0)
(num, den) = lp2lp(num0, den0, w0)
```

Description

`lp2lp` convert a lowpass analog filter prototype (with unit angular frequency) to a lowpass analog filter with the specified cutoff angular frequency `w0`. `lp2lp(z0,p0,k0,w0)` converts a filter given by its zeros, poles, and gain; `lp2lp(num0,den0,w0)` converts a filter given by its numerator and denominator coefficients in decreasing powers of s.

The new filter $F(s)$ is

$$F(s) = F_0\left(\frac{s}{\omega_0}\right)$$

where $F_0(s)$ is the filter prototype.

See also
`lp2hp`, `lp2bp`, `lp2bs`

11.8 lti

Library `lti` defines methods related to objects which represent linear time-invariant dynamical systems. LTI systems may be used to model many different systems: electro-mechanical devices, robots, chemical processes, filters, etc. LTI systems map one or more inputs u to one or more outputs y. The mapping is defined as a state-space model or as a matrix of transfer functions, either in continuous time or in discrete time. Methods are provided to create, combine, and analyze LTI objects.

Graphical methods are based on the corresponding graphical functions; the numerator and denominator coefficient vectors or the state-space matrices are replaced with an LTI object. They accept the same optional arguments, such as a character string for the style.

The following statement makes available functions defined in `lti`:

```
use lti
```

Methods for conversion to MathML are defined in library `lti_mathml`. Both libraries can be loaded with a single statement:

```
use lti, lti_mathml
```

Class overview

The LTI library defines six classes. The three central ones correspond to the main model structures used for linear time-invariant systems in automatic control: `ss` for state-space models, `tf` for rational transfer functions given by the coefficients of the numerator and denominator polynomials, and `zpk` for rational transfer functions given by their zeros, poles and gain. State-space representation is restricted to causal systems, while transfer functions can be non-causal. Three additional classes are more specialized: `frd` (frequency response data) for systems described by a discrete set of frequency/complex response pairs, and `pid` or `pidstd` for PID controllers.

LTI classes share many properties and methods. They can represent systems with single or multiple inputs and/or outputs. Inputs, outputs and internal states are continuous in time (*continuous-time systems*) or defined at a fixed sampling frequency (*discrete-time systems*).

The variable of the Laplace transform can be `'s'` or `'p'`. The variable of the z transform can be `'z'` or `'q'`. By multiplying the numerator and the denominator of a rational transfer function by a suitable

power of q^-1 (or z^-1), polynomials in q^-1 can be obtained, where q^-1 is the delay operator; this yields directly a recurrence relation.

Conversion

Conversion between `ss`, `tf` and `zpk` can be done simply by calling the target constructor. The only restriction is that systems to be converted to state-space models must be causal. For instance, a transfer function given by its zeros, poles and gain can be converted to a state-space model as follows:

```
use lti;
P = zpk([1], [-3+1j, -3-1j], 2)
  P =
    continuous-time zero-pole-gain transfer function
    2(s-1)/(s-(-3+1j))(s-(-3-1j))
S = ss(P)
  S =
    continuous-time LTI state-space system
    A =
   -6  -10
    1    0
    B =
    1
    0
    C =
    2   -2
    D =
    0
```

Conversion from `pid` or `pidstd` objects is performed the same way. Conversion to `pid` or `pidstd` objects is possible only if the system to be converted has the structure of a P, PI, PD, or PID controller, with or without filter on the derivative term.

Conversion to an `frd` object requires an array of frequency points where the frequency response is evaluated. Conversion of `frd` objects to other LTI objects is not possible.

Conversion between continuous-time and discrete-time objects of the same class is performed with `c2d` and `d2c`.

Building large systems

Simple systems can be combined to create larger ones. All systems can be seen as matrices mapping inputs to outputs via a matrix product. Larger systems can be created by matrix concatenation, addition or multiplication. More specialized connections can be obtained with methods `connect` and `feedback`.

Mixing objects of different classes is possible for all classes except for `frd` (where a frequency array must be provided explicitly, which can only be done with a call of the `frd` constructor). Continuous-time objects cannot be connected with discrete-time objects, and discrete-time objects must have the same sampling period.

Functions

frd::frd

LTI frequency response data constructor.

Syntax

```
use lti
a = frd
a = frd(resp, freq)
a = frd(resp, freq, Ts)
```

Description

`frd(response,frequency,Ts)` creates an LTI object which represents a discrete set of frequency response data. Argument `response` is an array of complex frequency responses corresping to frequency array `freq`.

A single-input single-output (SISO) PID controller has scalar parameters. If the parameters are matrices, they must all have the same size (scalar values are replicated as required), and the resulting controller has as many inputs as parameters have columns and as many outputs as parameters have rows; mapping from each input to each output is and independent SISO PID controller.

Examples

Simple continuous-time `frd` object:

```
use lti
freq = 0:100;
resp = 3 ./ (1 + 0.1 * freq * 1j) + 0.1 * randn(size(freq));
r = frd(resp, freq)
  r =
    continuous-time frequency response, units=rad/s
    1 input, 1 output
    101 frequencies
```

Conversion from a transfer function object:

```
freq = 0:100;
G = tf(1, [1, 2, 3, 4]);
r = frd(G, freq)
  r =
    continuous-time frequency response, units=rad/s
    1 input, 1 output
    101 frequencies
```

See also

`frd::frdata`

pid::pid

LTI PID controller constructor.

Syntax

```
use lti
a = pid
a = pid(Kp, Ki, Kd, Tf)
a = pid(Kp, Ki, Kd, Tf, Ts)
a = pid(Kp, Ki, Kd, Tf, Ts, var)
a = pid(..., IFormula=f1, DFormula=f2)
```

Description

`pid(Kp, Ki, Kd, Tf)` creates an LTI object which represents the continuous-time PID controller $K_p + K_i/s + K_d s/(T_f s + 1)$, where s is the variable of the Laplace transform. `Kp` is the proportional gain, `Ki` is the integral gain, `Kd` is the derivative gain, and `Tf` is the time constant of the first-order filter of the derivative term. Missing `Ki`, `Kd` or `Tf` default to 0; without any input argument, `Kp` defaults to 1. If `Tf=0` and `Kd!=0`, the derivative term is not filtered and the controller is not causal.

A single-input single-output (SISO) PID controller has scalar parameters. If the parameters are matrices, they must all have the same size (scalar values are replicated as required), and the resulting controller has as many inputs as parameters have columns and as many outputs as parameters have rows; mapping from each input to each output is an independent SISO PID controller.

`pid(Kp, Ki, Kd, Tf, Ts)` creates an LTI object which represents the discrete-time PID controller $K_p + K_i I_i(z) + K_d/(T_f + I_d(z))$, where $I_i(z)$ is the integration formula used for the integral term, $I_d(z)$ is the integration formula used for the derivative term, and z is the variable of the z transform. The formulae can be specified by named arguments `IFormula` and `DFormula`, strings with the following values:

Name	**Value**
`'ForwardEuler'`	$T_s/(z-1)$
`'BackwardEuler'`	$T_s z/(z-1)$
`'Trapezoidal'`	$T_s/2\,(z+1)/(z-1)$

The default formula for both the integral and the derivative terms is `'ForwardEuler'`.

An additional argument `var` may be used to specify the variable of the Laplace (`'s'` (default) or `'p'`) or z transform (`'z'` (default) or `'q'` for forward time shift, `'z^-1'` or `'q^-1'` for backward time shift).

For PID controllers based on the standard parameters `Kp`, `Ti` and `Td`, where `Ki=Kp/Ti` and `Kd=Kp*Td`, `pidstd` objects should be used instead.

Examples

Simple continuous-time PID controller:

```
use lti
C = pid(5,2,1)
  C =
    continuous-time PID controller
    Kp + Ki/s + Kd s/(Tf s + 1)
    Kp = 5  Ki = 2  Kd = 1  Tf = 0
```

Discrete-time PD controller where the derivative term, filtered with a time constant of 20ms, is approximated with the Backward Euler formula, with a sampling period of 1ms. The controller is displayed with the backward-shift operator `q^-1`.

```
C = pid(5,0,1,20e-3,1e-3,'q^-1',DFormula='BackwardEuler')
  C =
    discrete-time PD controller, Ts=1e-3
    Kp + Kd/(Tf + Id(q^-1))
    Id(q^-1) = Ts/(1-q^-1) (BackwardEuler)
    Kp = 5  Kd = 1  Tf = 2e-2
```

Conversion of a first-order continuous-time transfer function with pole at 0 (integrator effect) to a continuous-time PI controller:

```
G = tf([1, 2], [1, 0])
  G =
    continuous-time transfer function
    (s+2)/s
C = pid(G)
  C =
    continuous-time PI controller
    Kp + Ki/s
    Kp = 1  Ki = 2
```

Conversion of a discrete-time PID controller with the Backward Euler formula for the integral term and the Trapezoidal formula for the derivative term to a transfer function, and back to a PID controller:

```
C1 = pid(5, 2, 3, 0.1, 0.01,
         IFormula='BackwardEuler', DFormula='Trapezoidal')
  C1 =
    discrete-time PID controller, Ts=1e-2
    Kp + Ki Ii(z) + Kd/(Tf + Id(z))
    Ii(z) = Ts z/(z-1) (BackwardEuler)
    Id(z) = Ts/2 (z+1)/(z-1) (Trapezoidal)
 Kp = 5  Ki = 2  Kd = 3  Tf = 0.1
G = tf(C1)
  G =
    discrete-time transfer function, Ts=1e-2
    (3.5271z^2-7.0019z+3.475)/(0.105z^2-0.2z+9.5e-2)
C2 = pid(G, IFormula='BackwardEuler', DFormula='Trapezoidal')
  C2 =
    discrete-time PID controller, Ts=1e-2
    Kp + Ki Ii(z) + Kd/(Tf + Id(z))
    Ii(z) = Ts z/(z-1) (BackwardEuler)
    Id(z) = Ts/2 (z+1)/(z-1) (Trapezoidal)
    Kp = 5  Ki = 2  Kd = 3  Tf = 10e-2
```

See also

`pidstd::pidstd`, `tf::tf`

pidstd::pidstd

LTI standard PID controller constructor.

Syntax

```
use lti
a = pidstd
a = pidstd(Kp, Ti, Td, N)
a = pidstd(Kp, Ti, Td, N, Ts)
a = pidstd(Kp, Ti, Td, N, Ts, var)
a = pidstd(..., IFormula=f1, DFormula=f2)
```

Description

`pidstd(Kp,Ti,Td,N)` creates an LTI object which represents the standard continuous-time PID controller $K_p(1/T_i s + T_d s/(T_d s/N + 1)$, where s is the variable of the Laplace transform. `Kp` is the proportional gain, `Ti` is the integral time, `Td` is the derivative time, and `N` is the relative frequency of the first-order filter of the derivative term. Missing `Ti` defaults to infinity (no integral term), missing `Td` to zero (no derivative

term), and missing `N` to infinity (no filter on the derivative term, which means that the controller is noncausal if `Td` is nonzero).

A single-input single-output (SISO) PID controller has scalar parameters. If the parameters are matrices, they must all have the same size (scalar values are replicated as required), and the resulting controller has as many inputs as parameters have columns and as many outputs as parameters have rows; mapping from each input to each output is and independent SISO PID controller.

`pid(Kp,Ti,Td,N,Ts)` creates an LTI object which represents the standard discrete-time PID controller $K_p(I_i(z)/T_i + T_d/(T_d/N + I_d(z)))$, where $I_i(z)$ is the integration formula used for the integral term, $I_d(z)$ is the integration formula used for the derivative term, and z is the variable of the z transform. The formulae can be specified by named arguments `IFormula` and `DFormula`, strings with the following values:

Name	**Value**
`'ForwardEuler'`	$T_s/(z-1)$
`'BackwardEuler'`	$T_s z/(z-1)$
`'Trapezoidal'`	$T_s/2\ (z+1)/(z-1)$

The default formula for both the integral and the derivative terms is `'ForwardEuler'`.

An additional argument `var` may be used to specify the variable of the Laplace (`'s'` (default) or `'p'`) or z transform (`'z'` (default) or `'q'` for forward time shift, `'z^-1'` or `'q^-1'` for backward time shift).

For PID controllers based on the gain parameters `Kp`, `Ki=Kp/Ti`, `Kd=Kp*Td`, and `Tf=Td/N`, `pid` objects should be used instead. Class `pidstd` is a subclass of `pid`. The only differences are the arguments of their constructors and the way their objects are displayed by `char`, `disp` and `mathml`.

Examples

Simple standard continuous-time PID controller:

```
use lti
C = pidstd(5,4,1)
  C =
    continuous-time PID controller
    Kp (1 + 1/(Ti s) + Td s/(Td/N s + 1))
    Kp = 5  Ti = 4  Td = 1  N = inf
```

Conversion to a `pid` object:

```
C1 = pid(C)
  C1 =
    continuous-time PID controller
    Kp + Ki/s + Kd s/(Tf s + 1)
    Kp = 5  Ki = 1.25  Kd = 5  Tf = 0
```

Standard discrete-time PD controller where the derivative term, filtered with a time constant 20 times smaller than the derivator time, is approximated with the Backward Euler formula, with a sampling period of 1ms. The controller is displayed with the backward-shift operator q^-1.

```
C = pidstd(5,0,1,20,1e-3,'q^-1',DFormula='BackwardEuler')
  C =
    discrete-time PID controller, Ts=1e-3
    Kp (1 + Ii(q^-1)/Ti + Td/(Td/N + Id(q^-1)))
    Ii(q^-1) = Ts q^-1/(1-q^-1) (ForwardEuler)
    Id(q^-1) = Ts/(1-q^-1) (BackwardEuler)
    Kp = 5  Ti = 0  Td = 1  N = 20
```

See also

`pid::pid`, `tf::tf`

ss::ss

LTI state-space constructor.

Syntax

```
use lti
a = ss
a = ss(A, B, C, D)
a = ss(A, B, C, D, Ts)
a = ss(A, B, C, D, Ts, var)
a = ss(A, B, C, D, b)
a = ss(b)
```

Description

`ss(A,B,C,D)` creates an LTI object which represents the continuous-time state-space model

```
x'(t) = A x(t) + B u(t)
y(t)  = C x(t) + D u(t)
```

`ss(A,B,C,D,Ts)` creates an LTI object which represents the discrete-time state-space model with sampling period Ts

```
x(k+1) = A x(k) + B u(k)
y(k)   = C x(k) + D u(k)
```

In both cases, if `D` is `0`, it is resized to match the size of `B` and `C` if necessary. An additional argument `var` may be used to specify the variable of the Laplace (`'s'` (default) or `'p'`) or z transform (`'z'` (default) or `'q'`).

`ss(A,B,C,D,b)`, where `b` is an LTI object, creates a state-space model of the same kind (continuous/discrete time, sampling time and variable) as `b`.

`ss(b)` converts the LTI object `b` to a state-space model.

Examples

```
use lti
sc = ss(-1, [1,2], [2;5], 0)
  sc =
    continuous-time LTI state-space system
    A =
        -1
    B =
         1     2
    C =
         2
         5
    D =
         0     0
         0     0
sd = ss(tf(1,[1,2,3,4],0.1))
  sd =
    discrete-time LTI state-space system, Ts=0.1
    A =
      -2    -3    -4
       1     0     0
       0     1     0
    B =
       1
       0
       0
    C =
       0     0     1
    D =
       0
```

See also

`tf::tf`

tf::tf

LTI transfer function constructor.

Syntax

```
use lti
a = tf
a = tf(num, den)
```

```
a = tf(numlist, denlist)
a = tf(..., Ts)
a = tf(..., Ts, var)
a = tf(..., b)
a = tf(gain)
a = tf(b)
```

Description

`tf(num,den)` creates an LTI object which represents the continuous-time transfer function specified by descending-power coefficient vectors `num` and `den`. `tf(num,den,Ts)` creates an LTI object which represents a discrete-time transfer function with sampling period `Ts`.

In both cases, `num` and `den` can be replaced with cell arrays of coefficients whose elements are the descending-power coefficient vectors. The number of rows is the number of system outputs, and the number of columns is the number of system inputs.

An additional argument `var` may be used to specify the variable of the Laplace (`'s'` (default) or `'p'`) or z transform (`'z'` (default) or `'q'`).

`tf(...,b)`, where `b` is an LTI object, creates a transfer function of the same kind (continuous/discrete time, sampling time and variable) as `b`.

`tf(b)` converts the LTI object `b` to a transfer function.

`tf(gain)`, where `gain` is a matrix, creates a matrix of gains.

Examples

Simple continuous-time system with variable `p` (`p` is used only for display):

```
use lti
sc = tf(1,[1,2,3,4],'p')
  sc =
    continuous-time transfer function
    1/(pˆ3+2pˆ2+3p+4)
```

Matrix of discrete-time transfer functions for one input and two outputs, with a sampling period of 1ms:

```
sd = tf({0.1; 0.15}, {[1, -0.8]; [1; -0.78]}, 1e-3)
  sd =
    discrete-time transfer function, Ts=1e-3
    y1/u1: 0.1/(s-0.8)
    y2/u1: 0.15/(s-0.78)
```

See also

`zpk::zpk`, `pid::pid`, `pidstd::pidstd`, `ss::ss`

zpk::zpk

LTI zero-pole-gain constructor.

Syntax

```
use lti
a = zpk(z, p, k)
a = zpk(Z, P, K)
a = zpk(..., Ts)
a = zpk(..., Ts, var)
a = zpk(..., b)
a = zpk(b)
```

Description

`zpk` creates a zero-pole-gain LTI object. It accepts a vector of zeros, a vector of poles, and a scalar gain for a simple-input simple-output (SISO) system; or a cell array of zeros, a cell array of poles, and a real array of gains for multiple-input multiple-output (MIMO) systems. `zpk(z,p,k,Ts)` creates an LTI object which represents a discrete-time transfer function with sampling period `Ts`.

In both cases, `z` and `p` can be replaced with cell arrays of coefficients whose elements are the zeros and poles vectors, and `k` with a matrix of the same size. The number of rows is the number of system outputs, and the number of columns is the number of system inputs.

An additional argument `var` may be used to specify the variable of the Laplace (`'s'` (default) or `'p'`) or z transform (`'z'` (default) or `'q'`).

`zpk(...,b)`, where `b` is an LTI object, creates a zero-pole-gain transfer function of the same kind (continuous/discrete time, sampling time and variable) as `b`.

`zpk(b)` converts the LTI object `b` to a zero-pole-gain transfer function.

Example

```
use lti
sd = zpk(0.3, [0.8+0.5j; 0.8-0.5j], 10, 0.1)
  discrete-time zero-pole-gain transfer function, Ts=0.1
  10(z-0.3)/(z-(0.8+0.5j)(z-(0.8-0.5j)
```

See also

`tf::tf`, `pid::pid`, `pidstd::pidstd`, `ss::ss`

lti::append

Append the inputs and outputs of systems.

Syntax

```
use lti
b = append(a1, a2, ...)
```

Description

`append(a1,a2)` builds a system with inputs `[u1;u2]` and outputs `[y1;y2]`, where u1 and u2 are the inputs of a1 and y1 and y2 their outputs, respectively. `append` accepts any number of input arguments.

See also

`lti::connect`, `ss::augstate`

ss::augstate

Extend the output of a system with its states.

Syntax

```
use lti
b = augstate(a)
```

Description

`augstate(a)` extends the `ss` object a by adding its states to its outputs. The new output is `[y;x]`, where y is the output of a and x is its states.

See also

`lti::append`

lti::beginning

First index.

Syntax

```
use lti
var(...beginning...)
```

Description

In an expression used as an index between parenthesis, `beginning(a)` gives the first valid value for an index. It is always 1.

See also

`lti::end`, `lti::subsasgn`, `lti::subsref`

lti::c2d

Conversion from continuous time to discrete time.

Syntax

```
use lti
b = c2d(a, Ts)
b = c2d(a, Ts, method)
```

Description

`c2d(a,Ts)` converts the continuous-time system a to a discrete-time system with sampling period `Ts`.

`c2d(a,Ts,method)` uses the specified conversion method. `method` is one of the methods supported by `c2dm` for classes `ss`, `tf` and `zpk`, and `'ForwardEuler'`, `'BackwardEuler'` or `'Trapezoidal'` for classes `pid` and `pidstd`.

See also

`lti::d2c`, `c2dm`

lti::connect

Arbitrary feedback connections.

Syntax

```
use lti
b = connect(a, links, in, out)
```

Description

`connect(a,links,in,out)` modifies `lti` object a by connecting some of the outputs to some of the inputs and by keeping some of the inputs and some of the outputs. Connections are specified by the rows of matrix `link`. In each row, the first element is the index of the system input where the connection ends; other elements are indices to system outputs which are summed. The sign of the indices to outputs gives the sign of the unit weight in the sum. Zeros are ignored. Arguments `in` and `out` specify which input and output to keep.

See also

`lti::feedback`

lti::ctranspose

Conjugate transpose.

Syntax

```
use lti
b = a'
b = ctranspose(a)
```

Description

`a'` or `ctranspose(a)` gives the conjugate transpose of a.

The conjugate of the single-input single-output (SISO) continuous-time transfer function `G(s)` is defined as `G(-s)`, and the conjugate of the SISO discrete-time transfer function `G(z)` is defined as `G(1/z)`; the conjugate transpose is the conjugate of the transpose of the original system.

See also

`lti::transpose`, operator `'`

ss::ctrb

Controllability matrix.

Syntax

```
use lti
C = crtb(a)
```

Description

`ctrb(a)` gives the controllability matrix of system a, which is full-rank if and only if a is controllable.

See also

`ss::obsv`

lti::d2c

Conversion from discrete time to continuous time.

Syntax

```
use lti
b = d2c(a)
b = d2c(a, method)
```

Description

`d2c(a)` converts the discrete-time system a to a continuous-time system.

`d2c(a,method)` uses the specified conversion method. `method` is one of the methods supported by `d2cm` for classes `ss`, `tf` and `zpk`, and is ignored for class `pid` and `pidstd`.

See also

`lti::c2d`, `d2cm`

lti::dcgain

Steady-state gain.

Syntax

```
use lti
g = dcgain(a)
```

Description

`dcgain(a)` gives the steady-state gain of system a.

See also

`lti::norm`

lti::end

Last index.

Syntax

```
use lti
var(...end...)
```

Description

In an expression used as an index between parenthesis, end gives the last valid value for that index. It is `size(var,1)` or `size(var,2)`.

Example

Time response when the last input is a step:

```
use lti
P = ss([1,2;-3,-4],[1,0;0,1],[3,5]);
P1 = P(:, end)
  continuous-time LTI state-space system
  A =
     1   2
    -3  -4
  B =
     0
     1
  C =
     3   5
  D =
     0
step(P1);
```

See also

`lti::beginning`, `lti::subsasgn`, `lti::subsref`

lti::evalfr

Frequency value.

Syntax

```
use lti
y = evalfr(a, x)
```

Description

`evalfr(a,x)` evaluates system a at complex value or values x. If x is a vector of values, results are stacked along the third dimension.

Example

```
use lti
sys = [tf(1, [1,2,3]), tf(2, [1,2,3,4])];
evalfr(sys, 0:1j:3j)
  ans =
    1x2x4 array
    (:,:,1) =
      0.3333                  0.5
    (:,:,2) =
      0.25     -0.25j         0.5      -0.5j
    (:,:,3) =
     -5.8824e-2-0.2353j      -0.4      +0.2j
    (:,:,4) =
     -8.3333e-2-8.3333e-2j   -5.3846e-2+6.9231e-2j
```

See also

`polyval`

frd::fcat

Frequency concatenation.

Syntax

```
use lti
c = fcat(a, b)
```

Description

`fcat(a,b)` concatenates the frequency response data of `frd` objects a and b along the frequency axis, and sort data by increasing frequency. The size of a and b must be the same (same numbers of inputs and outputs).

Example

```
use lti
G = tf(1, [1, 2, 3, 4]);
a = frd(G, 0:5);
b = frd(G, 6:20);
c = fcat(a, b);
d = frd(G, 0:20); // same as c
```

See also

`frd::frd`

lti::feedback

Feedback connection.

Syntax

```
use lti
c = feedback(a, b)
c = feedback(a, b, sign)
c = feedback(a, b, ina, outa)
c = feedback(a, b, ina, outa, sign)
```

Description

`feedback(a,b)` connects all the outputs of `lti` object a to all its inputs via the negative feedback `lti` object b.

`feedback(a,b,sign)` applies positive feedback with weight `sign`; the default value of `sign` is `-1`.

`feedback(a,b,ina,outa)` specifies which inputs and outputs of a to use for feedback. The inputs and outputs of the result always correspond to the ones of a.

See also

`lti::connect`

frd::frdata

Get frequency response data.

Syntax

```
use lti
(resp, freq) = frdata(f)
(resp, freq, Ts) = frdata(f)
```

Description

`frdata(f)`, where `f` is an `frd` object, gives the complex frequency response, the corresponding frequencies, and optionally the sampling period or the empty array `[]` for continuous-time systems.

See also

`frd::frd`

frd::fselect

Frequency selection.

Syntax

```
use lti
b = fselect(a, ix)
b = fselect(a, sel)
b = fselect(a, freqmin, freqmax)
```

Description

`fselect(a,ix)` selects frequencies of `frd` object a whose index are in array `ix`. The frequencies of the result are `a.freq(ix)`.

`fselect(a,sel)` selects frequencies of `frd` object a corresponding to `true` values in logical array `sel`. The frequencies of the result are `a.freq(sel)`.

`fselect(a,freqmin,freqmax)` selects frequencies of `frd` object a which are greater than or equal to `freqmin` and less than or equal to `freqmax`. The frequencies of the result are `a.freq(a.freq>=freqmin&a.freq<=freqmax)`.

See also

`frd::frd`, operator `()`

frd::interp

Frequency interpolation.

Syntax

```
use lti
b = interp(a, freq)
b = interp(a, freq, method)
```

Description

`interp(a,freq)` interpolates response data of `frd` object a at the frequencies in array `freq`. The frequencies of the result are `freq`. The interpolation method is linear. Interpolation for frequencies outside the frequency range of a yields nan (not a number).

`interp(a,freq,method)` use the specified method for interpolation. Method is one of the strings accepted by `interp1` (`'0'` or `'nearest'`, `'<'`, `'>'`, `'1'` or `'linear'`, `'3'` or `'cubic'`, `'p'` or `'pchip'`).

See also

`frd::frd`, `interp1`

lti::inv

System inverse.

Syntax

```
use lti
b = inv(a)
```

Description

`inv(a)` gives the inverse of system a.

See also

`lti::mldivide`, `lti::mrdivide`

isct

Test for a continous-time LTI.

Syntax

```
use lti
b = isct(a)
```

Description

`isct(a)` is true if system a is continuous-time or static, and false otherwise.

See also

`isdt`

isdt

Test for a discrete-time LTI.

Syntax

```
use lti
b = isdt(a)
```

Description

`isdt(a)` is true if system a is discrete-time or static, and false otherwise.

See also

`isct`

lti::isempty

Test for an LTI without input/output.

Syntax

```
use lti
b = isempty(a)
```

Description

`isempty(a)` is true if system a has no input and/or no output, and false otherwise.

See also

`lti::size`, `lti::issiso`

lti::isproper

Test for a proper (causal) LTI.

Syntax

```
use lti
b = isproper(a)
```

Description

`isproper(a)` is `true` if `lti` object a is causal, or `false` otherwise. An `ss` object is always causal. A `tf` object is causal if all the transfer functions are proper, i.e. if the degrees of the denominators are at least as large as the degrees of the numerators.

lti::issiso

Test for a single-input single-output LTI.

Syntax

```
use lti
b = issiso(a)
```

Description

`issiso(a)` is `true` if `lti` object a has one input and one output (single-input single-output system, or SISO), or `false` otherwise.

`lti::size`, `lti::isempty`

tf::mathml zpk::mathml pid::mathml pidstd::mathml

Conversion to MathML.

Syntax

```
use lti, lti_mathml
str = mathml(G)
str = mathml(G, false)
str = mathml(..., Format=f, NPrec=n)
```

Description

`mathml(x)` converts its argument `x` to MathML presentation, returned as a string.

By default, the MathML top-level element is `<math>`. If the result is to be used as a MathML subelement of a larger equation, a last input argument equal to the logical value `false` can be specified to suppress `<math>`.

By default, `mathml` converts numbers like format `'%g'` of `sprintf`. Named arguments can override them: `format` is a single letter format recognized by `sprintf` and `NPrec` is the precision (number of decimals).

Example

```
use lti, lti_mathml
G = zpk(-1, [1, 2+j, 2-j], 2);
m = mathml(G);
math(0, 0, m);
```

See also

`mathml`, `sprintf`

lti::minreal

Minimum realization.

Syntax

```
use lti
b = minreal(a)
b = minreal(a, tol)
```

Description

`minreal(a)` modifies `lti` object `a` in order to remove states which are not controllable and/or not observable. For `tf` objects, identical zeros and poles are canceled out.

`minreal(a,tol)` uses tolerance `tol` to decide whether to discard a state or a pair of pole/zero.

lti::minus

System difference.

Syntax

```
use lti
c = a - b
c = minus(a, b)
```

Description

`a-b` computes the system whose inputs are fed to both `a` and `b` and whose outputs are the difference between outputs of `a` and `b`. If `a` and `b` are transfer functions or matrices of transfer functions, this is equivalent to a difference of matrices.

See also

`lti::parallel`, `lti::plus`, `lti::uminus`

lti::mldivide

System left division.

Syntax

```
use lti
c = a \ b
c = mldivide(a, b)
```

Description

`a/b` is equivalent to `inv(a)*b`.

See also

`lti::mrdivide`, `lti::times`, `lti::inv`

lti::mrdivide

System right division.

Syntax

```
use lti
c = a / b
c = mrdivide(a, b)
```

Description

a/b is equivalent to a∗inv(b).

See also

lti::mldivide, lti::times, lti::inv

lti::mtimes

System product.

Syntax

```
use lti
c = a ∗ b
c = mtimes(a, b)
```

Description

a∗b connects the outputs of lti object b to the inputs of lti object a. If a and b are transfer functions or matrices of transfer functions, this is equivalent to a product of matrices.

See also

lti::series

lti::norm

H2 norm.

Syntax

```
use lti
h2 = norm(a)
```

Description

norm(a) gives the H2 norm of the system a.

See also

lti::dcgain

ss::obsv

Observability matrix.

Syntax

```
use lti
O = obsv(a)
```

Description

`obsv(a)` gives the observability matrix of system a, which is full-rank if and only if a is observable.

See also

`ss::ctrb`

lti::parallel

Parallel connection.

Syntax

```
use lti
c = parallel(a, b)
c = parallel(a, b, ina, inb, outa, outb)
```

Description

`parallel(a,b)` connects `lti` objects a and b in such a way that the inputs of the result is applied to both a and b, and the outputs of the result is their sum.

`parallel(a,b,ina,inb,outa,outb)` specifies which inputs are shared between a and b, and which outputs are summed. The inputs of the result are partitioned as `[ua,uab,ub]` and the outputs as `[ya,yab,yb]`. Inputs uab are fed to inputs `ina` of a and `inb` of b; inputs ua are fed to the remaining inputs of a, and ub to the remaining inputs of b. Similarly, outputs yab are the sum of outputs `outa` of a and outputs `outb` of b, and ya and yb are the remaining outputs of a and b, respectively.

See also

`lti::series`

lti::piddata

Get PID parameters.

Syntax

```
use lti
(Kp, Ki, Kd, Tf) = piddata(a)
(Kp, Ki, Kd, Tf, Ts) = piddata(a)
```

Description

`piddata(a)`, where a is any kind of LTI object which has the structure of a PID controller except for `frd`, gives the PID parameters `Kp`, `Ki`, `Kd` and `Tf`, and optionally the sampling period or the empty array `[]` for continuous-time systems. The parameters are given as matrices; the rows correspond to the outputs, and their columns to the inputs.

See also

`pid::pid`, `lti::pidstddata`, `lti::tfdata`

lti::pidstddata

Get standard PID parameters.

Syntax

```
use lti
(Kp, Ti, Td, N) = pidstddata(a)
(Kp, Ti, Td, N, Ts) = pidstddata(a)
```

Description

`pidstddata(a)`, where a is any kind of LTI object which has the structure of a PID controller except for `frd`, gives the standard PID parameters `Kp`, `Ti`, `Td` and `N`, and optionally the sampling period or the empty array `[]` for continuous-time systems. The parameters are given as matrices; the rows correspond to the outputs, and their columns to the inputs.

See also

`pidstd::pidstd`, `lti::piddata`, `lti::tfdata`

lti::plus

System sum.

Syntax

```
use lti
c = a + b
c = plus(a, b)
```

Description

a+b computes the system whose inputs are fed to both a and b and whose outputs are the sum of the outputs of a and b. If a and b are transfer functions or matrices of transfer functions, this is equivalent to a sum of matrices.

See also

`lti::parallel`, `lti::minus`

lti::repmat

Replicate a system.

Syntax

```
use lti
b = repmat(a, n)
b = repmat(a, [m,n])
b = repmat(a, m, n)
```

Description

`repmat(a,m,n)`, when `a` is an `lti` object and `m` and `n` are positive integers, creates a new system of the same class with `m` times as many outputs and `n` times as many inputs. If `a` is a matrix of transfer functions, it is replicated `m` times vertically and `n` horizontally, as if `a` were a numeric matrix. If `a` is a state-space system, matrices B, C, and D are replicated to obtain the same effect.

`repmat(a,[m,n])` gives the same result as `repmat(a,m,n)`; `repmat(a,n)` gives the same result as `repmat(a,n,n)`.

See also

`lti::append`

lti::series

Series connection.

Syntax

```
use lti
c = series(a, b)
c = series(a, b, outa, inb)
```

Description

`series(a,b)` connects the outputs of `lti` object `a` to the inputs of `lti` object `b`.

`series(a,b,outa,inb)` connects outputs `outa` of `a` to inputs `inb` of `b`. Unconnected outputs of `a` and inputs of `b` are discarded.

See also

`lti::mtimes`, `lti::parallel`

lti::size

Number of outputs and inputs.

Syntax

```
use lti
s = size(a)
(nout, nin) = size(a)
n = size(a, dim)
```

Description

With one output argument, `size(a)` gives the row vector `[nout,nin]`, where `nout` is the number of outputs of system `a` and `nin` its number of inputs. With two output arguments, `size(a)` returns these results separately as scalars.

`size(a,1)` gives only the number of outputs, and `size(a,2)` only the number of inputs.

See also

`lti::isempty`, `lti::issiso`

lti::ssdata

Get state-space matrices.

Syntax

```
use lti
(A, B, C, D) = ssdata(a)
(A, B, C, D, Ts) = ssdata(a)
```

Description

`ssdata(a)`, where `a` is any kind of LTI object except for `frd`, gives the four matrices of the state-space model, and optionally the sampling period or the empty array `[]` for continuous-time systems.

See also

`ss::ss`, `lti::tfdata`

lti::subsasgn

Assignment to a part of an LTI system.

Syntax

```
use lti
var(i,j) = a
var(ix) = a
var(select) = a
var.field = value
a = subsasgn(a, s, b)
```

Description

The method `subsasgn(a)` permits the use of all kinds of assignments to a part of an LTI system. If the variable is a matrix of transfer functions, `subsasgn` produces the expected result, converting the right-hand side of the assignment to a matrix of transfer function if required. If the variable is a state-space model, the result is equivalent; the result remains a state-space model. For state-space models, changing all the inputs or all the outputs with the syntax `var(expr,:)=sys` or `var(:,expr)=sys` is much more efficient than specifying both subscripts or a single index.

The syntax for field assignment, `var.field=value`, is defined for the following fields: for state-space models, `A`, `B`, `C`, and `D` (matrices of the state-space model); for transfer functions, `num` and den (cell arrays of coefficients); for zero-pole-gain transfer functions, z and p (cell arrays of zero or pole vectors), and k (gain matrix); for PID controllers, Kp, `Ki`, Kd, `Tf`, `Ti` and `Td` (controller parameter matrices); for all LTI objects, `var` (string) and `Ts` (scalar, or empty array for continuous-time systems). Field assignment must preserve the size of matrices and arrays.

The syntax with braces (`var{i}=value`) is not supported.

See also

`lti::subsref`, operator `()`, `subsasgn`

lti::subsref

Extraction of a part of an LTI system.

Syntax

```
use lti
var(i,j)
var(ix)
var(select)
var.field
b = subsref(a, s)
```

Description

The method `subsref(a)` permits the use of all kinds of extraction of a part of an LTI system. If the variable is a matrix of transfer functions, `subsref` produces the expected result. If the variable is a state-space model, the result is equivalent; the result remains a state-space model, with the same state vector (the same matrix `A`) as the original system. For state-space models, extracting all the inputs or all the outputs with the syntax `var(expr,:)` or `var(:,expr)` is much more efficient than specifying both subscripts or a single index.

If the variable is an `frd` object, `var('freq',i)` produces a new `frd` object where the frequency vector is `var.frequency(i)` amd the response array contains the corresponding reponse. `i` can be a scalar index, a vector of indices or a logical array with the same size as `var.frequency`.

The syntax for field access, `var.field`, is defined for the following fields: for state-space models, `A`, `B`, `C`, and `D` (matrices of the state-space model); for transfer functions, `num` and `den` (cell arrays of coefficients); for zero-pole-gain transfer functions, `z` and `p` (cell arrays of zero or pole vectors), and `k` (gain matrix); for PID controllers, `Kp`, `Ki`, `Kd`, `Tf`, `Ti` and `Td` (controller parameter matrices); for all LTI objects, `var` (string) and `Ts` (scalar, or empty array for continuous-time systems).

The syntax with braces (`var{i}`) is not supported.

See also

`lti::subsasgn`, operator `()`, `subsasgn`

lti::tfdata

Get transfer functions.

Syntax

```
use lti
(num, den) = tfdata(a)
(num, den, Ts) = tfdata(a)
```

Description

`tfdata(a)`, where a is any kind of LTI object except for `frd`, gives the numerator and denominator of the transfer function model, and optionally the sampling period or the empty array `[]` for continuous-time systems. The numerators and denominators are given as a cell array of power-descending coefficient vectors; the rows of the cell arrays correspond to the outputs, and their columns to the inputs.

See also

`tf::tf`, `lti::zpkdata`, `lti::ssdata`

lti::transpose

Transpose.

Syntax

```
use lti
b = a.'
b = transpose(a)
```

Description

`a.'` or `transpose(a)` gives the transpose of a, i.e. `a.'(i,j)=a(j,i)`.

See also

`lti::ctranspose`, operator `.'`

lti::uminus

Negative.

Syntax

```
use lti
b = -a
b = uminus(a)
```

Description

`-a` multiplies all the outputs (or all the inputs) of system a by -1. If a is a transfer functions or a matrix of transfer functions, this is equivalent to the unary minus.

See also

`lti::minus`, `lti::uplus`

lti::uplus

Positive.

Syntax

```
use lti
b = +a
b = uplus(a)
```

Description

+a gives a.

See also

`lti::uminus`, `lti::plus`

lti::zpkdata

Get zeros, poles and gains.

Syntax

```
use lti
(z, p, k) = zpkdata(a)
(z, p, k, Ts) = zpkdata(a)
```

Description

`zpkdata(a)`, where a is any kind of LTI object except for `frd`, gives the zeros, poles and gains of the transfer function model, and optionally the sampling period or the empty array `[]` for continuous-time systems. The zeros and poles are given as a cell array of vectors; the rows of the cell arrays correspond to the outputs, and their columns to the inputs.

See also

`zpk::zpk`, `lti::tfdata`

11.9 lti (graphics)

In addition to the class definitions and the computational methods, library `lti` includes methods which provide for `lti` objects the same functionality as the native graphical functions of Sysquake for dynamical systems, such as `bodemag` for the magnitude of the Bode diagram or `step` for the step response. The system is provided as a single `lti` object instead of separate vectors for the numerator and denominator or four matrices for state-space models. For discrete-time systems, the sampling time is also obtained from the object, and the method name is the same as its continuous-time equivalent, without an initial `d` (e.g. `step(G)` is the discrete-time step response of `G` if `G` is a discrete-time `tf`, `zpk` or `ss` object).

The method definitions are stored in a separate file which is referenced in `lti` with `includeifexists`; this means that only `lti` must be loaded, with

```
use lti
```

Functions

lti::bodemag

Magnitude of the Bode plot.

Syntax

```
use lti
bodemag(a)
bodemag(a, style, id)
(mag, w) = bodemag(a)
```

Description

`bodemag(a)` plots the magnitude of the Bode diagram of system `a`, which can be any lti object with a single input (`size(a,2)` must be 1), continuous-time or discrete-time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `bodemag` gives the magnitude and the frequency as column vectors. No display is produced.

Examples

Green plot for $\left|1/(s^3 + 2s^2 + 3s + 4)\right|$ with $s = j\omega$ (see Fig. 10.9):

```
G = tf(1, [1, 2, 3, 4]);
bodemag(G, 'g');
```

The same plot, between $\omega = 0$ and $\omega = 10$, with a named argument for the color:

```
scale([0,10]);
bodemag(G, Color='green');
```

Frequency response of the discrete-time system $1/(z-0.9)(z-0.7-0.6j)(z-0.7+0.6j)$ with unit sampling period:

```
H = zpk([], [0.9,0.7+0.6j,0.7-0.6j], 1, 1);
bodemag(H);
```

See also

`lti::bodephase`, `lti::nichols`, `lti::nyquist`, `plotset`, `bodemag`

lti::bodephase

Phase of the Bode plot.

Syntax

```
use lti
bodephase(a)
bodephase(a, style, id)
(phase, w) = bodephase(a)
```

Description

`bodephase(a)` plots the phase of the Bode diagram of system a, which can be any lti object with a single input (`size(a,2)` must be 1), continuous-time or discrete-time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `bodephase` gives the phase and the frequency as column vectors. No display is produced.

See also

`lti::bodemag`, `lti::nichols`, `lti::nyquist`, `plotset`, `bodephase`

lti::impulse

Impulse response.

Syntax

```
use lti
impulse(a)
impulse(a, style, id)
(y, t) = impulse(a)
```

Description

`impulse(a)` plots the impulse response of system a, which can be any lti object with a single input (`size(a,2)` must be 1), continuous-time or discrete-time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `impulse` gives the output and the time as column vectors. No display is produced.

Example

Impulse response of the first order transfer function $1/(s/2+1)$:

```
G = tf(1, [1/2, 1]);
impulse(G);
```

See also

`lti::step`, `lti::lsim`, `ss::initial`, `plotset`, `impulse`

ss::initial

Time response with initial conditions.

Syntax

```
use lti
initial(a, x0)
initial(a, x0, style, id)
(y, t) = initial(a, x0)
```

Description

`initial(a,x0)` plots the time response of state-space system a with initial state `x0` and null input. System a can be continuous-time or discrete-time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `initial` gives the output and the time as column vectors. No display is produced.

Example

Response of a continuous-time system whose initial state is `[5;3]`:

```
a = ss([-0.3,0.1;-0.8,-0.4], [2;3], [1,3;2,1], [2;1]);
initial(a, [5;3])
```

See also

`lti::impulse`, `lti::step`, `lti::lsim`, `plotset`, `initial`

lti::lsim

Time response.

Syntax

```
use lti
lsim(a, u, t)
lsim(a, u, t, style, id)
(y, t) = lsim(a, u, t)
```

Description

`lsim(a,u,t)` plots the time response of system a. For continuous-time systems, the input is piece-wise linear; it is defined by points in real vectors `t` and `u`, which must have the same length. Input before `t(1)` and after `t(end)` is 0. For discrete-time systems, `u` is sampled at the rate given by the system, and `t` is ignored or can be omitted.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `lsim` gives the output and the time as column vectors. No display is produced.

Example

Response of continuous-time system given by its transfer function with an input defined by linear segments, displayed as a solid blue line:

```
G = tf(1, [1, 2, 3, 4]);
t = [0, 10, 20, 30, 50];
u = [1, 1, 0, 1, 1];
lsim(G, u, t, Color = 'blue');
```

See also

`lti::impulse`, `lti::step`, `ss::initial`, `plotset`, `lsim`

lti::nichols

Nichols plot.

Syntax

```
use lti
nichols(a, ...)
(mag, phase, w) = nichols(a, ...)
```

Description

`nichols(a)` plots the Nichols diagram of system a, which can be any lti object with a single input (`size(a,2)` must be 1), continuous-time or discrete-time.

The optional arguments `style` and `id` have their usual meaning.

With output arguments, `nichols` gives the magnitude, the phase and the corresponding frequency as column vectors. No display is produced.

See also

`lti::nyquist`, `lti::bodemag`, `lti::bodephase`, `plotset`, `nichols`

lti::nyquist

Nyquist plot.

Syntax

```
use lti
nyquist(a, ...)
(re, im, w) = nyquist(a, ...)
```

Description

nyquist(a) plots the Nyquist diagram of system a, which can be any lti object with a single input (size(a,2) must be 1), continuous-time or discrete-time.

The optional arguments style and id have their usual meaning.

With output arguments, nyquist gives the real part, the imaginary part and the corresponding frequency as column vectors. No display is produced.

See also

lti::nichols, lti::bodemag, lti::bodephase, plotset, nyquist

lti::pzmap

Pole/zero map.

Syntax

```
use lti
pzmap(a)
pzmap(a, style)
```

Description

pzmap(a) plots the poles and the zeros of system a in the complex plane. Poles are represented with crosses and zeros with circles. The system must be SISO (single-input, single-output).

With a second input argument, pzmap(a,style) uses the specified style for the poles and zeros. Typically, style is a structure array of two elements: the first element contains style options for the poles, and the second element, for the zeros. An empty structure (0 element) stands for the default style, and a simple structure uses the same style for the poles and the zeros.

Examples

Pole/zero map of a transfer function:

```
use lti
G = tf([2, 3, 4], [1, 2, 3, 4]);
pzmap(G);
```

Pole/zero map with the same scale along x and y axes, a grid showing relative damping and natural frequencies, and explicit style:

```
use lti
G = tf([2, 3, 4], [1, 2, 3, 4]);
scale equal;
sgrid;
plotoption fullgrid;
style = {
  Marker='x', MarkerEdgeColor='red';
  Marker='o', MarkerEdgeColor='navy', MarkerFaceColor='yellow'
}
pzmap(G, style);
```

See also

`lti::rlocus`, `plotset`, `plotroots`

lti::rlocus

Root locus.

Syntax

```
use lti
rlocus(a)
rlocus(a, style, id)
```

Description

`rlocus(a)` plots the root locus of system a, i.e. the locus of the poles of the system obtained by adding a feedback loop with a positive real gain. Only the root locus itself is displayed, as a solid line by default. Open-loop poles and zeros (the extremities of the root locus), which are typically displayed with special markers, can be added with `pzmap`.

The optional arguments `style` and `id` have their usual meaning.

Example

Root locus of a transfer function with open-loop poles and zeros displayed with `pzmap`. The scale is the same along x and y axes thanks to a call to `scale`, and a grid shows relative damping and natural frequencies.

```
use lti
G = tf([2, 3, 1], [1, 2, 3, 4]);
scale equal;
sgrid;
plotoption fullgrid;
rlocus(G);
pzmap(G);
```

See also

lti::pzmap, plotset, rlocus

lti::step

Step response.

Syntax

```
use lti
step(a)
step(a, style, id)
(y, t) = step(a)
```

Description

step(a) plots the step response of system a, which can be any lti object with a single input (size(a,2) must be 1), continuous-time or discrete-time.

The optional arguments style and id have their usual meaning.

With output arguments, step gives the output and the time as column vectors. No display is produced.

See also

lti::impulse, lti::lsim, ss::initial, plotset, step

11.10 sigenc

sigenc is a library which adds to LME functions for encoding and decoding scalar signals. It implements quantization, DPCM (differential pulse code modulation), and companders used in telephony.

The following statement makes available functions defined in sigenc:

```
use sigenc
```

Functions

alawcompress

A-law compressor.

Syntax

```
use sigenc
output = alawcompress(input)
output = alawcompress(input, a)
```

Description

`alawcompress(input,a)` compresses signal `input` with A-law method using parameter a. The signal is assumed to be in [-1,1]; values outside this range are clipped. `input` can be a real array of any size and dimension. The default value of a is 87.6.

The compressor and its inverse, the expander, are static, nonlinear filters used to improve the signal-noise ratio of quantized signals. The compressor should be used before quantization (or on a signal represented with a higher precision).

See also

`alawexpand`, `ulawcompress`

alawexpand

A-law expander.

Syntax

```
use sigenc
output = alawexpand(input)
output = alawexpand(input, a)
```

Description

`alawexpand(input,a)` expands signal `input` with A-law method using parameter a. `input` can be a real array of any size and dimension. The default value of a is 87.6.

See also

`alawcompress`, `ulawexpand`

dpcmdeco

Differential pulse code modulation decoding.

Syntax

```
use sigenc
output = dpcmdeco(i, codebook, predictor)
```

Description

dpcmdeco(i,codebook,predictor) reconstructs a signal encoded with differential pulse code modulation. It performs the opposite of dpcmenco.

See also

dpcmenco, dpcmopt

dpcmenco

Differential pulse code modulation encoding.

Syntax

```
use sigenc
i = dpcmenco(input, codebook, partition, predictor)
```

Description

dpcmenco(input,codebook,partition,predictor) quantizes the signal in vector input with differential pulse code modulation. It predicts the future response with the finite-impulse response filter given by polynomial predictor, and it quantizes the residual error with codebook and partition like quantiz. The output i is an array of codes with the same size and dimension as input.

The prediction $y^*(k)$ for sample k s

$$y^*(k) = \sum_{i=1}^{\text{degpredictor}} \text{predictor}_i \cdot y_q(k-i)$$

where $y_q(k)$ is the quantized (reconstructed) signal. The predictor must be strictly causal: predictor(0) must be zero. To encode the difference between in(k) and yq(k-1), predictor=[0,1]. Note that there is no drift between the reconstructed signal and the input [1], contrary to the case where the input is differentiated, quantized, and integrated.

Example

```
use sigenc
t = 0:0.1:10;
x = sin(t);
codebook = -.1:.01:.1;
partition = -.0:.01:.09;
predictor = [0, 1];
i = dpcmenco(x, codebook, partition, predictor);
y = dpcmdeco(i, codebook, predictor);
```

[1] Actually, there may be a drift if the arithmetic units used for encoding and decoding do not produce exactly the same results.

See also

`quantiz`, `dpcmdeco`, `dpcmopt`

dpcmopt

Differential pulse code modulation decoding.

Syntax

```
use sigenc
(predictor, codebook, partition) = dpcmopt(in, order, n)
(predictor, codebook, partition) = dpcmopt(in, order, codebook0)
(predictor, codebook, partition) = dpcmopt(in, predictor, ...)
(predictor, codebook, partition) = dpcmopt(..., tol)
predictor = dpcmopt(in, order)
```

Description

`dpcmopt(in,order,n)` gives the optimal predictor of order `order`, codebook of size `n` and partition to encode the signal in vector `in` with differential pulse code modulation. The result can be used with `dpcmenco` to encode signals with similar properties. If the second input argument is a vector, it is used as the predictor and not optimized further; its first element must be zero. If the third input argument is a vector, it is used as an initial guess for the codebook, which has the same length. An optional fourth input argument provides the tolerance (the default is 1e-7).

If only the predictor is required, only the input and the predictor order must be supplied as input arguments.

See also

`dpcmenco`, `dpcmdeco`, `lloyds`

lloyds

Optimal quantization.

Syntax

```
use sigenc
(partition, codebook) = lloyds(input, n)
(partition, codebook) = lloyds(input, codebook0)
(partition, codebook) = lloyds(..., tol)
```

Description

lloyds(input,n) computes the optimal partition and codebook for quantizing signal input with n codes, using the Lloyds algorithm.

If the second input argument is a vector, lloyds(input,codebook0) uses codebook0 as an initial guess for the codebook. The result has the same length.

A third argument can be used to specify the tolerance used as the stopping criterion of the optimization loop. The default is 1e-7.

Example

We start from a suboptimal partition and compute the distortion:

```
use sigenc
partition = [-1, 0, 1];
codebook = [-2, -0.5, 0.5, 2];
in = -5:0.6:3;
(i, out, dist) = quantiz(in, partition, codebook);
dist
  2.1421
```

The partition is optimized with lloyds, and the same signal is quantized again. The distortion is reduced.

```
(partition_opt, codebook_opt) = lloyds(in, codebook)
  partition_opt =
       -2.9  -0.5   1.3
  codebook_opt =
    -4.1  -1.7   0.4   2.2
(i, out, dist) = quantiz(in, partition_opt, codebook_opt);
dist
  1.0543
```

See also

quantiz, dpcmopt

quantiz

Table-based signal quantization.

Syntax

```
use sigenc
i = quantiz(input, partition)
(i, output, distortion) = quantiz(input, partition, codebook)
```

Description

`quantiz(input,partition)` quantizes signal `input` using `partition` as boundaries between different ranges. Range from −∞ to `partition(1)` corresponds to code 0, range from `partition(1)` to `partition(2)` corresponds to code 1, and so on. `input` may be a real array of any size and dimension; `partition` must be a sorted vector. The output `i` is an array of codes with the same size and dimension as `input`.

`quantiz(input,partition,codebook)` uses `codebook` as a look-up table to convert back from codes to signal. It should be a vector with one more element than `partition`. With a second output argument, `quantiz` gives `codebook(i)`.

With a third output argument, `quantiz` computes the distortion between `input` and `codebook(i)`, i.e. the mean of the squared error.

Example

```
use sigenc
partition = [-1, 0, 1];
codebook = [-2, -0.5, 0.5, 2];
in = randn(1, 5)
  in =
    0.1799 -9.7676e-2   -1.1431   -0.4986    1.0445
(i, out, dist) = quantiz(in, partition, codebook)
  i =
    2    1    0    1    2
  out =
    0.5 -0.5 -2   -0.5  0.5
  dist =
    0.259
```

See also

`lloyds`, `dpcmenco`

ulawcompress

mu-law compressor.

Syntax

```
use sigenc
output = ulawcompress(input)
output = ulawcompress(input, mu)
```

Description

`ulawcompress(input,mu)` compresses signal `input` with mu-law method using parameter `mu`. `input` can be a real array of any size and dimension. The default value of `mu` is 255.

The compressor and its inverse, the expander, are static, nonlinear filters used to improve the signal-noise ratio of quantized signals. The compressor should be used before quantization (or on a signal represented with a higher precision).

See also

`ulawexpand`, `alawcompress`

ulawexpand

mu-law expander.

Syntax

```
use sigenc
output = ulawexpand(input)
output = ulawexpand(input, mu)
```

Description

`ulawexpand(input,mu)` expands signal `input` with mu-law method using parameter a. `input` can be a real array of any size and dimension. The default value of `mu` is 255.

See also

`ulawcompress`, `alawexpand`

11.11 wav

wav is a library which adds to LME functions for encoding and decoding WAV files. WAV files contain digital sound. The wav library supports uncompressed, 8-bit and 16-bit, monophonic and polyphonic WAV files. It can also encode and decode WAV data in memory without files.

The following statement makes available functions defined in wav:

```
use wav
```

Functions

wavread

WAV decoding.

Syntax

```
use wav
(samples, samplerate, nbits) = wavread(filename)
(samples, samplerate, nbits) = wavread(filename, n)
(samples, samplerate, nbits) = wavread(filename, [n1,n2])
(samples, samplerate, nbits) = wavread(data, ...)
```

Description

`wavread(filename)` reads the WAV file `filename`. The result is a 2-d array, where each row corresponds to a sample and each column to a channel. Its class is the same as the native type of the WAV file, i.e. `int8` or `int16`.

`wavread(filename,n)`, where `n` is a scalar integer, reads the first `n` samples of the file. `wavread(filename,[n1,n2])`, where the second input argument is a vector of two integers, reads samples from `n1` to `n2` (the first sample corresponds to 1).

Instead of a file name string, the first input argument can be a vector of bytes, of class `int8` or `uint8`, which represents directly the contents of the WAV file.

In addition to the samples, `wavread` can return the sample rate in Hz (such as 8000 for phone-quality speech or 44100 for CD-quality music), and the number of bits per sample and channel.

See also

`wavwrite`

wavwrite

WAV encoding.

Syntax

```
use wav
wavwrite(samples, samplerate, nbits, filename)
data = wavwrite(samples, samplerate, nbits)
data = wavwrite(samples, samplerate)
```

Description

`wavwrite(samples,samplerate,nbits,filename)` writes a WAV file `filename` with samples in array `samples`, sample rate `samplerate` (in Hz), and `nbits` bits per sample and channel. Rows of `samples` corresponds to samples and columns to channels. `nbits` can be 8 or 16.

With 2 or 3 input arguments, `wavwrite` returns the contents of the WAV file as a vector of class `uint8`. The default word size is 16 bits per sample and channel.

Example

```
use wav
sr = 44100;
t = (0:sr)' / sr;
s = sin(2 * pi * 740 * t);
wavwrite(map2int(s, -1, 1, 'int16'), sr, 16, 'beep.wav');
```

See also

wavread

11.12 date

date is a library which adds to LME functions to convert date and time between numbers and strings.

The following statement makes available functions defined in date:

```
use date
```

Functions

datestr

Date to string conversion.

Syntax

```
use date
str = datestr(datetime)
str = datestr(date, format)
```

Description

datestr(datetime) converts the date and time to a string. The input argument can be a vector of 3 to 6 elements for the year, month, day, hour, minute, and second; a julian date as a scalar number; or a string, which is converted by datevec. The result has the following format:

```
jj-mmm-yyyy HH:MM:SS
```

where jj is the two-digit day, mmm the beginning of the month name, yyyy the four-digit year, HH the two-digit hour, MM the two-digit minute, and SS the two-digit second.

The format can be specified with a second input argument. When datestr scans the format string, it replaces the following sequences of characters and keeps the other ones unchanged:

Sequence	Replaced with
dd	day (2 digits)
ddd	day of week (3 char)
HH	hour (2 digits, 01-12 or 00-23)
MM	minute (2 digits)
mm	month (2 digits)
mmm	month (3 char)
PM	AM or PM
QQ	quarter (Q1 to Q4)
SS	second (2 digits)
sss	fraction of second (1-12 digits)
yy	year (2 digits)
yyyy	year (4 digits)

If the sequence PM is found, the hour is between 1 and 12; otherwise, between 0 and 23. Second fraction has as many digits as there are 's' characters in the format string.

Examples

```
use date
datestr(clock)
  18-Apr-2005 16:21:55
datestr(clock, 'ddd mm/dd/yyyy HH:MM PM')
  Mon 04/18/2005 04:23 PM
datestr(clock, 'yyyy-mm-ddTHH:MM:SS,sss')
  2008-08-23T02:41:37,515
```

See also

datevec, julian2cal, clock

datevec

String to date and time conversion.

Syntax

```
use date
datetime = datevec(str)
```

Description

datevec(str) converts the string str representing the date and/or the time to a row vector of 6 elements for the year, month, day, hour, minute, and second. The following formats are recognized:

Example	Value
20050418T162603	ISO 8601 date and time
2005-04-18	year, month and day
2005-Apr-18	year, month and day
18-Apr-2005	day, month and year
04/18/2005	month, day and year
04/18/00	month, day and year
18.04.2005	day, month and year
18.04.05	day, month and year
16:26:03	hour, minute and second
16:26	hour and minute
PM	afternoon

Unrecognized characters are ignored. If the year is given as two digits, it is assumed to be between 1951 and 2050.

Examples

```
use date
datevec('Date and time: 20050418T162603')
  2005   4  18  16  26   3
datevec('03:57 PM')
     0   0   0  15  57   0
datevec('01-Aug-1291')
  1291   8   1   0   0   0
datevec('At 16:30 on 11/04/07')
  2007  11   4  16  30   0
```

See also

`datestr`

weekday

Week day of a given date.

Syntax

```
use date
(num, str) = weekday(year, month, day)
(num, str) = weekday(datetime)
(num, str) = weekday(jd)
```

Description

weekday finds the week day of the date given as input. The date can be given with three input arguments for the year, the month and the day, or with one input argument for the date or date and time vector, or julian date.

The first output argument is the number of the day, from 1 for Sunday to 7 for Saturday; and the second output argument is its name as a string of 3 characters, such as `'Mon'` for Monday.

Example

Day of week of today:

```
use date
(num, str) = weekday(clock)
  num =
    2
  str =
    Mon
```

See also

`cal2julian`

11.13 constants

`constants` is a library which defines physical constants in SI units (meter, kilogram, second, ampere, kelvin, candela, mole).

The following statement makes available constants defined in `constants`:

```
use constants;
```

The following constants are defined:

Name	Value	Unit
avogadro_number	6.0221367e23	1/mole
boltzmann_constant	1.380658e-23	J/K
earth_mass	5.97370e24	kg
earth_radius	6.378140e6	m
electron_charge	1.60217733e-19	C
electron_mass	9.1093897e-31	kg
faraday_constant	9.6485309e4	C/mole
gravitational_constant	6.672659e-11	N mˆ2/kgˆ2
gravity_acceleration	9.80655	m/sˆ2
hubble_constant	3.2e-18	1/s
ice_point	273.15	K
induction_constant	1.256e-6	V s/A m
molar_gaz_constant	8.314510	J/K mole
molar_volume	22.41410e-3	mˆ3/mole
muon_mass	1.8835327e-28	kg
neutron_mass	1.6749286e-27	kg
plank_constant	6.6260755e-34	J s
plank_constant_reduced	1.0545727e-34	J s
plank_mass	2.17671e-8	kg
proton_mass	1.6726231e-27	kg
solar_radius	6.9599e8	m
speed_of_light	299792458	m/s
speed_of_sound	340.29205	m/s
stefan_boltzmann_constant	5.67051e-8	W/mˆ2 Kˆ-4
vacuum_permittivity	8.854187817e-12	A s/V m

11.14 colormaps

colormaps is a library containing functions related to color maps. Color maps are tables of colors which can be used with the colormap function; they are used by functions such as image and surf to map values to colors.

All functions accept at least the number of colors n as input argument, and produce an n-by-3 real double array which can be used directly as the argument of colormap. The default value of n is 256.

colormaps defines the following functions:

Function	Description
black2orangecm	color shades from black to orange
black2red2whitecm	color shades from black to red and white
blue2greencm	color shades from blue to green
blue2yellow2redcm	color shades from blue to yellow and red
cyan2magentacm	color shades from cyan to magenta
graycm	gray shades from black to white
green2yellowcm	color shades from green to yellow
huecm	color shades from red to red through green and blue
interprgbcm	colormap created with linear interpolation
magenta2yellowcm	color shades from magenta to yellow
red2yellowcm	color shades from red to yellow
sepiacm	sepia shades
whitecm	plain white

The following statement makes available functions defined in `colormaps`:

```
use colormaps
```

Functions are typically used directly as the argument of `colormap`:

```
colormap(blue2yellow2red);
```

Functions

black2orangecm

Colormap with shades from black to orange.

Syntax

```
use colormaps
cm = black2orangecm
cm = black2orangecm(n)
```

Description

`black2orangecm(n)` creates a color map with n entries corresponding to color shades from black to orange. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of `colormap`.

See also

`colormap`, `black2red2whitecm`, `blue2greencm`, `blue2yellow2redcm`, `cyan2magentacm`, `graycm`, `green2yellowcm`, `huecm`, `interprgbcm`, `magenta2yellowcm`, `red2yellowcm`, `sepiacm`, `whitecm`

black2red2whitecm

Colormap with shades from black to red and white.

Syntax

```
use colormaps
cm = black2red2whitecm
cm = black2red2whitecm(n)
```

Description

black2red2whitecm(n) creates a color map with n entries corresponding to color shades from black to red and white. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

colormap, black2orangecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

blue2greencm

Colormap with shades from blue to green.

Syntax

```
use colormaps
cm = blue2greencm
cm = blue2greencm(n)
```

Description

blue2greencm(n) creates a color map with n entries corresponding to color shades from blue to green. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

colormap, black2orangecm, black2red2whitecm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

blue2yellow2redcm

Colormap with shades from blue to yellow and red.

Syntax

```
use colormaps
cm = blue2yellow2redcm
cm = blue2yellow2redcm(n)
```

Description

`blue2yellow2redcm(n)` creates a color map with n entries corresponding to color shades from blue to yellow and red. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of `colormap`.

See also

`colormap`, `black2orangecm`, `black2red2whitecm`, `blue2greencm`, `cyan2magentacm`, `graycm`, `green2yellowcm`, `huecm`, `interprgbcm`, `magenta2yellowcm`, `red2yellowcm`, `sepiacm`, `whitecm`

cyan2magentacm

Colormap with shades from cyan to magenta.

Syntax

```
use colormaps
cm = cyan2magentacm
cm = cyan2magentacm(n)
```

Description

`cyan2magentacm(n)` creates a color map with n entries corresponding to color shades from cyan to magenta. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of `colormap`.

See also

`colormap`, `black2orangecm`, `black2red2whitecm`, `blue2greencm`, `blue2yellow2redcm`, `graycm`, `green2yellowcm`, `huecm`, `interprgbcm`, `magenta2yellowcm`, `red2yellowcm`, `sepiacm`, `whitecm`

graycm

Colormap with shades of gray.

Syntax

```
use colormaps
cm = graycm
cm = graycm(n)
```

Description

graycm(n) creates a color map with n entries corresponding to gray shades from black to white. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

green2yellowcm

Colormap with shades from green to yellow.

Syntax

```
use colormaps
cm = green2yellowcm
cm = green2yellowcm(n)
```

Description

green2yellowcm(n) creates a color map with n entries corresponding to color shades from green to yellow. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

huecm

Colormap with hue from red to red through green and blue.

Syntax

```
use colormaps
cm = huecm
cm = huecm(n)
```

Description

`huecm(n)` creates a color map with `n` entries corresponding to color shades with hue varying linearly from red back to red through green and blue. In HSV (hue-saturation-value) space, saturation and value are 1 (maximum). The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of `n` is 256.

The color map is suitable as the input argument of `colormap`.

See also

`colormap`, `black2orangecm`, `black2red2whitecm`, `blue2greencm`, `blue2yellow2redcm`, `cyan2magentacm`, `graycm`, `green2yellowcm`, `interprgbcm`, `magenta2yellowcm`, `red2yellowcm`, `sepiacm`, `whitecm`

interprgbcm

Colormap with entries obtained by linear interpolation.

Syntax

```
use colormaps
cm = interprgbcm(i, r, g, b)
cm = interprgbcm(i, r, g, b, n)
```

Description

`interprgbcm(i,r,b,g,n)` creates a color map with `n` entries. Color shades are interpolated between colors defined in RGB color space by corresponding elements of `r`, `g` and `b`, defined for input in `i`. These four arguments must be vectors of the same length larger or equal to 2 with elements between 0 and 1. Argument `i` must have monotonous entries with `i(1)=0` and `i(end)=1`. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of `n` is 256.

The color map is suitable as the input argument of `colormap`.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

magenta2yellowcm

Colormap with shades from magenta to yellow.

Syntax

```
use colormaps
cm = magenta2yellowcm
cm = magenta2yellowcm(n)
```

Description

magenta2yellowcm(n) creates a color map with n entries corresponding to color shades from magenta to yellow. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, red2yellowcm, sepiacm, whitecm

red2yellowcm

Colormap with shades from red to yellow.

Syntax

```
use colormaps
cm = red2yellowcm
cm = red2yellowcm(n)
```

Description

red2yellowcm(n) creates a color map with n entries corresponding to color shades from red to yellow. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of colormap.

See also

`colormap`, `black2orangecm`, `black2red2whitecm`, `blue2greencm`, `blue2yellow2redcm`, `cyan2magentacm`, `graycm`, `green2yellowcm`, `huecm`, `interprgbcm`, `magenta2yellowcm`, `sepiacm`, `whitecm`

sepiacm

Colormap with shades of sepia.

Syntax

```
use colormaps
cm = sepiacm
cm = sepiacm(n)
```

Description

`sepiacm(n)` creates a color map with n entries corresponding to shades of sepia. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of `colormap`.

See also

`colormap`, `black2orangecm`, `black2red2whitecm`, `blue2greencm`, `blue2yellow2redcm`, `cyan2magentacm`, `graycm`, `green2yellowcm`, `huecm`, `interprgbcm`, `magenta2yellowcm`, `red2yellowcm`, `whitecm`

whitecm

Colormap with plain white.

Syntax

```
use colormaps
cm = whitecm
cm = whitecm(n)
```

Description

`whitecm(n)` creates a color map with n identical entries corresponding to plain white. The color map is an n-by-3 array with one color per row; columns correspond to red, green, and blue components as real numbers between 0 to 1 (maximum intensity). The default value of n is 256.

The color map is suitable as the input argument of `colormap`.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm

11.15 polyhedra

Library polyhedra implements functions which create solid shapes with polygonal facesin 3D. Solids are displayed with plotpoly. They are defined by the coordinates of their vertices and by the list of vertex indices for each face. Other solids, such as cylinder and sphere, are generated with parametric equations and displayed with surf. Some solids have parameters, e.g. for the number of discrete values used for parameters. When called without output argument, with an optional trailing string argument for the edge style, the solid is displayed with the current scaling and color map. With output arguments, arrays X, Y, Z expected by surf, mesh and plotpoly, and index array expected by plotpoly, are produced. They can be modified to move, scale or stretch the solids.

The following statement makes available functions defined in polyhedra:

```
use polyhedra
```

Functions

cube

Create a cube.

Syntax

```
use polyhedra
cube;
cube(style);
(X, Y, Z, ind) = cube
```

Description

Without output argument, cube displays a cube, i.e. a convex solid whose six faces are squares. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, cube produces the X, Y, Z and ind arrays expected by plotpoly, and it does not display anything.

See also

`tetrahedron`, `octahedron`, `dodecahedron`, `icosahedron`, `plotpoly`

dodecahedron

Create a regular dodecahedron.

Syntax

```
use polyhedra
dodecahedron;
dodecahedron(style);
(X, Y, Z, ind) = dodecahedron
```

Description

Without output argument, `dodecahedron` displays a regular convex dodecahedron, i.e. a convex solid whose twelve faces are regular pentagons. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, `dodecahedron` produces the X, Y, Z and `ind` arrays expected by `plotpoly`, and it does not display anything.

See also

`tetrahedron`, `cube`, `octahedron`, `icosahedron`, `greatdodecahedron`, `greatstellateddodecahedron`, `smallstellateddodecahedron`, `plotpoly`

greatdodecahedron

Create a great dodecahedron.

Syntax

```
use polyhedra
greatdodecahedron;
greatdodecahedron(style);
(X, Y, Z, ind) = greatdodecahedron
```

Description

Without output argument, `greatdodecahedron` displays a great dodecahedron, i.e. a regular nonconvex solid whose twelve faces are regular pentagons. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, `greatdodecahedron` produces the X, Y, Z and `ind` arrays expected by `plotpoly`, and it does not display anything.

See also

dodecahedron, greatstellateddodecahedron, greaticosahedron, plotpoly

greaticosahedron

Create a great dodecahedron.

Syntax

```
use polyhedra
greaticosahedron;
greaticosahedron(style);
(X, Y, Z, ind) = greaticosahedron
```

Description

Without output argument, greaticosahedron displays a great icosahedron, i.e. a regular nonconvex solid whose twenty faces are equilateral triangles. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, greaticosahedron produces the X, Y, Z and ind arrays expected by plotpoly, and it does not display anything.

See also

icosahedron, greatdodecahedron, plotpoly

greatstellateddodecahedron

Create a great stellated dodecahedron.

Syntax

```
use polyhedra
greatstellateddodecahedron;
greatstellateddodecahedron(style);
(X, Y, Z, ind) = greatstellateddodecahedron
```

Description

Without output argument, greatstellateddodecahedron displays a great stellated dodecahedron, i.e. a regular nonconvex solid whose twelve faces are regular star pentagons and where each vertex is common to three faces. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, greatstellateddodecahedron produces the X, Y, Z and ind arrays expected by plotpoly, and it does not display anything.

See also

`dodecahedron`, `greatdodecahedron`, `smallstellateddodecahedron`, `plotpoly`

icosahedron

Create a regular icosahedron.

Syntax

```
use polyhedra
icosahedron;
icosahedron(style);
(X, Y, Z, ind) = icosahedron
```

Description

Without output argument, `icosahedron` displays a regular convex icosahedron, i.e. a convex solid whose twenty faces are equilateral triangles. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, `icosahedron` produces the X, Y, Z and `ind` arrays expected by `plotpoly`, and it does not display anything.

See also

`tetrahedron`, `cube`, `octahedron`, `dodecahedron`, `plotpoly`

octahedron

Create a regular octahedron.

Syntax

```
use polyhedra
octahedron;
octahedron(style);
(X, Y, Z, ind) = octahedron
```

Description

Without output argument, `octahedron` displays a regular octahedron, i.e. a convex solid whose eight faces are equilateral triangles. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, `octahedron` produces the X, Y, Z and `ind` arrays expected by `plotpoly`, and it does not display anything.

See also

tetrahedron, cube, dodecahedron, icosahedron, plotpoly

smallstellateddodecahedron

Create a small stellated dodecahedron.

Syntax

```
use polyhedra
smallstellateddodecahedron;
smallstellateddodecahedron(style);
(X, Y, Z, ind) = smallstellateddodecahedron
```

Description

Without output argument, smallstellateddodecahedron displays a small stellated dodecahedron, i.e. a regular nonconvex solid whose twelve faces are regular star pentagons and where each vertex is common to five faces. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, smallstellateddodecahedron produces the X, Y, Z and ind arrays expected by plotpoly, and it does not display anything.

See also

dodecahedron, greatdodecahedron, greatstellateddodecahedron, plotpoly

tetrahedron

Create a regular tetrahedron.

Syntax

```
use polyhedra
tetrahedron;
tetrahedron(style);
(X, Y, Z, ind) = tetrahedron
```

Description

Without output argument, tetrahedron displays a regular tetrahedron, i.e. a solid whose four faces are equilateral triangles. By default, edges are not drawn. An optional string input argument specifies the edge style.

With four output arguments, tetrahedron produces the X, Y, Z and ind arrays expected by plotpoly, and it does not display anything.

See also

`cube`, `octahedron`, `dodecahedron`, `icosahedron`, `plotpoly`

11.16 solids

Library `solids` implements functions which create solid shapes in 3D. Solids are generated with parametric equations and displayed with `surf`. When called without output argument, with an optional trailing string argument for the edge style, the solid is displayed with the current scaling and color map. With output arguments, arrays X, Y, Z expected by `surf` or `mesh` are produced. They can be modified to move, scale or stretch the solids.

The following statement makes available functions defined in `solids`:

```
use solids
```

Functions

cone

Cone.

Syntax

```
use solids
cone
cone(cap)
cone(cap, n)
cone(cap, n, style)
(X, Y, Z) = cone
(X, Y, Z) = cone(n)
```

Description

Without output argument, `cone` draws a cone approximated by a polyhedron. The optional first input argument, a logical value which is true by default, specifies if the cap is included. The optional second input argument, an integer, specifies the number of discrete values for the parameter which describes its surface.

By default, edges are not drawn. An optional third input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `cone` produces the X, Y, and Z arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`cylinder`, `sphere`, `cube`, `surf`

crosscap

Cross-cap.

Syntax

```
use solids
crosscap
crosscap(n)
crosscap(n, style)
(X, Y, Z) = crosscap
(X, Y, Z) = crosscap(n)
```

Description

Without output argument, `crosscap` draws a cross-cap (a self-intersecting surface) approximated by a polyhedron. With an input argument, `crosscap(n)` draws a cross-cap where the two parameters which describe its surface are sampled with `n` discrete values.

By default, edges are not drawn. An optional second input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `crosscap` produces the `X`, `Y`, and `Z` arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`klein`, `klein8`, `sphere`, `sphericon`, `surf`

cylinder

Cylinder.

Syntax

```
use solids
cylinder
cylinder(cap)
cylinder(cap, n)
cylinder(cap, n, style)
(X, Y, Z) = cylinder
(X, Y, Z) = cylinder(n)
```

Description

Without output argument, `cylinder` draws a cylinder approximated by a polyhedron. The optional first input argument, a logical value which is true by default, specifies if caps are included. The optional second input argument, an integer, specifies the number of discrete values for the parameter which describes its surface.

By default, edges are not drawn. An optional third input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `cylinder` produces the `X`, `Y`, and `Z` arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`cone`, `sphere`, `torus`, `cube`, `surf`

klein

Klein bottle.

Syntax

```
use solids
klein
klein(p)
klein(p, n)
klein(p, n, style)
(X, Y, Z) = ...
```

Description

Without output argument, `klein` draws a Klein bottle approximated by a polyhedron. With an input argument, `klein(p)` uses parameters stored in structure `p`. The following fields are used:

Field	Description	Default value
r0	average tube radius	0.7
d	tube variation	0.5
h	half height	3

With two input arguments, `klein(p,n)` draws a Klein bottle where the two parameters which describe its surface are sampled with `n` discrete values.

By default, edges are not drawn. An optional third input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `klein` produces the `X`, `Y`, and `Z` arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`klein8`, `crosscap`, `surf`

klein8

Figure 8 Klein bottle immersion.

Syntax

```
use solids
klein8
klein8(r)
klein8(r, n)
klein8(r, n, style)
(X, Y, Z) = ...
```

Description

Without output argument, `klein8` draws a figure 8 Klein bottle immersion (a closed, self-intersecting surface with one face) approximated by a polyhedron. With an input argument, `klein8(r)` draws the surface with a main radius of `r` (the default value is `1`).

With two input arguments, `klein8(r,n)` samples the two parameters which describe its surface with `n` discrete values.

By default, edges are not drawn. An optional third input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `klein8` produces the `X`, `Y`, and `Z` arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`klein`, `crosscap`, `surf`

sphere

Sphere.

Syntax

```
use solids
sphere
sphere(n)
sphere(n, style)
(X, Y, Z) = sphere
(X, Y, Z) = sphere(n)
```

Description

Without output argument, `sphere` draws a sphere approximated by a polyhedron. With an input argument, `sphere(n)` draws a sphere where the two parameters which describe its surface are sampled with `n` discrete values.

By default, edges are not drawn. An optional second input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `sphere` produces the X, Y, and Z arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`cylinder`, `cone`, `torus`, `cube`, `surf`

sphericon

Sphericon.

Syntax

```
use solids
sphericon
sphericon(n)
sphericon(n, style)
(X, Y, Z) = sphericon
(X, Y, Z) = sphericon(n)
```

Description

Without output argument, `sphericon` draws a sphericon (a 3D shape made from a bicone with a 90-degree apex, cut by a plane containing both apices, where one half is rotated by 90 degrees) approximated by a polyhedron. With an input argument, `sphericon(n)` draws a sphericon where the two parameters which describe its surface are sampled with `n` discrete values.

By default, edges are not drawn. An optional second input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `sphericon` produces the X, Y, and Z arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`sphere`, `crosscap`, `surf`

torus

Torus.

Syntax

```
use solids
torus
torus(r)
torus(r, n)
torus(r, n, style)
(X, Y, Z) = ...
```

Description

Without output argument, `torus` draws a torus approximated by a polyhedron with a main radius of 1 and a tube radius of 0.5. With an input argument, `torus(r)` draws a torus with tube radius `r`. With two input arguments, `torus(r,n)` draws a torus where the two parameters which describe its surface are sampled with `n` discrete values.

By default, edges are not drawn. An optional third input argument, a string, specifies the edge style; it corresponds to the style argument of `surf`.

With three output arguments, `torus` produces the X, Y, and Z arrays expected by `surf` or `mesh`, and it does not display anything.

See also

`sphere`, `cylinder`, `surf`

11.17 bench

Library bench implements functions to evaluate the performance of the LME implementation on the platform it is running on. It measures the amount of time required to execute different kinds of operations and gives numbers which can be seen as the equivalent frequency of the reference computer.

We intend to keep bench the same as long as it makes sense to make possible the comparison of successive generations of hardware.

The following statement makes available functions defined in bench:

```
use bench
```

The library if written in such a way that it is compatible with a Matlab M-file: the first function defined is the main entry point (all other functions are subfunctions which should not be called directly); and

features specific to LME, such as C- or C++-style comments, parenthesis for output arguments, and keywords `public` and `private` are avoided.

Function

bench

Run benchmark function.

Syntax

```
use bench
bench
bench(totaltime)
bench(totaltime, n)
(freq, names) = bench
(freq, names) = bench(totaltime)
```

Description

`bench` runs the benchmark, spending about 5 seconds for each phase. It displays the result of each phase and the average value, as the equivalent frequency of the reference platform in MHz, an Apple PowerBook G4 17" 1.33 GHz running under Mac OS 10.3.5 (therefore running `bench` gives about 1330). Finally, it displays in a table the comparison with a few other platforms.

With an input argument, `bench(totaltime)` computes the number of iterations of each phase so that the total time is about `totaltime` seconds, divided in equal amounts for each phase. If `totaltime` is negative, the number of iterations per second is displayed instead of the equivalent frequency of the reference platform. With two input arguments, `bench(totaltime,n)` performs the whole benchmark `n` times and keeps the best value for each phase. Reference values are obtained with `bench(35,10)`.

With output arguments, `(freq,names)=bench` returns in `freq` a vector of phase scores and in `names` a list of phase names.

Example

```
use bench
bench
Scores (PowerBook G4 at 1.33 GHz = 1330 MHz):
        lu: 789.336 MHz
       max: 774.279 MHz
 fibonacci: 776.552 MHz
     uint8: 777.402 MHz
```

```
    strfind: 772.787 MHz
       list: 786.320 MHz
    funcall: 785.705 MHz
    Average: 780.340 MHz
                                      AVERAGE
     Dell Dimension 2400 P4 3.06 GHz: 2749.3
         Apple PowerBook G4 1.33 GHz: 1330.0
                    ** This computer:  780.3
               Apple iBook G3 500 MHz:  497.4
                Sun Blade 100 500 MHz:  367.5
   Generic PC Pentium Win2K 300 MHz:  350.9
 Apple PowerBook 3400 ppc603 200 MHz:  133.7
        CerfBoard 255 XScale 400 MHz:   84.4
 Kontron X-board<861> SC1200 266 MHz:   76.9
 Palm Zire 71 OMAP 144 MHz, emu M68k:    1.2
```

See also

`tic`, `toc`

11.18 parbench

Library parbench implements functions to evaluate the performance of the LME implementation on the platform it is running on, using multiple parallel tasks. It measures the amount of time required to execute different kinds of operations and gives numbers which can be seen as the equivalent frequency of the reference computer. It is based on the same operations as bench; the results it produces can be compared directly to estimate the speedup parallel execution can provide on similar computations.

The following statement makes available functions defined in parbench:

```
use parbench
```

Function

parbench

Run parallel benchmark function.

Syntax

```
use parbench
parbench
parbench(totaltime)
```

```
parbench(totaltime, n)
(freq, names) = parbench
(freq, names) = parbench(totaltime)
```

Description

parbench runs the parallel benchmark, spending about 5 seconds for each phase. It displays the result of each phase and the average value, as the equivalent frequency of the reference platform in MHz, an Apple PowerBook G4 17" 1.33 GHz running the single-thread bench under Mac OS 10.3.5 (therefore running bench gives about 1330). Finally, it displays in a table the comparison with a few other platforms.

With an input argument, `parbench(totaltime)` computes the number of iterations of each phase so that the total time is about `totaltime` seconds, divided in equal amounts for each phase. If `totaltime` is negative, the number of iterations per second is displayed instead of the equivalent frequency of the reference platform. With two input arguments, `parbench(totaltime,n)` performs the whole parallel benchmark `n` times and keeps the best value for each phase. Reference values are obtained with `parbench(35,10)`.

With output arguments, `(freq,names)=parbench` returns in `freq` a vector of phase scores and in `names` a list of phase names.

Example

```
use parbench
parbench
Scores (PowerBook G4 at 1.33 GHz = 1330 MHz):
        lu: 4788.540 MHz
       max: 6418.464 MHz
 fibonacci: 1601.957 MHz
     uint8: 6233.715 MHz
   strfind: 6020.619 MHz
      list: 2393.057 MHz
   funcall: 7546.563 MHz
   Average: 5000.417 MHz
                                    AVERAGE   lu  max fibo uint strf list func
          i7-4790 3.6GHz win10 PAR:  42959  17k  35k  10k 101k  71k  12k  54k
                  ** This computer:  12568   7k  19k   3k  17k  22k   4k  16k
    MacBook Pro i7-620M 2.66GHz PAR:  10792   6k  15k   3k  16k  16k   4k  16k
             i7-4790 3.6GHz win10:   9619   5k   8k   2k  24k  13k   3k  13k
        MacPro Dual-Core Xeon 3 GHz:   5478   3829 7632 5311 5974 6469 3718 54
        MacBook Pro i7-620M 2.66GHz:   5149   3304 7359 1478 8201 6794 1847 70
   Apple Mac Mini Core Duo 1.66 GHz:   2516   2024 3185 2275 2465 3178 1852 26
     Apple PowerMac G5 Dual 2.5 GHz:   2411   2915 2633 2438 2239 2116 1835 27
        Apple PowerBook G4 1.33 GHz:   1330   1330 1330 1330 1330 1330 1330 13
DEC Workstation 500au Alpha 500 MHz:    402    227  516  359  663  457  230  3
Sun Blade 100 UltraSparcIIe 500 MHz:    367    194  400  488  414  400  326  3
```

See also
`bench`, `tic`, `toc`

Index