Calerga

Sysquake for LaTeX
User Manual

Embedding numerical results and graphics in LaTeX

November 2007

Sysquake for LATEX User Manual, August 2007.
Yves Piguet, Calerga Sàrl, Lausanne, Switzerland.

Most of the material in LyME 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:	http://www.calerga.com
E-mail:	sysquake@calerga.com
Mail:	Calerga Sàrl
	Av. de la Chablière 35
	1004 Lausanne
	Switzerland

Typesetting: 2007-11-29

Contents

6 LME Reference — 39

Chapter 1

Introduction

LaTeX is a typesetting environment for generating high-quality documents, for reading on-screen (typically as PDF) or for printing. Developed by Leslie Lamport with the support of many other contributors and the American Mathematics Society (AMS), LaTeX is extremely popular for scientific publications. LaTeX is a set of macros for TeX. From TeX, it inherits the high quality of text layout and equation formatting, and implementations on most computer platforms; it adds facilities for sections, page layout, table of contents, index, etc.

Like TeX, LaTeX takes as input a text file which contains the text to typeset and commands introduced with an escape character. To create a document, the author typically writes this file in a text editor, processes it with latex (a command-line tool), displays the result in a viewer, and continues with more iterations until a satisfactory result is obtained. Numerical results and graphics come usually from other applications such as Sysquake.

Sysquake for LaTeX is a LaTeX package and a command tool which let you embed Sysquake commands directly in LaTeX source files. With it, no need to copy and format numerical results or generate EPS file by hand; the package takes care of cumbersome tasks, letting you focus on the creative work. Sysquake for LaTeX uses the same language and libraries as Sysquake and other products of Calerga.

Chapter 2

Installing Sysquake for LaTeX

Installing Sysquake for LaTeX consists in three steps:

- Copy all Sysquake for LaTeX files to your hard disk. The default location is /opt/SysquakeLx on unix systems, and C:\Program Files\SysquakeForLaTeX on Windows.

- Update environment variables so that sysquakelatextool is found by the operating system and finds library files. This consists in adding the path of the directory containing sysquakelatextool to PATH, and setting a new environment variable SYSQUAKELXPATH to the semicolon-separated list of paths to library directories.

- Make your TeX distribution able to load the sysquake.sty package. In TDS-based distributions (TeX Live, teTex, MiKTeX, etc.), this means adding the path of TexFiles to the list of TDS roots.

LaTeX must also be able to execute external programs with \write18. Depending on the platform, this may require to add an option argument to pdflatex, such as -shell-escape or -enable-write18.

2.1 Windows

Install the latest release of MiKTeX (e.g. MiKTeX 2.6 2742). Then run Sysquake for LaTeX installer and follow the instructions.

The last step, as explained in the installer, is manual: make your TeX distribution aware of Sysquake for LaTeX. You should add a new root directory to the list of TDS directories.

In the Start menu, click MiKTeX>Settings and select the Roots tab. Click the Add button, select directory TexFiles of Sysquake for La-TeX, and confirm, clicking the main OK button of the MiKTeX Options window.

For other TeX distributions, please read their documentation. Sysquake for LaTeX installer is totally independent from any other software; you should just configure TeX so that it finds the file `sysquake.sty`.

2.2 Mac OS X

On Mac OS X, Sysquake for LaTeX is distributed as a package. To install it, just double-click its icon and accept everything. You will need an administrator password. An alias of the documentation entry file is put on the Desktop.

The Mac OS X package stores the distribution in /opt/SysquakeLx and creates the following symbolic links, also creating directories if they do not already exist:

```
ln -s /opt/SysquakeLx/bin/i386-darwin/sysquakelatextool \
  /usr/bin
ln -s /opt/SysquakeLx/TexFiles/tex/latex/sysquake.sty \
  ~/Library/texmf/tex/latex
ln -s /opt/SysquakeLx/TexFiles/tex/latex/sq-example.tex \
  ~/Library/texmf/tex/latex
ln -s /opt/SysquakeLx/TexFiles/doc/latex/sysquake/index.html \
  ~/Desktop/SQ-LaTeX.html
```


2.3 TeX Live on Linux and other Posix systems

Install the latest release of TeX Live (e.g. TeX Live 2007). Then follow the steps below:

– Unarchive the distribution of Sysquake for LaTeX for your platform to a suitable location, such as /opt/SysquakeLx.

– Have `bin/i386-linux/sysquakelatextool` in your PATH. There are two ways to do it: put a symbolic link to `bin/i386-linux/sysquakelatextool` in one of your existing program directories, such as /usr/bin or /usr/local/bin:

```
sudo ln -s /opt/SysquakeLx/bin/i386-linux/sysquakelatextool /usr/bin
```

or add /opt/SysquakeLx/SysquakeLx/bin/i386-linux to your environment variable PATH.

– Create a new environment variable named SYSQUAKELXPATH which contains a list of semicolon-separated paths for the directories containing .lml libraries. The default value is .;./Lib;/opt/SysquakeLx/Lib, which finds standard libraries if Sysquake for LaTeX is installed in /opt/SysquakeLx.

– Edit file texmf.cnf (type kpsewhich texmf.cnf in a terminal to find it) and append the full path of directory TexFiles to TEXM-FLOCAL, separated from previous path(s) with a semicolon. For example:

```
TEXMFLOCAL = $SELFAUTOPARENT/../texmf-local;/opt/SysquakeLx/TexFiles
```

Chapter 3

Sysquake for LaTeX Tutorial

Once installed, Sysquake for LaTeX is a LaTeX package which you can use as easily as any other package. In this tutorial, we will create a technical report which analyzes some of the properties of a rational function.

Sysquake for LaTeX works at the level of the TeX program, which, with LaTeX definitions, is usually called `latex` for creating DVI files, or `pdflatex` for creating PDF files. You can also choose among different applications which hide the details of running `latex` and displaying the result, such as emacs on unix platforms, TeXShop on Mac OS X, or Texmaker on Windows and other platforms. Sysquake for LaTeX is perfectly compatible with them; actually it just ignores them. So you can keep your favorite editing environment.

And now, let us begin!

Remark: the complete source file of another example comes with Sysquake for LaTeX.

3.1 Creating a LaTeX file

Use your usual method to create a LaTeX file and write a few things to check that everything runs correctly. For this tutorial, we will make a report. The LaTeX file, named for example `sq-tutorial.tex`, contains something like

```
\documentclass[11pt]{report}
\begin{document}

\title{Sysquake for \LaTeX{} tutorial}
\author{Calerga}
```

```
\date{13 August 2007}
\maketitle

\end{document}
```

3.2   Add the Sysquake package

Packages are sets of definitions which add new commands to LaTeX. They are listed in the preamble of LaTeX files, between `\documentclass` and `\begin{document}`. The Sysquake package is added like any other package:

```
\documentclass[11pt]{report}
\usepackage{sysquake}
\begin{document}
...
```

If there is an error when you typeset this file, the file `sysquake.sty` is probably not found. Please refer to section Sysquake for LaTeX Installation for more informations.

Assuming there is no error, we can now use LaTeX commands for invoking Sysquake.

3.3   Your first Sysquake computation

Since we are anxious to get a result, let us begin with something really simple. We will insert the result of 5+2 as an equation. Add the following line between `\maketitle` and `\end{document}`:

```
 We all know that $5+2=\sqexpr{5+2}$, don't we?
```

Running LaTeX twice will produce the result you expect. Why twice? Like other features of LaTeX, the Sysquake package needs two phases to do its magic. During the first phase, commands create a file named `sq-tutorial.lme` (the name is built after the .tex file) which contains Sysquake code, or more exactly code written in the programming language of Sysquake, LME. During the second phase, loading the sysquake package runs the command-line tool `sysquakelatextool` to interpret this file. The result is written to a file named `sq-tutorial.lmeout` and, if there are graphics, to EPS files. This file contains the result as LaTeX definitions; for instance, `\sqexpr{5+2}` above creates the following definition:

```
\def\lmefragi{%
7}
```

Here, the name is \lmefrag followed by the expression number in lowercase roman notation (this might change in future versions). Sysquake commands, in addition to producing `sq-tutorial.lme`, also invoke the corresponding definition to insert the result in the final typeset document.

Note that typesetting documents is often already performed by running `latex` twice, or even more times, to resolve cross-references and update the table of contents and the index. So there is nothing new.

3.4   Formatted results

Our first expression could have been computed in plain LaTeX, which gives access to the computation capabilities of TeX. So our next example will use an advanced LME function, `magic`, which produces a magic square in a matrix. Matrices are displayed in mathematical notation.

Since matrices are too large to fit nicely in a paragraph, we put it in an equation:

```
Magic square $M_3$ of order 3 is
\begin{equation}
M_3 = \sqexpr{magic(3)}
\end{equation}
```

3.5   Programs

\sqexpr is very convenient for simple expressions. However, it is not suited to more complicated programs, for two reasons: first, only a single expression can be evaluated which gives a single result. Second, LME syntax may interfer with LaTeX syntax: the backslash, for instance, is a normal character used for premultiplying with the inverse of a matrix in LME, while it is an escape character in LaTeX. We can always escape special characters in LaTeX, but Sysquake for LaTeX's aim is to make things simpler, not overly complicated.

The answer to \sqexpr's limitations is the `sysquake` environment which can contain any kind of code. Let us write another magic square with it:

```
Magic square $M_4$ of order 4 is
\begin{equation}
M_4 =
\begin{sysquake}
magic(4)
\end{sysquake}
\end{equation}
```

The `sysquake` environment can contain any number of lines with any code, unescaped, like `verbatim`. The only invalid character sequence `\end{sysquake}`, which marks the end of Sysquake code. If for an unlikely reason you have to write Sysquake code which contains this sequence of characters, add a space somewhere, or if it is in a string, write a character with an escape sequence, e.g. `\end{sysquak\145}`.

Code in `sysquake` environment does not have the restrictions of expressions written with `\sqexpr`. You can define variables, have conditional executions and loops, use libraries, and more.

Contrary to `\sqexpr` where a single result is formatted for LaTeX, the sysquake environment only inserts any text produced by the code it contains in the LaTeX source. This text is still processed by LaTeX. To format it, you can either do it yourself, inserting LaTeX commands, or use the Sysquake function `sqlxvalue`. In the code fragment above, `magic(4)` without semicolon writes the result as an assignment to variable ans, as raw text:

```
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

In a LaTeX equation, this does not look like a matrix. With `sqlxvalue(magic(4))`, we obtain

```
\ensuremath{\left[ \begin{array}{cccc}16
&2
&3
&13\\
5
&11
&10
&8\\
9
&7
&6
&12\\
4
&14
&15
&1\end{array}\right]}
```

which is rendered more nicely, like with `\sqexpr`.

Short code fragments, such as variable assignments, can be included with the `\sqeval` command. Note that its argument is processed by LaTeX, contrary to the contents of `sysquake` environment.

3.6   Graphics

Creating graphics for LaTeX documents is among the most cumbersome tasks an author is facing. Sysquake for LaTeX makes it as easy as in Sysquake itself.

Graphics are created by adding the size of the EPS image to be created after \begin{sysquake}. The result is inserted automatically in a picture environment of the same size. Package graphicx is used, as well as package epstopdf in PDFLaTeX; they should be imported explicitly with \usepackage.

Here is an example of a function plot:

```
\begin{sysquake}(300,200)
a = 7.2;
fplot(@(x) (x+0.3)^2+a*exp(-3*x^2), [-2,3], 'r')
\end{sysquake}
```

In LaTeX, figures are usually floating. This is done with the figure environment:

```
\begin{figure}
\begin{center}
\begin{sysquake}(300,200)
a = 7.2;
fplot(@(x) (x+0.3)^2+a*exp(-3*x^2), [-2,3], 'r')
\end{sysquake}
\caption{Function plot}
\end{center}
\end{figure}
```

3.7   Functions and libraries

Code fragments may rely on the large set of built-in operators and functions as well as on additional functions stored in libraries. Libraries are text files with a *.lml* suffix which contain the source code of sets of related functions. You can use those which come with Sysquake for LaTeX or write develop new ones, which you can share with Sysquake.

To make functions defined in a library available to Sysquake, a use statement should be placed before any call, in a sysquake environment or a \sqeval command.

```
\begin{verbatim}
\begin{figure}
\begin{center}
\begin{sysquake}(300,200)
use polyhedra;
smallstellateddodecahedron;
```

```
 colormap(blue2yellow2redcm);
 plotoption nomargin;
 plotoption noframe;
 plotoption fill3d;
 \end{sysquake}
 \caption{Great dodecahedron}
 \end{center}
 \end{figure}
 \end{verbatim}
```

Chapter 4

LME Tutorial

The remainder of this chapter introduces LME(TM) (Lightweight Matrix Engine), the interpreter for numerical 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.

4.1   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 ˆ symbol:

```
> 2ˆ5
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
```

4.2   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 func-
tions 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 opera-
tions 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 con-
jugate 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
```

4.3 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
```

4.4 Polynomials

LME handles only numerical values. Therefore, it cannot differentiate functions like $f(x) = sin(e^x)$. However, a class of functions has a paramount importance in numerical computing, the polynomials. Polynomials are weighted sums of powers of a variable, such as $2x^2 + 3x - 5$. LME, which handles only matrices, 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 $(x + 2)(2x^2) + 4x + 5 = 2x^3 + 8x^2 + 13x + 10$.

4.5 Strings

You type strings by bracketing 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
```

4.6 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
```

4.7   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
```

4.8   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 out-
put 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
```

4.9   Local and Global Variables

When a value is assigned to a variable which has never been refer-
enced, a new variable is created. It is visible only in the current con-
text: 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 else-
where. 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 already exists, it is discarded.

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

Chapter 5

LATEX Package Reference

This section documents the LaTeX package `sysquake.sty`. This package permits to embed in LaTeX source files (.tex suffix) fragments of code written in LME, the programming language of Sysquake. All these fragments are evaluated successively in the same context: definitions and variables can be reused across multiple fragments.

5.1 Package options

The package is imported in the main LaTeX source file with \usepackage:

```
\usepackage{sysquake}
```

Several options can be specified in a comma-separated list in square brackets; for example

```
\usepackage[latexingraphics,noseparatelogfile]{sysquake}
```

Here is a list of all supported options.

latexingraphics   By default, text in graphics is embedded in EPS files without further processing. In addition to axis scale labels, graphical commands which display explicit text are `label`, `legend`, `text`, and `title`.

With option `latexingraphics`, all the text which appears in graphics in handled by LaTeX. This has two benefits: first, text font and size mix much better with the document text, for a higher overall quality. Second, LaTeX constructs can appear anywhere in graphics, with greek letters, math symbols and equations. A small drawback is that some characters, such as \ and $, have a special meaning for LaTeX which make the code incompatible with other versions of Sysquake.

noexec   Each time the document is typeset, Sysquake code fragments which have been extracted during the previous iteration are evaluated by a platform-dependant program, `sysquakelatextool`. This program produces output data fragments which are inserted in the document, and EPS files which contain graphics.

With option noexec, `sysquakelatextool` is not executed. The typesetting result will still be correct if all its output files (.lmeout and .eps files) already exist and code fragments, and the size of all text strings displayed in graphics if option `latexingraphics` is used, are not modified.

The main purpose of this option is to let an author make minor modifications on a platform where `sysquakelatextool` is not supported.

noseparatelogfile   By default, standard error channel used when evaluating Sysquake code fragments is redirected to a file whose base name is the same as the main `.tex` file and the suffix is `.lmelog`. This file can be opened after typesetting to check that all code fragments were executed correctly. This is especially important when an error occurs, when a code fragment is replaced with **??sysquake??** in the typeset result.

The `noseparatelogfile` option merges the Sysquake standard error channel with other LaTeX messages.

5.2   Other packages

Package `sysquake.sty` depends on other packages for some tasks. You should also import them with \usepackage if necessary.

graphicx   Used to include EPS graphics generated by Sysquake for LaTeX.

epstopdf   Used by `pdflatex` to convert EPS graphics to PDF. `pdflatex` cannot include directly EPS files.

hyperref   Used for hypertext links created by writing text markup to file descriptor 4.

The preamble of an article which needs all these packages could be as follows. There is no harm in importing superfluous packages.

```
\documentclass{article}
\usepackage[latexingraphics,noseparatelogfile]{sysquake}
\usepackage{graphicx}
\usepackage{epstopdf}
\usepackage{hyperref}
```

```
\begin{document}
...
\end{document}
```

5.3 Commands

Any text output produced by code fragments is inserted in the LaTeX source code. It is subject to further evaluation by LaTeX.

\sqeval

\sqeval{code} evaluates code fragment code, which is processed as LaTeX code before being evaluated by Sysquake for LaTeX; hence characters aimed at Sysquake which have a meaning in LaTeX, such as \ and $, must be escaped.

Examples

Simple value:

```
The result is \sqeval{disp(1+2);}.
```

Passing a LaTeX counter:

```
\newcounter{n}
\setcounter{n}{10}
The square root of \arabic{n} is \sqeval{disp(sqrt(\arabic{n}));}.
```

\sqexpr

\sqexpr{expr} evaluates expression expr, formats it for LaTeX and inserts it in the result. The expression should not be terminated by a semicolon or comma.

Currently, the following types are supported: real or complex double, scalar or 2-dim arrays, and strings. Options set by format are used; scientific notation is displayed with the mathematical notation (a multiplication by 10 to some integer power) instead of the classic computer notation. \sqexpr can be used in text as well as in math mode.

Examples

Scalar value in an equation:

```
Some small number:
$\frac{\pi}{e^{10}} = \sqexpr{pi/exp(10)}$.
```

Magic square of size 5:

```
\[
M_5 = \sqexpr{magic(5)}
\]
```

5.4   Environment

\begin{sysquake} ... \end{sysquake}

Text in the sysquake environment is interpreted verbatim as LME code. All LaTeX constructs are ignored, except for the terminating string \end{sysquake}. Standard output produced by functions like disp and fprintf is inserted in the output, where it is processed by LaTeX.

Remark: the sysquake environment uses the verbatim package. In the *beamer* class, the [fragile] option should be added to frames where it is used. See below for an example.

Example

In the fragment below, the eigenvalues of a magic square are computed and displayed in a loop. Function fprintf is used to display symbols with and sqlxvalue to format the numbers.

```
$R$'s eigenvalues $\lambda_i$ are
\begin{sysquake}
R = magic(3);
lambda = eig(R);
for i = 1:length(lambda)
  fprintf('\\[ \\lambda_%d = %s \\]\n', i, sqlxvalue(lambda(i)));
end
\end{sysquake}
```

\begin{sysquake}(w,h) ... \end{sysquake}

\begin{sysquake}(w,h) starts an environment where sysquake code is evaluated, like \begin{sysquake}. In addition, an EPS file with width w and height h is created with all the output produced by graphical commands the code fragment contains. This EPS is inserted in the LaTeX document in a picture environment with the \includegraphics command of package graphicx.

Example

The fragment below defines a floating figure with the step response of the transfer function 1/(s^3+2s^2+3s+4). We assume the

latexingraphics option of the sysquake package, so that text is processed by LaTeX (see above).

```
\begin{figure}
\begin{center}
\begin{sysquake}(300,200)
step(1, [1,2,3,4]);
label('$t$','$\\psi$');
legend('$\\psi(t)=\\int_0^\\infty g(\\tau)s(t-\\tau){\\rm d}\\tau$');
\end{sysquake}
\caption{Step response}
\end{center}
\end{figure}
```

Here is an example of a plot in a document with class *beamer*. Note the use of the option [fragile] required by the frame environment.

```
\documentclass{beamer}
\usepackage[latexingraphics]{sysquake}
\usepackage{graphicx}
\usepackage{epstopdf}
\begin{document}
\begin{frame}[fragile] \frametitle{Graphics}
\begin{sysquake}(280,200)
fplot(@(x) (x+0.3)^2+7.2*exp(-3*x^2), [-2,3], 'r');
\end{sysquake}
\end{frame}
\end{document}
```

Chapter 6

LME Reference

This chapter describes LME (Lightweight Matrix Engine), the inter-
preter for numerical computing used by Sysquake.

6.1   Program format

Statements

An LME program, or a code fragment typed at a command line, is com-
posed 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 ig-
nored.

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 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.

6.2   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.

In some cases, a simpler syntax can be used when the function has only literal character strings as input arguments. 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.

```
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')
```

This *command syntax* is especially useful for functions which accept well-known options represented as strings, such as `format loose`.

6.3   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.

6.4   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-numerical 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 numerical 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 > uint64 > int64 > uint32 > int32 > uint16 > int16 > uint8 > int8

Functions which manipulate arrays (such as reshape which changes their size or repmat which replicates them) preserve their type.

To convert arrays to numerical, char, or logical arrays, use functions + (unary operator), char, or logical respectively. To convert the numerical 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 numerical 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	unicode UTF-16 code

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).

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 two ways to make structures: 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.

```
a = struct('name', 'Sysquake', ...
           'os', {'Windows', 'Mac OS X', 'Linux'});

b.x = 200;
b.y = 280;
b.radius = 90;

c.s = b;
```

With the assignments above, a.os{3} is 'Linux' and c.s.radius is 90.

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 numerical 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 numerical types in array concatenation. Numerical sets and list or cell arrays os strings cannot be mixed.

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 classes imports the definitions of methods for class polynom and others.

```
use classes;
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
```

6.5   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.

6.6   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

```
> factor({2})

Wrong type (stdlib:primes:164) 'ones'
-> stdlib:factor:174
```

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 `stdlib` is the library name, `primes` is the function name, and 164 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, `primes` was called by `factor` at line 174 in library `stdlib`.

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 numerical 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.

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 numerical 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 structures   Not enough memory to compile that many nested programming structures 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.

6.7 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 Mac OS X and Linux, Sysquake uses the UCS-2 character encoding (i.e. plain 16-bit Unicode characters); and on Windows, the current system character encoding.

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 Mac OS X 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:

`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.

6.8   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: [[http://www.calerga.com]]
* Link with link text: [[http://www.calerga.com | Calerga]]
* Link with styled link text: [[http://www.calerga.com | **Calerga**]]
```

6.9   List of Commands, Functions, and Operators

Programming keywords

```
break          error          repeat
case           for            rethrow
catch          function       return
clear          global         switch
continue       if             try
define         otherwise      until
endfunction    persistent     use
else           private        useifexists
elseif         public         while
```

Programming operators and functions

assert	fun2str	nargout
Variable assignment	help	sandbox
Operator ()	info	sandboxtrust
Operator @	inline	str2fun
builtin	inmem	str2obj
deal	isglobal	subsasgn
dumpvar	iskeyword	subsref
eval	lasterr	variables
exist	lasterror	which
feval	lookfor	
fevalx	nargin	

Arrays

[]	ind2sub	ones
,	interpn	permute
;	ipermute	rand
:	iscell	randn
beginning	isempty	repmat
cat	length	reshape
cell	magic	rot90
cellfun	matrixcol	size
diag	matrixrow	sort
end	meshgrid	squeeze
eye	ndgrid	sub2ind
find	ndims	tril
flipdim	nnz	triu
fliplr	num2cell	unique
flipud	numel	zeros

Strings

```
base64decode    isspace     strcmpi
base64encode    length      strmatch
char            lower       strtok
deblank         md5         strtrim
findstr         setstr      upper
ischar          sha1        utf8decode
isdigit         sprintf     utf8encode
isempty         sscanf
isletter        strcmp
```

Lists

```
{}          islist      num2list
apply       length      replist
join        list2num
isempty     map
```

Structures

```
fieldnames    isstruct       setfield
getfield      orderfields    struct
isfield       rmfield
```

Objects

```
class    isobject
isa      methods
```

Logical operators

```
==      >       |
===     <=      &&
˜=      >=      ||
˜==     ˜       ?
<       &
```

Logical functions

```
all        isfinite     isnumeric
any        isfloat      isscalar
false      isinf        isspace
find       isinteger    isvector
ischar     isletter     logical
isdigit    islogical    true
isempty    isnan        xor
```

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      intmax    pi
false    intmin    realmax
i        j         realmin
inf      nan       true
```

Arithmetic functions

```
+       \          diff
-       .\         kron
*       ^          mod
.*      .^         prod
/       cumprod    rem
./      cumsum     sum
```

Trigonometric and hyperbolic functions

```
acos     asinh     csch
acosh    atan      sec
acot     atan2     sech
acoth    atanh     sin
acsc     cos       sinh
acsch    cosh      tan
asec     cot       tanh
asech    coth
asin     csc
```

Other scalar math functions

abs	erf	log
angle	erfc	log10
beta	erfinv	log1p
betainc	exp	log2
betaln	expm1	nchoosek
cdf	factorial	pdf
conj	gamma	real
diln	gammainc	reallog
ellipam	gammaln	realpow
ellipe	gcd	realsqrt
ellipf	hypot	sign
ellipj	imag	sinc
ellipke	lcm	sqrt

Type conversion functions

cast	fix	swapbytes
ceil	floor	typecast
complex	round	
double	single	

Matrix math functions

'	expm	null
.'	fft	orth
balance	funm	pinv
care	hess	qr
chol	ifft	rank
cond	inv	schur
conv2	linprog	sqrtm
dare	logm	svd
det	lu	trace
dlyap	lyap	
eig	norm	

Geometry functions

```
cross      dot
```

Statistic functions

```
cov          mean         skewness
kurtosis     min          std
max          moment       var
```

Polynomial math functions

```
addpol     filter     polyint
conv       poly       polyval
deconv     polyder    roots
```

Triangulation functions

```
delaunay      griddatan     voronoi
delaunayn     tsearch       voronoin
griddata      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	

Quaternion other 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 numerical functions

fminbnd	ode23	optimset
fminsearch	ode45	quad
fzero	odeset	

Dynamical systems functions

c2dm	dmargin	ss2tf
d2cm	margin	tf2ss

Input/output

```
bwrite      fprintf          redirect
disp        fread            sprintf
error       fscanf           sread
fclose      fseek            sscanf
feof        ftell            swrite
fgetl       fwrite           warning
fgets       matfiledecode
format      matfileencode
```

Files

```
fopen
```

Basic graphics

```
bar         legend       subplotparam
barh        line         subplotpos
circle      plot         subplotprops
contour     plotoption   subplots
fontset     quiver       text
fplot       scale        title
image       scalefactor
label       subplot
```

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
```

Data compression

```
deflate     inflate
```

Image input/output

```
imageread     imageset      imagewrite
```

Date and time

```
cal2julian    julian2cal      toc
clock         tic
```

Unix

```
cd          getenv     sleep
cputime     pwd        unix
```

Sysquake Remote

```
beginfigure     htmlspecialchars     urldecode
endfigure       http                 urlencode
figurelist      httpheader
getclick        httpvars
```

6.10 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 cre-
ated them exits. Global variables must be declared with global in
each functions 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 functions 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 [1,1,1;2,2,2].

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 [1,2,3;1,2,3].

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 when s.type is a list, and for structures when s.type is '.'.

Examples

```
A = [1,2;3,4];
subsasgn(A, struct('type','()','subs',{1,':'}), 999)
   999 999
     3   4
subsasgn(A, struct('type','()','subs',{':',1}), [])
     2
     4
```

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 '.'.

Examples

```
A = [1,2;3,4];
subsref(A, struct('type','()','subs',{1,':'}))
  1 2
```

See also

Operator (), operator {}, subsasgn, beginning, end

6.11   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 keyword is reserved for future use:

```
goto
```

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

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 succes-
sively 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; how-
ever, 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 state-
ments, but where function definitions can help and separate libraries
are not wished (note that libraries cannot contain code outside func-
tion definitions; they do never require endfunction). Like function,
endfunction must be the first element of a line.

Not all of the input and output arguments are necessarily used.
The caller fixes the number of input and output arguments, which can
be retrieved by the called function with nargin and nargout, respec-
tively. The unused input arguments (from nargin+1 to the last one)
are undefined, unless a default value is provided in the function defi-
nition: with the definition function f(x,y=2), y is 2 when f is called
with a single input argument. The unused output arguments (from
nargout+1 to the last one) do not have to be set, but may be.

To redefine an operator (which is especially useful for object meth-
ods; 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 argu-
ments 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 nargin < 2
    b = a;
  end
  if nargin < 3
    ellipse = false;
  end
  S = ellipse ? pi * a * b / 4 : a * b;
```

See also

`return`, `nargin`, `nargout`, `define`, `inline`, `global`, `persistent`

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, `private`, `public`

otherwise

Conditional execution of statements depending on a number or a string.

See also

`switch`

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` may 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 expres-
sion. 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 ex-
pression 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

```
if
```

try

Error recovery.

Syntax

```
try
  ...
end

try
  ...
catch
  ...
end
```

Description

The statements after `try` are executed. If an error occurs, execution is switched to the statements following `try`, if any, or to the statements following end. 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 Mac OS X, 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 may be defined in separate files, called *libraries*. use makes them available in the current context, so that they may be called by the functions or statements which follow. Using a library does not make available functions defined in its sublibraries; however, libraries may be used multiple times, in each context where their functions are referenced.

All use statements are parsed before execution begins. They may 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`

6.12   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

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)
```


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.

   With an output argument, dumpvar stores result into a string and produces no output.

Examples

```
dumpvar(2+3)
  5
a = 6; dumpvar('a', a)
  a = 6;
s = 'abc'; dumpvar('string', s)
  string = 'abc';
```

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, ...)
```

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 may 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.

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'
```

See also

`warning`, `try`, `lasterr`, `lasterror`, `assert`, `fprintf`

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 variable ('variable').

Examples

```
exist('sin')
   true
exist('cos', 'function')
   false
```

See also

```
info
```

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 inline function.

Examples

```
y = feval('sin', 3:5)
   y =
      0.1411 -0.7568 -0.9589
y = feval(inline('sin(2*x)'), 3:5)
   y =
      -0.2794 0.9894 -0.544
```

See also

`builtin`, `eval`, `fevalx`, `apply`, `inline`, operator @

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 by either by its name as a string, a function reference, or an inline 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 `@`

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 functions
info methods
info variables
info global
info persistent
info libraries
info usedlibraries
info threads
str = info
list = 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 list of structures which describe each built-in function, with the following fields:

```
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. 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 list of structures which describe each user-defined function, with the following fields:

```
library   library name
name      function name
loaded    true if loaded
```

   info methods displays the list of methods. With an output argu-
ment, info('methods') gives a list of structures which describe each
method, with the following fields:

```
library   library name
class     class name
name      function name
loaded    true if loaded
```

   info variables displays the list of variables with their type and
size. With an output argument, info('methods') gives a list of struc-
tures which describe each variable, with the following fields:

```
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 persis-
tent variable names.
   info libraries displays the list of all loaded libraries. With an
output argument, info('libraries') gives the list of the library
names.
   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 threads displays the ID of all threads. With an output argu-
ment, info('threads') gives a list of structures which describe each
thread, with the following fields:

```
id            thread ID
totaltime   execution time in seconds
```

   Only the first character of the argument is meaningful; info b is
equivalent to info builtin.

Examples

```
info
  LME 4.5
info b
  abs
  acos
  acosh
  (etc.)
```

```
info v
  ans (1x1 complex)
vars = info('v');
dumpvar(vars)
  {struct('name','ans', ...
    'defined',true), ...
   struct('name','vars', ...
    'defined',false)}
```

See also

inmem, which, exist

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)
```

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 list of arguments which must be supplied when the inline function is called; `paramstruct` is a structure whose fields define fixed parameters.

Anonymous functions 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
```

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})
```

See also

function, operator @, feval, eval

inmem

List of functions loaded in memory.

Syntax

```
inmem
list = 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 a list of structures which describe each user-defined function loaded in memory, with the following fields:

```
library   library name
class     class name ('' for functions)
name      function name
```

See also

info, which

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, 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 other-
wise. 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','if',
   'otherwise','persistent','private','public','repeat',
   'return','switch','try','until','use','useifexists',
   'while'}
```

See also

info, which

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:

```
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`

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 may use the `nargin` function to know how many arguments were passed by the caller to avoid accessing the undefined variables.

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 two other ways to let a function accept a variable number of input arguments: 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 one argument, `nargin(fun)` returns the (maximum) number of input arguments a function accepts. `fun` may 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 written "+"):

```
nargin('plus')
  2
```

See also

nargout, varargin, function

nargout

Number of output arguments.

Syntax

```
n = nargout
n = nargout(fun)
```

Description

A function may 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 may 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)
```

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`.

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)
l = varargin
```

Description

varargin is a special variable which may be used to collect input arguments. In the function declaration, it must be used as the last (or unique) input argument. 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 may 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`, `varargout`, `function`

varargout

Remaining output arguments.

Syntax

```
function (..., varargout) = fun(...)
varargout = ...
```

Description

`varargout` is a special variable which may 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 argu-
ments, 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` may 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 de-
fined 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`). If it is a
variable, it is prefixed with (`var`). If name is neither a function nor a
variable, `which` returns the empty string.

Examples

```
which logspace
   stdlib/logspace
which polynom::plus
   classes/polynom::plus
which sin
   (builtin)/sin
x = 2;
which x
   (var)/x
```

See also

```
info
```

6.13   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`

6.14   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 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 numerical 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 sometimes 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. 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 [ ].

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)) = {[]};
```

See also

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}
v{index}
v{index1, index2, ...}
v{index} = expr
fun(...,v{:},...)
```

Description

A pair of braces is used to define a list or a cell array given by its elements. 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.

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
```

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.

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 numerical 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.

If the second operand is 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 ˆ

Matrix power.

Syntax

```
x ˆ y
M ˆ k
x ˆ M
mpower(x, y)
```

Description

xˆy 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 an integer;

- 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 xˆy. It can be used to redefine this operator for objects.

Examples

```
 2 ˆ 3
    8
 [1,2;3,4] ˆ 2
    7   10
    15 22
 2 ˆ [1,2;3,4]
    10.4827 14.1519
    21.2278 31.7106
```

See also

operator .ˆ, expm

Operator .ˆ

Scalar power.

Syntax

```
 M1 .ˆ M2
 x .ˆ M
 M .ˆ x
 power(x, y)
```

Description

M1.ˆM2 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.ˆy. It can be used to redefine this operator for objects.

Examples

```
 [1,2;3,4].ˆ2
    1   4
    9  16
 [1,2,3].ˆ[5,4,3]
    1  16 27
```

See also

operator ˆ, 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 oper-
ator 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.

Examples

```
˜true
   false
˜[1,0,3,false]
   F  T  F  T
```

See also

operator ˜=, bitcmp

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` state-
ments, 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 or matrices;
their number of rows must be the same, unless one of them is an
empty matrix.

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 rede-
fine this operator for objects.

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 or matrices; their number of columns must
be the same, unless one of them is an empty matrix.

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 rede-
fine 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((z - x1)/(x2 - x1)) <= 1
    add z to the vector
    z = z + step
end
```

This algorithm is robust enough to stop even if x2 is not on the complex straight line defined by x1 and 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 `quad`, but it may also be stored in lists, cell arrays, or

structures. A reference cannot be cast to a double (unlike characters or logical values), nor can it be stored in an array.

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.

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:

```
quad(@sin, 0, pi)
  2
```

Anonymous function:

```
a = 2;
fun = @(x) sin(a * x);
fun(3)
  -0.2794
quad(fun, 0, 2)
  0.8268
```

Without anonymous function, parameter a should be passed as an additional argument after all the input arguments defined for quad, including those which are optional when parameters are missing:

```
quad(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)
```

See also

fun2str, str2fun, inline, feval, apply

6.15   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

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 coor-
dinates 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

$$B(z, w) = \int_0^1 t^{z-1}(1 - t)^{w-1} \, dt$$

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

$$I_x(z, w) = \frac{1}{B(z, w)} \int_0^x t^{z-1}(1 - t)^{w-1} \, dt$$

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

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)
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	a and b

See also

pdf, icdf, 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

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

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

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, dt$$

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

$$\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} \, \mathrm{d}t$$

Example

```
 erfc(1)
   0.1573
```

See also

erf, erfinv

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, erfc

exp

Exponential.

Syntax

```
y = exp(x)
```

Description

`exp(x)` is the exponential of x, i.e. `2.7182818284590446...`ˆx.

Example

```
exp([0,1,0.5j*pi])
  1 2.7183 1j
```

See also

log, expm1, expm, operator .ˆ

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

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

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

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

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \, dt$$

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

$$\text{gammainc}(x, a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t}\, dt$$

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)

See also

`cdf`, `pdf`

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

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`

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, isvector

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, 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, 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

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

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

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

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 .ˆ, 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

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 be-
tween 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

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

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 numerical 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

6.16   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 columns 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=A.

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)
```

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

$$L = \text{eig}(A - BK)$$

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)
```

Description

`conv(v1,v2)` convolves the vectors v1 and v2, giving a vector whose length is length(v1)+length(v2)-1. 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.

Example

```
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
```

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 (nl1,nc1)=size(M1) and (nl2,nc2)=size(M2). With kind='full' (default value), the result M has nl1+nl2-1 lines and nc1+nc2-1 columns. With kind='same', the result M has nl1 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 nl1-nl2+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, 0)
M = cov(data, 1)
```

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 real and symmetric, even if `data` is complex. The diagonal is the variance of each column of `data`. `cov(data,0)` is the same as `cov(data)`.

`cov(data,1)` returns the m-by-m covariance matrix of the n-by-m matrix `data` which contains the whole population.

Example

```
cov([1,2;2,4;3,5])
   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 ambi-
guity, 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

cumprod

Cumulative products.

Syntax

```
M2 = cumprod(M1)
M2 = cumprod(M1,dim)
```

Description

cumprod(M1) returns a matrix M2 of the same size as M1, whose ele-
ments 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).

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

cumsum

Cumulative sums.

Syntax

```
M2 = cumsum(M1)
M2 = cumsum(M1,dim)
```

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).

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
```

See also

sum, diff, cumprod

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:

$$X = A'XA - A'XB(B'XB)^{-1}B'XA + Q$$

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.

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
```

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

```
eig([1,2;3,4])
  -0.3723
  5.3723
(V,D) = eig([1,2;2,1])
 V =
  0.7071 0.7071
  -0.7071 0.7071
 D =
  -1 0
  0 3
[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 ˆ, 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:

```
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 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, inline('(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

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:

$$\min c x$$
$$\text{s.t. } Ax \leq b$$

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:

$$\min c x$$
$$\text{s.t. } Ax \leq b$$
$$x \geq x_{lb}$$
$$x \leq x_{ub}$$

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`, `sum`, `prod`

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).ˆ2))	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))	largest row sum of abs
p	sum(abs(V).ˆp)ˆ(1/p)	invalid
'fro'	sqrt(sum(abs(v).ˆ2))	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)
```

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 negligible singular values).

Example

```
null([1,2,3;1,2,4;1,2,5])
  -0.8944
  0.4472
  8.0581e-17
```

See also

svd, orth

orth

Orthogonalization.

Syntax

```
Q = orth(A)
```

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).

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,e)
```

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. To 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.

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 polyno-
mial `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 poly-
nomial A. Both polynomials are given as vectors of their 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 their 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

Numerical 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 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)
(Qe, Re, e) = qr(A, false)
(Qe, 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 argu-
ments, qr(A) computes the QR decomposition without pivoting, such
that A=Q∗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.

Example

```
 (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,e)
```

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.

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'.

Example

```
(U,T) = schur([1,2;3,4])
  U =
   -0.8246    -0.5658
    0.5658    -0.8246
  T =
   -0.3723    -1
         0     5.3723
eig([1,2;3,4])
  ans =
   -0.3723
    5.3723
```

```
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)ˆ2=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)ˆ2-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
Yˆ2
    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 true, or by length(v) if p is false.
   std(M) gives a row vector which contains the standard deviation of
the columns of M. With a third argument, median(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

6.17 Array Functions

cat

Array concatenation.

Syntax

```
 cat(dim, A1, A2, ...)
```

Description

cat(dim,A1,A2,...) concatenates arrays A1, A2, etc. along dimen-
sion 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 = cell(fun, C, ...)
```

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.

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

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

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

interpn

Interpolation.

Syntax

```
Vi = interpn(x1, ..., xn, V, xi1, ..., xin)
Vi = interpn(x1, ..., xn, V, xi1, ..., xin, method)
```


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 (mono-
tonically 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 in-
dex 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 interpo-
lated value.

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;
```

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 numerical, 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

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 perm(i) in A becomes dimension i in the result.

Examples

```
size(permute(rand(3,4,5), [2,3,1]))
   5 3 4
```

See also

permute, ndims, squeeze

isempty

Test for empty matrices or empty lists.

Syntax

```
b = isempty(M)
b = isempty(list)
```

Description

isempty(obj) gives true if obj is the empty array [], the empty string '', or the empty list {}, and false otherwise.

Examples

```
isempty([])
   true
isempty(0)
   false
isempty('')
   true
isempty({})
   true
isempty({{}})
   false
```

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)
```


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 numerical array or a cell array, matching type of set s.

Example

```
s = {'a','bc','bbb','de'};
m = {'d','a','x';'de','a','z'};
b = ismember(m, s)
   b =
     F T F
     T T F
```


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`

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.

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 dimensions 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 numerical 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).

Example

```
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 i in A becomes dimension perm(i) in the result.

Examples

```
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
M = rand(n)
M = rand(n1, n2, ...)
M = rand([n1, n2, ...])
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 current implementation is based on a scalar 64-bit seed, which theoretically allows 2^{64} different numbers. This seed can be set with the arguments rand('seed', s), where s is a scalar or a vector of two components. rand('seed', s) returns the empty array [] as output argument. To discard it, the statement should be followed by a semicolon.

rand(n), rand(n1,n2,...) and rand([n1,n2,...]) return an n-by-n square matrix or an array of arbitrary size whose elements are pseudo-random numbers uniformly distributed between 0 and 1.

Examples

```
rand
  0.2361
rand(1, 3)
  0.6679 0.8195 0.2786
rand('seed',0);
rand
  0.2361
```

See also

randn

randn

Normally-distributed random number

Syntax

```
x = randn
M = randn(n)
M = randn(n1, n2, ...)
M = randn([n1, n2, ...])
randn('seed', s);
```

Description

randn builds a scalar pseudo-random number chosen from a normal distribution with zero mean and unit variance. The current implementation is based on a scalar 64-bit seed, which theoretically allows 2^{64} different numbers. This seed can be set with the arguments randn('seed', s), where s is a scalar or a vector of two components. The seed is not the same as the seed of rand. randn('seed', s) returns the empty array [] as output argument. To discard it, the statement should be followed by a semicolon.

   randn(n), randn(n1,n2,...) and randn([n1,n2,...]) return an n-by-n square matrix or an array of arbitrary size whose elements are pseudo-random numbers chosen from a normal distribution.

Examples

```
randn
  1.5927
randn(1, 3)
  0.7856 0.6489 -0.8141
randn('seed',0);
```

```
 randn
   1.5927
```

See also

rand

repmat

Replicate an array.

Syntax

```
 A2 = repmat(A1, n)
 A2 = repmat(A1, m, n)
 A2 = repmat(A1, [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. The number of copies is m in the vertical direction, and n in the horizontal direction. The type of the first argument (number, character or logical value) is preserved. With a vector as second argument, the array can be replicated along more than two dimensions.

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, which is read row-wise. 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 line-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.

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 ()

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 numerical, 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 numerical, 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)
```

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.

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 numerical, 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

zeros

Null 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

6.18   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

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

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 Delauny 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

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 Delauny triangulation of 20 random points:

```
x = randn(20, 2);
t = delaunayn(x);
xi = [0, 0; 0, 1];
ix = tsearchn(x, t, xi);
```

See also

delaunayn, voronoin

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 func-
tions. 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 poly-
gons. 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

6.19   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.

`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
int8([50:50:400])
   1x8 int8 array
     50  100 -106  -56   -6   44   94 -112
```

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 ʼuint8ʼ, ʼuint16ʼ, ʼuint32ʼ, ʼuint64ʼ, ʼint8ʼ, ʼint16ʼ, ʼint32ʼ, or ʼint64ʼ (64-bit integers are not supported on all platforms). The result has the corresponding integer type.

Examples

```
intmax
  2147483647int32
intmax(ʼuint16ʼ)
  65535uint16
```

See also

intmin, realmax, 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 ʼuint8ʼ, ʼuint16ʼ, ʼuint32ʼ, ʼuint64ʼ, ʼint8ʼ, ʼint16ʼ, ʼint32ʼ, or ʼint64ʼ (64-bit integers are not supported on all platforms). The result has the corresponding integer type.

Examples

```
intmin
  -2147483648int32
intmin(ʼuint16ʼ)
  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.

6.20 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 if 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
```

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):

```
mfunc = @(X) ...
   X(1) < 1 ...
      ? -(X(1)*exp(-(X(1)*X(2))^2)*X(1)*X(2) - 0.1*X(1)^2) ...
      : inf;
fminsearch(mfunc, [1, 2])
   1     0.7071
```

See also

optimset, fminbnd, 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

ode23 ode45

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) = ode45(fun,[t0,tend],y0)
(t,y) = ode45(fun,[t0,tend],y0,options)
(t,y) = ode45(fun,[t0,tend],y0,options,...)
```

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. The function to be integrated is either spec-
ified 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 sec-
ond 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.)

   Additional input arguments of ode45 are given as additional input
arguments to the function specified by fun. They permit to parame-
terize the ODE.

Example

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 may 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. 6.1):

Van der Pol equation, mu=1
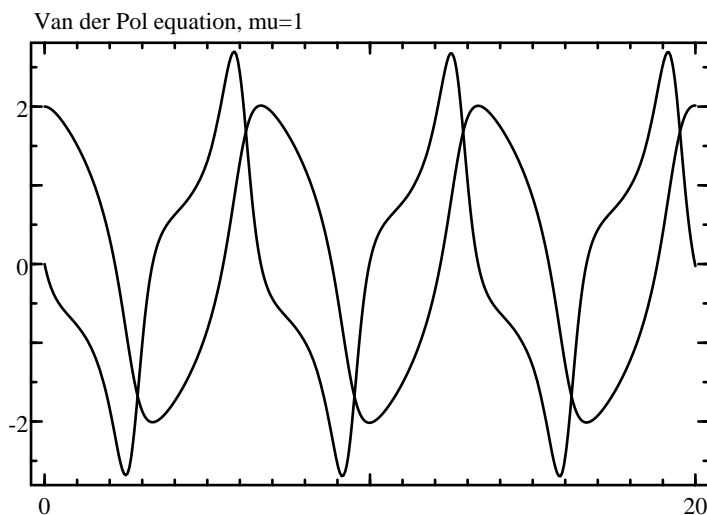
Figure 6.1 Van der Pol equation with $\mu = 1$ integrated with ode45

```
(t,y)=ode45(@f,[0,10],[2;0],[],1);
```

The plot command expects traces along the second dimension; consequently, the result of ode45 should be transposed.

```
plot(t', y');
```

See also

odeset, quad, inline, operator @, expm

odeset

Options for ordinary differential equation integration.

Syntax

```
options = odeset
options = odeset(name1, value1, ...)
options = odeset(options0, name1, value1, ...)
```

Description

odeset(name1,value1,...) creates the option argument used by ode23 and ode45. 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, odeset creates a structure with all the default options. Note that ode23 and ode45 also interpret 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, 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
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
PreArg	{}	list of prepended input arguments
Refine	[] (1, 4 for ode45)	refinement factor
RelTol	1e-3	maximum relative error
Stats	false	statistics display

Time steps and output

Several options control how the time step is tuned during the numerical 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

`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 with fun(1,'abc',t,y), the output function as outfun(1,'abc',tn,yn), and so on.

Examples

Default options

```
odeset
  AbsTol: 1e-6
```
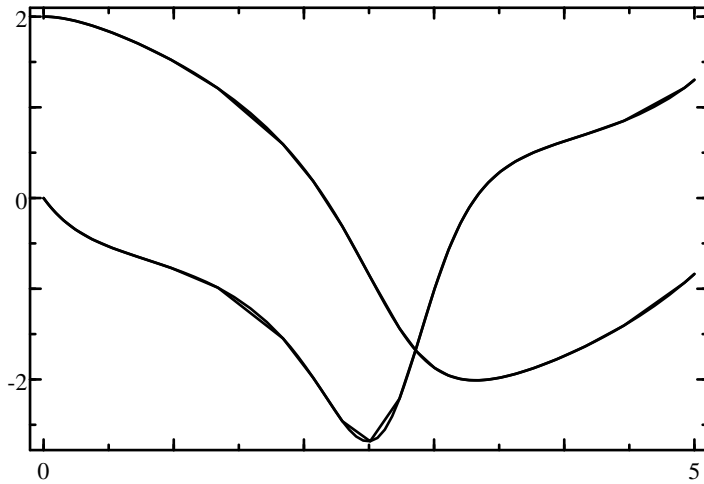
Figure 6.2 Van der Pol equation with Refine set to 1 and 4

```
Events: []
EventTime: []
InitialStep: []
MaxStep: []
MinStep: []
NormControl: false
OnEvent: []
OutputFcn: []
PreArg: {}
Refine: []
RelTol: 1e-3
Stats: false
```

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. 6.2). 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], ...
                 odeset('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], ...
                   odeset('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. 6.3) with

```
 opt = odeset('Events', @ballevents, ...
              'OnEvent', @ballonevent);
 (t, y) = ode45(@ballfun, [0, 5], [2; 0], opt, 1);
 plot(t', y');
```

Bouncing ball integrated with events
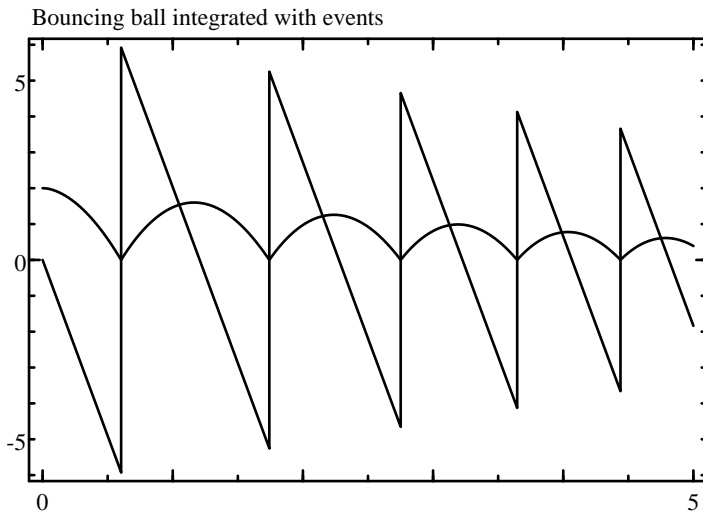
Figure 6.3 Bouncing ball integrated with events

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');
```

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 $u = 0$.

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 integrator and y(2) the output of the relay. Only y(1) is integrated:
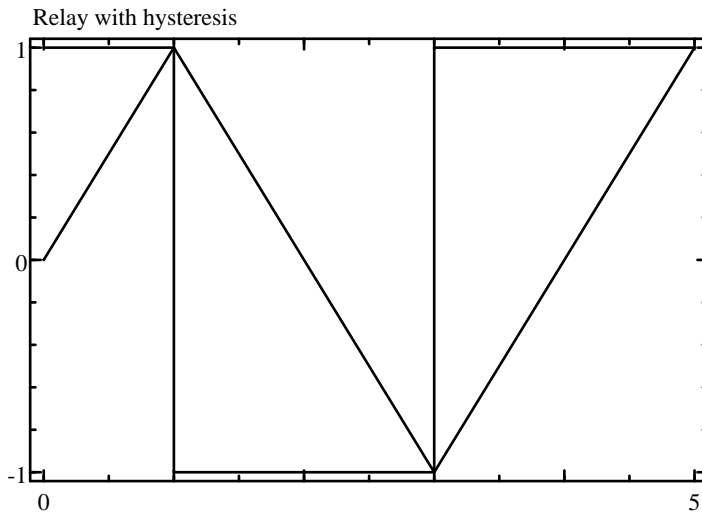
Figure 6.4   Relay with hysteresis integrated with events

```
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. 6.4) with

```
(t, y) = ode45(@relayfun, [0, 5], [0; 1], ...
  odeset('Events', @relayevents, 'OnEvent', @relayonevent));
plot(t', y');
```

See also

ode23, ode45, optimset

optimset

Options for minimization and zero finding.

Syntax

```
options = optimset
options = optimset(name1, value1, ...)
options = optimset(options0, name1, value1, ...)
```


Description

optimset(name1,value1,...) creates the option argument used by fminbnd, fminsearch, and fzero. 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, 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.

   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, odeset

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 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
```

See also

sum, ode45, inline, operator @

6.21   String Functions

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

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 '='; 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 1521.

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

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

findstr

Find a substring in a string.

Syntax

```
pos = findstr(str, sub)
```

Description

findstr(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

```
findstr('ababcdbaaab','ab')
  1 3 10
findstr('ababcdbaaab','ac')
  []
findstr('aaaaaa','aaa')
  1 2 3
```

See also

find, strcmp, strmatch, strtok

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 digits.

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.

Examples

```
isdigit('a123bAB12∗ ')
  F T T T F F F T T F F
```

See also

isletter, isspace, lower, upper, ischar

isletter

Test for letters.

Syntax

```
b = isletter(s)
```

Description

For each character of string s, isletter(s) is true if it is a letter and false otherwise. Letters with diacritical signs are not considered as letters.

Examples

```
isletter('abAB12* ')
  T T T T F F F F
```

See also

isdigit, isspace, lower, upper, ischar

isspace

Test for spaces.

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.

Example

```
isspace('a\tb c\nd')
  0 1 0 1 0 1 0
```

See also

isletter, isdigit, ischar

lower

Convert all uppercase letters to lowercase.

Syntax

```
s2 = lower(s1)
```

Description

lower(s1) converts all the uppercase letters of string s1 to lowercase. Currently, only ASCII letters (without diacritic) are converted.

Example

```
lower('abcABC123')
   abcabc123
```

See also

upper, isletter

md5

Calculate MD5 digest.

Syntax

```
digest = md5(strb)
digest = md5(fd)
```

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.

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

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

Calculate SHA1 digest.

Syntax

```
digest = sha1(strb)
digest = sha1(fd)
```

Description

sha1(strb) calculates the SHA1 digest of strb which represents bi-
nary 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 SHA1 digest of the bytes read from file descriptor fd until the end of the file. The file is left open.

SHA1 digest is an Internet standard described in RFC 3174.

Example

SHA1 digest of the three characters 'a', 'b', and 'c':

```
sha1('abc')
  a9993e364706816aba3e25717850c26c9cd0d89d
```

See also

md5

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','abc',5)
  false
```

See also

strcmpi, operator ===, operator ˜==, operator ==, findstr, 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 ==, `findstr`, `strmatch`

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 `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, findstr

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, findstr, 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

upper

Convert all lowercase letters to lowercase.

Syntax

```
 s2 = upper(s1)
```

Description

upper(s1) converts all the lowercase letters of string s1 to uppercase. Currently, only ASCII letters (without diacritic) are converted.

Example

```
upper('abcABC123')
   ABCABC123
```

See also

lower, isletter

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 one, two, or three bytes of UTF-8 code. The result is a standard character array with a single row. Invalid codes (for example when the beginning of the decoded data does not correspond to a character boundary) are ignored.

See also

utf8encode

utf8encode

Encode a string of Unicode characters using UTF-8.

Syntax

```
b = utf8encode(str)
```

Description

utf8encode(b) encodes the contents of character array str using UTF-8. Each Unicode character in str corresponds to one, two, or three 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);
+str
  1x6 uint16 array
     97     98     99    200   2000 20000
```

See also

utf8decode

6.22   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 to 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 ˆ are defined as a convenience (they are equivalent to .∗, ./, .\, and .ˆ 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.

Compatibility note: native functions for quaternions replace library `quaternion` which defined quaternion scalars and matrices. It is much faster and supports arrays of more than two dimensions; on the other hand, matrix-oriented functions are not supported anymore, and the result of `dumpvar` is not directly compatible.

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

$\begin{bmatrix} v_x\ v_y\ v_z \end{bmatrix}^T$. 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`

6.23   List Functions

apply

Function evaluation with arguments in lists.

Syntax

```
listout = apply(fun, listin)
listout = apply(fun, listin, nargout)
```

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 either by its name as a string or by an inline function.

The number of expected output arguments can be specified with an optional third input argument nargout.

Examples

```
apply('min', {5, 7})
   {5}
apply('size',{magic(3)},2)
   {3, 3}
apply(inline('2*x+3*y','x','y'), {5, 10})
   {40}
```

See also

map, feval, inline, operator @

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'}}
```

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. fun is the name of a function as a string, a function reference, or an inline function.

Examples

```
map('max', {[2,6,4], [7,-1], 1:100})
   {6, 7, 100}
map(inline('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
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

6.24   Structure Functions

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(struct('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(struct('a', 1:3, 'x', 'abc'), 'x')
   true
 isfield(struct('a', 1:3, 'x', 'abc'), 'X')
   false
```

See also

`isstruct`, `struct`

isstruct

Test for a structure object.

Syntax

```
 b = isstruct(obj)
```

Description

`isstruct(obj)` is true if the object `obj` is a structure, false otherwise. Structures are lists whose at least one field has a name.

Examples

```
isstruct(struct('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 = struct('a',123,'c',1:3,'b','123')
  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 re-
moved. If field name does not exist, strctout is the same as strctin.

Example

```
x = rmfield(struct('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`. `setfield` 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`, `getfield`, `rmfield`

struct

Creation of a structure

Syntax

```
 strct = struct(fieldname1, value1, fieldname2, value2, ...)
```

Description

`struct` builds a new structure. 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.

Example

```
 x = struct('a', 1, 'b', 2:5);
 x.a
   1
 x.b
   2 3 4 5
```

See also

isstruct, isfield, rmfield, fieldnames, operator {}

6.25 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, complex, or logical scalar or array
char	character or character array
list	list or structure
inline	inline function
funref	function reference

Examples

```
o1 = class(struct('fld1', 1, 'fld2', rand(4)), 'c1');
o2 = class(struct('fld3', 'abc'), 'c2', o1);
class(o2)
  c2
```

See also

map, isa, isobject, methods

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.

Example

```
 isa(pi,'double')
   true
```

See also

class, isobject, methods

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

6.26   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 vec-
tor) 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 argu-
ment 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 corre-
sponding 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 |

6.27   Dynamical System Functions

This section describes functions related to linear time-invariant dy-
namical 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

```
'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

$$
\begin{aligned}
\frac{dx}{dt}(t) &= A_c x(t) + B_c u(t) \\
y(t) &= C_c x(t) + D_c u(t)
\end{aligned}
$$

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

`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

$$
\begin{aligned}
\frac{dx}{dt}(t) &= A_c x(t) + B_c u(t) \\
y(t) &= C_c x(t) + D_c u(t)
\end{aligned}
$$

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`

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 null, and one less if D is null.

Example

```
(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 trans-
fer 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

6.28   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

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)
```

Description

disp(obj) displays the object obj. Command format may be used to
control how numbers are formatted.

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.

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.

See also

`fgetl, fscanf`

format

Default output format.

Syntax

```
format
format short
format short e
format short g
format long
format long e
format long g
format bank
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 g	general format with up to 4 digits
long	fixed format with 0 or 15 digits
long e	exponential format with 15 digits
long g	general format with up to 15 digits
bank	fixed format with 2 digits (for currencies)
+	'+', '-' 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 '+' 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

fprintf

Formatted output.

Syntax

```
n = fprintf(fd,format,a,b,...)
n = fprintf(format,a,b,...)
```

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. 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
```

Caveat

Same limitations as `sprintf`

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 bytes 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.

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).

See also

fwrite, sread

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:

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

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 out-
put 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, ...)
```

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	Meaning
-	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 or the number of characters for a string (one or more decimal digits; by default, it is 4 for a number or the length of the string for a string), and 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
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
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.

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

```
sprintf('%d %.2f %.2e %.2E %.2g',pi∗ones(1,5))
  3 3.14 3.14e0 3.14E0 3.14
sprintf('%.1k ', 0.001, 0.11, 111, 1000)
  1e-3 0.11 111 1e3
sprintf('∗%8.3f∗%8.6s∗%-8.6s∗',pi,'abcdefgh','abcdefgh')
  ∗   3.142∗  abcdef∗abcdef  ∗
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_
sprintf('%∗.∗f', 15, 7, pi)
      3.1415927
```

Caveat

Exotic formats unsupported.

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

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(fd, data) stores the contents of the matrix data in a string. The third 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

6.29   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
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

See also

`fclose`

6.30 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 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

`tic`, `toc`

tic

Start stopwatch.

Syntax

```
tic
```

Description

`tic` resets the stopwatch. Typically, `tic` is used once at the beginning of the block to be timed.

See also

`toc`, `clock`

toc

Elapsed time of stopwatch.

Syntax

```
elapsed_time = toc
```

Description

`toc` gets the time elapsed since the last execution of `tic`. Typically, `toc` is used at the end of the block of statements to be timed.

On multi-tasking operating systems like Windows, Mac OS X and Unix, `toc` measures only the time spent in the LME application. Other processes do not have a large impact. For instance, typing `tic` at the command-line prompt, waiting 5 seconds, and typing `toc` will show a value much smaller than 5.

Example

```
tic; x = eig(rand(200)); toc
  0.3046
```

See also

tic, clock

6.31   MAT-files

matfiledecode

Decode the contents of a MATLAB MAT-file.

Syntax

```
var = matfiledecode(fd)
var = matfiledecode(data)
var = matfiledecode(..., ignoreErr)
```

Description

matfiledecode(fd) reads data from file descriptor fd until the end
of the file. The data must be the contents of a MATLAB-compatible
MAT-file. They are made of 8-bit bytes; no text conversion must take
place. The result is a structure whose fields have the name and the
contents of the variables saved in the MAT-file.

   Instead of a file descriptor, the data can be provided directly as the
argument. In that case, the argument data must be an array, which
can be read from the actual file with fread or obtained from a network
connection.

   Only arrays are supported (scalar, matrices, arrays of more than
two dimensions, real or complex, numerical, logical or char). A second
input argument can be used to specify how to handle data of unsup-
ported types: with false (default value), unsupported types cause an
error; with true, they are ignored.

Example

```
fd = fopen('data.mat');
s = matfiledecode(fd);
fclose(fd);
s
  s =
    x: real 1x1024
    y: real 1x1024
```

See also

matfileencode

matfileencode

Encode the contents of a MATLAB MAT-file.

Syntax

```
matfileencode(fd, s)
matfileencode(s)
```

Description

`matfileencode(fd,s)` writes the contents of structure s to file descriptor fd as a MATLAB-compatible MAT-file. Each field of s corresponds to a separate variable in the MAT-file. With one argument, `matfileencode(s)` writes to the standard output (which should be uncommon since MAT-files contain non-printable bytes).

Only arrays are supported (scalar, matrices, arrays of more than two dimensions, real or complex, numerical, logical or char).

Examples

```
s.a = 123;
s.b = 'abc';
fd = fopen('data.mat', 'wb');
matfileencode(fd, s);
fclose(fd);
```

Function `variables` can be used to save all variables:

```
v = variables;
fd = fopen('var.mat', 'wb');
matfileencode(fd, v);
fclose(fd);
```

See also

`matfiledecode`, `variables`

6.32 Shell

This section describes functions related to the Unix or Windows shell. They are available only on Windows and on Unix (or Unix-like) systems, such as Mac OS X.

The versions for Unix and Windows have significant differences:

– Most functions described here are defined on both Unix and Windows, to avoid errors when loading functions which contain conditional code for Unix and Windows. Functions with an empty implementation return the error "Not supported". Table below gives the status of all commands.

Command	Unix	Windows
cd	supported	supported
cputime	supported	undefined
dir	supported	supported
dos	not supported	supported
getenv	supported	supported
ispc	false	true
pwd	supported	supported
setenv	supported	not supported
sleep	supported	supported
unix	supported	not supported
unsetenv	supported	not supported

– On Windows, some of the functionality of unix is provided by dos. The main difference is that dos does not give any output, except for the status code of the command.

cd

Set or get the current directory.

Syntax

```
cd(str)
str = cd
```

Description

cd(str) changes the current directory. Without input argument, cd gives the current directory, like pwd.

The current directory is the root directory where files specified by relative paths are searched by functions like fopen and dir. LME libraries are specified by name, not by path; the places where they are searched is specified by a list of search paths, typically specified with a path command or a dialog box in a graphical user interface.

Example

```
cd('/usr/include');
```

See also

pwd, dir

cputime

Amount of processing time since the beginning of the process.

Syntax

```
 t = cputime
```

Description

cputime gives the amount of processing time spent since the application has been launched.

See also

clock

dir

List of files and directories.

Syntax

```
 dir
 dir(path)
 r = dir
 r = dir(path)
```

Description

dir displays the list of files and directories in the current path. A string input argument can specify the path.

With an output argument, dir gives a list of files and directories. Each element is a structure with the following fields:

Name	Value
name	file name or directory name
isdir	false for files, true for directories
altname	alternate name (Windows only)

Field isdir may be missing on some patforms. On Windows, altname contains the DOS-compatible name (a.k.a. "8.3") if it exists, or an empty string otherwise.

See also

cd, pwd

dos

Execute a command under Windows.

Syntax

```
status = dos(str)
```

Description

dos(str) executes a command with the system Windows function. No
input can be provided, and output is discarded. dos returns the status
code of the command, which is normally 0 for successful execution.

Example

```
dos('del C:/tmp/data.txt');
```

See also

unix

getenv

Get the value of an environment variable.

Syntax

```
value = getenv(name)
```

Description

getenv(name) gives the value of the environment variable of the spec-
ified name. If no such environment variable exists, getenv returns an
empty string.

Example

```
user = getenv('USER');
```

See also

setenv, unsetenv

ispc

Check whether platform is a PC.

Syntax

```
 b = ispc
```

Description

ispc returns true on Windows (where dos is defined), false on other platforms.

See also

isunix, dos, unix

isunix

Check whether platform runs under unix.

Syntax

```
 b = isunix
```

Description

isunix returns true on unix (or unix-like) platforms (where unix is defined), false on other platforms.

See also

ispc, dos, unix

pwd

Get the current directory.

Syntax

```
 str = pwd
```

Description

pwd ("print working directory") gives the current directory. It has the same effect as `cd` without input argument.

See also

`cd`, `dir`

setenv

Set the value of an environment variable.

Syntax

```
setenv(name, value)
setenv(name)
```

Description

`setenv(name,value)` sets the value of the environment variable of the specified name. Both arguments are strings. If no such environment variable exists, it is created.

   With a single input argument, `setenv` creates an empty environment variable (or remove the value of an exisisting environment variable).

   Environment variables are defined in the context of the application; they can be accessed in the application or in processes it launches. Environment variables of the calling process (command shell, for instance) are not changed.

   `setenv` is not defined for Windows.

Example

```
setenv('CONTROLDEBUG', '1');
```

See also

`getenv`, `unsetenv`

sleep

Suspend execution for a specified amount of time.

Syntax

```
sleep(t)
```

Description

`sleep(t)` suspend execution during `t` seconds with a resolution of a microsecond.

Example

```
sleep(1e-3);
```

unix

Execute a Unix command.

Syntax

```
unix(str)
```

Description

`unix(str)` executes a command with the default shell. No input can be provided, and output is directed to the standard output of LME.

Examples

```
unix ls
unix('cc -o calc calc.c; ./calc')
```

See also

dos

unsetenv

Remove an environment variable.

Syntax

```
unsetenv(name)
```

Description

unsetenv(name) removes the definition of the environment variable of the specified name. Argument is a string. If no such environment variable exists, unsetenv does nothing.

Environment variables are defined in the context of the application; they can be accessed in the application or in processes it launches. Environment variables of the calling process (command shell, for instance) are not changed.

unsetenv is not defined for Windows.

Example

```
unsetenv('CONTROLDEBUG');
```

See also

getenv, setenv

6.33 Graphics

LME provides 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.

Here is the list of these commands:

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
```

Scaling and labels

```
label     plotoption    scalefactor
legend    scale         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
```

6.34   Remarks on graphics

Most 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. The possible values 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
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 '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]);
```

6.35 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, 1);
```

See also

`plot`, `image`

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.

Example

```
area(rand(20,10))
```

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.
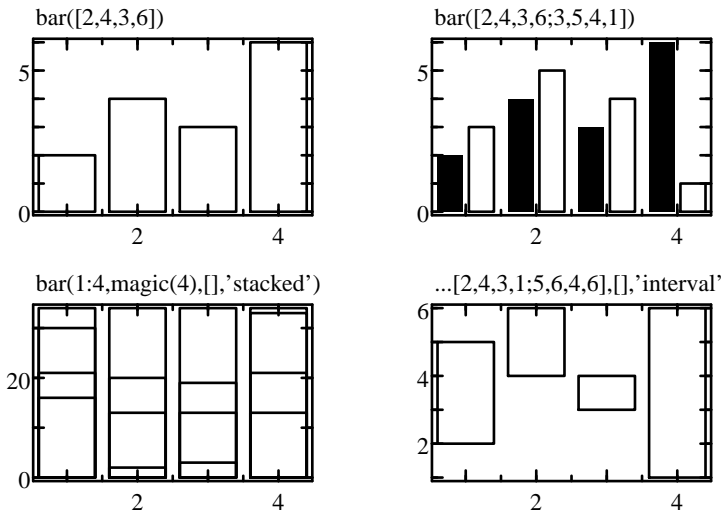
Figure 6.5   Example of bar with different options

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. 6.5):

```
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');
```

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 de-
scription 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 axis 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.

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. 6.6):

```
(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]);
```
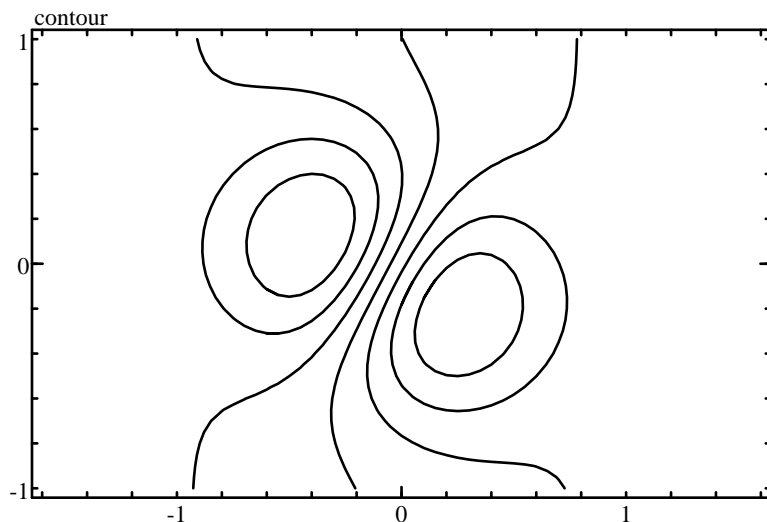
contour

Figure 6.6   Example of `contour`

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`

fontset

Options for fonts.

Syntax

```
options = fontset
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. 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.

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
```

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 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);
```

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(..., style)
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), or by a uint8 matrix in the range 0 (black) to 255 (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, or between 0 (black) to 255 (maximum intensity) with uint8 values.

   The position is defined by the the minimum and maximum coordinates along the horizontal and vertical axis. The raster image is scaled to fit. The first line of the matrix or matrices is displayed at the top. If `style` 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 `style` is '1', and the bicubic method if `style` is '3'.

Examples

Two ways to display a table of 10-by-10 random color cells (see Fig. 6.7):

```
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');
```

See also

`contour`, `quiver`
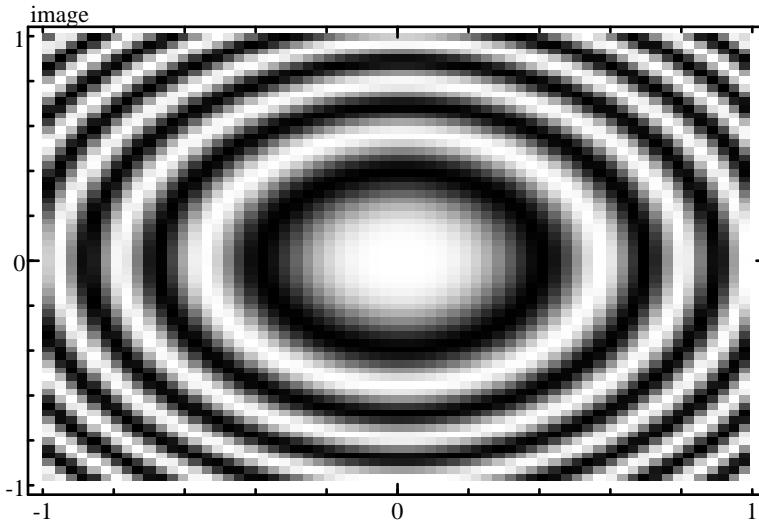
image

Figure 6.7   Example of image

label

Plot labels.

Syntax

```
label(label_x)
label(label_x, label_y)
```

Description

`label(label_x, label_y)` displays labels for the X and Y axis. Its arguments are strings. The label for the Y axis may be omitted.

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;
```

See also

`text`, `legend`, `title`
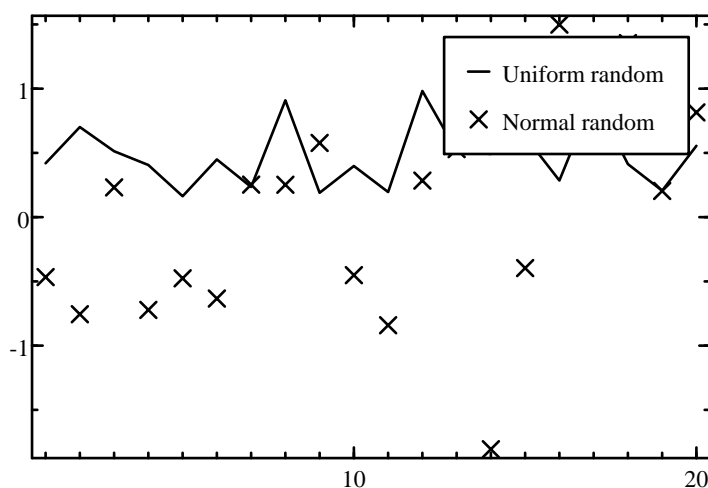
Figure 6.8 Example of legend

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.

   With a single input argument, legend(str) uses the default style 'k'.

Example

Legend for two traces (see Fig. 6.8).

```
plot(1:20, [rand(1,20); randn(1,20)], '_x');
legend('Uniform random\nNormal random', '_x');
```

See also

label, title

line

Plot lines.

Syntax

```
line(A, b)
line(A, b, style)
line(A, b, style, id)
```

Description

line displays one or several straight line(s). Each line is defined by an equation 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; each row corresponds to a different line. The second argument is a column vector which contains the coefficients b. If one of these 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 lines at y=2 and y=3:

```
line([0,1],[2;3],'b',1)
```

See also

plot, circle

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, X 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`

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');
```

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, 'rf', 1);  % filled red triangle with 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(str)
```

Description

`plotoption` sets the initial value of one of the plot options the user can change. Its single argument, a character string, can take one of the following values.

`'frame'`   Rectangular frame with tick marks and a white background around the plot.

`'noframe'`   No frame, no tickmarks, no white background.

'label'   Subplot name above the frame.

'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 (space for title and labels).

'nomargin'   No margin.

'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 axis.

'noxygrid'   No grid for the x and y axis.

'grid'   Normal details for grids displayed by sgrid, zgrid, etc.

'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.

Example

Display of a photographic image without frame:

```
plotoption noframe;
image(photo);
```

See also

scale, legend

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 axis, 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
```

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 axis or only for the x axis in 2D graphics, or for x, y and z axis 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 axis. 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])`.

'lock'　　See below.

'linlin'　　Linear scale for both axis.

'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 axis.

'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.

This last 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 loga-
rithmic or linear scale, and enabling a logarithmic scale always permits
to choose a linear scale. The 'equal' option cannot be combined with
anything else.

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 axis
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 numerical 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 an output argument, `scale` returns the current scale as a vec-
tor `[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 x and y 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.

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 real positive numbers to set separately the x axis and the y axis, or a real positive scalar to set the same factor for both axis. 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

text

Display text in a figure.

Syntax

```
text(x, y, string)
text(x, y, string, justification)
text(..., font)
```

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.

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);
```

See also

label, fontset, sprintf

title

Subplot title.

Syntax

```
title(string)
```

Description

title(string) sets or changes the title of the current subplot.

See also

label, legend, text, sprintf

6.36   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).

6.37   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

6.38   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.

6.39   Functions

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

axisbounds

Bounds of display area along each axis.

Syntax

```
axisbounds(bnds)
bnds = axisbounds
```

Description

axisbounds(bnds) sets the bounds of displayed box along each axis.
Argument bnds is an array of 2, 4, or 6 elements. With 2 or 4 elements,
axisbounds has the same effect as scale.

   ...
   Merge with scale ???

See also

scale, daspect

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(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 equations of the form $a_1 x + a_2 y + a_3 z = b$. 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.

The optional third and fourth arguments are the same as for all graphical commands.

Example

Vertical line at x=5, y=6:

```
line3([1,0;0,1],[5;6])
```

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.

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.

See also

line, 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 poly-
gons. Color of each polygon is mapped linearly from the z coordinate
of the center of gravity of its vertices to the color map. Each poly-
gon 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 ver-
tex 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 sur-
face 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. Ar-
guments 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 miss-
ing, 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.

See also

plot3, mesh, plotpoly

6.40 Graphical Functions 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. 6.9):

```
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

bodephase

Phase Bode diagram for a continuous-time system.

scale('logdb'); bodemag(1, [1,2,3,4])
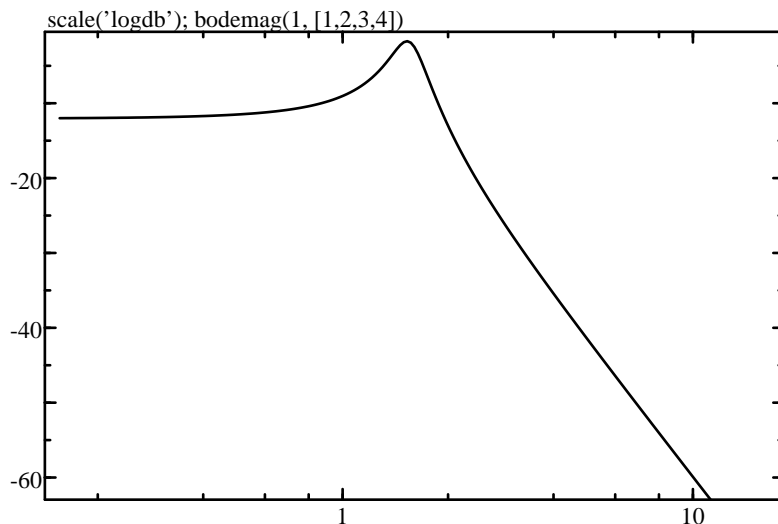
Figure 6.9 `scale('logdb'); bodemag(1, [1,2,3,4])`

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.

`bodemag(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

scale('loglin'); bodephase(1, [1,2,3,4])
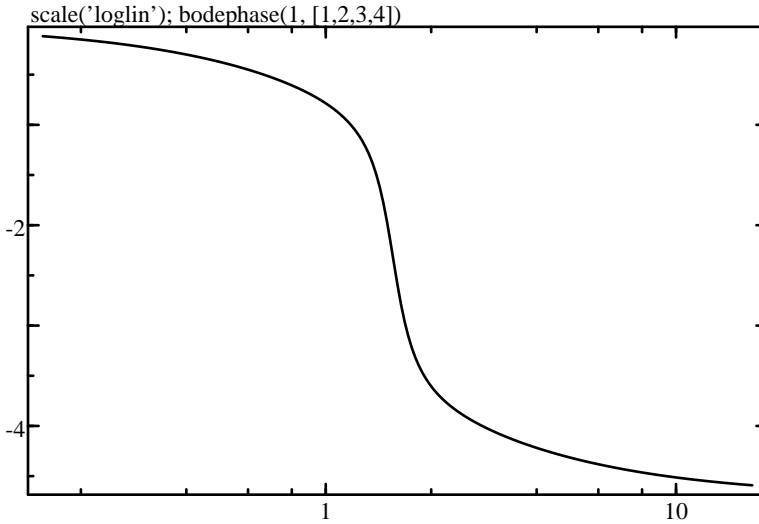
Figure 6.10 `scale('loglin'); bodephase(1, [1,2,3,4])`

$$
\begin{aligned}
j\omega X(j\omega) &= A_c X(j\omega) + B_c U(j\omega) \\
Y(j\omega) &= C_c X(j\omega) + D_c U(j\omega)
\end{aligned}
$$

With output arguments, bodephase gives the phase and the frequency
as column vectors. No display is produced.

Example

Green plot for $\arg(1/(s^3 + 2s^2 + 3s + 4))$, with $s = j\omega$ (see Fig. 6.10):

```
bodephase(1, [1, 2, 3, 4], 'g');
```

See also

bodemag, dbodephase, responseset

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

$$
\begin{aligned}
zX(z) &= A_d X(z) + B_d U(z) \\
Y(z) &= C_d X(z) + D_d U(z)
\end{aligned}
$$

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

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 spec-
ified 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 fre-
quency 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

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

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, Bd, Cd, Dd, Ts, x0, 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.

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`

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.

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. 6.11):

```
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

dnichols

Nichols diagram of a discrete-time system.
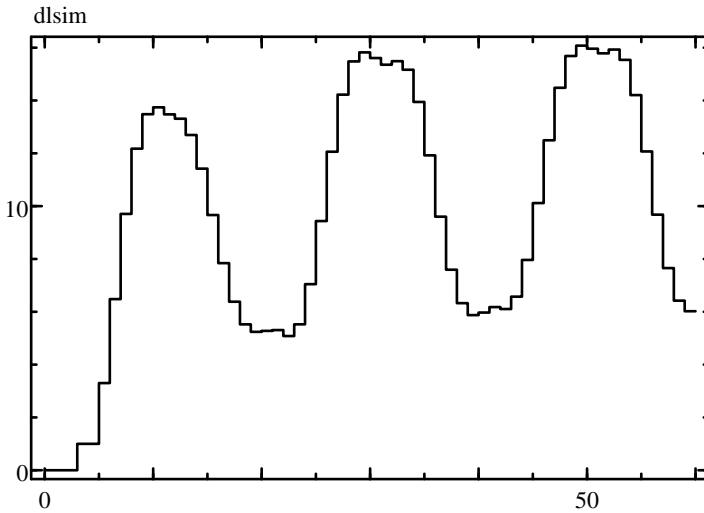
dlsim

Figure 6.11 dlsim(1, poly([0.9,0.7+0.6j,0.7-0.6j]), u)

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 ar-

guments `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

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.
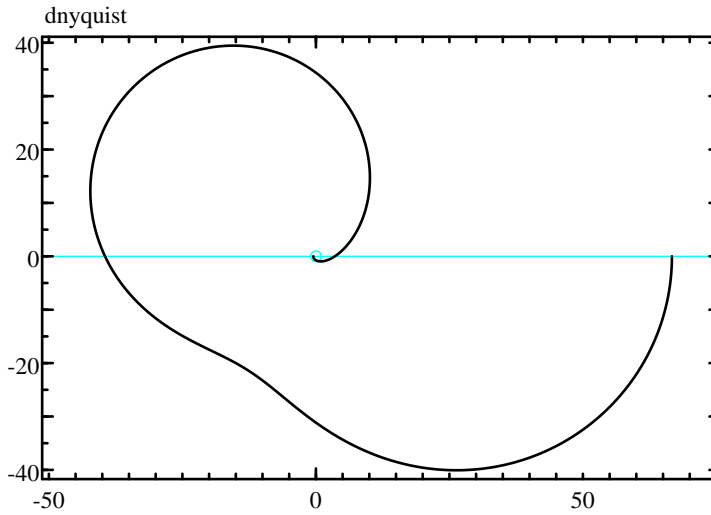
dnyquist

Figure 6.12 dnyquist(3, poly([0.9,0.7+0.6j,0.7-0.6j]))

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) (see Fig. 6.12)

```
scale equal;
hgrid;
dnyquist(3, poly([0.9,0.7+0.6j,0.7-0.6j]))
```

See also

nyquist, hgrid, dnichols, responseset

dsigma

Singular value plot for a discrete-time state-space model.

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

$$
\begin{aligned}
zX(z) &= A_d X(z) + B_d U(z) \\
Y(z) &= C_d X(z) + D_d U(z)
\end{aligned}
$$

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

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.

Example

Step response of a discrete-time third-order system (see Fig. 6.13):

```
dstep(1, poly([0.9,0.7+0.6j,0.7-0.6j]), 1, 'g');
```

See also

dimpulse, dlsim, step, hstep, responseset

erlocus

Root locus of a polynomial with coefficients bounded by an ellipsoid.

Syntax

```
erlocus(C0, P)
erlocus(C0, P, sizes, colors)
```

dstep(1, poly([0.9,0.7+0.6j,0.7-0.6j]), 0.2, 's')
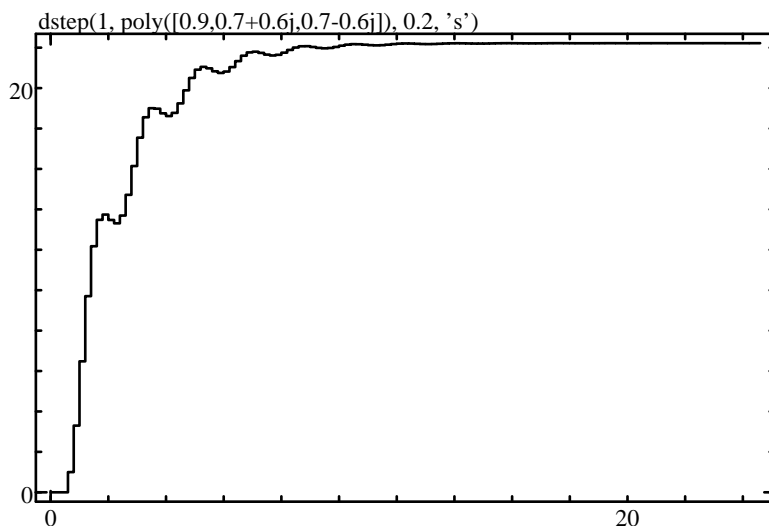
Figure 6.13 `dstep(1,poly([.9,.7+.6j,.7-.6j]),0.2,'s')`

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.

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. 6.14).

```
scale('equal', [-2,2,-2,2]);
erlocus(poly([0.8, 0.7+0.6j, 0.7-0.6j]), eye(3));
zgrid;
```
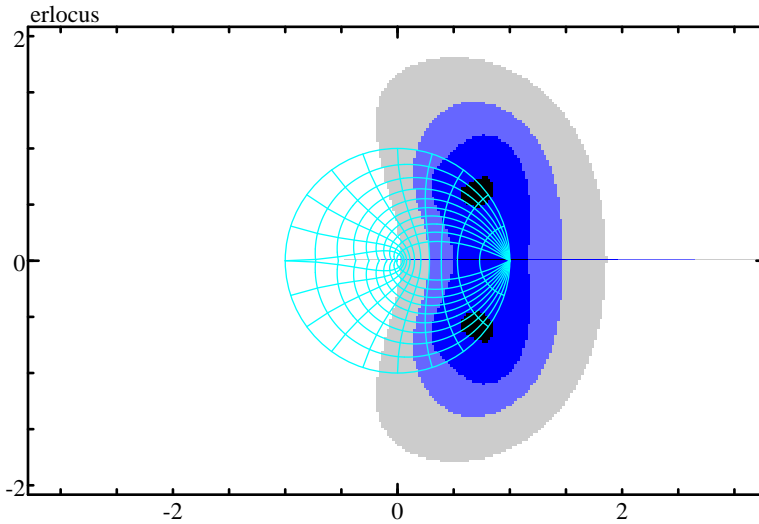
Figure 6.14 `erlocus(poly([.8,.7+.6j,.7-.6j]), eye(3))`

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.

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 0.2, 0.5, 0.8, 1, 1/0.8, 2, and 5; and to a closed-loop phase of plus or minus 0, 10, 20, 30, 45, 60, and 75 degrees.

scale('equal', [-2,2,-2,2]); hgrid;
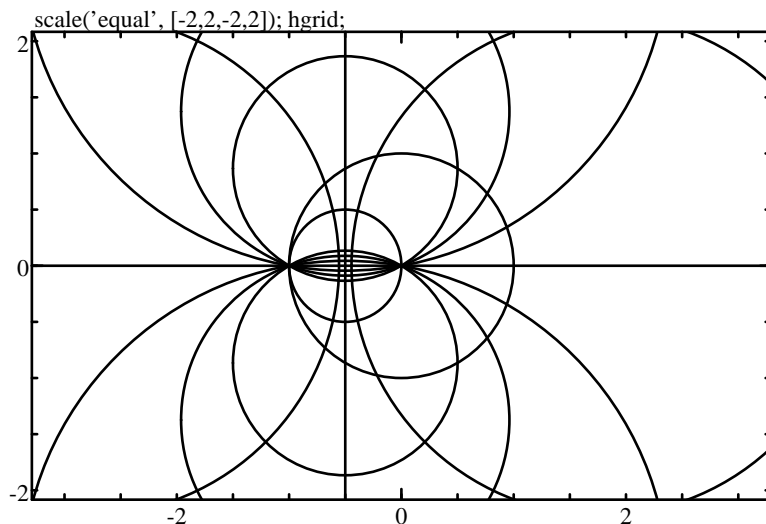
Figure 6.15 `scale('equal', [-2,2,-2,2]); hgrid`

Example

Hall chart grid with a Nyquist diagram (see Fig. 6.15):

```
scale('equal', [-1.5, 1.5, -1.5, 1.5]);
hgrid;
nyquist(20, poly([-1,-2+1j,-2-1j]))
```

See also

`ngrid`, `nyquist`, `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)
```

scale([0,5]); hstep(1,[1,-0.5],1,1,[1,5])
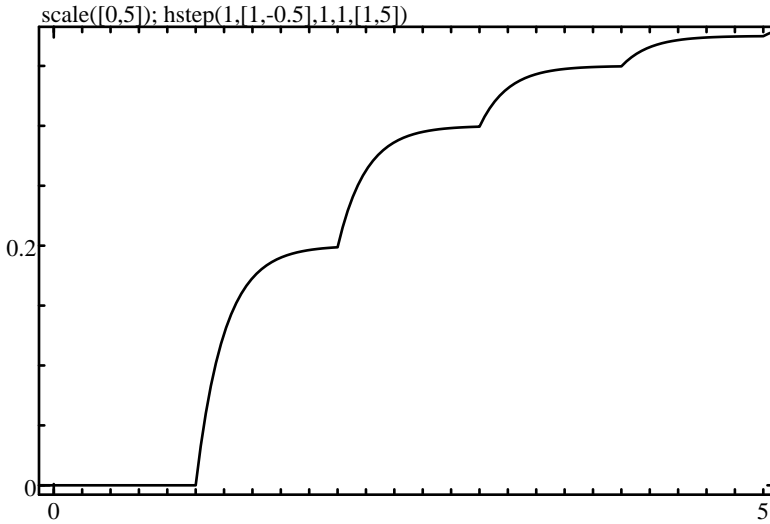
Figure 6.16 `scale([0,5]); hstep(1,[1,-0.5],1,1,[1,5])`

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 be used to 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. 6.16):

```
% 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

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

$$
\frac{\mathrm{d}x}{\mathrm{d}t}(t) = A_c x(t) + B_c u(t)
$$
$$
y(t) = C_c x(t) + D_c u(t)
$$

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

initial

Time response plot for a continuous-time state-space model with initial conditions.

Syntax

```
initial(Ac, Bc, Cc, Dc, 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

$$\frac{dx}{dt}(t) = A_c x(t) + B_c u(t)$$
$$y(t) = C_c x(t) + D_c u(t)$$

where u(t) is null. The state-space model may have a scalar or vector output.

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.
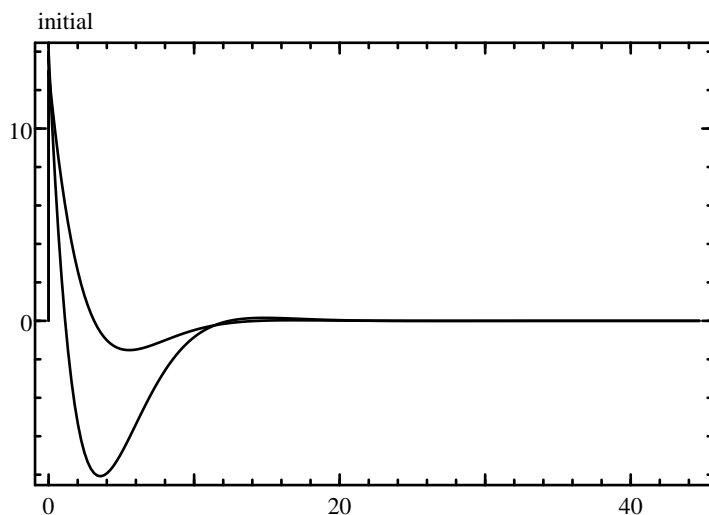
Figure 6.17 Example of `initial`

Example

Response of a continuous-time system whose initial state is [5;3] (see Fig. 6.17):

```
initial([-0.3,0.1;-0.8,-0.4],[2;3],[1,3;2,1],[2;1],[5;3])
```

See also

`dinitial, impulse, responseset`

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

$$
\begin{aligned}
\frac{dx}{dt}(t) &= A_c x(t) + B_c u(t) \\
y(t) &= C_c x(t) + D_c u(t)
\end{aligned}
$$

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. 6.18):

```
t = [0, 10, 20, 30, 50];
u = [1, 1, 0, 1, 1];
lsim(1, [1, 2, 3, 4], u, t, 'b');
```
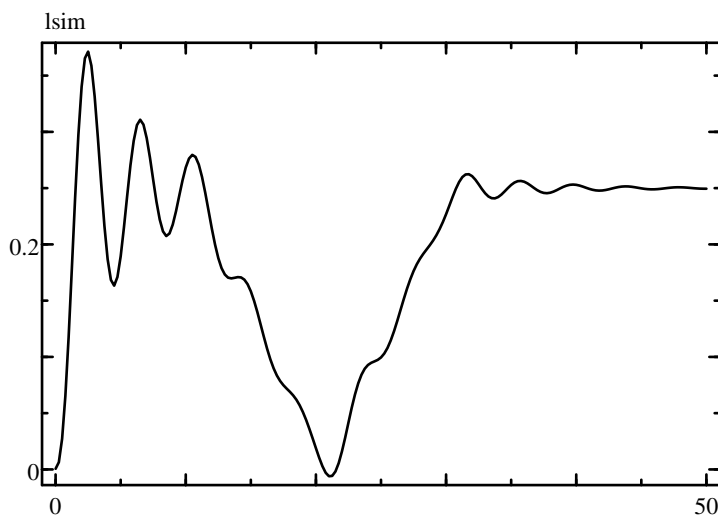
Figure 6.18 `lsim(1, [1,2,3,4], u, t)`

See also

`step`, `impulse`, `initial`, `dlsim`

ngrid

Nichols chart grid.

Syntax

```
ngrid
ngrid(style)
```

Description

`ngrid` plots a Nichols chart in the complex plane of the Nichols diagram (see Fig. 6.19). The Nichols chart is a set of lines which correspond to the same magnitude of the closed-loop frequency response. The optional argument specifies the style.

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.

scale('lindb',[-4*pi,0,0.01,100]); ngrid
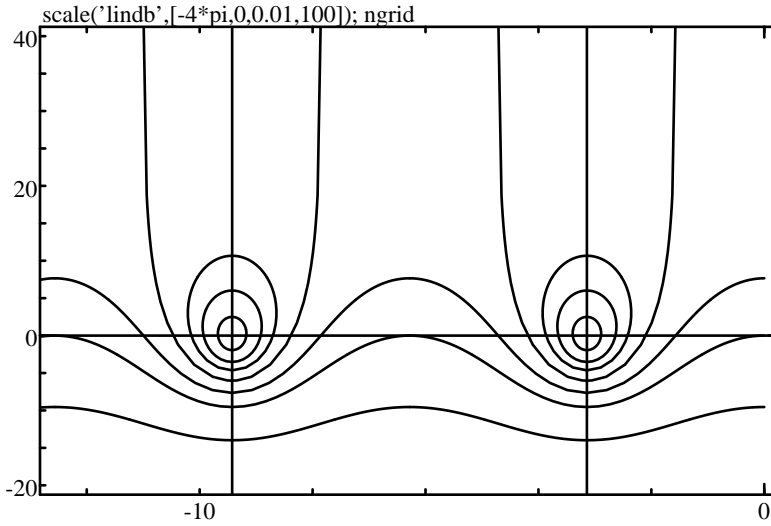
Figure 6.19 `scale('lindb',[-4*pi,0,0.01,100]); ngrid`

Example

```
ngrid;
nichols(7, 1:3);
```

See also

hgrid, nichols, 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(...)
```

scale('lindb'); nichols(1,1:4)
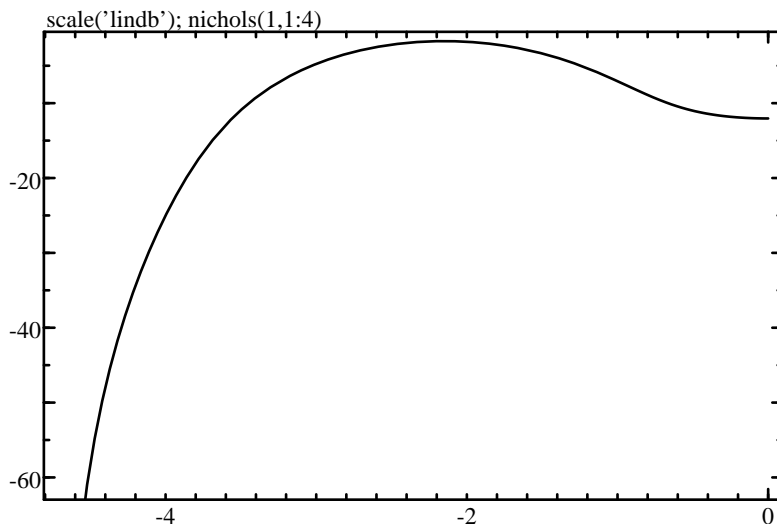
Figure 6.20 `scale('lindb'); nichols(1,1:4)`

Description

`nichols(numc,denc)` displays the Nichols diagram of the continuous-time transfer function given by polynomials numc and denc. In continuous 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 $\_q$.

Example

Nichols diagram of a third-order system (see Fig. 6.20):

```
scale('lindb');
ngrid;
nichols(20,poly([-1,-2+1j,-2-1j]))
```

See also

dnichols, ngrid, nyquist, responseset

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.

scale('equal'); nyquist(1,1:3)
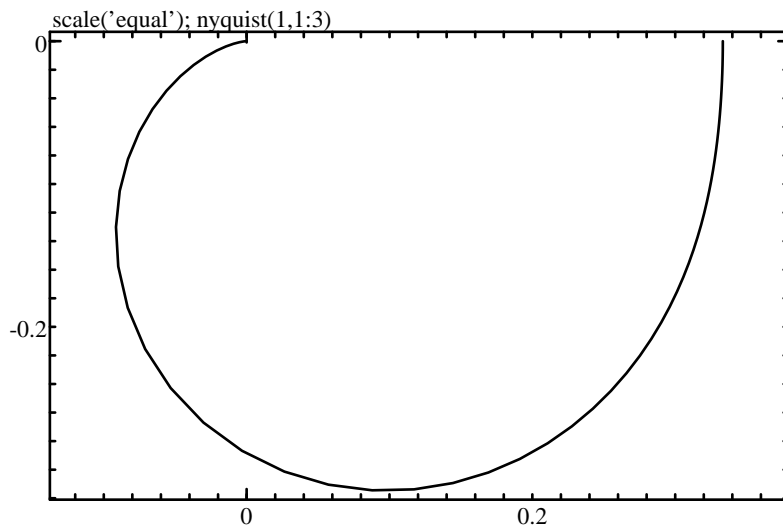
Figure 6.21 `scale equal; nyquist(1,[1,2,3])`

Example

Nyquist diagram of a third-order system (see Fig. 6.21):

```
scale equal;
hgrid;
nyquist(20, poly([-1,-2+1j,-2-1j]))
```

See also

dnyquist, hgrid, nichols, responseset

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.

Example

```
scale equal;
plotroots(den,'x');
plotroots(num,'o');
```

See also

`rlocus`, `erlocus`, `sgrid`, `zgrid`, `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 numerical 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
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 `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)
```

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 $\text{num} + k \cdot \text{den}$, with $k \geq 0$. `rlocus` requires a causal system with real coefficients, i.e. `length(den)>=length(num)`. 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.

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. 6.22):

```
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.
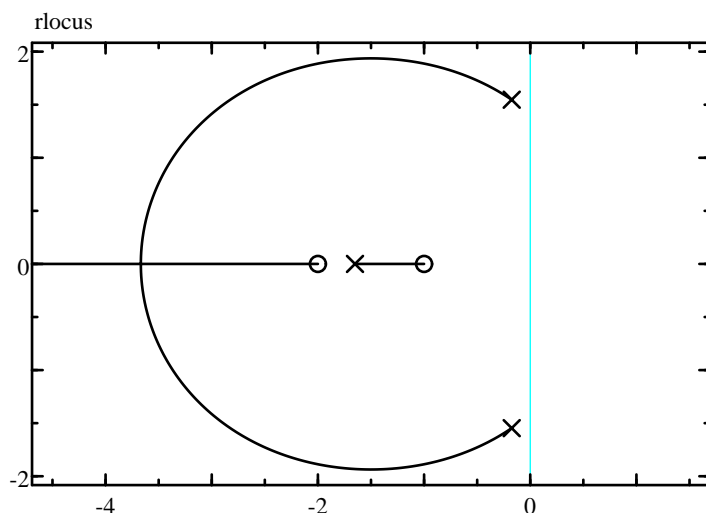
Figure 6.22   Example of rlocus

```
figure "Root Locus"
  draw myPlotRLocus(num, den);
  mousedrag num = myDragRLocus(num, _q);

function
{@
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.

scale('equal', [-2,2,-2,2]); sgrid
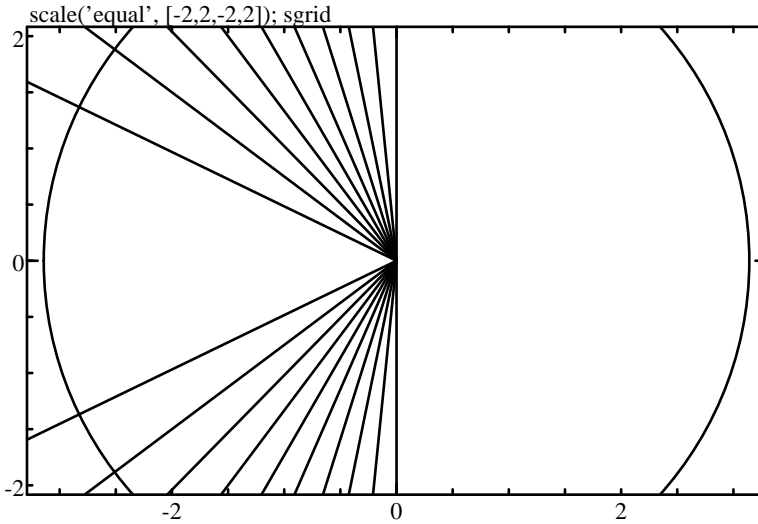
Figure 6.23 `scale('equal', [-2,2,-2,2]); sgrid`

See also

`plotroots`, `erlocus`, `sgrid`, `zgrid`

sgrid

Relative damping and natural frequency grid for the poles of a continuous-time system.

Syntax

```
sgrid
sgrid(style)
sgrid(damping, freq)
sgrid(damping, freq, style)
```

Description

With no or one argument, `sgrid` plots a grid of lines with constant relative damping and natural frequencies in the complex plane of s (see Fig. 6.23). The `style` argument has its usual meaning.

   With two or three 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 $|\operatorname{Im} p| = \sqrt{1 - \zeta^2}\,\operatorname{Re} p$ with $\operatorname{Re} p < 0$. The locus of roots with a constant natural frequency ω is a circle of radius ω.

 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) is displayed.

Example

Typical use for poles or zeros displayed in the s plane:

```
scale equal;
sgrid;
plotroots(pol);
```

See also

`zgrid`, `plotroots`, `hgrid`, `ngrid`, `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

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

$$\frac{dx}{dt}(t) = A_c x(t) + B_c u(t)$$
$$y(t) = C_c x(t) + D_c u(t)$$

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.
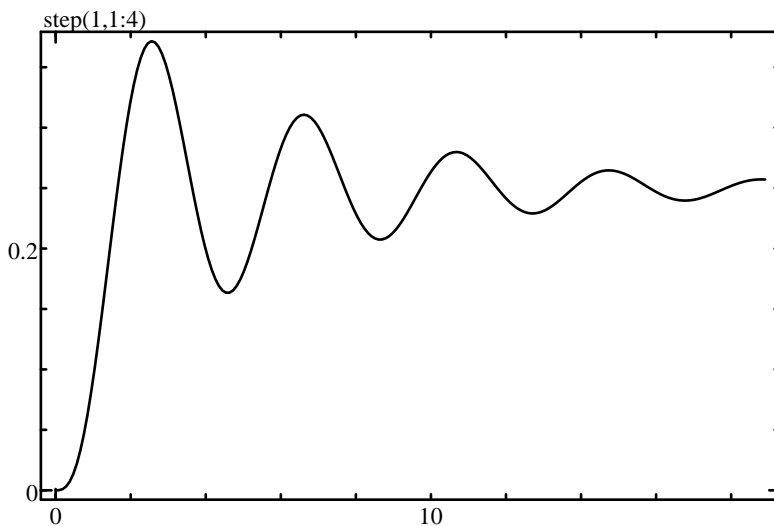
Figure 6.24 `step(1, [1,2,3,4])`

Example

Step response of the continuous-time system $1/(s^3 + 2s^2 + 3s + 4)$ (see Fig. 6.24):

```
step(1, 1:4, 'b');
```

See also

impulse, lsim, dstep, hstep, responseset

zgrid

Relative damping and natural frequency grid for the poles of a discrete-time system.

Syntax

```
zgrid
zgrid(style)
zgrid(damping, freq)
zgrid(damping, freq, style)
```
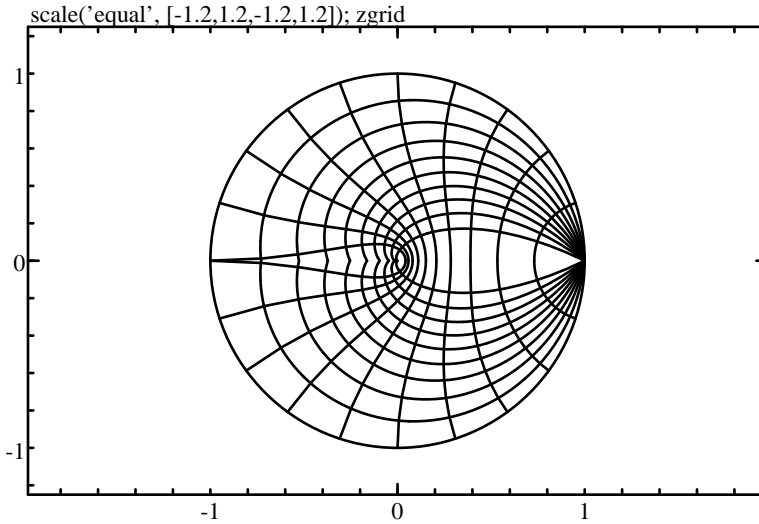
Figure 6.25 `scale('equal',[-1.2,1.2,-1.2,1.2]);zgrid`

Description

With no or one argument, `zgrid` plots a grid of lines with constant relative damping and natural frequencies in the complex plane of z (see Fig. 6.25). The `style` argument has its usual meaning.

With two or three 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 $\bar{z}$ are generated by $z = e^{(-1+j\sqrt{1-\zeta^2}/\zeta)u}$, with $0 \leq u \leq u_{max}$ 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 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) 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`, `plotoption`

6.41 Sysquake for LATEX Functions

Functions described in this section are specific to Sysquake for LaTeX. Some of them are documented because they appear in .lme files generated when .tex files which include file sysquake.sty are typeset, but they should not be called directly.

sqlxendfragment

End fragment of code.

Syntax

```
sqlxendfragment(fragid)
```

Description

`sqlxendfragment(fragid)` ends a fragment of code beginning with `beginlmefragment`. Argument `fragid` (a string) must match the first argument of the previous `beginlmefragment`.

`sqlxendfragment` is used by the Sysquake package; it should not be called from .tex files.

See also

beginlmefragment

sqlxvalue

Format value for an equation.

Syntax

```
sqlxvalue(v)
str = sqlxvalue(v)
```

Description

sqlxvalue(v) inserts value v in the LaTeX output, enclosed in
\ensuremath so that it can be used in math mode as well as in text
mode.

    With an output argument, sqlxvalue(v) returns its result in a
string.

sqlxstartfragment

Start a new fragment of code.

Syntax

```
 sqlxstartfragment(fragid, width, height)
```

Description

sqlxstartfragment(fragid,width,height) starts a new fragment
of code which corresponds to command \sqexpr or environment
sysquake in LaTeX. Argument fragid is a string made only of letters
which is used as fragment identifier; it is typically the fragment index
as a lowercase roman number starting at 1. Arguments width and
height are the dimensions of the figure reserved in LaTeX, or
negative if the result is text without graphics.

    sqlxstartfragment is used by the Sysquake package; it should
not be called from .tex files.

See also

sqlxendfragment, beginfigure

Chapter 7

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.

classes   classes implements constructors and methods for polynomial and rational functions. With them, you can use standard operator notations such as + or *.

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.

`polyhedra`   `polyhedra` implements functions which create solid shapes with polygonal faces in 3D.

`ratio`   `ratio` implements constructors and methods for rational numbers based on long integers. Standard arithmetic and boolean operators can be used.

`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.

`wsserver`   `wsserver` implements a server for web services using the XML-RPC or SOAP protocol. It requires the TCP/IP and WebServices functions.

7.1 stdlib

`stdlib` is a library which extends the native LME functions in the following areas:

- creation of matrices: `blkdiag`, `compan`, `hankel`, `linspace`, `logspace`, `toeplitz`

- geometry: `cart2sph`, `cart2pol`, `pol2cart`, `sph2cart`, `subspace`

- functions on integers: `factor`, `isprime`, `primes`

- statistics: `corrcoef`, `median`, `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
```

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

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

circshift

Shift the elements of a matrix in a circular way.

Syntax

```
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

```
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

```
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

```
X = compan(pol)
```

Description

compan(pol) gives the companion matrix of polynomial pol, a square matrix whose eigenvalues are the roots of pol.

Example

```
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

```
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 diag-
onal 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

```
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 numerical integration with trapezoidal approximation.

Syntax

```
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

```
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`

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`

fftshift

Shift DC frequency of FFT from beginning to center of spectrum.

Syntax

```
 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

```
 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

```
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

```
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

```
(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

```
(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

```
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`

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

primes, factor

isreal

Test for a real number.

Syntax

```
b = isreal(x)
```

Description

isreal(x) is true if x is a real scalar or a matrix whose entries are all real.

Examples

```
isreal([2,5])
    true
isreal([2,3+2j])
    false
isreal(exp(pi*1j))
    true
```

See also

isnumeric, isfloat, isscalar

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 100 values spaced linearly from x1 and x2 inclusive. With a third argument, linspace(x1,x2,n) produces a row vector of n values.

Examples

```
linspace(1,10)
   1.0000 1.0909 1.1818 ... 9.9091 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 100 values spaced log-
arithmically from 10ˆx1 and 10ˆx2 inclusive. With a third argument,
logspace(x1,x2,n) produces a row vector of n values.

Example

```
logspace(0,1)
   1.0000 1.0235 1.0476 ... 9.5455 9.7701 10.0000
```

See also

linspace, operator :

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.

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`

perms

Array of permutations.

Syntax

```
M = perms(v)
```

Description

perm(v) gives an array whose rows are all the possible permutations of vector v.

Example

```
perms(1:3)
   3   2   1
   3   1   2
   2   3   1
   1   3   2
   2   1   3
   1   2   3
```

See also

sort

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

polyfit

Polynomial fit.

Syntax

```
pol = polyfit(x, y, n)
```

Description

polyfit(x,y,n) calculates the polynomial (given as a vector of de-
scending power coefficients) of order n which best fits the points given
by vectors x and y. The least-square algorithm is used.

Example

```
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

```
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

```
polyvalm([1,2,8],[2,1;0,1])
  16   5
   0  11
```

See also

polyval

primes

List of primes.

Syntax

```
v = primes(n)
```

Description

primes(n) gives a row vector which contains the primes up to n.

Example

```
primes(20)
   2   3   5   7  11  13  17  19
```

See also

```
isprime
```

sortrows

Sort matrix rows.

Syntax

```
(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

```
sortrows([3, 1, 2; 2, 2, 1; 2, 1, 2])
   2   1   2
   2   2   1
   3   1   2
```

See also

```
sort
```

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

subspace

Angle between two subspaces.

Syntax

```
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:

```
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 R^3:

```
subspace([1;1;1], [2,7;5,1;3,0])
  0.2226
```

toeplitz

Toeplitz matrix.

Syntax

```
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

```
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

Numerical integration with trapezoidal approximation.

Syntax

```
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

```
trapz([2, 3, 5])
   6.5
trapz([1, 2, 5], [2, 3, 5])
   14.5
```

See also

sum, cumtrapz

7.2   stat

stat is a library which adds to LME advanced statistical functions.
    The following statement makes available functions defined in stat:

```
use stat
```

bootstrp

Bootstrap estimate.

Syntax

```
(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

```
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

```
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

```
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

```
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

```
harmmean(1:10)
  3.4142
mean(1:10)
  5.5
```

See also

`geomean`, `mean`

iqr

Interquartile range.

Syntax

```
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

```
iqr(rand(1,1000))
  0.5158
```

See also

`trimmean`, `prctile`

mad

Mean absolute deviation.

Syntax

```
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

```
mad(rand(1,1000))
   0.2446
```

See also

trimmean, mean, iqr

nancorrcoef

Correlation coefficients after discarding NaNs.

Syntax

```
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 equiva-
lent to nancorrcoef([X1(:),X2(:)]).

See also

nanmean, nanstd, nancov, `corrcoef`

nancov

Covariance after discarding NaNs.

Syntax

```
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

```
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

```
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

```
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

nansum

Sum after discarding NaNs.

Syntax

```
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

```
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 resized to 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:

'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

```
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

```
 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

Mean absolute deviation.

Syntax

```
 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

Resize the output of pdist to a square matrix.

Syntax

```
D = squareform(d)
```

Description

squareform(d) resize 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

```
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

```
 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.

7.3 classes

Library `classes` 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 `classes`:

```
 use classes
```

polynom::polynom

Polynom object constructor.

Syntax

```
a = polynom
a = 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.

+	plus		rem
-	minus	^	mpower
	mtimes	-	uminus
/	mrdivide	+	uplus
\	mldivide		roots

Examples

```
p = polynom([3,0,1,-4,2])
  p =
    3xˆ4+xˆ2-4x+2
q = 3 * pˆ2 + 8
  q =
    27xˆ8+18xˆ6-72xˆ5+39xˆ4-24xˆ3+60xˆ2-48x+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

```
disp(a)
```

Description

`disp(a)` displays polynomial a. It is also executed implicitly when LME displays the polynom result of an expression which does not end with a semicolon.

Example

```
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

```
coef = double(a)
```

Description

`double(a)` converts polynomial a to a row vector of descending-power coefficients.

Example

```
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

```
subst(a, b)
```

Description

subst(a,b) substitutes the variable of polynom a with polynom b.

Example

```
p = polynom([1,2,3])
  p =
    x^2+3x+9
q = polynom([2,0])
  q =
    2x
r = subst(p, q)
  r =
    4x^2+6x+9
```

See also

polynom::polynom, polynom::feval

polynom::diff

Polynom derivative.

Syntax

```
diff(a)
```

Description

diff(a) differentiates polynomial a.

Example

```
p = polynom([3,0,1,-4,2]);
q = diff(p)
  q =
    12x^3+2x-4
```

See also

polynom::polynom, polynom::int, polyder

polynom::int

Polynom integral.

Syntax

```
int(a)
```

Description

int(a) integrates polynomial a.

Example

```
p = polynom([3,0,1,-4,2]);
q = int(p)
  q =
    0.6x^5+0.3333x^3-2x^2+2x
```

See also

polynom::polynom, polynom::diff, polyint

polynom::inline

Conversion from polynom object to inline function.

Syntax

```
fun = inline(a)
```

Description

inline(a) converts polynomial a to an inline function which can then
be used with functions such as feval and ode45.

Example

```
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

```
y = feval(a, x)
```

Description

feval(a,x) evaluates polynomial a 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

```
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

ratfun::ratfun

Ratfun object constructor.

Syntax

```
a = ratfun
a = ratfun(coefnum)
a = 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.

```
    inv              ^   mpower
 -  minus            +   plus
    mtimes           -   uminus
 /  mrdivide         +   uplus
 \  mldivide
```

Example
```
 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
```
 disp(a)
```

Description
disp(a) displays rational function a. 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
```
 coef = num(a)
```

Description

`num(a)` gets the numerator of a 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

```
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

```
diff(a)
```

Description

`diff(a)` differentiates ratfun a.

Example

```
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

```
fun = inline(a)
```

Description

`inline(a)` converts ratfun a 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

```
y = feval(a, x)
```

Description

`feval(a,x)` evaluates ratfun a 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

```
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
```

7.4 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
```

ratio::ratio

Ratio object constructor.

Syntax

```
 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.

```
 +   plus          ˜=   ne
 -   minus         <    lt
 *   mtimes        <=   le
 /   mrdivide      >    gt
 \   mldivide      >=   ge
 ˆ   mpower             inv
 -   uminus             min(a,b)
 +   uplus              max(a,b)
 ==  eq
```

Examples

```
 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

```
 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

```
 disp(r)
```

Description

disp(r) displays ratio r with the same format as char. It is also ex-
ecuted 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

```
x = double(r)
```

Description

`double(r)` converts ratio r to a floating-point number of class double.

Example

```
r = ratio(2, 3);
double(r)
   0.6666
```

See also

```
ratio::ratio
```

7.5 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
```

bitfield::beginning

First bit position in a bitfield.

Syntax

```
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

```
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.

+	plus	&	and
-	minus	|	or
	mtimes	~	not
/	mrdivide		xor
\	mldivide	==	eq
-	uminus	~=	ne
+	uplus		

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

```
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

```
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

```
n = double(a)
```

Description

`double(a)` converts bitfield a to double number.

Example

```
a = bitfield(123, 16);
double(a)
   123
```

See also

`bitfield::bitfield`

bitfield::end

Last bit position in a bitfield.

Syntax

```
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

```
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

```
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

```
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

```
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

```
wordlength = length(a)
```

Description

length(a) gives the number of bits of bitfield a.

Example

```
a = bitfield(123, 16);
length(a)
   16
```

See also

bitfield::bitfield, length

bitfield::sign

Get the sign of a bitfield.

Syntax

```
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

```
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

```
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

```
a = bitfield(1234, 16);
uint8(a)
  210
```

See also

uint8, uint16, uint32, bitfield::int8, bitfield::int16, bitfield::int32, bitfield::double, bitfield::bitfield

7.6   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 com-
  pute the zeros, poles and gain of the prototype of analog low-pass
  filter with a cutoff frequency of 1 rad/s. They correspond respec-
  tively 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 band-
  stop 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 nu-
merator 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.

besselap

Bessel analog filter prototype.

Syntax

```
 (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

```
(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 nth-order digital low-pass filter 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 2nth-order digital bandpass filter with pass-band between wl and wh relatively to half the sampling frequency.

`besself(n,w0,'high')` gives a nth-order digital highpass filter 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 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'`, `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

```
(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

```
(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

```
(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

```
 (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.

See also

`cheby1`, `cheb2ap`, `ellipap`, `besselap`, `buttap`

cheb2ap

Chebyshev type 2 analog filter prototype.

Syntax

```
(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.

See also

cheby1, cheb1ap, ellipap, besselap, buttap

cheby1

Chebyshev type 1 filter.

Syntax

```
(z, p, k) = cheby1(n, w0)
(num, den) = cheby1(n, w0)
(...) = cheby1(n, [wl, wh])
(...) = cheby1(n, w0, 'high')
(...) = cheby1(n, [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,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.

cheby1(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.

cheby1(n,w0,'high') gives a nth-order digital highpass filter with a cutoff frequency of w0 relatively to half the sampling frequency.

cheby1(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', 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

```
(z, p, k) = cheby2(n, w0)
(num, den) = cheby2(n, w0)
(...) = cheby2(n, [wl, wh])
(...) = cheby2(n, w0, 'high')
(...) = cheby2(n, [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,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.

cheby2(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.

cheby2(n,w0,'high') gives a nth-order digital highpass filter with a cutoff frequency of w0 relatively to half the sampling frequency.

cheby2(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', 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

```
(z, p, k) = ellip(n, w0)
(num, den) = ellip(n, w0)
(...) = ellip(n, [wl, wh])
(...) = ellip(n, w0, 'high')
(...) = ellip(n, [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,w0), where w0 is a scalar, gives a nth-order digital low-pass filter with a cutoff frequency of w0 relatively to half the sampling frequency.

ellip(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.

ellip(n,w0,'high') gives a nth-order digital highpass filter with a cutoff frequency of w0 relatively to half the sampling frequency.

ellip(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', 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

```
(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.

See also

`ellip, cheb1ap, cheb1ap, besselap, buttap`

lp2bp

Lowpass prototype to bandpass filter conversion.

Syntax

```
(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

```
(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 fre-
quency) 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

```
(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 fre-
quency) to a highpass analog filter with the specified cutoff angular
frequency w0. lp2hp(z0,p0,k0,w0) converts a filter given by its ze-
ros, 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

```
(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 fre-
quency) to a lowpass analog filter with the specified cutoff angular
frequency w0. lp2lp(z0,p0,k0,w0) converts a filter given by its ze-
ros, 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

7.7   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 func-
tions; 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
```

ss::ss

LTI state-space constructor.

Syntax

```
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

```
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

LTI transfer function constructor.

Syntax

```
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 may 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):

```
sc = tf(1,[1,2,3,4],'p')
  sc =
    continuous-time transfer function
    1/(p^3+2p^2+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

ss

lti::append

Append the inputs and outputs of systems.

Syntax

```
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

connect

ss::augstate

Extend the output of a system with its states.

Syntax

```
b = augstate(a)
```

Description

augstate(a) extends the ss object a by adding its states to its outputs. The new output yext is [y;x], where y is the output of a and x is its states.

lti::beginning

First index.

Syntax

```
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

end, subsasgn, subsref

lti::c2d

Conversion from continuous time to discrete time.

Syntax

```
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.

See also

d2c, c2dm

lti::connect

Arbitrary feedback connections.

Syntax

```
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

feedback

lti::d2c

Conversion from discrete time to continuous time.

Syntax

```
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.

See also

c2d, d2cm

lti::end

Last index.

Syntax

```
 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:

```
 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

beginning, subsasgn, subsref

lti::evalfr

Frequency value.

Syntax

```
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

```
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

ss::ctrb

Controllability matrix.

Syntax

```
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

`obsv`

lti::dcgain

Steady-state gain.

Syntax

```
 g = dcgain(a)
```

Description

`dcgain(a)` gives the steady-state gain of system a.

See also

`norm`

lti::feedback

Feedback connection.

Syntax

```
 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

`connect`

lti::inv

System inverse.

Syntax

```
b = inv(a)
```

Description

inv(a) gives the inverse of system a.

See also

mldivide, mrdivide

lti::isct

Test for a continous-time LTI.

Syntax

```
b = isct(a)
```

Description

isct(a) is true if system a is continuous-time or static, and false otherwise.

See also

isdt

lti::isdt

Test for a discrete-time LTI.

Syntax

```
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

```
 b = isempty(a)
```

Description

`isempty(a)` is true if system a has no input and/or no output, and false otherwise.

See also

`size, issiso`

lti::isproper

Test for a proper (causal) LTI.

Syntax

```
 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

```
 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.
    size, isempty

lti::minreal

Minimum realization.

Syntax

```
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

```
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

parallel, plus, uminus

lti::mldivide

System left division.

Syntax

```
c = a \ b
c = mldivide(a, b)
```

Description

a/b is equivalent to inv(a)∗b.

See also

mrdivide, times, inv

lti::mrdivide

System right division.

Syntax

```
c = a / b
c = mrdivide(a, b)
```

Description

a/b is equivalent to a∗inv(b).

See also

mldivide, times, inv

lti::mtimes

System product.

Syntax

```
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

`series`

lti::norm

H2 norm.

Syntax

```
h2 = norm(a)
```

Description

`norm(a)` gives the H2 norm of the system a.

See also

`dcgain`

ss::obsv

Observability matrix.

Syntax

```
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

`ctrb`

lti::parallel

Parallel connection.

Syntax

```
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

`series`

lti::plus

System sum.

Syntax

```
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

`parallel`, `minus`

lti::series

Series connection.

Syntax

```
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

`mtimes`, `parallel`

lti::repmat

Replicate a system.

Syntax

```
b = repmat(a, n)
b = repmat(a, [m,n])
b = repmat(a, m, n)
```

Description

`repmat(a,n)`, when a is a transfer function or a matrix of transfer functions, creates a new system described by a matrix of transfer functions where a is repeated n times horizontally and vertically. If a is a state-space system, matrices B, C, and D are replicated to obtain the same effect.

`repmat(a,[m,n])` or `repmat(a,m,n)` repeats matrix a m times vertically and n times horizontally.

See also

append

lti::size

Number of outputs and inputs.

Syntax

```
 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

isempty, issiso

lti::ssdata

Get state-space matrices.

Syntax

```
 (A, B, C, D) = ssdata(a)
 (A, B, C, D, Ts) = ssdata(a)
```

Description

ssdata(a), where a is any kind of LTI object, gives the four matrices of the state-space model, and optionally the sampling period or the empty array [] for continuous-time systems.

See also

tfdata

lti::subsasgn

Assignment to a part of an LTI system.

Syntax

```
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 both, `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

`subsref`, operator (), `subsasgn`

lti::subsref

Extraction of a part of an LTI system.

Syntax

```
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. 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.

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 both, `var` (string) and Ts (scalar, or empty array for continuous-time systems).

The syntax with braces (`var{i}`) is not supported.

See also

subsasgn, operator (), subsasgn

lti::tfdata

Get transfer functions.

Syntax

```
(num, den) = tfdata(a)
(num, den, Ts) = ssdata(a)
```

Description

`tfdata(a)`, where a is any kind of LTI object, 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 correcpond to the outputs, and their columns to the inputs.

See also

ssdata

zpk

LTI transfer function constructor using zeros and poles.

Syntax

```
a = zpk(z, p, k)
a = zpk(zeroslist, poleslist, gainlist)
a = zpk(..., Ts)
a = zpk(..., Ts, var)
a = zpk(..., b)
a = zpk(b)
```

Description

zpk creates transfer-function LTI systems like `tf`. Instead of using transfer function coefficients as input, it accepts a vector of zeros, a vector of poles, and a gain for a simple-input simple-output (SISO) system; or lists of sublists of zeros, poles and gains for multiple-input multiple-output (MIMO) systems.

Examples

```
sd = zpk(0.3, [0.8+0.5j; 0.8-0.5j], 10, 0.1)
  sd =
    discrete-time transfer function, Ts=0.1
    (10z-3)/(z^2-1.6z+0.89)
```

See also

`tf::tf`

lti::bodemag

Magnitude of the Bode plot.

Syntax

```
bodemag(a, ...)
... = bodemag(a, ...)
```

Description

bodemag(a) plots the magnitude of the Bode diagram of system a.

See also

`bodephase`, `nichols`, `nyquist`

lti::bodephase

Phase of the Bode plot.

Syntax

```
bodephase(a, ...)
... = bodephase(a, ...)
```

Description

bodephase(a) plots the magnitude of the Bode diagram of system a.

See also

bodemag, nichols, nyquist

lti::impulse

Impulse response.

Syntax

```
impulse(a, ...)
... = impulse(a, ...)
```

Description

impulse(a) plots the impulse response of system a.

See also

step, lsim, initial

lti::initial

Time response with initial conditions.

Syntax

```
initial(a, x0, ...)
... = initial(a, x0, ...)
```

Description

initial(a,x0) plots the time response of state-space system a with initial state x0 and null input.

See also

impulse, step, lsim

lti::lsim

Time response.

Syntax

```
lsim(a, u, 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.

See also

impulse, step, initial

lti::nichols

Nichols plot.

Syntax

```
nichols(a, ...)
... = nichols(a, ...)
```

Description

nichols(a) plots the Nichols diagram of system a.

See also

nyquist, bodemag, bodephase

lti::nyquist

Nyquist plot.

Syntax

```
nyquist(a, ...)
... = nyquist(a, ...)
```

Description

nyquist(a) plots the Nyquist diagram of system a.

See also

nichols, bodemag, bodephase

lti::step

Step response.

Syntax

```
step(a, ...)
... = step(a, ...)
```

Description

step(a) plots the step response of system a.

See also

impulse, lsim, initial

lti::uminus

Negative.

Syntax

```
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

minus, uplus

lti::uplus

Negative.

Syntax

```
b = +a
b = uplus(a)
```

Description

+a gives a.

See also

uminus, plus

7.8   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
```

alawcompress

A-law compressor.

Syntax

```
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

```
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

```
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

```
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.

---

[1]Actually, there may be a drift if the arithmetic units used for encoding and decoding do not produce exactly the same results.

Example

```
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);
```

See also

quantiz, dpcmdeco, dpcmopt

dpcmopt

Differential pulse code modulation decoding.

Syntax

```
(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

```
(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:

```
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

```
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 $-\infty$ 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

```
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

```
output = ulawcompress(input)
output = ulawcompress(input, mu)
```

Description

ulawcompress(input,a) 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, nonlin-
ear 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

```
output = ulawexpand(input)
output = ulawexpand(input, mu)
```

Description

ulawexpand(input,a) expands signal input with mu-law method us-
ing parameter a. input can be a real array of any size and dimension.
The default value of mu is 255.

See also

ulawcompress, alawexpand

7.9 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 un-
compressed, 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
```

wavread

WAV decoding.

Syntax

```
(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

```
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` corre-
sponds 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

```
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`

7.10 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
```

datestr

Date to string conversion.

Syntax

```
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)
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.

Examples

```
datestr(clock)
  18-Apr-2005 16:21:55
datestr(clock, 'ddd mm/dd/yyyy HH:MM PM')
  Mon 04/18/2005 04:23 PM
```

See also

`datevec`, `julian2cal`, `clock`

datevec

String to date and time conversion.

Syntax

```
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

```
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

```
(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:

```
(num, str) = weekday(clock)
  num =
    2
  str =
    Mon
```

See also

cal2julian

7.11 constants

constants is a library which defines physical constants in SI units (meter, kilogram, second, ampere).

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 m^2/kg^2
gravity_acceleration	9.80655	m/s^2
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	m^3/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/m^2 K^-4
vacuum_permittivity	8.854187817e-12	A s/V m

7.12   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 with hue from red to red throw green and blu
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);
```

black2orangecm

Colormap with shades from black to orange.

Syntax

```
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.

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

```
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.

See also

colormap, black2orangecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

blue2greencm

Colormap with shades from blue to green.

Syntax

```
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.

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

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

cyan2magentacm

Colormap with shades from cyan to magenta.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

graycm

Colormap with shades of gray.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm,
blue2yellow2redcm, cyan2magentacm, green2yellowcm, huecm,
interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

green2yellowcm

Colormap with shades from green to yellow.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm,
blue2yellow2redcm, cyan2magentacm, graycm, huecm, interprgbcm,
magenta2yellowcm, red2yellowcm, sepiacm, whitecm

huecm

Colormap with hue from red to red throw green and blue.

Syntax

```
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 throw 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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

interprgbcm

Colormap with entries obtained by linear interpolation.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm,
blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm,
huecm, magenta2yellowcm, red2yellowcm, sepiacm, whitecm

magenta2yellowcm

Colormap with shades from black to orange.

Syntax

```
cm = magenta2yellowcm
cm = magenta2yellowcm(n)
```

Description

`magenta2yellowcm(n)` creates a color map with n entries correspond-
ing 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 inten-
sity). The default value of n is 256.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm,
blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm,
huecm, interprgbcm, red2yellowcm, sepiacm, whitecm

red2yellowcm

Colormap with shades from black to orange.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, sepiacm, whitecm

sepiacm

Colormap with shades of sepia.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, whitecm

whitecm

Colormap with shades from black to orange.

Syntax

```
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.

See also

colormap, black2orangecm, black2red2whitecm, blue2greencm, blue2yellow2redcm, cyan2magentacm, graycm, green2yellowcm, huecm, interprgbcm, magenta2yellowcm, red2yellowcm, sepiacm

7.13   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
```

cube

Create a cube.

Syntax

```
 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

```
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

```
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

```
greaticosahedron;
greaticosahedron(style);
(X, Y, Z, ind) = greaticosahedron
```

Description

Without output argument, greaticosahedron displays a great icosa-
hedron, i.e. a regular nonconvex solid whose twenty faces are equi-
lateral 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

```
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 com-
mon to three faces. By default, edges are not drawn. An optional
string input argument specifies the edge style.

With four output arguments, greatstellateddodecahedron pro-
duces 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

```
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

```
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

```
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

```
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`

7.14 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
```

cone

Cone.

Syntax

```
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 poly-hedron. 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

```
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

```
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

```
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
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

```
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

```
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

```
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

```
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`

Index