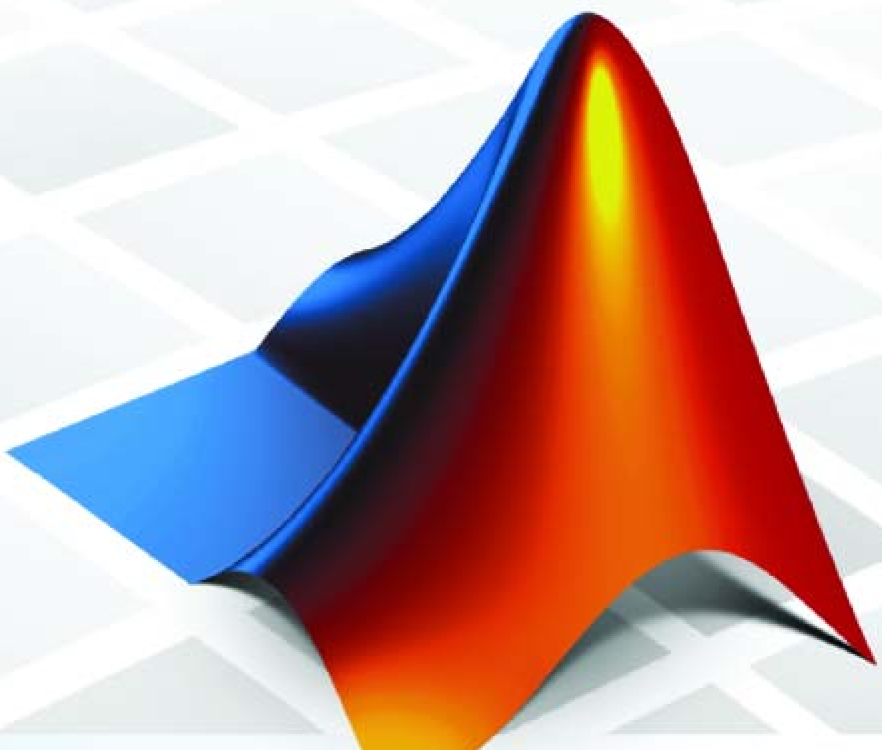
Symbolic Math Toolbox 3
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Symbolic Math Toolbox User's Guide

© COPYRIGHT 1993–2007 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

August 1993	First printing	
October 1994	Second printing	
May 1997	Third printing	Revised for Version 2
May 2000	Fourth printing	Minor changes
June 2001	Fifth printing	Minor changes
July 2002	Online only	Revised for Version 2.1.3 (Release 13)
October 2002	Online only	Revised for Version 3.0.1
December 2002	Sixth printing	
June 2004	Seventh printing	Revised for Version 3.1 (Release 14)
October 2004	Online only	Revised for Version 3.1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.1.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.1.3 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1.4 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.5 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.2.2 (Release 2007b)

Contents

3 Functions — By Category

4 Functions — Alphabetical List

Index

Getting Started

This section introduces you to Symbolic Math Toolbox, and describes how to create and use symbolic objects. The section covers the following topics:

What Are Symbolic Math Toolboxes?

Symbolic Math Toolboxes incorporate symbolic computation into the numeric environment of MATLAB®. These toolboxes supplement MATLAB numeric and graphical facilities with several other types of mathematical computation, which are summarized in following table.

Facility	**Covers**
Calculus	Differentiation, integration, limits, summation, and Taylor series
Linear Algebra	Inverses, determinants, eigenvalues, singular value decomposition, and canonical forms of symbolic matrices
Simplification	Methods of simplifying algebraic expressions
Solution of Equations	Symbolic and numerical solutions to algebraic and differential equations
Special Mathematical Functions	Special functions of classical applied mathematics
Variable-Precision Arithmetic	Numerical evaluation of mathematical expressions to any specified accuracy
Transforms	Fourier, Laplace, z-transform, and corresponding inverse transforms

The computational engine underlying the toolboxes is the kernel of Maple®, a system developed primarily at the University of Waterloo, Canada and, more recently, at the Eidgenössiche Technische Hochschule, Zürich, Switzerland. Maple is marketed and supported by Waterloo Maple, Inc.

There are two toolboxes:

- The basic Symbolic Math Toolbox is a collection of more than 100 MATLAB functions that provide access to the Maple kernel using a syntax and style that is a natural extension of the MATLAB language. The basic toolbox also allows you to access functions in the Maple linear algebra package.

- Extended Symbolic Math Toolbox augments this functionality to include access to all nongraphics Maple packages, Maple programming features, and user-defined procedures. With both toolboxes, you can write your own M-files to access Maple functions and the Maple workspace.

If you have the version of Maple consistent with the version of Symbolic Math Toolboxes you are using, you can use that instead of the copy of the Maple Library that is distributed with Symbolic Math Toolboxes by changing the path to the library in the MATLAB M-file `mapleinit.m`. See the `mapleinit` reference page to learn how to do this.

Symbolic Objects

Symbolic Math Toolbox defines a new MATLAB data type called a *symbolic object*. (See "Data Types" in the MATLAB Programming documentation for an introduction to MATLAB classes and objects.) Internally, a symbolic object is a data structure that stores a string representation of the symbol. Symbolic Math Toolbox uses symbolic objects to represent symbolic variables, expressions, and matrices. The actual computations involving symbolic objects are performed primarily by Maple, mathematical software developed by Waterloo Maple, Inc.

MATLAB Data Types and the Corresponding Symbolic Objects

The following example illustrates the difference between a standard MATLAB data type, such as `double`, and the corresponding symbolic object. The MATLAB command

```
sqrt(2)
```

returns a floating-point decimal number:

```
ans =
    1.4142
```

On the other hand, if you convert 2 to a symbolic object using the `sym` command, and then take its square root by entering

```
a = sqrt(sym(2))
```

the result is

```
a =
2^(1/2)
```

MATLAB gives the result `2^(1/2)`, which means $2^{1/2}$, using symbolic notation for the square root operation, without actually calculating a numerical value. MATLAB records this symbolic expression in the string that represents `2^(1/2)`. You can always obtain the numerical value of a symbolic object with the `double` command:

```
double(a)
ans =
      1.4142
```

Notice that the result is indented, which tells you it has data type `double`. Symbolic results are not indented.

When you create a fraction involving symbolic objects, MATLAB records the numerator and denominator. For example:

```
sym(2)/sym(5)
ans =
2/5
```

MATLAB performs arithmetic on symbolic objects differently than it does on standard data types. If you add two fractions that are of data type `double`, MATLAB gives the answer as a decimal fraction. For example:

```
2/5 + 1/3
ans =
0.7333
```

If you add the same fractions as symbolic objects, MATLAB finds their common denominator and combines them by the usual procedure for adding rational numbers:

```
sym(2)/sym(5) + sym(1)/sym(3)
ans =
11/15
```

Symbolic Math Toolbox enables you to perform a variety of symbolic calculations that arise in mathematics and science. These are described in detail in Chapter 2, “Using Symbolic Math Toolbox”.

Creating Symbolic Variables and Expressions

In this section...

sym and syms Commands

The `sym` command lets you construct symbolic variables and expressions. For example, the commands

```
x = sym('x')
a = sym('alpha')
```

create a symbolic variable `x` that prints as `x` and a symbolic variable `a` that prints as `alpha`.

Suppose you want to use a symbolic variable to represent the golden ratio

$$p = \frac{1+\sqrt{5}}{2}$$

The command

```
rho = sym('(1 + sqrt(5))/2')
```

achieves this goal. Now you can perform various mathematical operations on `rho`. For example,

```
f = rho^2 - rho - 1
```

returns

```
f =

(1/2+1/2*5^(1/2))^2-3/2-1/2*5^(1/2)
```

You can simplify this answer by entering

```
simplify(f)
```

which returns

```
ans =
0
```

Now suppose you want to study the quadratic function $f = ax^2 + bx + c$. One approach is to enter the command

```
f = sym('a*x^2 + b*x + c')
```

which assigns the symbolic expression $ax^2 + bx + c$ to the variable `f`. However, in this case, Symbolic Math Toolbox does not create variables corresponding to the terms of the expression, a, b, c, and x. To perform symbolic math operations (e.g., integration, differentiation, substitution, etc.) on `f`, you need to create the variables explicitly. A better alternative is to enter the commands

```
a = sym('a')
b = sym('b')
c = sym('c')
x = sym('x')
```

or simply

```
syms a b c x
```

Then enter

```
f = sym('a*x^2 + b*x + c')
```

In general, you can use `sym` or `syms` to create symbolic variables. We recommend you use `syms` because it requires less typing.

Note To create a symbolic expression that is a constant, you must use the `sym` command. For example, to create the expression whose value is `5`, enter `f = sym('5')`. Note that the command `f = 5` does *not* define `f` as a symbolic expression.

If you set a variable equal to a symbolic expression, and then apply the `syms` command to the variable, MATLAB removes the previously defined expression from the variable. For example,

```
syms a b
f = a + b
```

returns

```
f =
a+b
```

If you then enter

```
syms f
f
```

MATLAB returns

```
f =
f
```

You can use the `syms` command to clear variables of definitions that you assigned to them previously in your MATLAB session. However, `syms` does not clear the properties of the variables in the Maple workspace. See “Clearing Variables in the Maple Workspace” on page 1-15 for more information.

findsym Command

To determine what symbolic variables are present in an expression, use the `findsym` command. For example, given the symbolic expressions `f` and `g` defined by

```
syms a b n t x z
f = x^n; g = sin(a*t + b);
```

you can find the symbolic variables in `f` by entering

```
findsym(f)
ans =
n, x
```

Similarly, you can find the symbolic variables in `g` by entering

```
findsym(g)
ans =
a, b, t
```

Substituting for Symbolic Variables

In this section...

subs Command

You can substitute a numerical value for a symbolic variable using the `subs` command. For example, to substitute the value $x = 2$ in the symbolic expression,

```
f = 2*x^2 - 3*x + 1
```

enter the command

```
subs(f,2)
```

This returns $f(2)$:

```
ans =
     3
```

Note To substitute a matrix `A` into the symbolic expression `f`, use the command `polyvalm(sym2poly(f), A)`, which replaces all occurrences of `x` by `A`, and replaces the constant term of `f` with the constant times an identity matrix.

When your expression contains more than one variable, you can specify the variable for which you want to make the substitution. For example, to substitute the value $x = 3$ in the symbolic expression,

```
syms x y
f = x^2*y + 5*x*sqrt(y)
```

enter the command

```
subs(f, x, 3)
```

This returns

```
ans =
9*y+15*y^(1/2)
```

On the other hand, to substitute $y = 3$, enter

```
subs(f, y, 3)
ans =
3*x^2+5*x*3^(1/2)
```

Default Symbolic Variable

If you do not specify a variable to substitute for, MATLAB chooses a default variable according to the following rule. For one-letter variables, MATLAB chooses the letter closest to x in the alphabet. If there are two letters equally close to x, MATLAB chooses the one that comes later in the alphabet. In the preceding function, `subs(f, 3)` returns the same answer as `subs(f, x, 3)`.

You can use the `findsym` command to determine the default variable. For example,

```
syms s t
g = s + t;
findsym(g,1)
```

returns the default variable:

```
ans =
t
```

See "Substitutions" on page 2-50 to learn more about substituting for variables.

Symbolic and Numeric Conversions

In this section...

Floating-Point Symbolic Expressions

Consider the ordinary MATLAB quantity

```
t = 0.1
```

The `sym` function has four options for returning a symbolic representation of the numeric value stored in `t`. The `'f'` option

```
sym(t,'f')
```

returns a symbolic floating-point representation

```
'1.999999999999a'*2^(-4)
```

Rational Symbolic Expressions

The `'r'` option

```
sym(t,'r')
```

returns the rational form

```
1/10
```

This is the default setting for `sym`. That is, calling `sym` without a second argument is the same as using `sym` with the `'r'` option:

```
sym(t)
```

```
ans =
1/10
```

The third option `'e'` returns the rational form of `t` plus the difference between the theoretical rational expression for `t` and its actual (machine) floating-point value in terms of `eps` (the floating-point relative accuracy):

```
sym(t,'e')

ans =
1/10+eps/40
```

Decimal Symbolic Expressions

The fourth option `'d'` returns the decimal expansion of `t` up to the number of significant digits specified by `digits`:

```
sym(t,'d')

ans =
.10000000000000000555111512312578
```

The default value of `digits` is 32 (hence, `sym(t,'d')` returns a number with 32 significant digits), but if you prefer a shorter representation, use the `digits` command as follows:

```
digits(7)
sym(t,'d')

ans =
.1000000
```

Converting Symbolic Matrices to Numeric Form

A particularly effective use of `sym` is to convert a matrix from numeric to symbolic form. The command

```
A = hilb(3)
```

generates the 3-by-3 Hilbert matrix:

```
A =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

By applying `sym` to `A`

```
A = sym(A)
```

you can obtain the symbolic (infinitely precise) form of the 3-by-3 Hilbert matrix:

```
A =

[   1, 1/2, 1/3]
[ 1/2, 1/3, 1/4]
[ 1/3, 1/4, 1/5]
```

Constructing Real and Complex Variables

The `sym` command allows you to specify the mathematical properties of symbolic variables by using the `'real'` option. That is, the statements

```
x = sym('x','real'); y = sym('y','real');
```

or more efficiently

```
syms x y real
z = x + i*y
```

create symbolic variables `x` and `y` that have the added mathematical property of being real variables. Specifically this means that the expression

```
f = x^2 + y^2
```

is strictly nonnegative. Hence, `z` is a complex variable and can be manipulated as such. Thus, the commands

```
conj(x), conj(z), expand(z*conj(z))
```

return

```
x, x-i*y, x^2+y^2
```

respectively. The `conj` command is the complex conjugate operator for the toolbox. If `conj(x) == x` returns 1, then `x` is a real variable.

Clearing Variables in the Maple Workspace

When you declare `x` to be real with the command

```
syms x real
```

`x` becomes a symbolic object in the MATLAB workspace and a positive real variable in the Maple kernel workspace. If you later want to remove the `real` property from `x`, enter

```
syms x unreal
```

If you want to clear all variable definitions in the Maple kernel workspace, enter

```
maple restart
```

Note that entering

```
clear x
```

only clears `x` in the MATLAB workspace. If you then enter `syms x`, without having also cleared `x` from the Maple kernel workspace, MATLAB still treats `x` as a positive real number.

Creating Abstract Functions

If you want to create an abstract (i.e., indeterminant) function $f(x)$, type

```
f = sym('f(x)')
```

Then `f` acts like $f(x)$ and can be manipulated by the toolbox commands. For example, to construct the first difference ratio, type

```
df = (subs(f,'x','x+h') - f)/'h'
```

or

```
syms x h
df = (subs(f,x,x+h)-f)/h
```

which returns

```
df =
(f(x+h)-f(x))/h
```

This application of sym is useful when computing Fourier, Laplace, and *z*-transforms.

Using sym to Access Maple Functions

Similarly, you can access Maple's factorial function k! using sym:

```
kfac = sym('k!')
```

To compute 6! or n!, type

```
syms k n
subs(kfac,k,6), subs(kfac,k,n)

ans =
720

ans =
n!
```

Creating a Symbolic Matrix Example

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. You can create the circulant matrix A whose elements are a, b, and c, using the commands

```
syms a b c
A = [a b c; b c a; c a b]
```

which return

```
A =
[ a, b, c ]
[ b, c, a ]
[ c, a, b ]
```

Since A is circulant, the sum over each row and column is the same. To check this for the first row and second column, enter the command

```
sum(A(1,:))
```

which returns

```
ans =
a+b+c
```

The command

```
sum(A(1,:)) == sum(A(:,2))  % This is a logical test.
```

returns

```
ans =
     1
```

Now replace the (2,3) entry of A with beta and the variable b with alpha. The commands

```
syms alpha beta;
A(2,3) = beta;
A = subs(A,b,alpha)
```

return

```
A =
[     a, alpha,     c]
[ alpha,     c,  beta]
[     c,     a, alpha]
```

From this example, you can see that using symbolic objects is very similar to using regular MATLAB numeric objects.

Creating Symbolic Math Functions

In this section...

Using Symbolic Expressions

The sequence of commands

```
syms x y z
r = sqrt(x^2 + y^2 + z^2)
t = atan(y/x)
f = sin(x*y)/(x*y)
```

generates the symbolic expressions `r`, `t`, and `f`. You can use `diff`, `int`, `subs`, and other Symbolic Math Toolbox functions to manipulate such expressions.

Creating an M-File

M-files permit a more general use of functions. Suppose, for example, you want to create the `sinc` function `sin(x)/x`. To do this, create an M-file in the `@sym` directory:

```
function z = sinc(x)
%SINC The symbolic sinc function
%     sin(x)/x. This function
%     accepts a sym as the input argument.
if isequal(x,sym(0))
   z = 1;
else
   z = sin(x)/x;end
```

You can extend such examples to functions of several variables. See the MATLAB topic "Data Types" in the online MATLAB Programming documentation for a more detailed discussion on object-oriented programming.

Using Symbolic Math Toolbox

This section explains how to use Symbolic Math Toolbox to perform many common mathematical operations. The section covers the following topics:

Calculus

In this section...

Differentiation

To illustrate how to take derivatives using Symbolic Math Toolbox, first create a symbolic expression:

```
syms x
f = sin(5*x)
```

The command

```
diff(f)
```

differentiates `f` with respect to `x`:

```
ans =
5*cos(5*x)
```

As another example, let

```
g = exp(x)*cos(x)
```

where `exp(x)` denotes e^x, and differentiate `g`:

```
diff(g)
ans =
exp(x)*cos(x)-exp(x)*sin(x)
```

To take the second derivative of g, enter

```
diff(g,2)
ans =
-2*exp(x)*sin(x)
```

You can get the same result by taking the derivative twice:

```
diff(diff(g))
ans =
-2*exp(x)*sin(x)
```

In this example, MATLAB automatically simplifies the answer. However, in some cases, MATLAB might not simply an answer, in which case you can use the `simplify` command. For an example of this, see "More Examples" on page 2-5.

Note that to take the derivative of a constant, you must first define the constant as a symbolic expression. For example, entering

```
c = sym('5');
diff(c)
```

returns

```
ans =
0
```

If you just enter

```
diff(5)
```

MATLAB returns

```
ans =
     []
```

because 5 is not a symbolic expression.

Derivatives of Expressions with Several Variables

To differentiate an expression that contains more than one symbolic variable, you must specify the variable that you want to differentiate with respect to.

The `diff` command then calculates the partial derivative of the expression with respect to that variable. For example, given the symbolic expression

```
syms s t
f = sin(s*t)
```

the command

```
diff(f,t)
```

calculates the partial derivative $\partial f / \partial t$. The result is

```
ans =
cos(s*t)*s
```

To differentiate `f` with respect to the variable `s`, enter

```
diff(f,s)
```

which returns:

```
ans =
cos(s*t)*t
```

If you do not specify a variable to differentiate with respect to, MATLAB chooses a default variable by the same rule described in "Substituting for Symbolic Variables" on page 1-10. For one-letter variables, the default variable is the letter closest to x in the alphabet. In the preceding example, `diff(f)` takes the derivative of `f` with respect to `t` because `t` is closer to x in the alphabet than s is. To determine the default variable that MATLAB differentiates with respect to, use the `findsym` command:

```
findsym(f,1)
ans =
t
```

To calculate the second derivative of `f` with respect to `t`, enter

```
diff(f,t,2)
```

which returns

```
ans =
-sin(s*t)*s^2
```

Note that diff(f,2) returns the same answer because t is the default variable.

More Examples

To further illustrate the diff command, define a, b, x, n, t, and theta in the MATLAB workspace by entering

```
syms a b x n t theta
```

The table below illustrates the results of entering diff(f).

f	diff(f)
x^n	x^n*n/x
sin(a*t+b)	cos(a*t+b)*a
exp(i*theta)	i*exp(i*theta)

In the first example, MATLAB does not automatically simplify the answer. To simplify the answer, enter

```
simplify(diff(x^n))
ans =
x^(n-1)*n
```

To differentiate the Bessel function of the first kind,besselj(nu,z), with respect to z, type

```
syms nu z
b = besselj(nu,z);
db = diff(b)
```

which returns

```
db =
-besselj(nu+1,z)+nu/z*besselj(nu,z)
```

The diff function can also take a symbolic matrix as its input. In this case, the differentiation is done element-by-element. Consider the example

```
syms a x
A = [cos(a*x),sin(a*x);-sin(a*x),cos(a*x)]
```

which returns

```
A =
[  cos(a*x),  sin(a*x)]
[ -sin(a*x),  cos(a*x)]
```

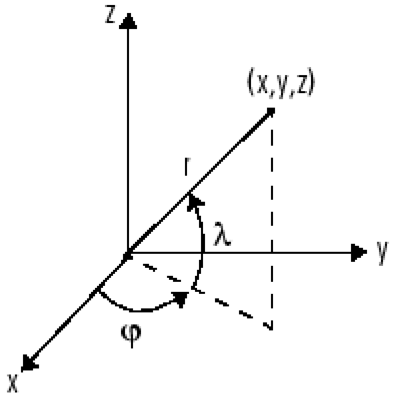
The command

```
diff(A)
```

returns

```
ans =
[ -sin(a*x)*a,  cos(a*x)*a]
[ -cos(a*x)*a, -sin(a*x)*a]
```

You can also perform differentiation of a column vector with respect to a row vector. Consider the transformation from Euclidean (x, y, z) to spherical (r,λ,φ) coordinates as given by $x = r\cos\lambda\cos\varphi$, $y = r\cos\lambda\cos\phi$, and $z = r\sin\lambda$. Note that λ corresponds to elevation or latitude while φ denotes azimuth or longitude.



To calculate the Jacobian matrix, J, of this transformation, use the `jacobian` function. The mathematical notation for J is

$$J = \frac{\partial(x, y, z)}{\partial(r, \lambda\varphi)}$$

For the purposes of toolbox syntax, use `l` for λ and `f` for φ. The commands

```
syms r l f
x = r*cos(l)*cos(f); y = r*cos(l)*sin(f); z = r*sin(l);
J = jacobian([x; y; z], [r l f])
```

return the Jacobian

```
J =
[    cos(l)*cos(f), -r*sin(l)*cos(f), -r*cos(l)*sin(f)]
[    cos(l)*sin(f), -r*sin(l)*sin(f),  r*cos(l)*cos(f)]
[           sin(l),         r*cos(l),                0]
```

and the command

```
detJ = simple(det(J))
```

returns

```
detJ =
-cos(l)*r^2
```

Notice that the first argument of the `jacobian` function must be a column vector and the second argument a row vector. Moreover, since the determinant of the Jacobian is a rather complicated trigonometric expression, you can use the `simple` command to make trigonometric substitutions and reductions (simplifications). The section “Simplifications and Substitutions” on page 2-42 discusses simplification in more detail.

A table summarizing `diff` and `jacobian` follows.

Mathematical Operator	MATLAB Command
$\frac{df}{dx}$	`diff(f)` or `diff(f,x)`
$\frac{df}{da}$	`diff(f,a)`
$\frac{d^2f}{db^2}$	`diff(f,b,2)`
$J = \frac{\partial(r,t)}{\partial(u,v)}$	`J = jacobian([r;t],[u,v])`

Limits

The fundamental idea in calculus is to make calculations on functions as a variable "gets close to" or approaches a certain value. Recall that the definition of the derivative is given by a limit

$$f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

provided this limit exists. Symbolic Math Toolbox enables you to calculate the limits of functions directly. The commands

```
syms h n x
limit( (cos(x+h) - cos(x))/h,h,0 )
```

which return

```
ans =
-sin(x)
```

and

```
limit( (1 + x/n)^n,n,inf )
```

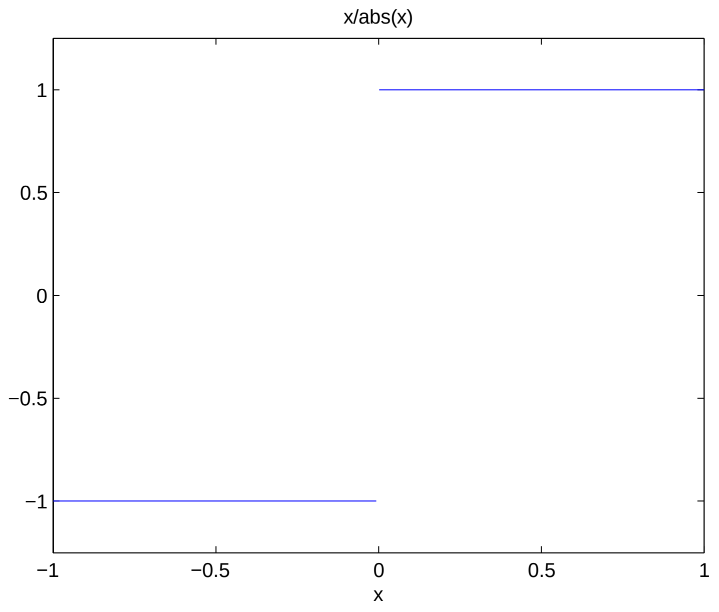
which returns

```
ans =
exp(x)
```

illustrate two of the most important limits in mathematics: the derivative (in this case of cos x) and the exponential function.

One-Sided Limits

You can also calculate one-sided limits with Symbolic Math Toolbox. For example, you can calculate the limit of $x/|x|$, whose graph is shown in the following figure, as x approaches 0 from the left or from the right.



To calculate the limit as x approaches 0 from the left,

$$\lim_{x \to 0^-} \frac{x}{|x|}$$

enter

```
limit(x/abs(x),x,0,'left')
```

This returns

```
ans =
 -1
```

To calculate the limit as x approaches 0 from the right,

$$\lim_{x \to 0^+} \frac{x}{|x|} = 1$$

enter

```
limit(x/abs(x),x,0,'right')
```

This returns

```
ans =
1
```

Since the limit from the left does not equal the limit from the right, the two-sided limit does not exist. In the case of undefined limits, MATLAB returns `NaN` (not a number). For example,

```
limit(x/abs(x),x,0)
```

returns

```
ans =
NaN
```

Observe that the default case, `limit(f)` is the same as `limit(f,x,0)`. Explore the options for the `limit` command in this table, where `f` is a function of the symbolic object `x`.

Mathematical Operation	MATLAB Command
$\lim_{x \to 0} f(x)$	`limit(f)`
$\lim_{x \to a} f(x)$	`limit(f,x,a)` or `limit(f,a)`
$\lim_{x \to a^-} f(x)$	`limit(f,x,a,'left')`
$\lim_{x \to a^+} f(x)$	`limit(f,x,a,'right')`

Integration

If `f` is a symbolic expression, then

```
int(f)
```

attempts to find another symbolic expression, `F`, so that `diff(F) = f`. That is, `int(f)` returns the indefinite integral or antiderivative of `f` (provided one exists in closed form). Similar to differentiation,

```
int(f,v)
```

uses the symbolic object `v` as the variable of integration, rather than the variable determined by `findsym`. See how `int` works by looking at this table.

Mathematical Operation	MATLAB Command
$\int x^n dx = \frac{x^{n+1}}{n+1}$	`int(x^n)` or `int(x^n,x)`
$\int_0^{\pi/2} \sin(2x)dx = 1$	`int(sin(2*x),0,pi/2)` or `int(sin(2*x),x,0,pi/2)`

Mathematical Operation	MATLAB Command
$g = cos(at + b)$ $\int g(t)dt = \sin(at+b)/a$	`g = cos(a*t + b) int(g)` or `int(g,t)`
$\int J_1(z)dz = -J_0(z)$	`int(besselj(1,z))` or `int(besselj(1,z),z)`

In contrast to differentiation, symbolic integration is a more complicated task. A number of difficulties can arise in computing the integral:

- The antiderivative, `F`, may not exist in closed form.
- The antiderivative may define an unfamiliar function.
- The antiderivative may exist, but the software can't find the it.
- The software could find the antiderivative on a larger computer, but runs out of time or memory on the available machine.

Nevertheless, in many cases, MATLAB can perform symbolic integration successfully. For example, create the symbolic variables

```
syms a b theta x y n u z
```

The following table illustrates integration of expressions containing those variables.

f	int(f)
`x^n`	`x^(n+1)/(n+1)`
`y^(-1)`	`log(y)`
`n^x`	`1/log(n)*n^x`
`sin(a*theta+b)`	`-1/a*cos(a*theta+b)`
`1/(1+u^2)`	`atan(u)`
`exp(-x^2)`	`1/2*pi^(1/2)*erf(x)`

In the last example, `exp(-x^2)`, there is no formula for the integral involving standard calculus expressions, such as trigonometric and exponential functions. In this case, MATLAB returns an answer in terms of the error function `erf`.

If MATLAB is unable to find an answer to the integral of a function `f`, it just returns `int(f)`.

Definite integration is also possible. The commands

```
int(f,a,b)
```

and

```
int(f,v,a,b)
```

are used to find a symbolic expression for

$$\int_a^b f(x)dx$$

and

$$\int_a^b f(v)dv$$

respectively.

Here are some additional examples.

f	a, b	int(f,a,b)
`x^7`	`0, 1`	`1/8`
`1/x`	`1, 2`	`log(2)`
`log(x)*sqrt(x)`	`0, 1`	`-4/9`
`exp(-x^2)`	`0, inf`	`1/2*pi^(1/2)`
`besselj(1,z)^2`	`0, 1`	`1/12*hypergeom([3/2, 3/2], [2, 5/2, 3],-1)`

For the Bessel function (besselj) example, it is possible to compute a numerical approximation to the value of the integral, using the double function. The commands

```
syms z
a = int(besselj(1,z)^2,0,1)
```

return

```
a =
1/12*hypergeom([3/2, 3/2],[2, 5/2, 3],-1)
```

and the command

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

returns

```
a =
  0.0717
```

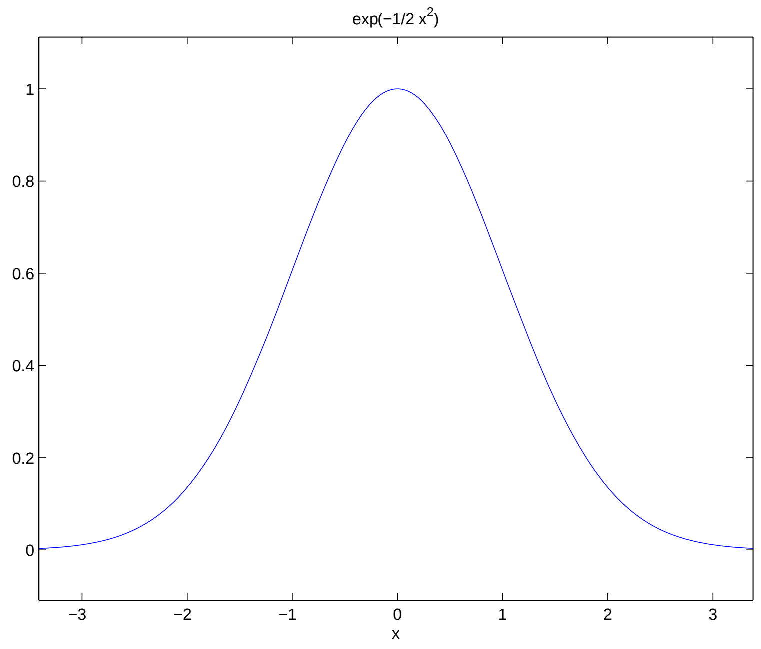
Integration with Real Parameters

One of the subtleties involved in symbolic integration is the “value” of various parameters. For example, if a is any positive real number, the expression

$$e^{-ax^2}$$

is the positive, bell shaped curve that tends to 0 as x tends to $\pm\infty$. You can create an example of this curve, for a = 1/2, using the following commands:

```
syms x
a = sym(1/2);
f = exp(-a*x^2);
ezplot(f)
```



However, if you try to calculate the integral

$$\int_{-\infty}^{\infty} e^{-ax^2}\,dx$$

without assigning a value to a, MATLAB assumes that a represents a complex number, and therefore returns a complex answer. If you are only interested in the case when a is a positive real number, you can calculate the integral as follows:

```
syms a positive;
```

The argument `positive` in the `syms` command restricts `a` to have positive values. Now you can calculate the preceding integral using the commands

```
syms x;
f = exp(-a*x^2);
```

```
int(f,x,-inf,inf)
```

This returns

```
ans =
1/(a)^(1/2)*pi^(1/2)
```

If you want to calculate the integral

$$\int_{-\infty}^{\infty} e^{-ax^2}\,dx$$

for any real number a, not necessarily positive, you can declare a to be real with the following commands:

```
syms a real
f=exp(-a*x^2);
F = int(f, x, -inf, inf)
```

MATLAB returns

```
F =
PIECEWISE([1/a^(1/2)*pi^(1/2), signum(a) = 1],[Inf, otherwise])
```

You can put this in a more readable form by entering

```
pretty(F)
                            {   1/2
                            { pi
                            { -----        signum(a~) = 1
                            {   1/2
                            { a~
                            {
                            {  Inf            otherwise
```

The ~ after a is simply a reminder that a is real, and signum(a~) is the sign of a. So the integral is

$$\frac{\sqrt{\pi}}{\sqrt{a}}$$

when a is positive, just as in the preceding example, and ∞ when a is negative.

You can also declare a sequence of symbolic variables w, y, x, z to be real by entering

```
syms w x y z real
```

Integration with Complex Parameters

To calculate the integral

$$\int_{-\infty}^{\infty} e^{-ax^2}\,dx$$

for complex values of a, enter

```
syms a x unreal %
f = exp(-a*x^2);
F = int(f, x, -inf, inf)
```

Note that syms is used with the unreal option to clear the real property that was assigned to a in the preceding example — see "Clearing Variables in the Maple Workspace" on page 1-15.

The preceding commands produce the complex output

```
F =
PIECEWISE([1/a^(1/2)*pi^(1/2), csgn(a) = 1],[Inf, otherwise])
```

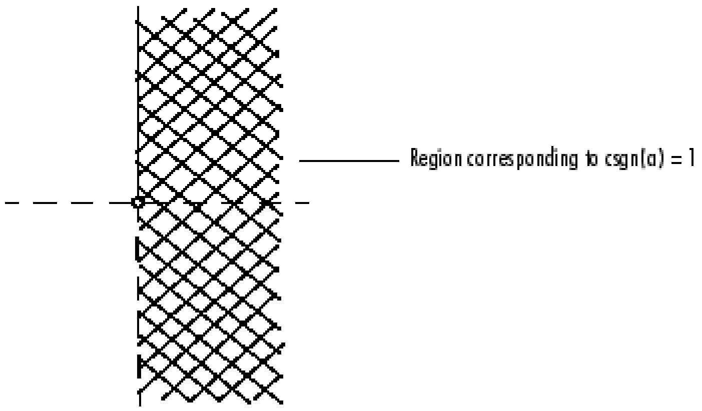
You can make this output more readable by entering

```
pretty(F)
                          {   1/2
                          { pi
                          { -----        csgn(a) = 1
                          {  1/2
                          { a
                          {
                          {  Inf          otherwise
```

The expression `csgn(a)` (complex sign of `a`) is defined by

$$c\,\mathrm{sgn}(a) = \begin{cases} 1 & \text{if } \mathrm{Re}(a) > 0, \text{or } \mathrm{Re}(a) = 0 \text{ and } \mathrm{Im}(a) > 0 \\ -1 & \text{if } \mathrm{Re}(a) < 0, \text{or } \mathrm{Re}(a) = 0 \text{ and } \mathrm{Im}(a) < 0 \end{cases}$$

The condition csgn(a) = 1 corresponds to the shaded region of the complex plane shown in the following figure.



The square root of `a` in the answer is the unique square root lying in the shaded region.

Symbolic Summation

You can compute symbolic summations, when they exist, by using the `symsum` command. For example, the p-series

$$1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$$

sums to $\pi^2/6$, while the geometric series

$$1 + x + x^2 + \ldots$$

sums to $1/(1-x)$, provided $|x| < 1$. Three summations are demonstrated below:

```
syms x k
s1 = symsum(1/k^2,1,inf)
s2 = symsum(x^k,k,0,inf)

s1 =

1/6*pi^2

s2 =

-1/(x-1)
```

Taylor Series

The statements

```
syms x
f = 1/(5+4*cos(x))
T = taylor(f,8)
```

return

```
T =
1/9+2/81*x^2+5/1458*x^4+49/131220*x^6
```

which is all the terms up to, but not including, order eight in the Taylor series for *f*(*x*):

$$\sum_{n=0}^{\infty}(x-a)^n\frac{f^{(n)}(a)}{n!}$$

Technically, `T` is a Maclaurin series, since its basepoint is `a = 0`.

The command

```
pretty(T)
```

prints `T` in a format resembling typeset mathematics:

```
            2            4      49    6
1/9 + 2/81 x  + 5/1458 x  + ------ x
```

```
                        131220
```

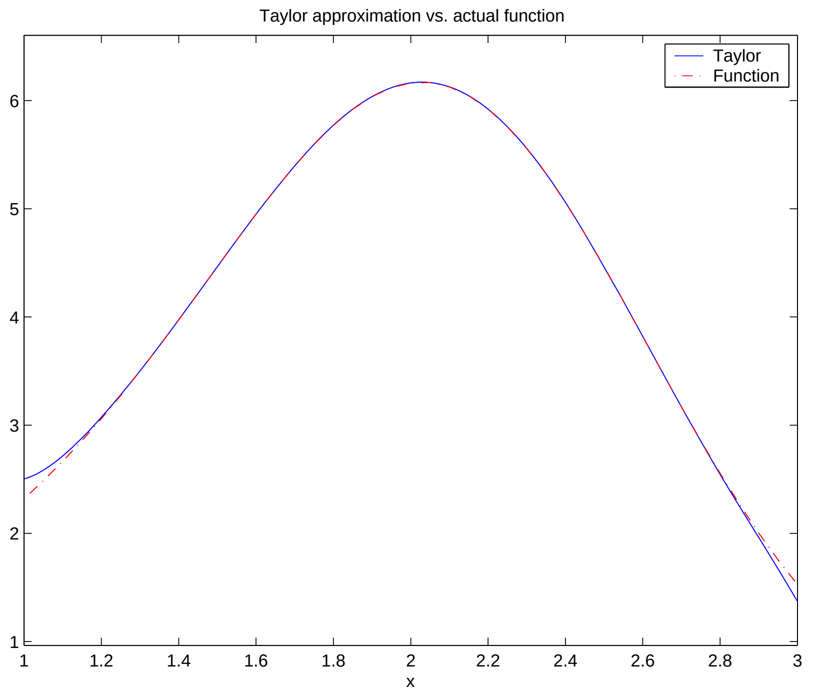
These commands

```
syms x
g = exp(x*sin(x))
t = taylor(g,12,2);
```

generate the first 12 nonzero terms of the Taylor series for `g` about `x = 2`.

Next, plot these functions together to see how well this Taylor approximation compares to the actual function `g`:

```
xd = 1:0.05:3; yd = subs(g,x,xd);
ezplot(t, [1,3]); hold on;
plot(xd, yd, 'r-.')
title('Taylor approximation vs. actual function');
legend('Taylor','Function')
```



Special thanks to Professor Gunnar Bäckstrøm of UMEA in Sweden for this example.

Calculus Example

This section describes how to analyze a simple function to find its asymptotes, maximum, minimum, and inflection point. The section covers the following topics:

- "Defining the Function" on page 2-22
- "Finding the Asymptotes" on page 2-23
- "Finding the Maximum and Minimum" on page 2-25
- "Finding the Inflection Point" on page 2-27

Defining the Function

The function in this example is

$$f(x)=\frac{3x^2+6x-1}{x^2+x-3}$$

To create the function, enter the following commands:

```
syms x
num = 3*x^2 + 6*x -1;
denom = x^2 + x - 3;
f = num/denom
```

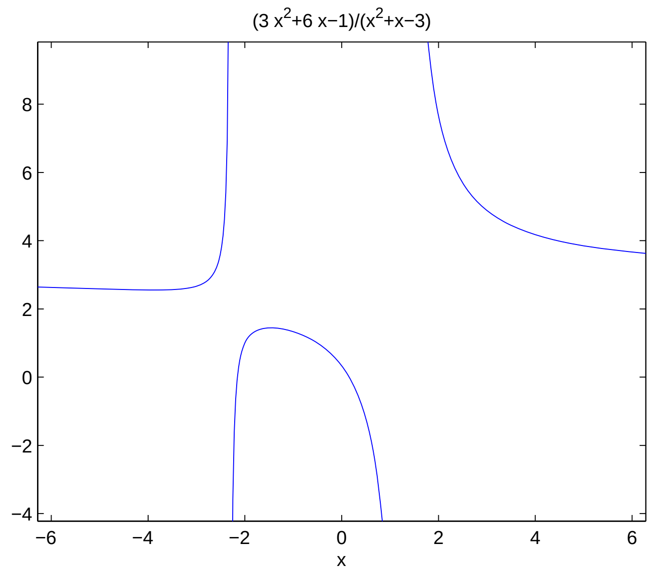
This returns

```
f =
(3*x^2+6*x-1)/(x^2+x-3)
```

You can plot the graph of `f` by entering

```
ezplot(f)
```

This displays the following plot.



Finding the Asymptotes

To find the horizontal asymptote of the graph of `f`, take the limit of `f` as `x` approaches positive infinity:

```
limit(f, inf)
ans =
3
```

The limit as x approaches negative infinity is also 3. This tells you that the line $y = 3$ is a horizontal asymptote to the graph.

To find the vertical asymptotes of `f`, set the denominator equal to 0 and solve by entering the following command:

```
`roots = solve(denom)
```

This returns to solutions to $x^2 + x - 3 = 0$:

```
roots =
[ -1/2+1/2*13^(1/2)]
[ -1/2-1/2*13^(1/2)]
```

This tells you that vertical asymptotes are the lines

$$x = \frac{-1+\sqrt{13}}{2}$$

and

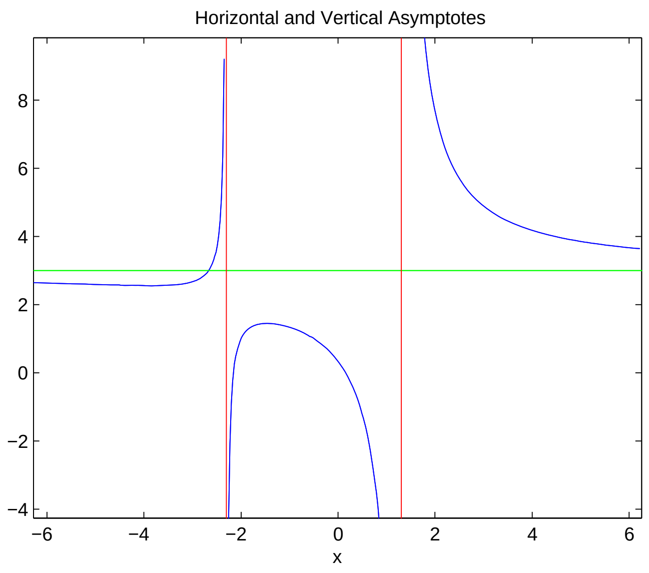
$$x = \frac{-1-\sqrt{13}}{2}$$

You can plot the horizontal and vertical asymptotes with the following commands:

```
ezplot(f)
hold on % Keep the graph of f in the figure
% Plot horizontal asymptote
plot([-2*pi 2*pi], [3 3],'g')
% Plot vertical asymptotes
plot(double(roots(1))*[1 1], [-5 10],'r')
plot(double(roots(2))*[1 1], [-5 10],'r')
title('Horizontal and Vertical Asymptotes')
hold off
```

Note that `roots` must be converted to `double` to use the `plot` command.

The preceding commands display the following figure.



To recover the graph of `f` without the asymptotes, enter

```
ezplot(f)
```

Finding the Maximum and Minimum

You can see from the graph that `f` has a local maximum somewhere between the points x = 2 and x = 3, and might have a local minimum between x = -4 and x = -2. To find the x-coordinates of the maximum and minimum, first take the derivative of `f`:

```
f1 = diff(f)
```

This returns

```
f1 = (6*x+6)/(x^2+x-3)-(3*x^2+6*x-1)/(x^2+x-3)^2*(2*x+1)
```

To simplify this expression, enter

```
f1 = simplify(f1)
```

which returns

```
f1 = -(3*x^2+16*x+17)/(x^2+x-3)^2
```

You can display `f1` in a more readable form by entering

```
pretty(f1)
```

which returns

```
                   2
              3 x   + 16 x + 17
            - ----------------
                  2         2
               (x   + x - 3)
```

Next, set the derivative equal to 0 and solve for the critical points:

```
crit_pts = solve(f1)
```

This returns

```
ans =
[ -8/3-1/3*13^(1/2)]
[ -8/3+1/3*13^(1/2)]
```

It is clear from the graph of `f` that it has a local minimum at

$$x_1 = \frac{-8-\sqrt{13}}{3}$$

and a local maximum at

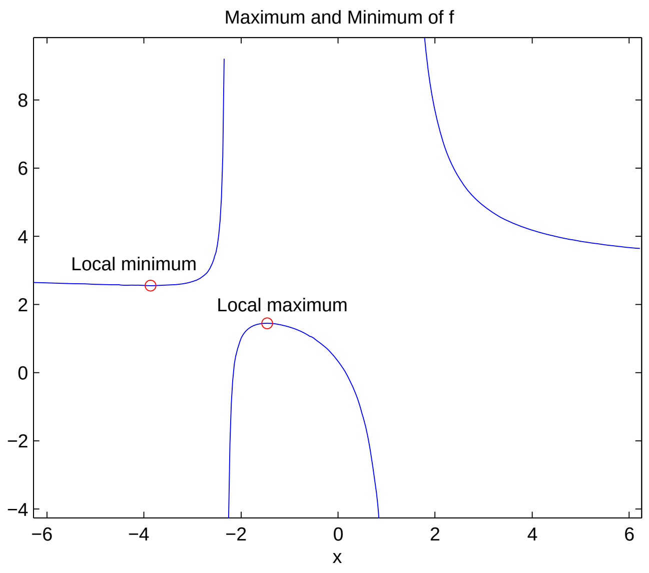
$$x_2 = \frac{-8+\sqrt{13}}{3}$$

Note MATLAB does not always return the roots to an equation in the same order.

You can plot the maximum and minimum of `f` with the following commands:

```
ezplot(f)
hold on
plot(double(crit_pts), double(subs(f,crit_pts)),'ro')
title('Maximum and Minimum of f')
text(-5.5,3.2,'Local minimum')
text(-2.5,2,'Local maximum')
hold off
```

This displays the following figure.



Finding the Inflection Point

To find the inflection point of `f`, set the second derivative equal to 0 and solve.

```
f2 = diff(f1);
inflec_pt = solve(f2);
double(inflec_pt)
```

This returns

```
ans =
   -5.2635
   -1.3682 - 0.8511i
   -1.3682 + 0.8511i
```

In this example, only the first entry is a real number, so this is the only inflection point. (Note that in other examples, the real solutions might not be the first entries of the answer.) Since you are only interested in the real solutions, you can discard the last two entries, which are complex numbers.

```
inflec_pt = inflec_pt(1)
```

To see the symbolic expression for the inflection point, enter

```
pretty(simplify(inflec_pt))
```

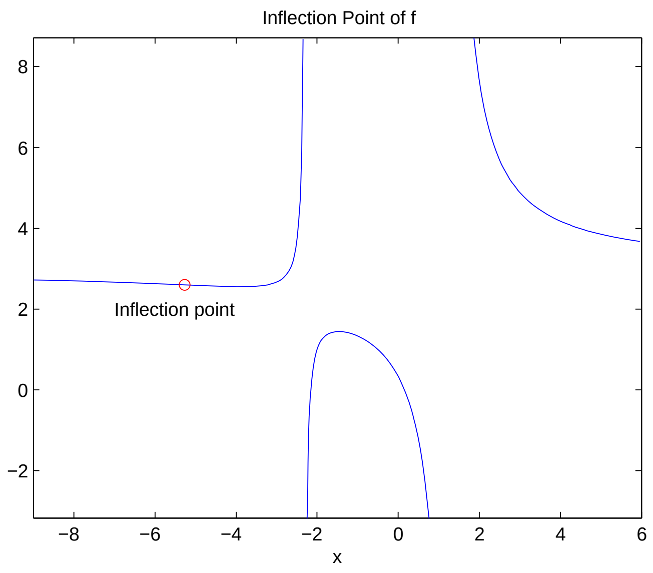
This returns

```
                    1/2 2/3                        1/2 1/3
       (676 + 156 13   )    + 52 + 16 (676 + 156 13   )
- 1/6 -----------------------------------------------------
                                    1/2 1/3
                      (676 + 156 13   )
```

To plot the inflection point, enter

```
ezplot(f, [-9 6])
hold on
plot(double(inflec_pt), double(subs(f,inflec_pt)),'ro')
title('Inflection Point of f')
text(-7,2,'Inflection point')
hold off
```

The extra argument, `[-9 6]`, in `ezplot` extends the range of x values in the plot so that you see the inflection point more clearly, as shown in the following figure.



Extended Calculus Example

This section presents an extended example that illustrates how to find the maxima and minima of a function. The section covers the following topics:

- "Defining the Function" on page 2-29
- "Finding the Zeros of f3" on page 2-31
- "Finding the Maxima and Minima of f2" on page 2-34
- "Integrating" on page 2-36

Defining the Function

The starting point for the example is the function

$$f(x) = \frac{1}{5 + 4\cos(x)}$$

You can create the function with the commands

```
syms x
f = 1/(5+4*cos(x))
```

which return

```
f =
1/(5+4*cos(x))
```

The example shows how to find the maximum and minimum of the second derivative of *f*(*x*). To compute the second derivative, enter

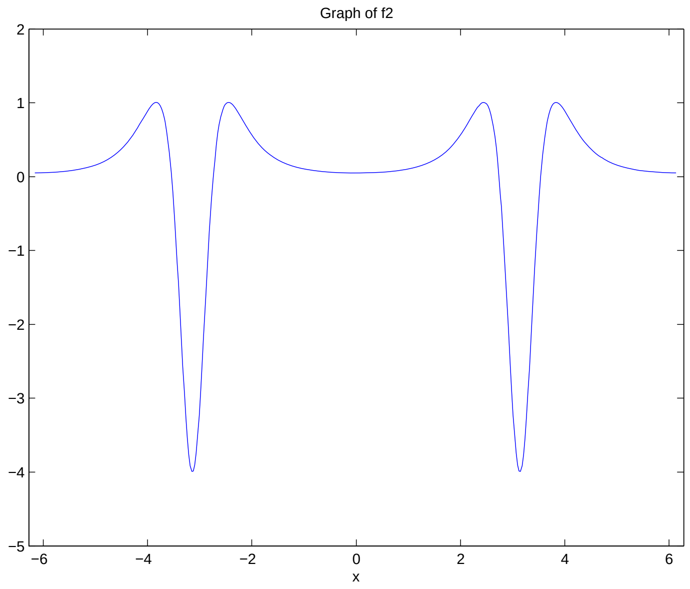
```
f2 = diff(f,2)
```

which returns

```
f2 =
32/(5+4*cos(x))^3*sin(x)^2+4/(5+4*cos(x))^2*cos(x)
```

Equivalently, you can type `f2 = diff(f,x,2)`. The default scaling in `ezplot` cuts off part of the graph of `f2`. You can set the axes limits manually to see the entire function:

```
ezplot(f2)
axis([-2*pi 2*pi -5 2])
title('Graph of f2')
```



From the graph, it appears that the maximum value of $f''(x)$ is 1 and the minimum value is -4. As you will see, this is not quite true. To find the exact values of the maximum and minimum, you only need to find the maximum and minimum on the interval $[-\pi\pi]$. This is true because $f''(x)$ is periodic with period 2π, so that the maxima and minima are simply repeated in each translation of this interval by an integer multiple of 2π. The next two sections explain how to do find the maxima and minima.

Finding the Zeros of f3

The maxima and minima of $f''(x)$ occur at the zeros of $f'''(x)$. The statements

```
f3 = diff(f2);
pretty(f3)
```

compute $f'''(x)$ and display it in a more readable form:

```
                  3
```

```
        sin(x)               sin(x) cos(x)           sin(x)
384 -------------- + 96 -------------- - 4 --------------
                  4                   3                 2
    (5 + 4 cos(x))      (5 + 4 cos(x))     (5 + 4 cos(x))
```

You can simplify this expression using the statements

```
f3 = simple(f3);
pretty(f3)
                   2                        2
  sin(x) (96 sin(x)  + 80 cos(x) + 80 cos(x)  - 25)
4 -------------------------------------------------
                                4
                  (5 + 4 cos(x))
```

Now, to find the zeros of $f'''(x)$, enter

```
zeros = solve(f3)
```

This returns a 5-by-1 symbolic matrix

```
zeros =
[                                                0]
[       atan((-255-60*19^(1/2))^(1/2),10+3*19^(1/2))]
[      atan(-(-255-60*19^(1/2))^(1/2),10+3*19^(1/2))]
[  atan((-255+60*19^(1/2))^(1/2)/(10-3*19^(1/2)))+pi]
[ -atan((-255+60*19^(1/2))^(1/2)/(10-3*19^(1/2)))-pi]
```

each of whose entries is a zero of $f'''(x)$. The commands

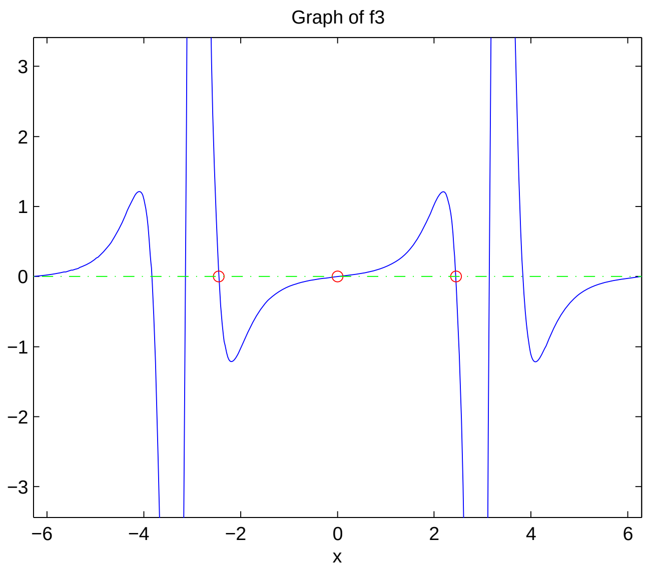
```
format; % Default format of 5 digits
zerosd = double(zeros)
```

convert the zeros to double form:

```
zerosd =
        0
        0+ 2.4381i
        0- 2.4381i
   2.4483
  -2.4483
```

So far, you have found three real zeros and two complex zeros. However, as the following graph of f3 shows, these are not all its zeros:

```
ezplot(f3)
hold on;
plot(zerosd,0*zerosd,'ro') % Plot zeros
plot([-2*pi,2*pi], [0,0],'g-.'); % Plot x-axis
title('Graph of f3')
```



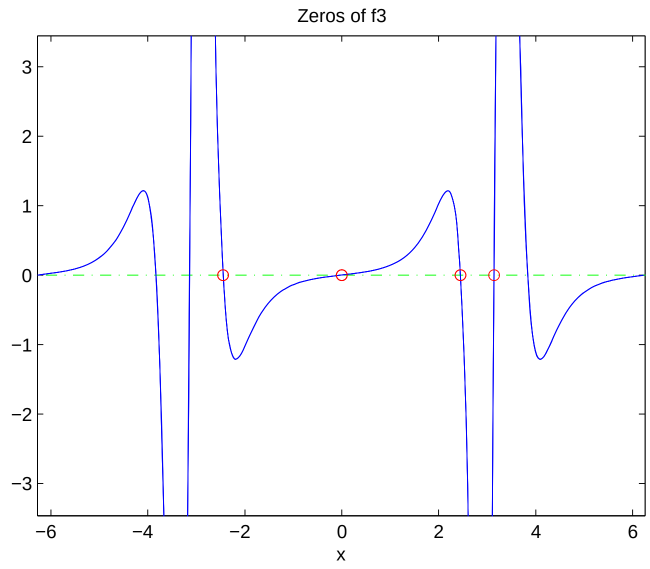
The red circles in the graph correspond to zerosd(1), zerosd(4), and zerosd(5). As you can see in the graph, there are also zeros at $\pm\pi$. The additional zeros occur because $f'''(x)$ contains a factor of $\sin(x)$, which is zero at integer multiples of π. The function, solve(sin(x)), however, only finds the zero at $x = 0$.

A complete list of the zeros of $f'''(x)$ in the interval $[-\pi\pi]$ is

```
zerosd = [zerosd(1) zerosd(4) zerosd(5) pi];
```

You can display these zeros on the graph of $f'''(x)$ with the following commands:

```
ezplot(f3)
hold on;
plot(zerosd,0*zerosd,'ro')
plot([-2*pi,2*pi], [0,0],'g-.'); % Plot x-axis
title('Zeros of f3')
hold off;
```



Finding the Maxima and Minima of f2

To find the maxima and minima of $f''(x)$, calculate the value of $f''(x)$ at each of the zeros of $f'''(x)$. To do so, substitute `zeros` into `f2` and display the result below `zeros`:

```
[zerosd; subs(f2,zerosd)]
ans =
        0     2.4483   -2.4483    3.1416
   0.0494     1.0051    1.0051   -4.0000
```

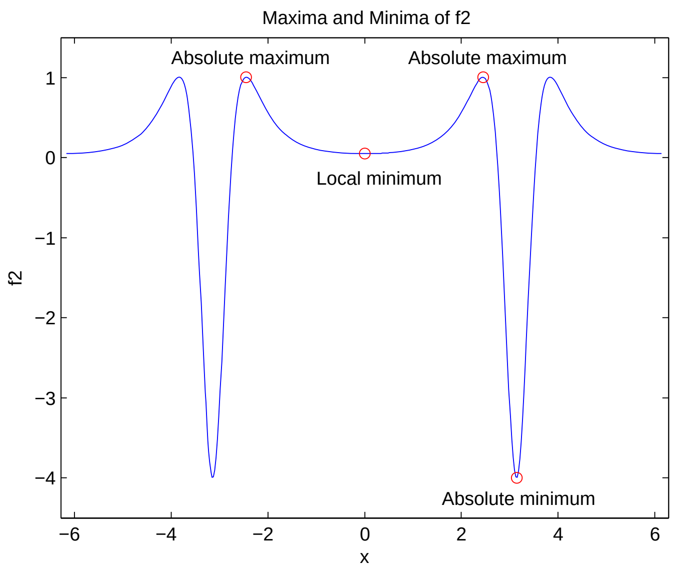
This shows the following:

- $f''(x)$ has an absolute maximum at $x = \pm 2.4483$, whose value is 1.0051.
- $f''(x)$ has an absolute minimum at $x = \pi$, whose value is -4.
- $f''(x)$ has a local minimum at $x = \pi$, whose value is 0.0494.

You can display the maxima and minima with the following commands:

```
clf
ezplot(f2)
axis([-2*pi 2*pi -4.5 1.5])
ylabel('f2');
title('Maxima and Minima of f2')
hold on
plot(zeros, subs(f2,zeros), 'ro')
text(-4, 1.25, 'Absolute maximum')
text(-1,-0.25,'Local minimum')
text(.9, 1.25, 'Absolute maximum')
text(1.6, -4.25, 'Absolute minimum')
hold off;
```

This displays the following figure.



The preceding analysis shows that the actual range of $f''(x)$ is [-4, 1.0051].

Integrating

To see whether integrating $f''(x)$ twice with respect to x recovers the original function $f(x) = 1/(5 + 4\ cosx)$, enter the command

```
g = int(int(f2))
```

which returns

```
g =
-8/(tan(1/2*x)^2+9)
```

This is certainly not the original expression for $f(x)$. Now look at the difference $f(x) - g(x)$.

```
d = f - g
pretty(d)
     1                    8
------------ + ---------------
```

```
5 + 4 cos(x)              2
               tan(1/2 x)  + 9
```

You can simplify this using `simple(d)` or `simplify(d)`. Either command produces

```
ans =
1
```

This illustrates the concept that differentiating $f(x)$ twice, then integrating the result twice, produces a function that may differ from $f(x)$ by a linear function of x.

Finally, integrate $f(x)$ once more:

```
F = int(f)
```

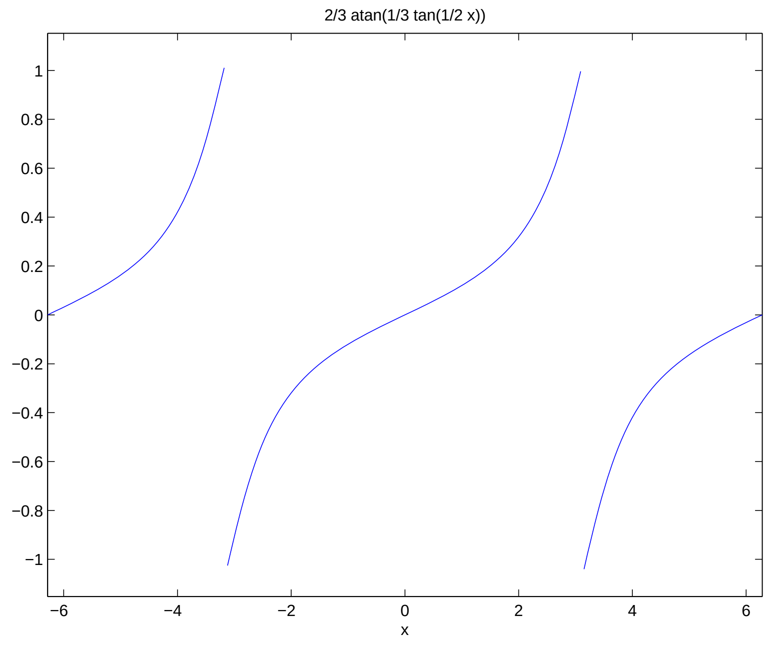
The result

```
F =
2/3*atan(1/3*tan(1/2*x))
```

involves the arctangent function.

Note that $F(x)$ is not an antiderivative of $f(x)$ for all real numbers, since it is discontinuous at odd multiples of π, where tan (x) is singular. You can see the gaps in $F(x)$ in the following figure.

```
ezplot(F)
```



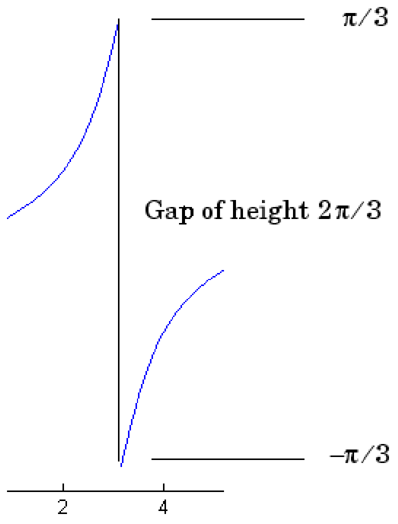
To change $F(x)$ into a true antiderivative of $f(x)$ that is differentiable everywhere, you can add a step function to $F(x)$. The height of the steps is the height of the gaps in the graph of $F(x)$. You can determine the height of the gaps by taking the limits of $F(x)$ as x approaches π from the left and from the right. The limit from the left is

```
limit(F, x, pi, 'left')
ans =
1/3*pi
```

On the other hand, the limit from the right is

```
limit(F, x, pi, 'right')
ans =-1/3*pi
```

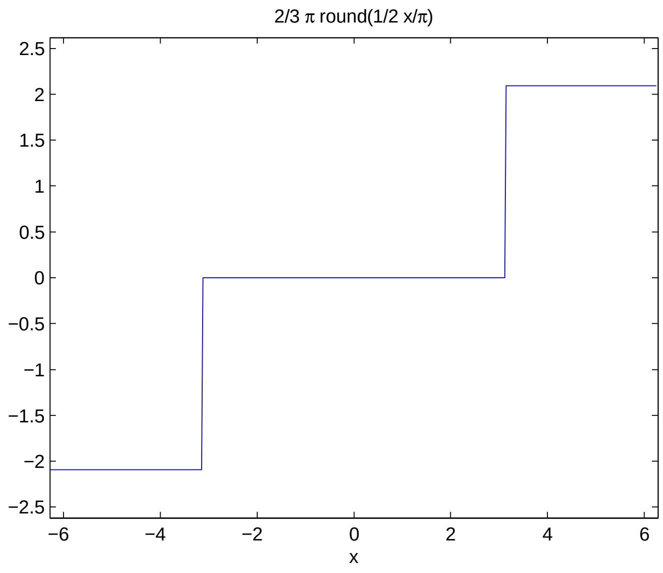
The height of the gap is the distance between the left-and right-hand limits, which is $2\pi/3$, as shown in the following figure.



You can create the step function using the `round` function, which rounds numbers to the nearest integer, as follows:

```
J = sym(2*pi/3)*sym('round(x/(2*pi))');
```

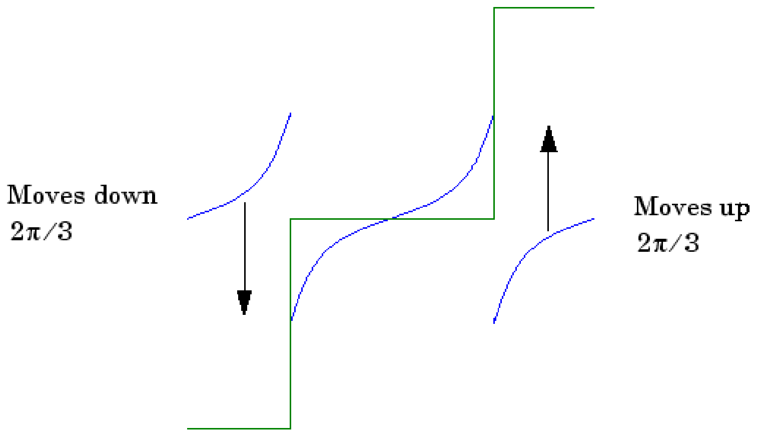
Each step has width 2π and the jump from one step to the next is $2\pi/3$, as shown in the following figure.



Next, add the step function $J(x)$ to $F(x)$ with the following code:

```
F1 = F+J
F1 =
2/3*atan(1/3*tan(1/2*x))+2/3*pi*round(1/2*x/pi)
```

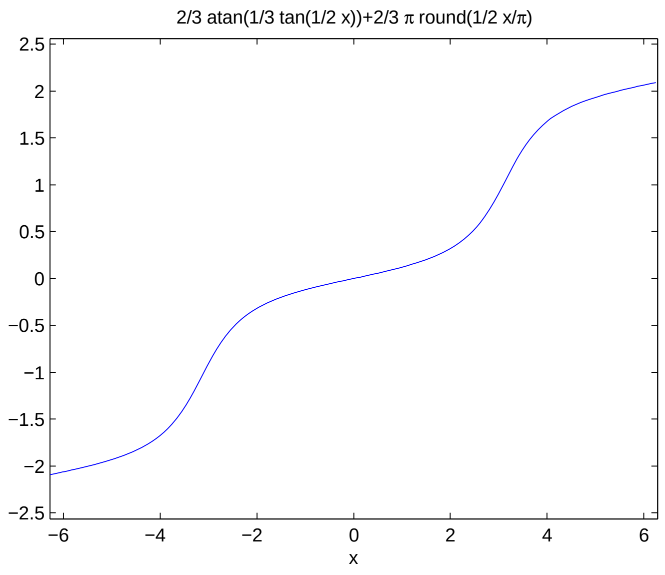
Adding the step function raises the section of the graph of $F(x)$ on the interval $[\pi\ 3\pi]$ up by $2\pi/3$, lowers the section on the interval $[-3\pi\ -\pi]$ down by $2\pi/3$, and so on, as shown in the following figure.



When you plot the result by entering

```
ezplot(F1)
```

you see that this representation does have a continuous graph.



Simplifications and Substitutions

In this section...

Simplifications

Here are three different symbolic expressions.

```
syms x
f = x^3-6*x^2+11*x-6
g = (x-1)*(x-2)*(x-3)
h = -6+(11+(-6+x)*x)*x
```

Here are their prettyprinted forms, generated by

```
pretty(f), pretty(g), pretty(h)
 3     2
x - 6 x  + 11 x - 6

(x - 1) (x - 2) (x - 3)

-6 + (11 + (-6 + x) x) x
```

These expressions are three different representations of the same mathematical function, a cubic polynomial in x.

Each of the three forms is preferable to the others in different situations. The first form, f, is the most commonly used representation of a polynomial. It is simply a linear combination of the powers of x. The second form, g, is the factored form. It displays the roots of the polynomial and is the most accurate for numerical evaluation near the roots. But, if a polynomial does not have such simple roots, its factored form may not be so convenient. The third form, h, is the Horner, or nested, representation. For numerical evaluation, it involves the fewest arithmetic operations and is the most accurate for some other ranges of x.

The symbolic simplification problem involves the verification that these three expressions represent the same function. It also involves a less clearly defined objective — which of these representations is "the simplest"?

This toolbox provides several functions that apply various algebraic and trigonometric identities to transform one representation of a function into another, possibly simpler, representation. These functions are `collect`, `expand`, `horner`, `factor`, `simplify`, and `simple`.

collect

The statement

```
collect(f)
```

views `f` as a polynomial in its symbolic variable, say `x`, and collects all the coefficients with the same power of `x`. A second argument can specify the variable in which to collect terms if there is more than one candidate. Here are a few examples.

f	collect(f)
`(x-1)*(x-2)*(x-3)`	`x^3-6*x^2+11*x-6`
`x*(x*(x-6)+11)-6`	`x^3-6*x^2+11*x-6`
`(1+x)*t + x*t`	`2*x*t+t`

expand

The statement

```
expand(f)
```

distributes products over sums and applies other identities involving functions of sums as shown in the examples below.

f	expand(f)
`a*(x + y)`	`a*x + a*y`
`(x-1)*(x-2)*(x-3)`	`x^3-6*x^2+11*x-6`
`x*(x*(x-6)+11)-6`	`x^3-6*x^2+11*x-6`

f	expand(f)
`exp(a+b)`	`exp(a)*exp(b)`
`cos(x+y)`	`cos(x)*cos(y)-sin(x)*sin(y)`
`cos(3*acos(x))`	`4*x^3-3*x`

horner

The statement

```
horner(f)
```

transforms a symbolic polynomial `f` into its Horner, or nested, representation as shown in the following examples.

f	horner(f)
`x^3-6*x^2+11*x-6`	`-6+(11+(-6+x)*x)*x`
`1.1+2.2*x+3.3*x^2`	`11/10+(11/5+33/10*x)*x`

factor

If `f` is a polynomial with rational coefficients, the statement

```
factor(f)
```

expresses `f` as a product of polynomials of lower degree with rational coefficients. If `f` cannot be factored over the rational numbers, the result is `f` itself. Here are several examples.

f	factor(f)
`x^3-6*x^2+11*x-6`	`(x-1)*(x-2)*(x-3)`
`x^3-6*x^2+11*x-5`	`x^3-6*x^2+11*x-5`
`x^6+1`	`(x^2+1)*(x^4-x^2+1)`

Here is another example involving `factor`. It factors polynomials of the form `x^n + 1`. This code

```
syms x;
n = (1:9)';
p = x.^n + 1;
f = factor(p);
[p, f]
```

returns a matrix with the polynomials in its first column and their factored forms in its second.

```
[                      x+1,                                x+1 ]
[                    x^2+1,                              x^2+1 ]
[                    x^3+1,                    (x+1)*(x^2-x+1) ]
[                    x^4+1,                              x^4+1 ]
[                    x^5+1,            (x+1)*(x^4-x^3+x^2-x+1) ]
[                    x^6+1,                (x^2+1)*(x^4-x^2+1) ]
[                    x^7+1, (x+1)*(1-x+x^2-x^3+x^4-x^5+x^6) ]
[                    x^8+1,                              x^8+1 ]
[                    x^9+1,      (x+1)*(x^2-x+1)*(x^6-x^3+1) ]
```

As an aside at this point, `factor` can also factor symbolic objects containing integers. This is an alternative to using the `factor` function in the MATLAB `specfun` directory. For example, the following code segment

```
N = sym(1);
for k = 2:11
   N(k) = 10*N(k-1)+1;
end
[N' factor(N')]
```

displays the factors of symbolic integers consisting of 1s:

```
[                     1,                        1]
[                    11,                     (11)]
[                   111,                (3)*(37)]
[                  1111,              (11)*(101)]
[                 11111,              (41)*(271)]
[                111111, (3)*(7)*(11)*(13)*(37)]
[               1111111,            (239)*(4649)]
[              11111111,  (11)*(73)*(101)*(137)]
[             111111111,     (3)^2*(37)*(333667)]
[            1111111111, (11)*(41)*(271)*(9091)]
[           11111111111,       (513239)*(21649)]
```

simplify

The `simplify` function is a powerful, general purpose tool that applies a number of algebraic identities involving sums, integral powers, square roots and other fractional powers, as well as a number of functional identities involving trig functions, exponential and log functions, Bessel functions, hypergeometric functions, and the gamma function. Here are some examples.

f	simplify(f)
`x*(x*(x-6)+11)-6`	`x^3-6*x^2+11*x-6`
`(1-x^2)/(1-x)`	`x+1`
`(1/a^3+6/a^2+12/a+8)^(1/3)`	`((2*a+1)^3/a^3)^(1/3)`
`syms x y positive` `log(x*y)`	`log(x)+log(y)`
`exp(x) * exp(y)`	`exp(x+y)`
`besselj(2,x) + besselj(0,x)`	`2/x*besselj(1,x)`
`gamma(x+1)-x*gamma(x)`	`0`
`cos(x)^2 + sin(x)^2`	`1`

simple

The `simple` function has the unorthodox mathematical goal of finding a simplification of an expression that has the fewest number of characters. Of course, there is little mathematical justification for claiming that one expression is “simpler” than another just because its ASCII representation is shorter, but this often proves satisfactory in practice.

The `simple` function achieves its goal by independently applying `simplify`, `collect`, `factor`, and other simplification functions to an expression and keeping track of the lengths of the results. The `simple` function then returns the shortest result.

The `simple` function has several forms, each returning different output. The form

```
simple(f)
```

displays each trial simplification and the simplification function that produced it in the MATLAB command window. The `simple` function then returns the shortest result. For example, the command

```
simple(cos(x)^2 + sin(x)^2)
```

displays the following alternative simplifications in the MATLAB command window:

```
simplify:
1

radsimp:
cos(x)^2+sin(x)^2

combine(trig):
1

factor:
 cos(x)^2+sin(x)^2

expand:
cos(x)^2+sin(x)^2

combine:
1

convert(exp):
 (1/2*exp(i*x)+1/2/exp(i*x))^2-1/4*(exp(i*x)-1/exp(i*x))^2

convert(sincos):
cos(x)^2+sin(x)^2

convert(tan):
(1-tan(1/2*x)^2)^2/(1+tan(1/2*x)^2)^2+
4*tan(1/2*x)^2/(1+tan(1/2*x)^2)^2
```

```
collect(x):
cos(x)^2+sin(x)^2
```

and returns

```
ans =
1
```

This form is useful when you want to check, for example, whether the shortest form is indeed the simplest. If you are not interested in how `simple` achieves its result, use the form

```
f = simple(f)
```

This form simply returns the shortest expression found. For example, the statement

```
f = simple(cos(x)^2+sin(x)^2)
```

returns

```
f =
1
```

If you want to know which simplification returned the shortest result, use the multiple output form:

```
[F, how] = simple(f)
```

This form returns the shortest result in the first variable and the simplification method used to achieve the result in the second variable. For example, the statement

```
[f, how] = simple(cos(x)^2+sin(x)^2)
```

returns

```
f =
1

how =
combine
```

The `simple` function sometimes improves on the result returned by `simplify`, one of the simplifications that it tries. For example, when applied to the examples given for `simplify`, `simple` returns a simpler (or at least shorter) result in two cases.

f	**simplify(f)**	**simple(f)**
`(1/a^3+6/a^2+12/a+8)^(1/3)`	`((2*a+1)^3/a^3)^(1/3)`	`(2*a+1)/a`

`syms x y positive`
 `log(x*y)`
	`log(x)+log(y)`	`log(x*y)`

In some cases, it is advantageous to apply `simple` twice to obtain the effect of two different simplification functions. For example, the statements

```
f = (1/a^3+6/a^2+12/a+8)^(1/3);
simple(simple(f))
```

return

```
2+1/a
```

The first application, `simple(f)`, uses `radsimp` to produce `(2*a+1)/a`; the second application uses `combine(trig)` to transform this to `1/a+2`.

The `simple` function is particularly effective on expressions involving trigonometric functions. Here are some examples.

f	**simple(f)**
`cos(x)^2+sin(x)^2`	`1`
`2*cos(x)^2-sin(x)^2`	`3*cos(x)^2-1`
`cos(x)^2-sin(x)^2`	`cos(2*x)`
`cos(x)+(-sin(x)^2)^(1/2)`	`cos(x)+i*sin(x)`
`cos(x)+i*sin(x)`	`exp(i*x)`
`cos(3*acos(x))`	`4*x^3-3*x`

Substitutions

There are two functions for symbolic substitution: subexpr and subs.

subexpr

These commands

```
syms a x
s = solve(x^3+a*x+1)
```

solve the equation x^3+a*x+1 = 0 for x:

```
s =
[                            1/6*(-108+12*(12*a^3+81)^(1/2))^(1/3)-2*a/
                                  (-108+12*(12*a^3+81)^(1/2))^(1/3)]
[ -1/12*(-108+12*(12*a^3+81)^(1/2))^(1/3)+a/
      (-108+12*(12*a^3+81)^(1/2))^(1/3)+1/2*i*3^(1/2)*(1/
      6*(-108+12*(12*a^3+81)^(1/2))^(1/3)+2*a/
      (-108+12*(12*a^3+81)^(1/2))^(1/3))]
[ -1/12*(-108+12*(12*a^3+81)^(1/2))^(1/3)+a/
      (-108+12*(12*a^3+81)^(1/2))^(1/3)-1/2*i*3^(1/2)*(1/
      6*(-108+12*(12*a^3+81)^(1/2))^(1/3)+2*a/
      (-108+12*(12*a^3+81)^(1/2))^(1/3))]
```

Use the pretty function to display s in a more readable form:

```
pretty(s)

s =
    [                              1/3       a                         ]
    [                       1/6 %1     - 2 -----                       ]
    [                                        1/3                       ]
    [                                      %1                          ]
    [                                                                  ]
    [            1/3      a              1/2 /       1/3        a   \]
    [- 1/12 %1     + ----- + 1/2 i 3     |1/6 %1     + 2 -----|]
    [                  1/3               |                    1/3|]
    [                %1                  \                 %1    /]
    [                                                                  ]
    [            1/3      a              1/2 /       1/3        a   \]
    [- 1/12 %1     + ----- - 1/2 i 3     |1/6 %1     + 2 -----|]
    [                    1/3                |                  1/3|]
    [                  %1                   \                %1   /]

                                              3       1/2
                              %1 := -108 + 12 (12 a   + 81)
```

The pretty command inherits the %n (n, an integer) notation from Maple to denote subexpressions that occur multiple times in the symbolic object. The subexpr function allows you to save these common subexpressions as well as the symbolic object rewritten in terms of the subexpressions. The subexpressions are saved in a column vector called sigma.

Continuing with the example

```
r = subexpr(s)
```

returns

```
sigma =
-108+12*(12*a^3+81)^(1/2)
r =
[                                 1/6*sigma^(1/3)-2*a/sigma^(1/3)]
[ -1/12*sigma^(1/3)+a/sigma^(1/3)+1/2*i*3^(1/2)*(1/6*sigma^
     (1/3)+2*a/sigma^(1/3))]
[ -1/12*sigma^(1/3)+a/sigma^(1/3)-1/2*i*3^(1/2)*(1/6*sigma^
     (1/3)+2*a/sigma^(1/3))]
```

Notice that subexpr creates the variable sigma in the MATLAB workspace. You can verify this by typing whos, or the command

```
 sigma
```

which returns

```
sigma =
-108+12*(12*a^3+81)^(1/2)
```

subs

The following code finds the eigenvalues and eigenvectors of a circulant matrix A:

```
syms a b c
A = [a b c; b c a; c a b];
[v,E] = eig(A)
v =

[ -(a+(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2)-b)/(a-c),
          -(a-(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2)-b)/(a-c),  1]
[ -(b-c-(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2))/(a-c),
          -(b-c+(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2))/(a-c),  1]
[ 1,
         1,                                              1]

E =

[ (b^2-b*a-c*b-
   c*a+a^2+c^2)^(1/2),                    0,            0]
[                    0,   -(b^2-b*a-c*b-
                          c*a+a^2+c^2)^(1/2),           0]
[                    0,                   0,        b+c+a]
```

Note MATLAB might return the eigenvalues that appear on the diagonal of E in a different order. In this case, the corresponding eigenvectors, which are the columns of v, will also appear in a different order.

Suppose you want to replace the rather lengthy expression

```
(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2)
```

throughout v and E. First, use subexpr

```
v = subexpr(v,'S')
```

which returns

```
S =
(b^2-b*a-c*b-c*a+a^2+c^2)^(1/2)

v =
[ -(a+S-b)/(a-c), -(a-S-b)/(a-c),                1]
[ -(b-c-S)/(a-c), -(b-c+S)/(a-c),                1]
[              1,              1,                1]
```

Next, substitute the symbol S into E with

```
E = subs(E,S,'S')
E =
[     S,     0,     0]
[     0,    -S,     0]
[     0,     0, b+c+a]
```

Now suppose you want to evaluate v at a = 10. You can do this using the subs command:

```
subs(v,a,10)
```

This replaces all occurrences of a in v with 10.

```
[ -(10+S-b)/(10-c), -(10-S-b)/(10-c),                1]
[  -(b-c-S)/(10-c),  -(b-c+S)/(10-c),                1]
[                1,                1,                1]
```

Notice, however, that the symbolic expression that S represents is unaffected by this substitution. That is, the symbol a in S is not replaced by 10. The subs command is also a useful function for substituting in a variety of values for several variables in a particular expression. For example, suppose that in addition to substituting a = 10 in S, you also want to substitute the values for 2 and 10 for b and c, respectively. The way to do this is to set values for a, b, and c in the workspace. Then subs evaluates its input using the existing symbolic and double variables in the current workspace. In the example, you first set

```
a = 10; b = 2; c = 10;
subs(S)
ans =
8
```

To look at the contents of the workspace, type whos, which gives

```
  Name      Size         Bytes  Class

  A         3x3            878  sym object
  E         3x3              888  sym object
  S         1x1              186  sym object
  a         1x1                8  double array
  ans       1x1              140  sym object
  b         1x1                8  double array
  c         1x1                8  double array
  v         3x3              982  sym object
```

a, b, and c are now variables of class double while A, E, S, and v remain symbolic expressions (class sym).

If you want to preserve a, b, and c as symbolic variables, but still alter their value within S, use this procedure.

```
syms a b c
subs(S,{a,b,c},{10,2,10})

ans =
8
```

Typing whos reveals that a, b, and c remain 1-by-1 sym objects.

The subs command can be combined with double to evaluate a symbolic expression numerically. Suppose you have the following expressions

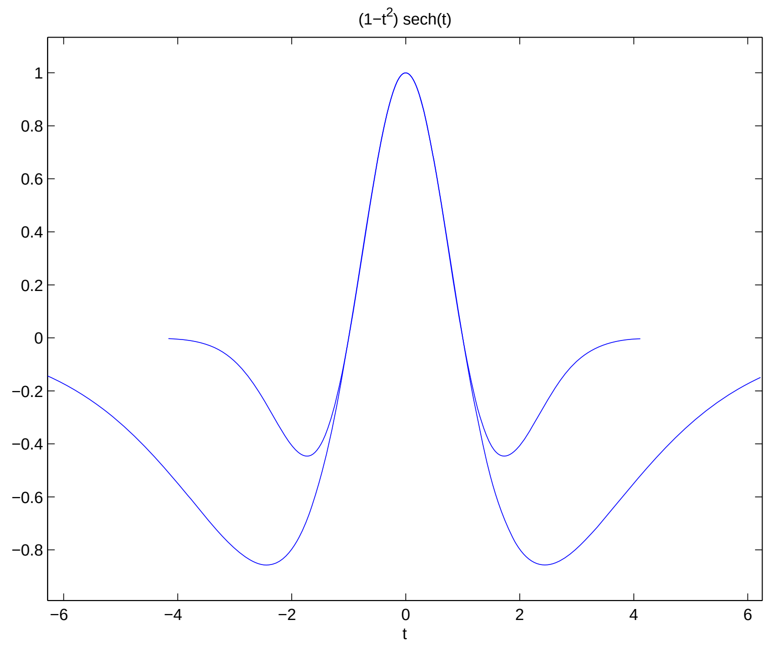
```
syms t
M = (1-t^2)*exp(-1/2*t^2);
P = (1-t^2)*sech(t);
```

and want to see how M and P differ graphically.

One approach is to type

```
ezplot(M); hold on; ezplot(P); hold off;
```

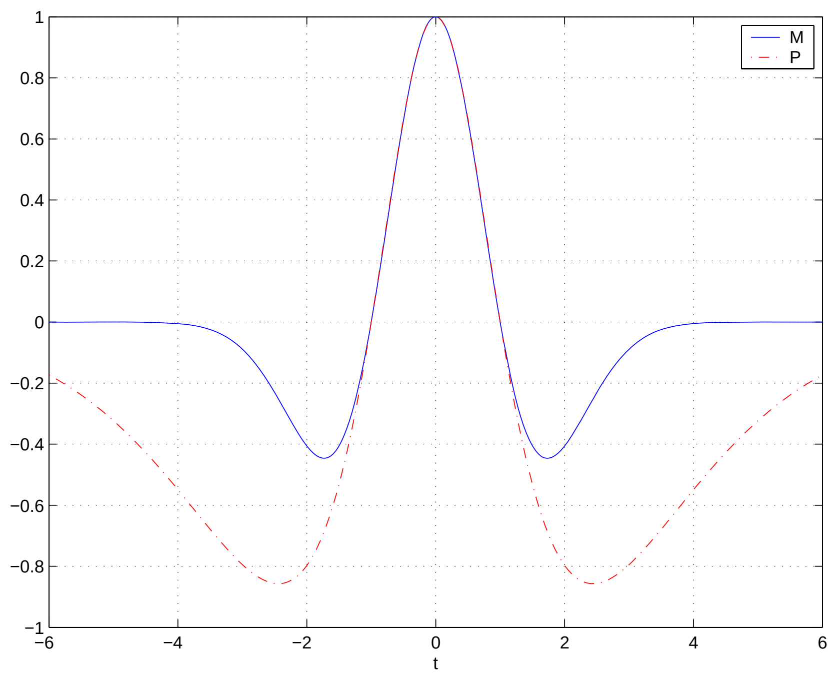
but this plot does not readily help you identify the curves.



Instead, combine `subs`, `double`, and `plot`

```
T = -6:0.05:6;
MT = double(subs(M,t,T));
PT = double(subs(P,t,T));
plot(T,MT,'b',T,PT,'r-.')
title(' ')
legend('M','P')
xlabel('t'); grid
```

to produce a multicolored graph that indicates the difference between `M` and `P`.



Finally the use of `subs` with strings greatly facilitates the solution of problems involving the Fourier, Laplace, or z-transforms.

Variable-Precision Arithmetic

In this section...

Overview

There are three different kinds of arithmetic operations in this toolbox:

Numeric	MATLAB floating-point arithmetic
Rational	Maple's exact symbolic arithmetic
VPA	Maple's variable-precision arithmetic

For example, the MATLAB statements

```
format long
1/2+1/3
```

use numeric computation to produce

```
0.83333333333333
```

With Symbolic Math Toolbox, the statement

```
sym(1/2)+1/3
```

uses symbolic computation to yield

```
5/6
```

And, also with the toolbox, the statements

```
digits(25)
vpa('1/2+1/3')
```

use variable-precision arithmetic to return

```
.8333333333333333333333333
```

The floating-point operations used by numeric arithmetic are the fastest of the three, and require the least computer memory, but the results are not exact. The number of digits in the printed output of MATLAB double quantities is controlled by the `format` statement, but the internal representation is always the eight-byte floating-point representation provided by the particular computer hardware.

In the computation of the numeric result above, there are actually three roundoff errors, one in the division of 1 by 3, one in the addition of 1/2 to the result of the division, and one in the binary to decimal conversion for the printed output. On computers that use IEEE floating-point standard arithmetic, the resulting internal value is the binary expansion of 5/6, truncated to 53 bits. This is approximately 16 decimal digits. But, in this particular case, the printed output shows only 15 digits.

The symbolic operations used by rational arithmetic are potentially the most expensive of the three, in terms of both computer time and memory. The results are exact, as long as enough time and memory are available to complete the computations.

Variable-precision arithmetic falls in between the other two in terms of both cost and accuracy. A global parameter, set by the function `digits`, controls the number of significant decimal digits. Increasing the number of digits increases the accuracy, but also increases both the time and memory requirements. The default value of `digits` is 32, corresponding roughly to floating-point accuracy.

The Maple documentation uses the term “hardware floating-point” for what you are calling “numeric” or “floating-point” and uses the term “floating-point arithmetic” for what you are calling “variable-precision arithmetic.”

Example: Using the Different Kinds of Arithmetic

Rational Arithmetic

By default, Symbolic Math Toolbox uses rational arithmetic operations, i.e., Maple’s exact symbolic arithmetic. Rational arithmetic is invoked when you create symbolic variables using the `sym` function.

The sym function converts a double matrix to its symbolic form. For example, if the double matrix is

```
A =
1.1000    1.2000    1.3000
2.1000    2.2000    2.3000
3.1000    3.2000    3.3000
```

its symbolic form, S = sym(A), is

```
S =
[11/10,  6/5, 13/10]
[21/10, 11/5, 23/10]
[31/10, 16/5, 33/10]
```

For this matrix A, it is possible to discover that the elements are the ratios of small integers, so the symbolic representation is formed from those integers. On the other hand, the statement

```
E = [exp(1) (1+sqrt(5))/2; log(3) rand]
```

returns a matrix

```
E =
   2.71828182845905   1.61803398874989
   1.09861228866811   0.76209683302739
```

whose elements are not the ratios of small integers, so sym(E) reproduces the floating-point representation in a symbolic form:

```
ans =
[ 6121026514868074*2^(-51), 7286977268806824*2^(-52)]
[ 4947709893870346*2^(-52), 6864358026484820*2^(-53)]
```

Variable-Precision Numbers

Variable-precision numbers are distinguished from the exact rational representation by the presence of a decimal point. A power of 10 scale factor, denoted by 'e', is allowed. To use variable-precision instead of rational arithmetic, create your variables using the vpa function.

For matrices with purely double entries, the vpa function generates the representation that is used with variable-precision arithmetic. For example, if you apply vpa to the matrix S defined in the preceding section, with digits(4), by entering

```
vpa(S)
```

MATLAB returns the output

```
S =
[1.100, 1.200, 1.300]
[2.100, 2.200, 2.300]
[3.100, 3.200, 3.300]
```

Applying vpa to the matrix E defined in the preceding section, with digits(25), by entering

```
digits(25)
F = vpa(E)
```

returns

```
F =
[2.718281828459045534884808, 1.414213562373094923430017]
[1.098612288668110004152823, .2189591863280899719512718]
```

Converting to Floating-Point

To convert a rational or variable-precision number to its MATLAB floating-point representation, use the double function.

In the example, both double(sym(E)) and double(vpa(E)) return E.

Another Example Using Different Kinds of Arithmetic

The next example is perhaps more interesting. Start with the symbolic expression

```
f = sym('exp(pi*sqrt(163))')
```

The statement

```
double(f)
```

produces the printed floating-point value

```
ans =
    2.625374126407687e+017
```

Using the second argument of `vpa` to specify the number of digits,

```
vpa(f,18)
```

returns

```
262537412640768744.
```

whereas

```
vpa(f,25)
```

returns

```
262537412640768744.0000000
```

you suspect that `f` might actually have an integer value. This suspicion is reinforced by the 30 digit value, `vpa(f,30)`

```
262537412640768743.999999999999
```

Finally, the 40–digit value, `vpa(f,40)`

```
262537412640768743.9999999999992500725944
```

shows that `f` is very close to, but not exactly equal to, an integer.

Linear Algebra

In this section...

Basic Algebraic Operations

Basic algebraic operations on symbolic objects are the same as operations on MATLAB objects of class `double`. This is illustrated in the following example.

The Givens transformation produces a plane rotation through the angle `t`. The statements

```
syms t;
G = [cos(t) sin(t); -sin(t) cos(t)]
```

create this transformation matrix.

```
G =
[  cos(t),  sin(t) ]
[ -sin(t),  cos(t) ]
```

Applying the Givens transformation twice should simply be a rotation through twice the angle. The corresponding matrix can be computed by multiplying `G` by itself or by raising `G` to the second power. Both

```
A = G*G
```

and

```
A = G^2
```

produce

```
A =
[cos(t)^2-sin(t)^2,   2*cos(t)*sin(t)]
[ -2*cos(t)*sin(t), cos(t)^2-sin(t)^2]
```

The `simple` function

```
A = simple(A)
```

uses a trigonometric identity to return the expected form by trying several different identities and picking the one that produces the shortest representation.

```
A =
[ cos(2*t), sin(2*t)]
[-sin(2*t), cos(2*t)]
```

The Givens rotation is an orthogonal matrix, so its transpose is its inverse. Confirming this by

```
I = G.' *G
```

which produces

```
I =
[cos(t)^2+sin(t)^2,                 0]
[                0, cos(t)^2+sin(t)^2]
```

and then

```
I = simple(I)
I =
[1, 0]
[0, 1]
```

Linear Algebraic Operations

The following examples show how to do several basic linear algebraic operations using Symbolic Math Toolbox.

The command

```
H = hilb(3)
```

generates the 3-by-3 Hilbert matrix. With `format short`, MATLAB prints

```
H =
1.0000    0.5000    0.3333
0.5000    0.3333    0.2500
0.3333    0.2500    0.2000
```

The computed elements of `H` are floating-point numbers that are the ratios of small integers. Indeed, `H` is a MATLAB array of class `double`. Converting `H` to a symbolic matrix

```
H = sym(H)
```

gives

```
[  1, 1/2, 1/3]
[1/2, 1/3, 1/4]
[1/3, 1/4, 1/5]
```

This allows subsequent symbolic operations on `H` to produce results that correspond to the infinitely precise Hilbert matrix, `sym(hilb(3))`, not its floating-point approximation, `hilb(3)`. Therefore,

```
inv(H)
```

produces

```
[  9,  -36,   30]
[-36,  192, -180]
[ 30, -180,  180]
```

and

```
det(H)
```

yields

```
1/2160
```

You can use the backslash operator to solve a system of simultaneous linear equations. For example, the commands

```
b = [1 1 1]'
```

```
x = H\b    % Solve Hx = b
```

produce the solution

```
[  3]
[-24]
[ 30]
```

All three of these results, the inverse, the determinant, and the solution to the linear system, are the exact results corresponding to the infinitely precise, rational, Hilbert matrix. On the other hand, using `digits(16)`, the command

```
V = vpa(hilb(3))
```

returns

```
[                1., .5000000000000000, .3333333333333333]
[.5000000000000000, .3333333333333333, .2500000000000000]
[.3333333333333333, .2500000000000000, .2000000000000000]
```

The decimal points in the representation of the individual elements are the signal to use variable-precision arithmetic. The result of each arithmetic operation is rounded to 16 significant decimal digits. When inverting the matrix, these errors are magnified by the matrix condition number, which for `hilb(3)` is about 500. Consequently,

```
inv(V)
```

which returns

```
ans =
[  9.000000000000179, -36.00000000000080,  30.00000000000067]
[ -36.00000000000080,  192.0000000000042, -180.0000000000040]
[  30.00000000000067, -180.0000000000040,  180.0000000000038]
```

shows the loss of two digits. So does

```
det(V)
```

which gives

```
.462962962962953e-3
```

and

```
V\b
```

which is

```
[ 3.000000000000041]
[-24.00000000000021]
[ 30.00000000000019]
```

Since `H` is nonsingular, calculating the null space of `H` with the command

```
null(H)
```

returns an empty matrix, and calculating the column space of `H` with

```
colspace(H)
```

returns a permutation of the identity matrix. A more interesting example, which the following code shows, is to find a value `s` for `H(1,1)` that makes `H` singular. The commands

```
syms s
H(1,1) = s
Z = det(H)
sol = solve(Z)
```

produce

```
H =
[   s, 1/2, 1/3]
[1/2, 1/3, 1/4]
[1/3, 1/4, 1/5]

Z =
1/240*s-1/270
sol =
8/9
```

Then

```
H = subs(H,s,sol)
```

substitutes the computed value of sol for s in H to give

```
H =
[8/9, 1/2, 1/3]
[1/2, 1/3, 1/4]
[1/3, 1/4, 1/5]
```

Now, the command

```
det(H)
```

returns

```
ans =
0
```

and

```
inv(H)
```

produces an error message

```
??? error using ==> inv
Error, (in inverse) singular matrix
```

because H is singular. For this matrix, Z = null(H) and C = colspace(H) are nontrivial:

```
Z =
[   1]
[  -4]
[10/3]

C =
[     1,     0]
[     0,     1]
[ -3/10,   6/5]
```

It should be pointed out that even though H is singular, vpa(H) is not. For any integer value d, setting

```
digits(d)
```

and then computing

```
det(vpa(H))
inv(vpa(H))
```

results in a determinant of size `10^(-d)` and an inverse with elements on the order of `10^d`.

Eigenvalues

The symbolic eigenvalues of a square matrix `A` or the symbolic eigenvalues and eigenvectors of `A` are computed, respectively, using the commands

```
E = eig(A)
[V,E] = eig(A)
```

The variable-precision counterparts are

```
E = eig(vpa(A))
[V,E] = eig(vpa(A))
```

The eigenvalues of `A` are the zeros of the characteristic polynomial of `A`, `det(A-x*I)`, which is computed by

```
poly(A)
```

The matrix `H` from the last section provides the first example:

```
H =
[8/9, 1/2, 1/3]
[1/2, 1/3, 1/4]
[1/3, 1/4, 1/5]
```

The matrix is singular, so one of its eigenvalues must be zero. The statement

```
[T,E] = eig(H)
```

produces the matrices `T` and `E`. The columns of `T` are the eigenvectors of `H`:

```
T =

[    1, 28/153+2/153*12589^(1/2),  28/153-2/153*12589^(12)]
[    -4,                        1,                        1]
[  10/3, 92/255-1/255*12589^(1/2), 292/255+1/255*12589^(12)]
```

Similarly, the diagonal elements of `E` are the eigenvalues of `H`:

```
E =

[0,                       0,                       0]
[0, 32/45+1/180*12589^(1/2),                       0]
[0,                       0, 32/45-1/180*12589^(1/2)]
```

It may be easier to understand the structure of the matrices of eigenvectors, `T`, and eigenvalues, `E`, if you convert `T` and `E` to decimal notation. To do so, proceed as follows. The commands

```
Td = double(T)
Ed = double(E)
```

return

```
Td =
    1.0000    1.6497   -1.2837
   -4.0000    1.0000    1.0000
    3.3333    0.7051    1.5851
Ed =
    0         0         0
    0    1.3344         0
    0         0    0.0878
```

The first eigenvalue is zero. The corresponding eigenvector (the first column of `Td`) is the same as the basis for the null space found in the last section. The other two eigenvalues are the result of applying the quadratic formula to

```
x^2-64/45*x+253/2160
```

which is the quadratic factor in `factor(poly(H))`.

```
syms x
g = simple(factor(poly(H))/x);
solve(g)

ans =
[ 32/45+1/180*12589^(1/2)]
[ 32/45-1/180*12589^(1/2)]
```

Closed form symbolic expressions for the eigenvalues are possible only when the characteristic polynomial can be expressed as a product of rational polynomials of degree four or less. The Rosser matrix is a classic numerical analysis test matrix that illustrates this requirement. The statement

```
R = sym(gallery('rosser'))
```

generates

```
 R =
[ 611   196  -192   407    -8   -52   -49    29]
[ 196   899   113  -192   -71   -43    -8   -44]
[-192   113   899   196    61    49     8    52]
[ 407  -192   196   611     8    44    59   -23]
[  -8   -71    61     8   411  -599   208   208]
[ -52   -43    49    44  -599   411   208   208]
[ -49    -8     8    59   208   208    99  -911]
[  29   -44    52   -23   208   208  -911    99]
```

The commands

```
p = poly(R);
pretty(factor(p))
```

produce

```
                2                  2                    2
 x (x - 1020) (x  - 1020 x + 100)(x  - 1040500) (x - 1000)
```

The characteristic polynomial (of degree 8) factors nicely into the product of two linear terms and three quadratic terms. You can see immediately that four of the eigenvalues are 0, 1020, and a double root at 1000. The other four roots are obtained from the remaining quadratics. Use

```
eig(R)
```

to find all these values

```
ans =
[                0]
[             1020]
[    10*10405^(1/2)]
[   -10*10405^(1/2)]
[ 510+100*26^(1/2)]
[ 510-100*26^(1/2)]
[             1000]
[             1000]
```

The Rosser matrix is not a typical example; it is rare for a full 8-by-8 matrix to have a characteristic polynomial that factors into such simple form. If you change the two "corner" elements of `R` from 29 to 30 with the commands

```
S = R;  S(1,8) = 30;  S(8,1) = 30;
```

and then try

```
p = poly(S)
```

you find

```
p =
 x^8-4040*x^7+5079941*x^6+82706090*x^5-5327831918568*x^4+
4287832912719760*x^3-1082699388411166000*x^2+51264008540948000*
x+40250968213600000
```

You also find that `factor(p)` is `p` itself. That is, the characteristic polynomial cannot be factored over the rationals.

For this modified Rosser matrix

```
F = eig(S)
```

returns

```
F =
[   .21803980548301606860857564424981]
[  999.94691786044276755320289228602]
[  1000.1206982933841335712817075454]
[  1019.5243552632016358324933278291]
[  1019.9935501291629257348091808173]
[  1020.4201882015047278185457498840]
[ -.17053529728768998575200874607757]
[ -1020.0532142558915165931894252600]
```

Notice that these values are close to the eigenvalues of the original Rosser matrix. Further, the numerical values of F are a result of Maple's floating-point arithmetic. Consequently, different settings of digits do not alter the number of digits to the right of the decimal place.

It is also possible to try to compute eigenvalues of symbolic matrices, but closed form solutions are rare. The Givens transformation is generated as the matrix exponential of the elementary matrix

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Symbolic Math Toolbox commands

```
syms t
A = sym([0 1; -1 0]);
G = expm(t*A)
```

return

```
[  cos(t),  sin(t)]
[ -sin(t),  cos(t)]
```

Next, the command

```
g = eig(G)
```

produces

```
g =
[ cos(t)+(cos(t)^2-1)^(1/2)]
[ cos(t)-(cos(t)^2-1)^(1/2)]
```

you can use simple to simplify this form of g. Indeed, a repeated application of simple

```
for  j = 1:4
   [g,how] = simple(g)
```

```
end
```

produces the best result:

```
g =
[ cos(t)+(-sin(t)^2)^(1/2)]
[ cos(t)-(-sin(t)^2)^(1/2)]

how =
mwcos2sin

g =
[ cos(t)+i*sin(t)]
[ cos(t)-i*sin(t)]

how =
radsimp

g =
[   exp(i*t)]
[ 1/exp(i*t)]

how =
convert(exp)

g =
[  exp(i*t)]
[ exp(-i*t)]

how =
simplify
```

Notice the first application of `simple` uses `mwcos2sin` to produce a sum of sines and cosines. Next, `simple` invokes `radsimp` to produce `cos(t) + i*sin(t)` for the first eigenvector. The third application of `simple` uses `convert(exp)` to change the sines and cosines to complex exponentials. The last application of `simple` uses `simplify` to obtain the final form.

Jordan Canonical Form

The Jordan canonical form results from attempts to diagonalize a matrix by a similarity transformation. For a given matrix `A`, find a nonsingular matrix `V`, so that `inv(V)*A*V`, or, more succinctly, `J = V\A*V`, is "as close to diagonal as possible." For almost all matrices, the Jordan canonical form is the diagonal matrix of eigenvalues and the columns of the transformation matrix are the eigenvectors. This always happens if the matrix is symmetric or if it has distinct eigenvalues. Some nonsymmetric matrices with multiple eigenvalues cannot be diagonalized. The Jordan form has the eigenvalues on its diagonal, but some of the superdiagonal elements are one, instead of zero. The statement

```
J = jordan(A)
```

computes the Jordan canonical form of `A`. The statement

```
[V,J] = jordan(A)
```

also computes the similarity transformation. The columns of `V` are the generalized eigenvectors of `A`.

The Jordan form is extremely sensitive to perturbations. Almost any change in `A` causes its Jordan form to be diagonal. This makes it very difficult to compute the Jordan form reliably with floating-point arithmetic. It also implies that `A` must be known exactly (i.e., without round-off error, etc.). Its elements must be integers, or ratios of small integers. In particular, the variable-precision calculation, `jordan(vpa(A))`, is not allowed.

For example, let

```
A = sym([12,32,66,116;-25,-76,-164,-294;
         21,66,143,256;-6,-19,-41,-73])
A =
[   12,   32,   66,  116]
[  -25,  -76, -164, -294]
[   21,   66,  143,  256]
[   -6,  -19,  -41,  -73]
```

Then

```
[V,J] = jordan(A)
```

produces

```
V =
[   4,  -2,   4,   3]
[  -6,   8, -11,  -8]
[   4,  -7,  10,   7]
[  -1,   2,  -3,  -2]

J =
[ 1, 1, 0, 0]
[ 0, 1, 0, 0]
[ 0, 0, 2, 1]
[ 0, 0, 0, 2]
```

Therefore `A` has a double eigenvalue at 1, with a single Jordan block, and a double eigenvalue at 2, also with a single Jordan block. The matrix has only two eigenvectors, `V(:,1)` and `V(:,3)`. They satisfy

```
A*V(:,1) = 1*V(:,1)
A*V(:,3) = 2*V(:,3)
```

The other two columns of `V` are generalized eigenvectors of grade 2. They satisfy

```
A*V(:,2) = 1*V(:,2) + V(:,1)
A*V(:,4) = 2*V(:,4) + V(:,3)
```

In mathematical notation, with v_j = `v(:,j)`, the columns of `V` and eigenvalues satisfy the relationships

$$(A - \lambda_1 I)v_2 = v_1$$

$$(A - \lambda_2 I)v_4 = v_3$$

Singular Value Decomposition

Only the variable-precision numeric computation of the complete singular vector decomposition is available in the toolbox. One reason for this is that the formulas that result from symbolic computation are usually too long and complicated to be of much use. If A is a symbolic matrix of floating-point or variable-precision numbers, then

```
S = svd(A)
```

computes the singular values of A to an accuracy determined by the current setting of `digits`. And

```
[U,S,V] = svd(A);
```

produces two orthogonal matrices, U and V, and a diagonal matrix, S, so that

```
A = U*S*V';
```

Consider the n-by-n matrix A with elements defined by

```
A(i,j) = 1/(i-j+1/2)
```

For `n = 5`, the matrix is

```
[  2    -2   -2/3   -2/5   -2/7]
[2/3     2     -2   -2/3   -2/5]
[2/5   2/3      2     -2   -2/3]
[2/7   2/5    2/3      2     -2]
[2/9   2/7    2/5    2/3      2]
```

It turns out many of the singular values of these matrices are close to π.

The most obvious way of generating this matrix is

```
for i=1:n
    for j=1:n
      A(i,j) = sym(1/(i-j+1/2));
   end
end
```

The most efficient way to generate the matrix is

```
[J,I] = meshgrid(1:n);
A = sym(1./(I - J+1/2));
```

Since the elements of A are the ratios of small integers, vpa(A) produces a variable-precision representation, which is accurate to digits precision. Hence

```
S = svd(vpa(A))
```

computes the desired singular values to full accuracy. With n = 16 and digits(30), the result is

```
S =
[ 1.20968137605668985332455685357 ]
[ 2.69162158686066606774782763594 ]
[ 3.07790297231119748658424727354 ]
[ 3.13504054399744654843898901261 ]
[ 3.14106044663470063805218371924 ]
[ 3.14155754359918083691050658260 ]
[ 3.14159075458605848728982577119 ]
[ 3.14159256925492306470284863102 ]
[ 3.14159265052654880815569479613 ]
[ 3.14159265349961053143856838564 ]
[ 3.14159265358767361712392612384 ]
[ 3.14159265358975439206849907220 ]
[ 3.14159265358979270342635559051 ]
[ 3.14159265358979323325290142781 ]
[ 3.14159265358979323843066846712 ]
[ 3.14159265358979323846255035974 ]
```

There are two ways to compare S with pi, the floating-point representation of π. In the vector below, the first element is computed by subtraction with variable-precision arithmetic and then converted to a double. The second element is computed with floating-point arithmetic:

```
format short e
[double(pi*ones(16,1)-S)  pi-double(S)]
```

The results are

```
    1.9319e+00   1.9319e+00
    4.4997e-01   4.4997e-01
    6.3690e-02   6.3690e-02
    6.5521e-03   6.5521e-03
```

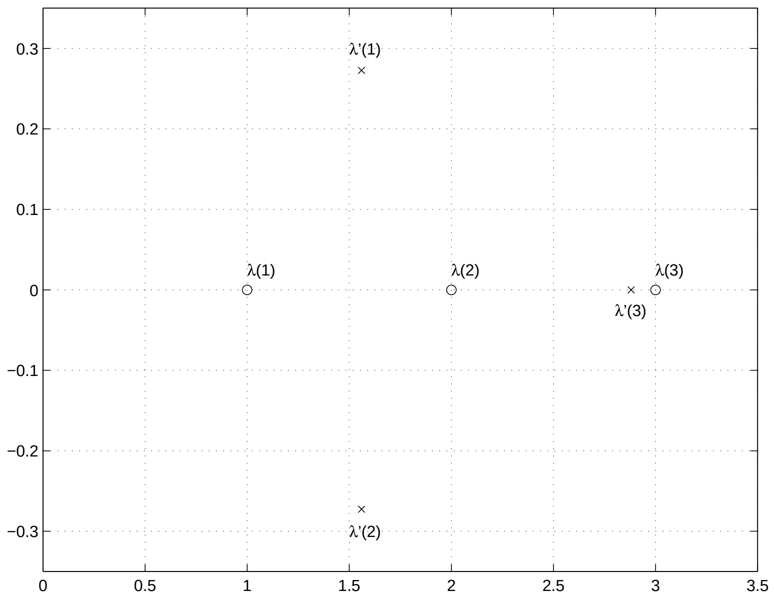
```
5.3221e-04   5.3221e-04
3.5110e-05   3.5110e-05
1.8990e-06   1.8990e-06
8.4335e-08   8.4335e-08
3.0632e-09   3.0632e-09
9.0183e-11   9.0183e-11
2.1196e-12   2.1196e-12
3.8846e-14   3.8636e-14
5.3504e-16   4.4409e-16
5.2097e-18            0
3.1975e-20            0
9.3024e-23            0
```

Since the relative accuracy of `pi` is `pi*eps`, which is `6.9757e-16`, either column confirms the suspicion that four of the singular values of the `16`-by-`16` example equal π to floating-point accuracy.

Eigenvalue Trajectories

This example applies several numeric, symbolic, and graphic techniques to study the behavior of matrix eigenvalues as a parameter in the matrix is varied. This particular setting involves numerical analysis and perturbation theory, but the techniques illustrated are more widely applicable.

In this example, you consider a 3-by-3 matrix A whose eigenvalues are 1, 2, 3. First, you perturb A by another matrix E and parameter $t : A \to A + tE$. As t increases from 0 to 10^{-6}, the eigenvalues $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$ change to $\lambda_1' = 1.5596 + 0.2726i$, $\lambda_2' = 1.5596 - 0.2726i$, $\lambda_3' = 2.8808$.



This, in turn, means that for some value of $t = \tau, 0 < \tau < 10^{-6}$, the perturbed matrix $A(t) = A + tE$ has a double eigenvalue $\lambda_1 = \lambda_2$. The example shows how to find the value of `t`, called τ, where this happens.

The starting point is a MATLAB test example, known as `gallery(3)`.

```
A = gallery(3)
A =
   -149        -50       -154
    537        180        546
    -27         -9        -25
```

This is an example of a matrix whose eigenvalues are sensitive to the effects of roundoff errors introduced during their computation. The actual computed eigenvalues may vary from one machine to another, but on a typical workstation, the statements

```
format long
e = eig(A)
```

produce

```
e =
   1.00000000001122
   1.99999999999162
   2.99999999999700
```

Of course, the example was created so that its eigenvalues are actually 1, 2, and 3. Note that three or four digits have been lost to roundoff. This can be easily verified with the toolbox. The statements

```
B = sym(A);
e = eig(B)'
p = poly(B)
f = factor(p)
```

produce

```
e =
[1,  2,  3]

p =
x^3-6*x^2+11*x-6

f =
(x-1)*(x-2)*(x-3)
```

Are the eigenvalues sensitive to the perturbations caused by roundoff error because they are “close together”? Ordinarily, the values 1, 2, and 3 would be regarded as “well separated.” But, in this case, the separation should be viewed on the scale of the original matrix. If `A` were replaced by `A/1000`, the eigenvalues, which would be .001, .002, .003, would “seem” to be closer together.

But eigenvalue sensitivity is more subtle than just “closeness.” With a carefully chosen perturbation of the matrix, it is possible to make two of its eigenvalues coalesce into an actual double root that is extremely sensitive to roundoff and other errors.

One good perturbation direction can be obtained from the outer product of the left and right eigenvectors associated with the most sensitive eigenvalue. The following statement creates

```
E = [130,-390,0;43,-129,0;133,-399,0]
```

the perturbation matrix

```
E =
130   -390     0
 43   -129     0
133   -399     0
```

The perturbation can now be expressed in terms of a single, scalar parameter `t`. The statements

```
syms x t
A = A+t*E
```

replace `A` with the symbolic representation of its perturbation:

```
A =
[-149+130*t, -50-390*t, -154]
[  537+43*t, 180-129*t,  546]
[ -27+133*t,  -9-399*t,  -25]
```

Computing the characteristic polynomial of this new `A`

```
p = poly(A)
```

gives

```
p =
x^3-6*x^2+11*x-t*x^2+492512*t*x-6-1221271*t
```

Prettyprinting

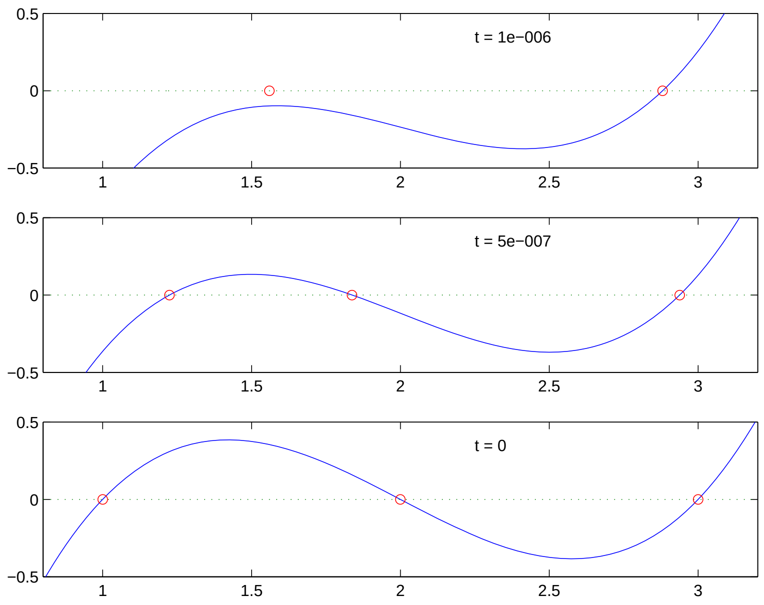
```
pretty(collect(p,x))
```

shows more clearly that `p` is a cubic in `x` whose coefficients vary linearly with `t`.

```
   3              2
  x  + (- t - 6) x  + (492512 t + 11) x - 6 - 1221271 t
```

It turns out that when `t` is varied over a very small interval, from 0 to 1.0e-6, the desired double root appears. This can best be seen graphically. The first figure shows plots of `p`, considered as a function of `x`, for three different values of `t: t` = 0, `t` = 0.5e-6, and `t` = 1.0e-6. For each value, the eigenvalues are computed numerically and also plotted:

```
x = .8:.01:3.2;
for k = 0:2
  c = sym2poly(subs(p,t,k*0.5e-6));
  y = polyval(c,x);
  lambda = eig(double(subs(A,t,k*0.5e-6)));
  subplot(3,1,3-k)
  plot(x,y,'-',x,0*x,':',lambda,0*lambda,'o')
  axis([.8 3.2 -.5 .5])
  text(2.25,.35,['t = ' num2str( k*0.5e-6 )]);
end
```



The bottom subplot shows the unperturbed polynomial, with its three roots at 1, 2, and 3. The middle subplot shows the first two roots approaching each other. In the top subplot, these two roots have become complex and only one real root remains.

The next statements compute and display the actual eigenvalues

```
e = eig(A);
pretty(e)
```

showing that `e(2)` and `e(3)` form a complex conjugate pair:

```
[                                      2             ]
[      1/3     492508/3 t - 1/3 - 1/9 t              ]
[1/3 %1    - 3 ------------------------ + 1/3 t + 2]
[                            1/3                     ]
[                          %1                        ]

[                                        2
[        1/3       492508/3 t - 1/3 - 1/9 t
[- 1/6 %1    + 3/2 ------------------------ + 1/3 t + 2
[                              1/3
[                            %1

              /                                    2\]
          1/2 |      1/3     492508/3 t - 1/3 - 1/9 t |]
 + 1/2 I 3    |1/3 %1    + 3 ------------------------|]
              |                            1/3        |]
              \                          %1           /]

[                                        2
[        1/3       492508/3 t - 1/3 - 1/9 t
[- 1/6 %1    + 3/2 ------------------------ + 1/3 t + 2
[                              1/3
[                            %1

              /                                    2\]
          1/2 |      1/3     492508/3 t - 1/3 - 1/9 t |]
 - 1/2 I 3    |1/3 %1    + 3 ------------------------|]
              |                            1/3        |]
              \                          %1           /]
```

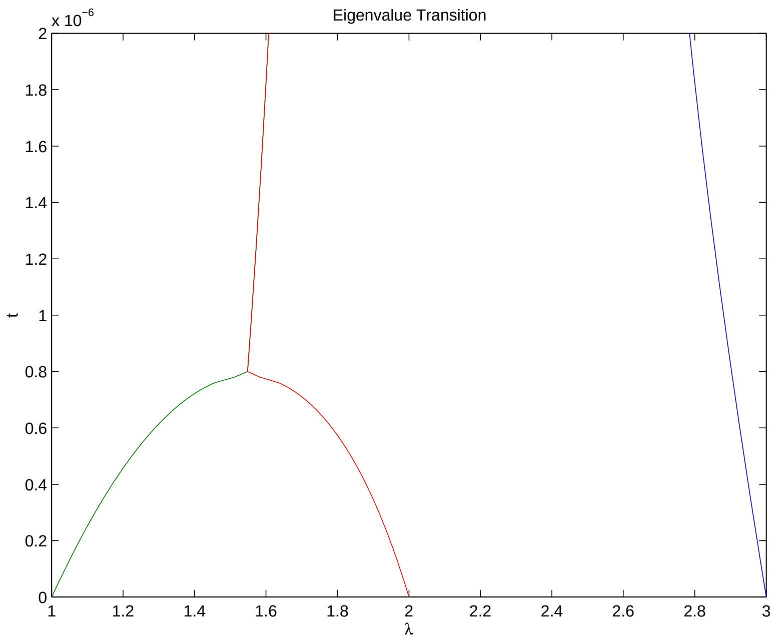
```
                          2                 3                           3
%1 := -2216286 t  + 3189393 t + t  + 3 (358392752910068940 t

                                  2                  4                 1/2
        - 1052829647418 t  - 181922388795 t  + 4432572 t - 3)
```

Next, the symbolic representations of the three eigenvalues are evaluated at many values of `t`

```
tvals = (2:-.02:0)' * 1.e-6;
r = size(tvals,1);
c = size(e,1);
lambda = zeros(r,c);
for k = 1:c
   lambda(:,k) = double(subs(e(k),t,tvals));
end
plot(lambda,tvals)
xlabel('\lambda'); ylabel('t');
title('Eigenvalue Transition')
```

to produce a plot of their trajectories.



Above `t = 0.8e`$^{-6}$, the graphs of two of the eigenvalues intersect, while below `t` = 0.8e^{-6}, two real roots become a complex conjugate pair. What is the precise value of `t` that marks this transition? Let τ denote this value of `t`.

One way to find the *exact* value of τ involves polynomial discriminants. The discriminant of a quadratic polynomial is the familiar quantity under the square root sign in the quadratic formula. When it is negative, the two roots are complex.

There is no `discrim` function in the toolbox, but there is one in Maple and it can be accessed through the toolbox's `maple` function. The statement

```
mhelp discrim
```

provides a brief explanation. Use these commands

```
syms a b c x
maple('discrim', a*x^2+b*x+c, x)
```

to show the generic quadratic's discriminant, $b^2 - 4ac$:

```
ans =
-4*a*c+b^2
```

The discriminant for the perturbed cubic characteristic polynomial is obtained, using

```
discrim = maple('discrim',p,x)
```

which produces

```
[discrim =
4-5910096*t+1403772863224*t^2-477857003880091920*t^3+2425631850
60*t^4]
```

The quantity τ is one of the four roots of this quartic. You can find a numeric value for τ with the following code.

```
digits(24)
s = solve(discrim);
tau = vpa(s)

tau =
[                        1970031.04061804553618914]
[                                 .783792490598e-6]
[ .1076924816047e-5+.259079734916440764385242e-5*i]
[ .1076924816047e-5-.259079734916440764385242e-5*i]
```

Of the four solutions, you know that

```
tau = tau(2)
```

is the transition point

```
tau =
.783792490602e-6
```

because it is closest to the previous estimate.

A more generally applicable method for finding τ is based on the fact that, at a double root, both the function and its derivative must vanish. This results in two polynomial equations to be solved for two unknowns. The statement

```
sol = solve(p,diff(p,'x'))
```

solves the pair of algebraic equations `p = 0` and `dp/dx = 0` and produces

```
sol =
    t: [4x1 sym]
    x: [4x1 sym]
```

Find τ now by

```
format short
tau = double(sol.t(2))
```

which reveals that the second element of `sol.t` is the desired value of τ:

```
tau =
   7.8379e-07
```

Therefore, the second element of `sol.x`

```
sigma = double(sol.x(2))
```

is the double eigenvalue

```
sigma =
   1.5476
```

To verify that this value of τ does indeed produce a double eigenvalue at $\sigma = 1.5476$, substitute τ for t in the perturbed matrix $A(t) = A + tE$ and find the eigenvalues of $A(t)$. That is,

```
e = eig(double(subs(A,t,tau)))
e =
   1.5476 + 0.0000i
   1.5476 - 0.0000i
   2.9048
```

confirms that $\sigma = 1.5476$ is a double eigenvalue of $A(t)$ for t = 7.8379e-07.

Solving Equations

In this section...

Solving Algebraic Equations

If S is a symbolic expression,

```
solve(S)
```

attempts to find values of the symbolic variable in S (as determined by findsym) for which S is zero. For example,

```
syms a b c x
S = a*x^2 + b*x + c;
solve(S)
```

uses the familiar quadratic formula to produce

```
ans =
[1/2/a*(-b+(b^2-4*a*c)^(1/2))]
[1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

This is a symbolic vector whose elements are the two solutions.

If you want to solve for a specific variable, you must specify that variable as an additional argument. For example, if you want to solve S for b, use the command

```
b = solve(S,b)
```

which returns

```
b =
-(a*x^2+c)/x
```

Note that these examples assume equations of the form $f(x) = 0$. If you need to solve equations of the form $f(x) = q(x)$, you must use quoted strings. In particular, the command

```
s = solve('cos(2*x)+sin(x)=1')
```

returns a vector with four solutions.

```
s =
[      0]
[     pi]
[ 1/6*pi]
[ 5/6*pi]
```

Several Algebraic Equations

This section explains how to solve systems of equations using Symbolic Math Toolbox. As an example, suppose you have the system

$$x^2y^2 = 0$$

$$x - \frac{y}{2} = \alpha$$

and you want to solve for x and y. First, create the necessary symbolic objects.

```
syms x y alpha
```

There are several ways to address the output of `solve`. One is to use a two-output call

```
[x,y] = solve(x^2*y^2, x-y/2-alpha)
```

which returns

```
x =
[     0]
[     0]
[ alpha]
[ alpha]
```

```
y =
[ -2*alpha]
[ -2*alpha]
[        0]
[        0]
```

Consequently, the solution vector

```
v = [x, y]
```

appears to have redundant components. This is due to the first equation $x^2y^2 = 0$, which has two solutions in x and y: $x = \pm 0$, $y = \pm 0$. Changing the equations to

```
eqs1 = 'x^2*y^2=1, x-y/2-alpha'
[x,y] = solve(eqs1)
```

produces four distinct solutions:

```
x =
[ 1/2*alpha+1/2*(alpha^2+2)^(1/2)]
[ 1/2*alpha-1/2*(alpha^2+2)^(1/2)]
[ 1/2*alpha+1/2*(alpha^2-2)^(1/2)]
[ 1/2*alpha-1/2*(alpha^2-2)^(1/2)]


y =
[ -alpha+(alpha^2+2)^(1/2)]
[ -alpha-(alpha^2+2)^(1/2)]
[ -alpha+(alpha^2-2)^(1/2)]
[ -alpha-(alpha^2-2)^(1/2)]
```

Since you did not specify the dependent variables, `solve` uses `findsym` to determine the variables.

This way of assigning output from `solve` is quite successful for “small” systems. Plainly, if you had, say, a 10-by-10 system of equations, typing

```
[x1,x2,x3,x4,x5,x6,x7,x8,x9,x10] = solve(...)
```

is both awkward and time consuming. To circumvent this difficulty, `solve` can return a structure whose fields are the solutions. In particular, consider the system `u^2-v^2 = a^2`, `u + v = 1`, `a^2-2*a = 3`. The command

```
S = solve('u^2-v^2 = a^2','u + v = 1','a^2-2*a = 3')
```

returns

```
S =
    a: [2x1 sym]
    u: [2x1 sym]
    v: [2x1 sym]
```

The solutions for `a` reside in the "a-field" of `S`. That is,

```
S.a
```

produces

```
ans =
[  3]
[ -1]
```

Similar comments apply to the solutions for `u` and `v`. The structure `S` can now be manipulated by field and index to access a particular portion of the solution. For example, if you want to examine the second solution, you can use the following statement

```
s2 = [S.a(2), S.u(2), S.v(2)]
```

to extract the second component of each field.

```
s2 =
[ -1,  1,  0]
```

The following statement

```
M = [S.a, S.u, S.v]
```

creates the solution matrix `M`

```
M =
[  3,  5, -4]
[ -1,  1,  0]
```

whose rows comprise the distinct solutions of the system.

Linear systems of simultaneous equations can also be solved using matrix division. For example,

```
clear u v x y
syms u v x y
S = solve(x+2*y-u, 4*x+5*y-v);
sol = [S.x;S.y]
```

and

```
A = [1 2; 4 5];
b = [u; v];
z = A\b
```

result in

```
sol =

[ -5/3*u+2/3*v]
[  4/3*u-1/3*v]

z =
[ -5/3*u+2/3*v]
[  4/3*u-1/3*v]
```

Thus `s` and `z` produce the same solution, although the results are assigned to different variables.

Single Differential Equation

The function `dsolve` computes symbolic solutions to ordinary differential equations. The equations are specified by symbolic expressions containing the letter `D` to denote differentiation. The symbols `D2`, `D3`, ... `DN`, correspond to the second, third, ..., Nth derivative, respectively. Thus, `D2y` is the Symbolic Math Toolbox equivalent of d^2y/dt^2. The dependent variables are those preceded by `D` and the default independent variable is `t`. Note that names of symbolic variables should not contain `D`. The independent variable can be changed from t to some other symbolic variable by including that variable as the last input argument.

Initial conditions can be specified by additional equations. If initial conditions are not specified, the solutions contain constants of integration, C1, C2, etc.

The output from dsolve parallels the output from solve. That is, you can call dsolve with the number of output variables equal to the number of dependent variables or place the output in a structure whose fields contain the solutions of the differential equations.

Example 1

The following call to dsolve

```
dsolve('Dy=1+y^2')
```

uses y as the dependent variable and t as the default independent variable.

The output of this command is

```
ans =
tan(t+C1)
```

To specify an initial condition, use

```
y = dsolve('Dy=1+y^2','y(0)=1')
```

This produces

```
y =
tan(t+1/4*pi)
```

Notice that y is in the MATLAB workspace, but the independent variable t is not. Thus, the command diff(y,t) returns an error. To place t in the workspace, type syms t.

Example 2

Nonlinear equations may have multiple solutions, even when initial conditions are given:

```
x = dsolve('(Dx)^2+x^2=1','x(0)=0')
```

results in

```
x =
[ sin(t)]
[ -sin(t)]
```

Example 3

Here is a second-order differential equation with two initial conditions. The commands

```
y = dsolve('D2y=cos(2*x)-y','y(0)=1','Dy(0)=0', 'x');
simplify(y)
```

produce

```
ans =
4/3*cos(x)-2/3*cos(x)^2+1/3
```

The key issues in this example are the order of the equation and the initial conditions. To solve the ordinary differential equation

$$\frac{d^3u}{dx^3} = u$$

$$u(0) = 1, u'(0) = -1, u''(0) = \pi$$

simply type

```
u = dsolve('D3u=u','u(0)=1','Du(0)=-1','D2u(0) = pi','x')
```

Use `D3u` to represent d^3u/dx^3 and `D2u(0)` for $u''(0)$.

Several Differential Equations

The function `dsolve` can also handle several ordinary differential equations in several variables, with or without initial conditions. For example, here is a pair of linear, first-order equations.

```
S = dsolve('Df = 3*f+4*g', 'Dg = -4*f+3*g')
```

The computed solutions are returned in the structure `S`. You can determine the values of `f` and `g` by typing

```
f = S.f
f =
exp(3*t)*(C1*sin(4*t)+C2*cos(4*t))

g = S.g
g =
exp(3*t)*(C1*cos(4*t)-C2*sin(4*t))
```

If you prefer to recover `f` and `g` directly as well as include initial conditions, type

```
[f,g] = dsolve('Df=3*f+4*g, Dg =-4*f+3*g', 'f(0) = 0, g(0) = 1')
f =
exp(3*t)*sin(4*t)

g =
exp(3*t)*cos(4*t)
```

This table details some examples and the Symbolic Math Toolbox syntax. Note that the final entry in the table is the Airy differential equation whose solution is referred to as the Airy function.

Differential Equation	MATLAB Command
$\frac{dy}{dt}+4y(t)=e^{-t}$ $y(0) = 1$	`y = dsolve('Dy+4*y = exp(-t)', 'y(0) = 1')`
$\frac{d^2y}{dx^2}=xy(x)$ $y(0)=0, y(3)=\frac{1}{\pi}K_{\frac{1}{3}}(2\sqrt{3})$ (The Airy equation)	`y = dsolve('D2y = x*y','y(0) = 0', 'y(3) = besselk(1/3, 2*sqrt(3))/pi', 'x')`

The Airy function plays an important role in the mathematical modeling of the dispersion of water waves. It is a nontrivial exercise to show that the Fourier transform of the Airy function is $\exp(iw^3/3)$.

Special Mathematical Functions

Over 50 of the special functions of classical applied mathematics are available in the toolbox. These functions are accessed with the `mfun` function, which numerically evaluates a special function for the specified parameters. This allows you to evaluate functions that are not available in standard MATLAB, such as the Fresnel cosine integral. In addition, you can evaluate several MATLAB special functions in the complex plane, such as the error function.

For example, suppose you want to evaluate the hyperbolic cosine integral at the points 2+i, 0, and 4.5. First, type

```
help mfunlist
```

to see the list of functions available for `mfun`. This list provides a brief mathematical description of each function, its Maple name, and the parameters it needs. From the list, you can see that the hyperbolic cosine integral is called `Chi`, and it takes one complex argument. For additional information, you can access Maple help on the hyperbolic cosine integral using

```
mhelp Chi
```

Now type

```
z = [2+i 0 4.5];
w = mfun('Chi',z)
```

which returns

```
w =
   2.0303 + 1.7227i     NaN       13.9658
```

`mfun` returns `NaN`s where the function has a singularity. The hyperbolic cosine integral has a singularity at $z = 0$.

These special functions can be used with the `mfun` function:

- Airy Functions
- Binomial Coefficients
- Riemann Zeta Functions

- Bernoulli Numbers and Polynomials
- Euler Numbers and Polynomials
- Harmonic Function
- Exponential Integrals
- Logarithmic Integral
- Sine and Cosine Integrals
- Hyperbolic Sine and Cosine Integrals
- Shifted Sine Integral
- Fresnel Sine and Cosine Integral
- Dawson's Integral
- Error Function
- Complementary Error Function and its Iterated Integrals
- Gamma Function
- Logarithm of the Gamma Function
- Incomplete Gamma Function
- Digamma Function
- Polygamma Function
- Generalized Hypergeometric Function
- Bessel Functions
- Incomplete Elliptic Integrals
- Complete Elliptic Integrals
- Complete Elliptic Integrals with Complementary Modulus
- Beta Function
- Dilogarithm Integral
- Lambert's *W* Function
- Dirac Delta Function (distribution)
- Heaviside Function (distribution)

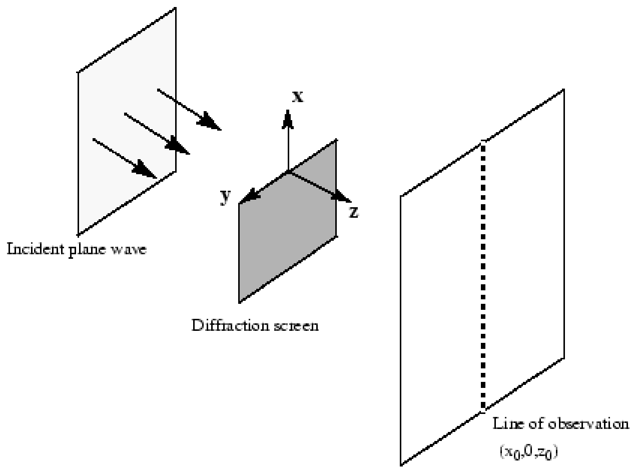
The orthogonal polynomials listed below are available with Extended Symbolic Math Toolbox:

- Gegenbauer
- Hermite
- Laguerre
- Generalized Laguerre
- Legendre
- Jacobi
- Chebyshev of the First and Second Kind

Diffraction Example

This example is from diffraction theory in classical electrodynamics. (J.D. Jackson, *Classical Electrodynamics*, John Wiley & Sons, 1962.)

Suppose you have a plane wave of intensity I_0 and wave number k. Assume that the plane wave is parallel to the xy-plane and travels along the z-axis as shown below. This plane wave is called the *incident wave*. A perfectly conducting flat diffraction screen occupies half of the xy-plane, that is $x < 0$. The plane wave strikes the diffraction screen, and you observe the diffracted wave from the line whose coordinates are $(x, 0, z_0)$, where $z_0 > 0$.



The intensity of the diffracted wave is given by

$$I = \frac{I_0}{2}\left[\left(C(\zeta)+\frac{1}{2}\right)^2 + \left(S(\zeta)+\frac{1}{2}\right)^2\right]$$

where

$$\zeta = \sqrt{\frac{k}{2z_0}} \cdot x$$

and $C(\zeta)$ and $S(\zeta)$ are the Fresnel cosine and sine integrals:

$$C(\zeta) = \int_0^{\zeta} \cos\left(\frac{\pi}{2}\cdot(t^2)\right)dt$$

$$S(\zeta) = \int_0^{\zeta} \sin\left(\frac{\pi}{2}\cdot(t^2)\right)dt$$

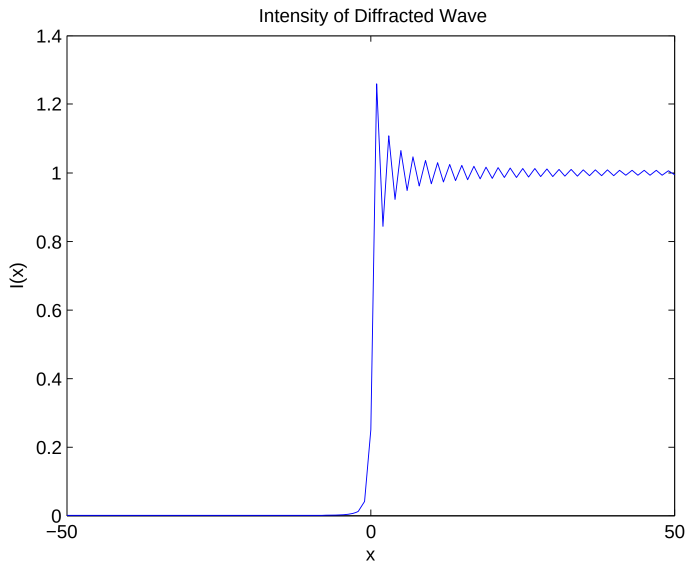
How does the intensity of the diffracted wave behave along the line of observation? Since k and z_0 are constants independent of x, you set

$$\sqrt{\frac{k}{2z_0}} = 1$$

and assume an initial intensity of $I_0 = 1$ for simplicity.

The following code generates a plot of intensity as a function of x:

```
x = -50:50;
C = mfun('FresnelC',x);
S = mfun('FresnelS',x);
I0 = 1;
T = (C+1/2).^2 + (S+1/2).^2;
I = (I0/2)*T;
plot(x,I);
xlabel('x');
ylabel('I(x)');
title('Intensity of Diffracted Wave');
```



You see from the graph that the diffraction effect is most prominent near the edge of the diffraction screen (`x = 0`), as you expect.

Note that values of x that are large and positive correspond to observation points far away from the screen. Here, you would expect the screen to have no effect on the incident wave. That is, the intensity of the diffracted wave should be the same as that of the incident wave. Similarly, x values that are large and negative correspond to observation points under the screen that are far away from the screen edge. Here, you would expect the diffracted wave to have zero intensity. These results can be verified by setting

```
x = [Inf -Inf]
```

in the code to calculate I.

Using Maple Functions

In this section...

Simple Example

The `maple` function lets you access Maple functions directly. This function takes sym objects, strings, and doubles as inputs. It returns a symbolic object, character string, or double corresponding to the class of the input. You can also use the `maple` function to debug symbolic math programs that you develop.

As an example, you can use the Maple function `gcd` to calculate the greatest common divisor of two integers or two polynomials. For example, to calculate the greatest common divisor of 14 and 21, enter

```
maple('gcd(14, 21)')
ans =
7
```

To calculate the greatest common divisor of *x*^2-*y*^2 and *x*^3-*y*^3 enter

```
maple('gcd(x^2-y^2,x^3-y^3)')
ans =
-y+x
```

To learn more about the function `gcd`, you can bring up its reference page by entering

```
doc gcd
```

As an alternative to typing the `maple` command every time you want to access `gcd`, you can write a simple M-file that does this for you. To do so, first create the M-file `gcd` in the subdirectory `toolbox/symbolic/@sym` of the directory where MATLAB is installed, and include the following commands in the M-file:

```
function g = gcd(a, b)
g = maple('gcd',a, b);
```

If you run this file

```
syms x y
z = gcd(x^2-y^2,x^3-y^3)
w = gcd(6, 24)
```

you get

```
z =
-y+x

w =
6
```

Now, extend the function so that you can take the `gcd` of two matrices in a pointwise fashion:

```
function g = gcd(a,b)

if any(size(a) ~= size(b))
  error('Inputs must have the same size.')
end

for k = 1: prod(size(a))
  g(k) = maple('gcd',a(k), b(k));
end

g = reshape(g,size(a));
```

Running this on some test data

```
A = sym([2 4 6; 3 5 6; 3 6 4]);
B = sym([40 30 8; 17 60 20; 6 3 20]);
gcd(A,B)
```

you get the result

```
ans =
[ 2, 2, 2 ]
[ 1, 5, 2 ]
[ 3, 3, 4 ]
```

Vectorized Example

Suppose you want to calculate the sine of a symbolic matrix. One way to do this is

```
function y = sin1(x)

for k = 1: prod(size(x))
   y(k) = maple('sin',x(k));
end

y = reshape(y,size(x));
```

So the statements

```
syms x y
A = [0 x; y pi/4]
sin1(A)
```

return

```
A =
[    0,    x ]
[    y, pi/4 ]

ans =
[           0,      sin(x) ]
[      sin(y), 1/2*2^(1/2) ]
```

A more efficient way to do this is to call Maple just once, using the Maple `map` function. The `map` function applies a Maple function to each element of an array. In the sine calculation example, this looks like

```
function y = sin2(x)

if  prod(size(x)) == 1
% scalar case
   y = maple('sin',x);
```

```
else
% array case
   y = maple('map','sin',x);
end
```

Note that the `sin2` function treats scalar and array cases differently. It applies the `map` function to arrays but not to scalars. This is because `map` applies a function to each operand of a scalar.

Because the `sin2` function calls Maple only once, it is considerably faster than the `sin1` function, which calls Maple `prod(size(A))` number of times.

The `map` function can also be used for Maple functions that require multiple input arguments. In this case, the syntax is

```
maple('map', Maple function, sym array, arg2, arg3, ..., argn)
```

For example, one way to call the `collect` M-file is `collect(S,x)`. In this case, the `collect` function collects all the coefficients with the same power of `x` for each element in `S`. The core section of the implementation is shown below.

```
r = maple('map','collect',sym(s),sym(x));
```

For additional information on the Maple `map` function, type

```
mhelp map
```

Debugging

The `maple` command provides two debugging facilities: trace mode and a status output argument.

Trace Mode

The command `maple traceon` causes all subsequent Maple statements and results to be printed to the screen. For example,

```
maple traceon
a = sym('a');
exp(2*a)
```

prints all calls made to the Maple kernel for calculating `exp(2*a)`:

```
statement:
   (2)*(a);
result:
   2*a
statement:
   2*a;
result:
   2*a
statement:
   exp(2*a);
result:
   exp(2*a)
statement:
   exp(2*a);
result:
   exp(2*a)

ans =

exp(2*a)
```

To revert back to suppressed printing, use `maple traceoff`.

Status Output Argument

The `maple` function optionally returns two output arguments, `result` and `status`. If the `maple` call succeeds, Maple returns the result in the `result` argument and zero in the `status` argument. If the call fails, Maple returns an error code (a positive integer) in the `status` argument and a corresponding warning/error message in the `result` argument.

For example, the Maple `discrim` function calculates the discriminant of a polynomial and has the syntax `discrim(p,x)`, where `p` is a polynomial in `x`. Suppose you forget to supply the second argument when calling the `discrim` function

```
syms a b c x
[result, status] = maple('discrim', a*x^2+b*x+c)
```

Maple returns

```
result =
Error, (in discrim) invalid arguments

status =
     2
```

If you then include x

```
[result, status] = maple('discrim', a*x^2+b*x+c, x)
```

you get the following

```
result =
-4*a*c+b^2

status =
     0
```

Extended Symbolic Math Toolbox

In this section...

Packages of Library Functions

Extended Symbolic Math Toolbox allows you to access all nongraphics Maple packages, Maple programming features, and Maple procedures. Extended Symbolic Math Toolbox thus provides access to a large body of mathematical software written in the Maple language.

Maple programming features include looping (`for ... do ... od`, `while ... do ... od`) and conditionals (`if ... elif ... else ... fi`). See *The Maple Handbook* for information on how to use these and other features.

You can access specialized libraries, or "packages," with the Extended Symbolic Math Toolbox. The available packages are listed in "Maple Packages" on page 2-110.

You can use the Maple `with` command to load these packages. For example, if you want to use the orthogonal polynomials package, first get the Maple name of this package, using the statement

```
mhelp index[packages]
```

which returns

```
Index of descriptions for packages of library functions

Description:
- The following packages are available:
    ...
    orthopoly   orthogonal polynomials
    ...
```

You can then can access information about the package `orthopoly` by entering

```
mhelp orthopoly
```

To load the package, type

```
maple('with(orthopoly);')
```

This returns

```
ans =
[G, H, L, P, T, U]
```

which is a listing of function names in the `orthopoly` package. These functions are now loaded in the Maple workspace, and you can use them as you would any regular Maple function.

Maple Packages

Extended Symbolic Math Toolbox contains the following packages of Maple functions:

- Combinatorial Functions
- Differential Equation Tools
- Differential Forms
- Domains of Computation
- Euclidean Geometry
- Gaussian Integers
- Gröbner Bases
- Permutation and Finitely Presented Groups
- Lie Symmetries
- Boolean Logic
- Graph Networks
- Newman-Penrose Formalism
- Number Theory

- Numerical Approximation
- Orthogonal Polynomials
- p-adic Numbers
- Formal Power Series
- Simplex Linear Optimization
- Statistics
- Total Orders on Names
- Galois Fields
- Linear Recurrence Relation Tools
- Financial Mathematics
- Rational Generating Functions
- Tensor Computations

Procedure Example

The following example shows how you can access a Maple procedure through Extended Symbolic Math Toolbox. The example computes either symbolic or variable-precision numeric approximations to π, using a method derived by Richard Brent based from the arithmetic-geometric mean algorithm of Gauss. Here is the Maple source code:

```
pie := proc(n)
  # pie(n) takes n steps of an arithmetic geometric mean
  # algorithm for computing pi. The result is a symbolic
  # expression whose length roughly doubles with each step.
  # The number of correct digits in the evaluated string also
  # roughly doubles with each step.

  # Example: pie(5) is a symbolic expression with 1167
  # characters which, when evaluated, agrees with pi to 84
  # decimal digits.

  local a,b,c,d,k,t;

  a := 1:
     b := sqrt(1/2):
     c := 1/4:
     t := 1:

     for k from 1 to n do
        d := (b-a)/2:
        b := sqrt(a*b):
        a := a+d:
        c := c-t*d^2:
        t := 2*t:
     od;

     (a+b)^2/(4*c):

  end;
```

Copy the source code and paste it into the MATLAB Editor. Then save the file as `pie.src` in your `Work` directory. Using Extended Symbolic Math Toolbox, the MATLAB statement

```
procread('pie.src')
```

reads the specified file, deletes comments and newline characters, and sends the resulting string to Maple. (The MATLAB `ans` variable then contains a string representation of the `pie.src` file.)

You can access the `pie` function using the `maple` function. The statement

```
p = maple('pie',5)
```

returns a string representing the solution that begins and ends with

```
p =
1/4*(1/32+1/64*2^(1/2)+1/32*2^(3/4)+ ...
     ... *2^(1/2))*2^(3/4))^(1/2))^(1/2))^2)
```

You can use the `sym` command to convert the string to a symbolic object.

The assignment to the variable `b` in the second executable line,

```
b := sqrt(1/2)
```

causes the entire computation to be done symbolically. You can change the computation from symbolic to numeric by modifying the assignment statement to include decimal points:

```
b := sqrt(1./2.)
```

With this assignment, the entire computation uses variable-precision arithmetic at the current setting of `digits`. The commands

```
digits(100)
procread('pie.src')
p = maple('pie',5)
```

produce a 100-digit result:

```
p =
3.14159265358979323 ... 5628703211672038
```

The last 16 digits differ from those of π because, with five iterations, the algorithm gives only 84 digits.

Note that you can define your own MATLAB M-file that accesses a Maple procedure:

```
function p = pie1(n)
p = maple('pie',n)
```

Additional References

For additional information, consult these references.

[1] Schwarz, David, *Introduction to Maple*, Apprentice Hall, 1999.

[2] Graven, Frank, *The Maple Book*, Chapman & Hall/CRC, 2001.

[3] Wright, Francis, *Computing with Maple*, Chapman & Hall/CRC, 2001.

Functions — By Category

Calculus

`diff`	Differentiate symbolic expression
`int`	Integrate
`jacobian`	Jacobian matrix
`limit`	Limit of symbolic expression
`symsum`	Symbolic summation of series
`taylor`	Taylor series expansion

Linear Algebra

`colspace`	Basis for column space
`det`	Symbolic matrix determinant
`diag`	Create or extract symbolic diagonals
`eig`	Symbolic eigenvalues and eigenvectors
`expm`	Symbolic matrix exponential
`inv`	Symbolic matrix inverse
`jordan`	Jordan canonical form
`null`	Basis for null space
`poly`	Characteristic polynomial of matrix
`rank`	Symbolic matrix rank
`rref`	Reduced row echelon form
`svd`	Symbolic singular value decomposition
`tril`	Symbolic lower triangle
`triu`	Symbolic upper triangle

Simplification

`coeffs`	Coefficients of multivariate polynomial
`collect`	Collect coefficients
`expand`	Symbolic expansion of polynomials and elementary functions
`factor`	Factorization
`horner`	Horner nested polynomial representation
`numden`	Numerator and denominator
`simple`	Search for simplest form of symbolic expression
`simplify`	Symbolic simplification
`subexpr`	Rewrite symbolic expression in terms of common subexpressions
`subs`	Symbolic substitution in symbolic expression or matrix

Solution of Equations

`compose`	Functional composition
`dsolve`	Symbolic solution of ordinary differential equations
`finverse`	Functional inverse
`solve`	Symbolic solution of algebraic equations

Variable Precision Arithmetic

`digits`	Variable precision accuracy
`vpa`	Variable precision arithmetic

Arithmetic Operations

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`.*`	Array multiplication
`\`	Left division
`.\`	Array left division
`/`	Right division
`./`	Array right division
`^`	Matrix or scalar raised to a power
`.^`	Array raised to a power
`'`	Complex conjugate transpose
`.'`	Real transpose

Special Functions

`cosint`	Cosine integral
`dirac`	Dirac delta
`heaviside`	Compute Heaviside step function
`hypergeom`	Generalized hypergeometric

`lambertw`	Lambert's *W* function
`sinint`	Sine integral
`zeta`	Riemann Zeta

Access to Maple

`maple`	Access Maple kernel
`mapleinit`	Initialize Maple kernel
`mfun`	Numeric evaluation of Maple function
`mfunlist`	List special functions for use with `mfun`
`procread`	Install Maple procedure

Pedagogical and Graphical Applications

`ezcontour`	Contour plotter
`ezcontourf`	Filled contour plotter
`ezmesh`	3-D mesh plotter
`ezmeshc`	Combined mesh and contour plotter
`ezplot3`	3-D parametric curve plotter
`ezpolar`	Polar coordinate plotter
`ezsurf`	3-D colored surface plotter
`ezsurfc`	Combined surface and contour plotter
`funtool`	Function calculator

`rsums`	Interactive evaluation of Riemann sums
`taylortool`	Taylor series calculator

Conversions

`double`	Convert symbolic matrix to MATLAB numeric form
`int8, int16, int32, int64`	Convert symbolic matrix to signed integers
`poly2sym`	Polynomial coefficient vector to symbolic polynomial
`single`	Convert symbolic matrix to single precision
`sym2poly`	Symbolic-to-numeric polynomial conversion
`uint8, uint16, uint32, uint64`	Convert symbolic matrix to unsigned integers

Basic Operations

`ccode`	C code representation of symbolic expression
`ceil`	Round symbolic matrix toward positive infinity
`conj`	Symbolic complex conjugate
`eq`	Symbolic equality test
`fix`	Round toward zero

`floor`	Round symbolic matrix toward negative infinity
`fortran`	Fortran representation of symbolic expression
`frac`	Symbolic matrix elementwise fractional parts
`imag`	Imaginary part of complex number
`latex`	LaTeX representation of symbolic expression
`log10`	Logarithm base 10 of entries of symbolic matrix
`log2`	Logarithm base 2 of entries of symbolic matrix
`mod`	Symbolic matrix elementwise modulus
`pretty`	Pretty-print symbolic expressions
`quorem`	Symbolic matrix elementwise quotient and remainder
`real`	Real part of imaginary number
`round`	Symbolic matrix elementwise round
`size`	Symbolic matrix dimensions
`sort`	Sort symbolic vectors or polynomials
`sym`	Symbolic numbers, variables, and objects
`syms`	Shortcut for constructing symbolic objects

Integral Transforms

`fourier`	Fourier integral transform
`iztrans`	Inverse z-transform
`laplace`	Laplace transform

Functions — Alphabetical List

Arithmetic Operations

Purpose Perform arithmetic operations on symbols

Syntax

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

Description

`+`	Matrix addition. `A+B` adds `A` and `B`. `A` and `B` must have the same dimensions, unless one is scalar.
`-`	Matrix subtraction. `A-B` subtracts `B` from `A`. `A` and `B` must have the same dimensions, unless one is scalar.
`*`	Matrix multiplication. `A*B` is the linear algebraic product of `A` and `B`. The number of columns of `A` must equal the number of rows of `B`, unless one is a scalar.
`.*`	Array multiplication. `A.*B` is the entry-by-entry product of `A` and `B`. `A` and `B` must have the same dimensions, unless one is scalar.
`\`	Matrix left division. `A\B` solves the symbolic linear equations `A*X=B` for `X`. Note that `A\B` is roughly equivalent to `inv(A)*B`. Warning messages are produced if `X` does not exist or is not unique. Rectangular matrices `A` are allowed, but the equations must be consistent; a least squares solution is *not* computed.

`.\`	Array left division. `A.\B` is the matrix with entries `B(i,j)/A(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`/`	Matrix right division. `B/A` solves the symbolic linear equation `X*A=B` for `X`. Note that `B/A` is the same as `(A.'\B.').'`. Warning messages are produced if `X` does not exist or is not unique. Rectangular matrices `A` are allowed, but the equations must be consistent; a least squares solution is not computed.
`./`	Array right division. `A./B` is the matrix with entries `A(i,j)/B(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`^`	Matrix power. `A^B` raises the square matrix `A` to the integer power `B`. If `A` is a scalar and `B` is a square matrix, `A^B` raises `A` to the matrix power `B`, using eigenvalues and eigenvectors. `A^B`, where `A` and `B` are both matrices, is an error.
`.^`	Array power. `A.^B` is the matrix with entries `A(i,j)^B(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`'`	Matrix Hermition transpose. If `A` is complex, `A'` is the complex conjugate transpose.
`.'`	Array transpose. `A.'` is the real transpose of `A`. `A.'` does not conjugate complex entries.

Examples

The following statements

```
syms a b c d;
A = [a b; c d];
A*A/A
A*A-A^2
```

return

```
[ a, b]
[ c, d]

[ 0, 0]
[ 0, 0]
```

The following statements

```
a11 a12 a21 a22 b1 b2;
A = [a11 a12; a21 a22];
B = [b1 b2];
X = B/A;
x1 = X(1)
x2 = X(2)
```

return

```
x1 =
(-a21*b2+b1*a22)/(a11*a22-a12*a21)

x2 =
(a11*b2-a12*b1)/(a11*a22-a12*a21)
```

See Also

`null`, `solve`

Purpose

C code representation of symbolic expression

Syntax

```
ccode(s)
```

Description

`ccode(s)` returns a fragment of C that evaluates the symbolic expression `s`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
ccode(f)
```

return

```
t0 = x-x*x/2.0+x*x*x/3.0-x*x*x*x/4.0+x*x*x*x*x/5.0;
```

The statements

```
H = sym(hilb(3));
ccode(H)
```

return

```
H[0][0] = 1.0;          H[0][1] = 1.0/2.0;      H[0][2] = 1.0/3.0;
H[1][0] = 1.0/2.0;      H[1][1] = 1.0/3.0;      H[1][2] = 1.0/4.0;
H[2][0] = 1.0/3.0;      H[2][1] = 1.0/4.0;      H[2][2] = 1.0/5.0;
```

See Also

`fortran`, `latex`, `pretty`

Purpose

Round symbolic matrix toward positive infinity

Syntax

```
Y = ceil(x)
```

Description

`Y = ceil(x)` is the matrix of the smallest integers greater than or equal to `x`.

Example

```
x = sym(-5/2)
[fix(x) floor(x) round(x) ceil(x) frac(x)]
= [ -2, -3, -3, -2, -1/2]
```

See Also

`round`, `floor`, `fix`, `frac`

Purpose

Coefficients of multivariate polynomial

Syntax

```
C = coeffs(p)
C = coeffs(p,x)
[C,T] = coeffs(p,x)
```

Description

`C = coeffs(p)` returns the coefficients of the polynomial `p` with respect to all the indeterminates of `p`.

`C = coeffs(p,x)` returns the coefficients of the polynomial `p` with respect to `x`.

`[C,T] = coeffs(p,x)` also returns an expression sequence of the terms of `p`. There is a one-to-one correspondence between the coefficients and the terms of `p`.

Examples

```
syms x
t = 2 + (3 + 4*log(x))^2 - 5*log(x);
coeffs(expand(t)) = [ 11, 19, 16]

syms a b c x
y = a + b*sin(x) + c*sin(2*x)
coeffs(y,sin(x)) = [a+c*sin(2*x), b]
coeffs(expand(y),sin(x)) = [a, b+2*c*cos(x)]

syms x y
z = 3*x^2*y^2 + 5*x*y^3
coeffs(z) = [3, 5]
coeffs(z,x) = [5*y^3, 3*y^2]
[c,t] = coeffs(z,y) returns c = [3*x^2, 5*x], t = [y^2, y^3]
```

See Also

`sym2poly`

Purpose

Collect coefficients

Syntax

```
R = collect(S)
R = collect(S,v)
```

Description

`R = collect(S)` returns an array of collected polynomials for each polynomial in the array `S` of polynomials.

`R = collect(S,v)` collects terms containing the variable `v`.

Examples

The following statements

```
syms x y;
R1 = collect((exp(x)+x)*(x+2))
R2 = collect((x+y)*(x^2+y^2+1), y)
R3 = collect([(x+1)*(y+1),x+y])
```

return

```
R1 =
x^2+(exp(x)+2)*x+2*exp(x)

R2 =
y^3+x*y^2+(x^2+1)*y+x*(x^2+1)

R3 =
[(y+1)*x+y+1, x+y]
```

See Also

`expand`, `factor`, `simple`, `simplify`, `syms`

Purpose Basis for column space

Syntax

```
B = colspace(A)
```

Description `B = colspace(A)` returns a matrix whose columns form a basis for the column space of `A`. `A` is a symbolic or numeric matrix. Note that `size(colspace(A),2)` returns the rank of `A`.

Examples The statements

```
A = sym([2,0;3,4;0,5])
B = colspace(A)
```

return

```
A =
[2,0]
[3,4]
[0,5]

B =
[    1,   0]
[    0,   1]
[-15/8, 5/4]
```

See Also `null`, `orth`

compose

Purpose

Functional composition

Syntax

```
compose(f,g)
compose(f,g,z)
compose(f,g,x,z)
compose(f,g,x,y,z)
```

Description

`compose(f,g)` returns `f(g(y))` where `f = f(x)` and `g = g(y)`. Here `x` is the symbolic variable of `f` as defined by `findsym` and `y` is the symbolic variable of `g` as defined by `findsym`.

`compose(f,g,z)` returns `f(g(z))` where `f = f(x)`, `g = g(y)`, and `x` and `y` are the symbolic variables of `f` and `g` as defined by `findsym`.

`compose(f,g,x,z)` returns `f(g(z))` and makes `x` the independent variable for `f`. That is, if `f = cos(x/t)`, then `compose(f,g,x,z)` returns `cos(g(z)/t)` whereas `compose(f,g,t,z)` returns `cos(x/g(z))`.

`compose(f,g,x,y,z)` returns `f(g(z))` and makes `x` the independent variable for `f` and `y` the independent variable for `g`. For `f = cos(x/t)` and `g = sin(y/u)`, `compose(f,g,x,y,z)` returns `cos(sin(z/u)/t)` whereas `compose(f,g,x,u,z)` returns `cos(sin(y/z)/t)`.

Examples

Suppose

```
syms x y z t u;
f = 1/(1 + x^2); g = sin(y); h = x^t; p = exp(-y/u);
```

Then

```
compose(f,g)       ->  1/(1+sin(y)^2)
compose(f,g,t)     ->  1/(1+sin(t)^2)
compose(h,g,x,z)   ->  sin(z)^t
compose(h,g,t,z)   ->  x^sin(z)
compose(h,p,x,y,z) ->  exp(-z/u)^t
compose(h,p,t,u,z) ->  x^exp(-y/z)
```

See Also

`finverse`, `subs`, `syms`

Purpose

Symbolic complex conjugate

Syntax

```
conj(X)
```

Description

`conj(X)` is the complex conjugate of `X`.

For a complex `X`, `conj(X) = real(X) - i*imag(X)`.

See Also

`real`, `imag`

Purpose Cosine integral

Syntax

```
Y = cosint(X)
```

Description `Y = cosint(X)` evaluates the cosine integral function at the elements of `X`, a numeric matrix, or a symbolic matrix. The cosine integral function is defined by

$$Ci(x) = \gamma + \ln(x) + \int_0^x \frac{\cos t - 1}{t} dt$$

where γ is Euler's constant 0.577215664...

Examples `cosint(7.2)` returns `0.0960`.

`cosint([0:0.1:1])` returns

```
  Columns 1 through 7

      Inf   -1.7279   -1.0422   -0.6492   -0.3788   -0.1778   -0.0223

  Columns 8 through 11

      0.1005    0.1983    0.2761    0.3374
```

The statements

```
syms x;
f = cosint(x);
diff(f)
```

return

```
cos(x)/x
```

See Also `sinint`

Purpose

Symbolic matrix determinant

Syntax

`r = det(A)`

Description

`r = det(A)` computes the determinant of `A`, where `A` is a symbolic or numeric matrix. `det(A)` returns a symbolic expression, if `A` is symbolic; a numeric value, if `A` is numeric.

Examples

The statements

```
syms a b c d;
det([a, b; c, d])
```

return

```
a*d - b*c
```

The statements

```
A = sym([2/3 1/3;1 1])
r = det(A)
```

return

```
A =
[ 2/3, 1/3]
[   1,   1]

r = 1/3
```

diag

Purpose

Create or extract symbolic diagonals

Syntax

```
diag(A,k)
diag(A)
```

Description

`diag(A,k)`, where `A` is a row or column vector with `n` components, returns a square symbolic matrix of order `n+abs(k)`, with the elements of `A` on the `k`-th diagonal. `k = 0` signifies the main diagonal; `k > 0`, the `k`-th diagonal above the main diagonal; `k < 0`, the `k`-th diagonal below the main diagonal.

`diag(A,k)`, where `A` is a square symbolic matrix, returns a column vector formed from the elements of the `k`-th diagonal of `A`.

`diag(A)`, where `A` is a vector with `n` components, returns an `n`-by-`n` diagonal matrix having `A` as its main diagonal.

`diag(A)`, where `A` is a square symbolic matrix, returns the main diagonal of `A`.

Examples

Suppose

```
v = [a b c]
```

Then both `diag(v)` and `diag(v,0)` return

```
[ a, 0, 0 ]
[ 0, b, 0 ]
[ 0, 0, c ]
```

`diag(v,-2)` returns

```
[ 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0]
[ a, 0, 0, 0, 0]
[ 0, b, 0, 0, 0]
[ 0, 0, c, 0, 0]
```

Suppose

```
A =
    [ a, b, c ]
    [ 1, 2, 3 ]
    [ x, y, z ]
```

`diag(A)` returns

```
[ a ]
[ 2 ]
[ z ]
```

`diag(A,1)` returns

```
[ b ]
[ 3 ]
```

See Also

`tril`, `triu`

Purpose

Differentiate symbolic expression

Syntax

```
diff(S)
diff(S,'v')
diff(S,n)
diff(S,'v',n)
```

Description

`diff(S)` differentiates a symbolic expression `S` with respect to its free variable as determined by `findsym`.

`diff(S,'v')` or `diff(S,sym('v'))` differentiates `S` with respect to `v`.

`diff(S,n)`, for a positive integer `n`, differentiates `S` `n` times.

`diff(S,'v',n)` and `diff(S,n,'v')` are also acceptable.

Examples

Assume

```
syms x t
```

Then

```
diff(sin(x^2))
```

returns

```
2*cos(x^2)*x
```

and

```
diff(t^6,6)
```

returns

```
720
```

See Also

`int`, `jacobian`, `findsym`

Purpose

Variable precision accuracy

Syntax

```
digits
digits(d)
d = digits
```

Description

`digits` specifies the number of significant decimal digits that Maple uses to do variable precision arithmetic (VPA). The default value is 32 digits.

`digits(d)` sets the current VPA accuracy to `d` digits.

`d = digits` returns the current VPA accuracy.

Examples

If

```
z = 1.0e-16
x = 1.0e+2
digits(14)
```

then

```
y = vpa(x*z+1)
```

uses 14-digit decimal arithmetic and returns

```
y =
1.0000000000000
```

Whereas

```
digits(15)
y = vpa(x*z+1)
```

used 15-digit decimal arithmetic and returns

```
y =
1.00000000000001
```

See Also `double`, `vpa`

Purpose

Dirac delta

Syntax

`dirac(x)`

Description

`dirac(x)` returns the Dirac delta function of `x`.

The Dirac delta function, `dirac`, has the value 0 for all `x` not equal to 0 and the value `Inf` for `x` = 0. Several functions in Symbolic Math Toolbox return answers in terms of `dirac`.

`dirac` has the property that

```
int(dirac(x-a)*f(x),-inf,inf) = f(a)
```

for any function `f` and real number `a`. It also has the following relationship to the function `heaviside`:

```
diff(heaviside(x),x)

ans =

dirac(x)
```

Example

```
syms x a
a = 5;
int(dirac(x-a)*sin(x),-inf, inf)

ans =

sin(5)
```

See Also

`heaviside`

Purpose

Convert symbolic matrix to MATLAB numeric form

Syntax

```
r = double(S)
```

Description

`r = double(S)` converts the symbolic object `S` to a numeric object. If `S` is a symbolic constant or constant expression, `double` returns a double-precision floating-point number representing the value of `S`. If `S` is a symbolic matrix whose entries are constants or constant expressions, `double` returns a matrix of double precision floating-point numbers representing the values of `S`'s entries.

Examples

`double(sym('(1+sqrt(5))/2'))` returns `1.6180`.

The following statements

```
a = sym(2*sqrt(2));
b = sym((1-sqrt(3))^2);
T = [a, b]
double(T)
```

return

```
ans =
    2.8284    0.5359
```

See Also

`sym`, `vpa`

Purpose

Symbolic solution of ordinary differential equations

Syntax

```
r = dsolve('eq1,eq2,...', 'cond1,cond2,...', 'v')
r = dsolve('eq1','eq2',...,'cond1','cond2',...,'v')
dsolve('eq1,eq2,...', 'cond1,cond2,...', 'v')
```

Description

`r = dsolve('eq1,eq2,...', 'cond1,cond2,...', 'v')` or `r = dsolve('eq1','eq2',...,'cond1','cond2',...,'v')` symbolically solves the ordinary differential equation(s) specified by `eq1`, `eq2,...` using `v` as the independent variable and the boundary and/or initial condition(s) specified by `cond1,cond2,...`.

The default independent variable is `t`.

The letter `D` denotes differentiation with respect to the independent variable; with the primary default, this is `d/dx`. A `D` followed by a digit denotes repeated differentiation. For example, `D2` is `d2/dx2`. Any character immediately following a differentiation operator is a dependent variable. For example, `D3y` denotes the third derivative of `y(x)` or `y(t)`.

Initial/boundary conditions are specified with equations like `y(a) = b` or `Dy(a) = b`, where `y` is a dependent variable and `a` and `b` are constants. If the number of initial conditions specified is less than the number of dependent variables, the resulting solutions will contain the arbitrary constants `C1`, `C2,...`.

You can also input each equation and/or initial condition as a separate symbolic equation. `dsolve` accepts up to 12 input arguments.

Three different types of output are possible.

- For one equation and one output, `dsolve` returns the resulting solution with multiple solutions to a nonlinear equation in a symbolic vector.
- For several equations and an equal number of outputs, `dsolve` sorts the results in lexicographic order and assigns them to the outputs.

- For several equations and a single output, `dsolve` returns a structure containing the solutions.

If `dsolve` cannot find a closed-form (explicit) solution, it attempts to find an implicit solution. When `dsolve` returns an implicit solution, it issues a warning. If `dsolve` cannot find either an explicit or an implicit solution, then it issues a warning and returns the empty `sym`. In such a case, you can find a numeric solution, using the MATLAB `ode23` or `ode45` functions. In some cases involving nonlinear equations, the output will be an equivalent lower order differential equation or an integral.

With no output arguments, `dsolve('eq1,eq2,...', 'cond1,cond2,...', 'v')` returns a list of solutions.

Examples

`dsolve('Dx = -a*x')` returns

```
C1*exp(-a*t)
```

`dsolve('Df = f + sin(t)')` returns

```
-1/2*cos(t)-1/2*sin(t)+exp(t)*C1
```

`dsolve('(Dy)^2 + y^2 = 1','s')` returns

```
[         -1]
[          1]
[  sin(s-C1)]
[ -sin(s-C1)]
```

`dsolve('Dy = a*y', 'y(0) = b')` returns

```
b*exp(a*t)
```

`dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0')` returns

```
cos(a*t)
```

`dsolve('Dx = y', 'Dy = -x')` returns

```
x: [1x1 sym]
y: [1x1 sym]
```

Diagnostics

If `dsolve` cannot find an analytic solution for an equation, it prints the warning

```
Warning: explicit solution could not be found
```

and return an empty `sym` object.

See Also

`syms`

Purpose

Symbolic eigenvalues and eigenvectors

Syntax

```
lambda = eig(A)
[V,D] = eig(A)
[V,D,P] = eig(A)
lambda = eig(vpa(A))
[V,D] = eig(vpa(A))
```

Description

`lambda = eig(A)` returns a symbolic vector containing the eigenvalues of the square symbolic matrix `A`.

`[V,D] = eig(A)` returns a matrix `V` whose columns are eigenvectors and a diagonal matrix `D` containing eigenvalues. If the resulting `V` is the same size as `A`, then `A` has a full set of linearly independent eigenvectors that satisfy `A*V = V*D`.

`[V,D,P] = eig(A)` also returns `P`, a vector of indices whose length is the total number of linearly independent eigenvectors, so that `A*V = V*D(P,P)`.

`lambda = eig(vpa(A))` and `[V,D] = eig(vpa(A))` compute numeric eigenvalues and eigenvectors, respectively, using variable precision arithmetic. If `A` does not have a full set of eigenvectors, the columns of `V` will not be linearly independent.

Examples

The statements

```
R = sym(gallery('rosser'));
eig(R)
```

return

```
ans =
[                0]
[             1020]
[ 510+100*26^(1/2)]
[ 510-100*26^(1/2)]
[   10*10405^(1/2)]
[   -10*10405^(1/2)]
[              1000]
[              1000]
```

`eig(vpa(R))` returns

```
ans =

[     -1020.0490184299968238463137913055]
[ .56512999999999999999999999999800e-28]
[   .98048640721516997177589097485157e-1]
[      1000.0000000000000000000000000002]
[      1000.0000000000000000000000000003]
[      1019.9019513592784830028224109024]
[      1020.0000000000000000000000000003]
[      1020.0490184299968238463137913055]
```

The statements

```
A = sym(gallery(5));
[v,lambda] = eig(A)
```

return

```
v =
[        0]
[   21/256]
[  -71/128]
[  973/256]
[        1]

lambda =

[ 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0]
[ 0, 0, 0, 0, 0]
```

See Also `jordan`, `poly`, `svd`, `vpa`

Purpose Symbolic equality test

Syntax `eq(A,B)`

Description `eq(A,B)` overloads the symbolic `A==B` and performs the element by element comparisons between `A` and `B`. The result is true if `A` and `B` have the same string representation. `eq` does not expand or simplify the string expressions before making the comparison.

See Also `sym`

expm

Purpose Symbolic matrix exponential

Syntax `expm(A)`

Description `expm(A)` is the matrix exponential of the symbolic matrix `A`.

Examples The statements

```
syms t;
A = [0 1; -1 0];
expm(t*A)
```

return

```
[  cos(t),  sin(t)]
[ -sin(t),  cos(t)]
```

Purpose

Symbolic expansion of polynomials and elementary functions

Syntax

```
expand(S)
```

Description

`expand(S)` writes each element of a symbolic expression `S` as a product of its factors. `expand` is most often used only with polynomials, but also expands trigonometric, exponential, and logarithmic functions.

Examples

`expand((x-2)*(x-4))` returns

```
x^2-6*x+8
```

`expand(cos(x+y))` returns

```
cos(x)*cos(y)-sin(x)*sin(y)
```

`expand(exp((a+b)^2))` returns

```
exp(a^2)*exp(a*b)^2*exp(b^2)
```

`expand([sin(2*t), cos(2*t)])` returns

```
[2*sin(t)*cos(t), 2*cos(t)^2-1]
```

See Also

`collect`, `factor`, `horner`, `simple`, `simplify`, `syms`

ezcontour

Purpose

Contour plotter

Syntax

```
ezcontour(f)
ezcontour(f,domain)
ezcontour(...,n)
```

Description

`ezcontour(f)` plots the contour lines of *f*(*x*,*y*), where `f` is a symbolic expression that represents a mathematical function of two variables, such as *x* and *y*.

The function *f* is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function *f* is not defined (singular) for points on the grid, then these points are not plotted.

`ezcontour(f,domain)` plots *f*(*x*,*y*) over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If *f* is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezcontour(u^2 - v^3,[0,1],[3,6])` plots the contour lines for u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

`ezcontour(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour` automatically adds a title and axis labels.

Examples

The following mathematical expression defines a function of two variables, *x* and *y*.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

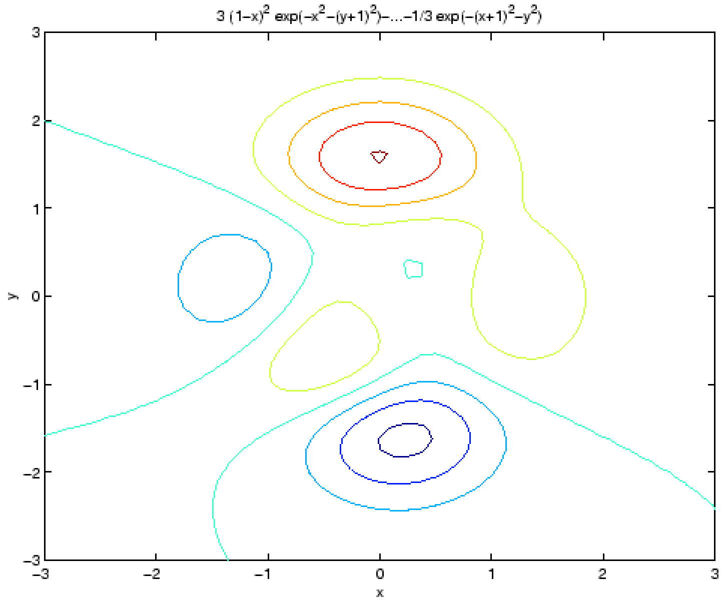
`ezcontour` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2) ...
   - 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2) ...
   - 1/3*exp(-(x+1)^2 - y^2);
```

For convenience, this expression is written on three lines.

Pass the sym `f` to `ezcontour` along with a domain ranging from -3 to 3 and specify a computational grid of 49-by-49.

```
ezcontour(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB abbreviates the string.

See Also

`contour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`

ezcontourf

Purpose

Filled contour plotter

Syntax

```
ezcontour(f)
ezcontour(f,domain)
ezcontourf(...,n)
```

Description

`ezcontour(f)` plots the contour lines of $f(x,y)$, where `f` is a `sym` that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezcontour(f,domain)` plots $f(x,y)$ over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezcontourf(u^2 - v^3,[0,1],[3,6])` plots the contour lines for u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

`ezcontourf(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf` automatically adds a title and axis labels.

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

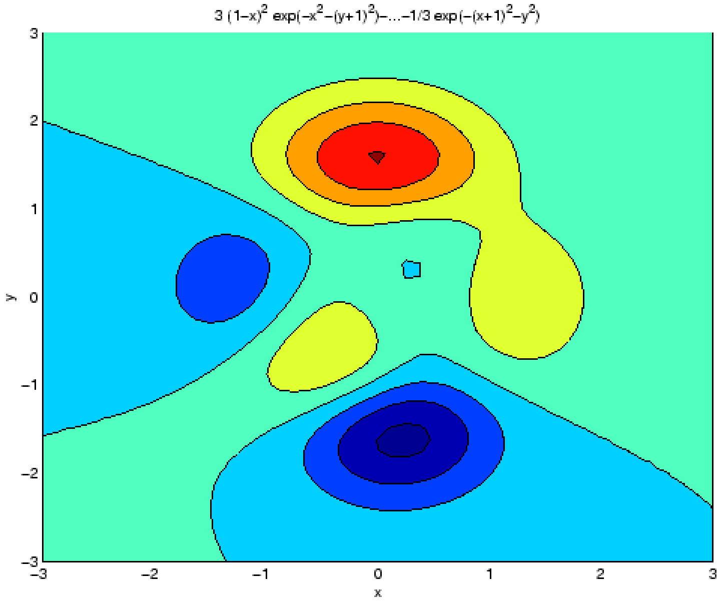
`ezcontourf` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2) ...
   - 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2) ...
   - 1/3*exp(-(x+1)^2 - y^2);
```

For convenience, this expression is written on three lines.

Pass the `sym` `f` to `ezcontourf` along with a domain ranging from -3 to 3 and specify a grid of 49-by-49.

```
ezcontourf(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB abbreviates the string.

See Also

`contourf`, `ezcontour`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`

Purpose

3-D mesh plotter

Syntax

```
ezmesh(f)
ezmesh(f,domain)
ezmesh(x,y,z)
ezmesh(x,y,z,[smin,smax,tmin,tmax])
ezmesh(x,y,z,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
```

Description

`ezmesh(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmesh(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezmesh(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezmesh(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmesh(x,y,z,[smin,smax,tmin,tmax])` or `ezmesh(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots f over a disk centered on the domain.

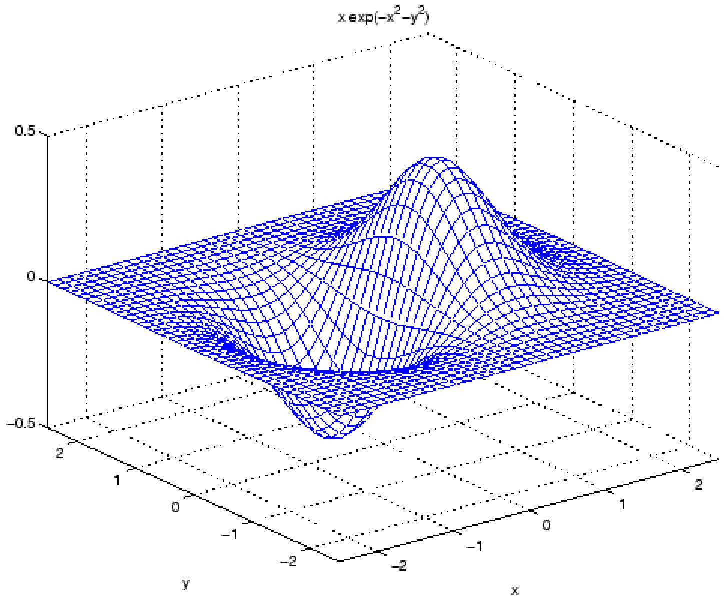
Examples

This example visualizes the function,

$$f(x, y) = xe^{-x^2-y^2}$$

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color.

```
syms x y
ezmesh(x*exp(-x^2-y^2),[-2.5,2.5],40)
colormap([0 0 1])
```



See Also

`ezcontour`, `ezcontourf`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `mesh`

ezmeshc

Purpose

Combined mesh and contour plotter

Syntax

```
ezmeshc(f)
ezmeshc(f,domain)
ezmeshc(x,y,z)
ezmeshc(x,y,z,[smin,smax,tmin,tmax])
ezmeshc(x,y,z,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
```

Description

`ezmeshc(f)` creates a graph of $f(x,y)$, where f is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$,

$-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmeshc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezmeshc(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezmeshc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmeshc(x,y,z,[smin,smax,tmin,tmax])` or `ezmeshc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots f over a disk centered on the domain.

Examples

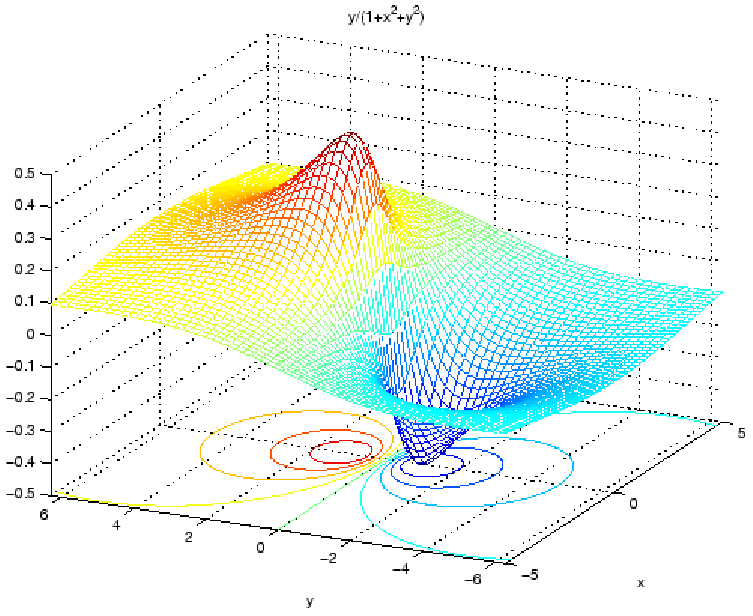
Create a mesh/contour graph of the expression,

$$f(x, y) = \frac{y}{1 + x^2 + y^2}$$

over the domain -5 < x < 5, -2*pi < y < 2*pi.

```
syms x y
ezmeshc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi])
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65 and elevation = 26).



See Also

`ezcontour`, `ezcontourf`, `ezmesh`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `meshc`

Purpose

Function plotter

Syntax

```
ezplot(f)
ezplot(f,[xmin xmax])
ezplot(f,[xmin xmax],fign)
ezplot(f,[xmin,xmax,ymin,ymax])
ezplot(x,y)
ezplot(x,y,[tmin,tmax])
ezplot(...,figure)
```

Description

`ezplot(f)` plots the expression $f = f(x)$ over the default domain $-2\pi < x < 2\pi$.

`ezplot(f,[xmin xmax])` plots $f = f(x)$ over the specified domain. It opens and displays the result in a window labeled **Figure No. 1**. If any plot windows are already open, `ezplot` displays the result in the highest numbered window.

`ezplot(f,[xmin xmax],fign)` opens (if necessary) and displays the plot in the window labeled `fign`.

For implicitly defined functions, $f = f(x,y)$.

`ezplot(f)` plots $f(x,y) = 0$ over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`ezplot(f,[xmin,xmax,ymin,ymax])` plots $f(x,y) = 0$ over `xmin` $< x <$ `xmax` and `ymin` $< y <$ `ymax`.

`ezplot(f,[min,max])`plots $f(x,y) = 0$ over `min` $< x <$ `max` and `min` $< y <$ `max`.

If f is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezplot(u^2 - v^2 - 1,[-3,2,-2,3])` plots $u^2 - v^2 - 1 = 0$ over $-3 < u < 2$, $-2 < v < 3$.

`ezplot(x,y)` plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default domain $0 < t < 2\pi$.

`ezplot(x,y,[tmin,tmax])` plots $x = x(t)$ and $y = y(t)$ over `tmin` $< t$ $<$ `tmax`.

`ezplot(...,figure)` plots the given function over the specified domain in the figure window identified by the handle `figure`.

Algorithm

If you do not specify a plot range, `ezplot` samples the function between `-2*pi` and `2*pi` and selects a subinterval where the variation is significant as the plot domain. For the range, `ezplot` omits extreme values associated with singularities.

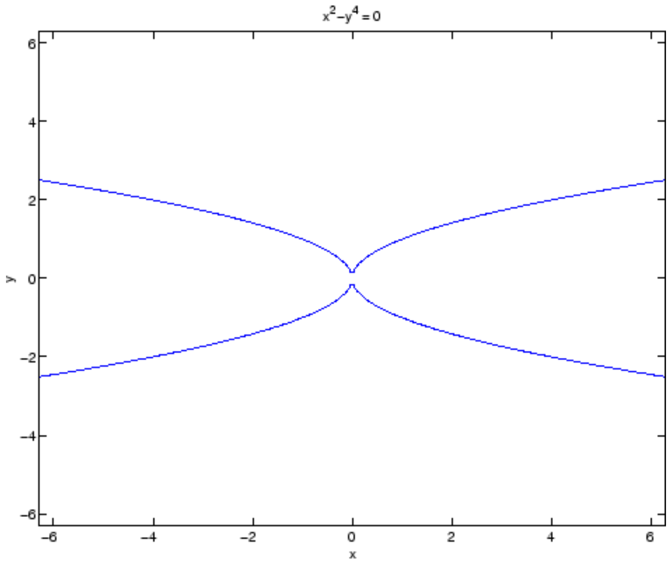
Examples

This example plots the implicitly defined function,

$x^2 - y^4 = 0$

over the domain $[-2\pi, 2\pi]$

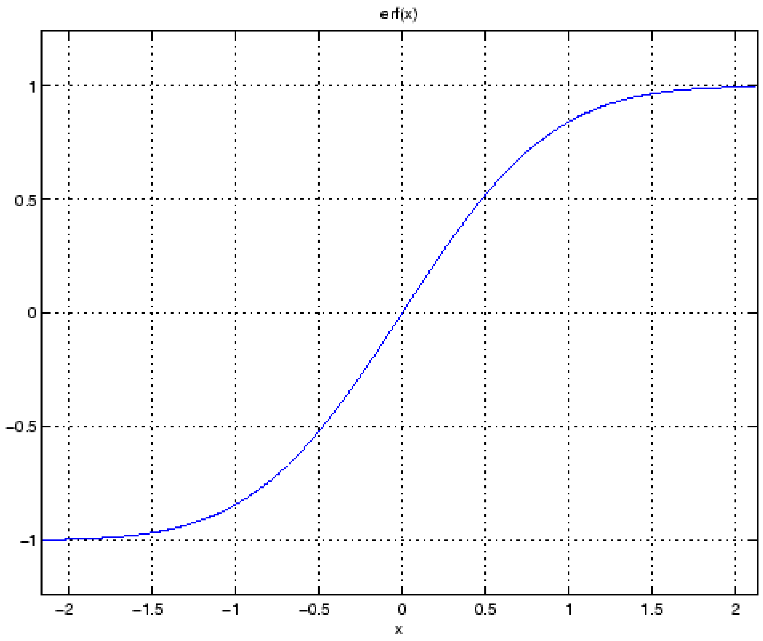
```
syms x y
ezplot(x^2-y^4)
```



The following statements

```
syms x
ezplot(erf(x))
grid
```

plot a graph of the error function.



See Also

`ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `plot`

Purpose

3-D parametric curve plotter

Syntax

```
ezplot3(x,y,z)
ezplot3(x,y,z,[tmin,tmax])
ezplot3(...,'animate')
```

Description

`ezplot3(x,y,z)` plots the spatial curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the default domain $0 < t < 2\pi$.

`ezplot3(x,y,z,[tmin,tmax])` plots the curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the domain `tmin` < t < `tmax`.

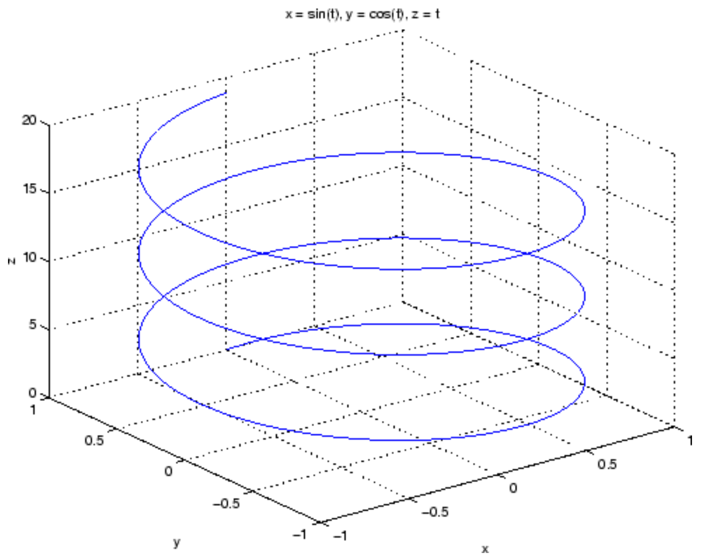
`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

Examples

This example plots the parametric curve, $x = \sin t, y = \cos t, z = t$

over the domain $[0, 6\pi]$

```
syms t; ezplot3(sin(t), cos(t), t,[0,6*pi])
```



See Also `ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezpolar`, `ezsurf`, `ezsurfc`, `plot3`

Purpose

Polar coordinate plotter

Syntax

```
ezpolar(f)
ezpolar(f,[a,b])
```

Description

`ezpolar(f)` plots the polar curve *rho* = *f*(*theta*) over the default domain $0 < theta > 2\pi$.

`ezpolar(f,[a,b])` plots *f* for `a` < *theta* < `b`.

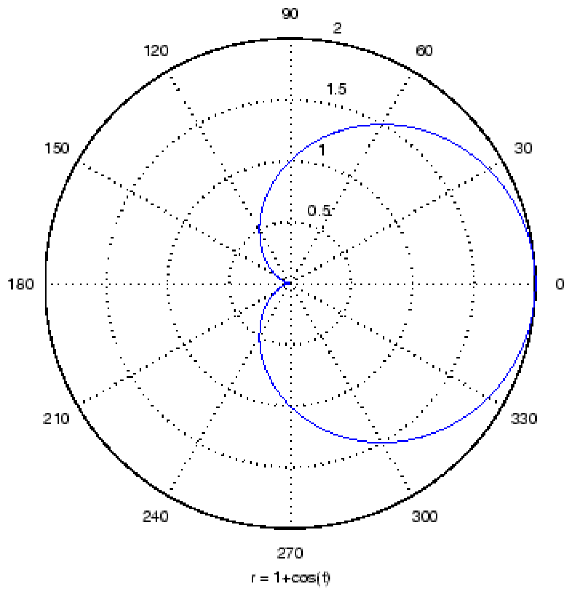
Example

This example creates a polar plot of the function,

$1 + cos(t)$

over the domain $[0, 2\pi]$

```
syms t
ezpolar(1+cos(t))
```



Purpose

3-D colored surface plotter

Syntax

```
ezsurf(f)
ezsurf(f,domain)
ezsurf(x,y,z)
ezsurf(x,y,z,[smin,smax,tmin,tmax])
ezsurf(x,y,z,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
```

`ezsurf(f)` plots over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function *f* is not defined (singular) for points on the grid, then these points are not plotted.

`ezsurf(f,domain)` plots *f* over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If *f* is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezsurf(u^2 - v^3,[0,1],[3,6])` plots u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

`ezsurf(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurf(x,y,z,[smin,smax,tmin,tmax])` or `ezsurf(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezsurf(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurf(...,'circ')` plots *f* over a disk centered on the domain.

Examples

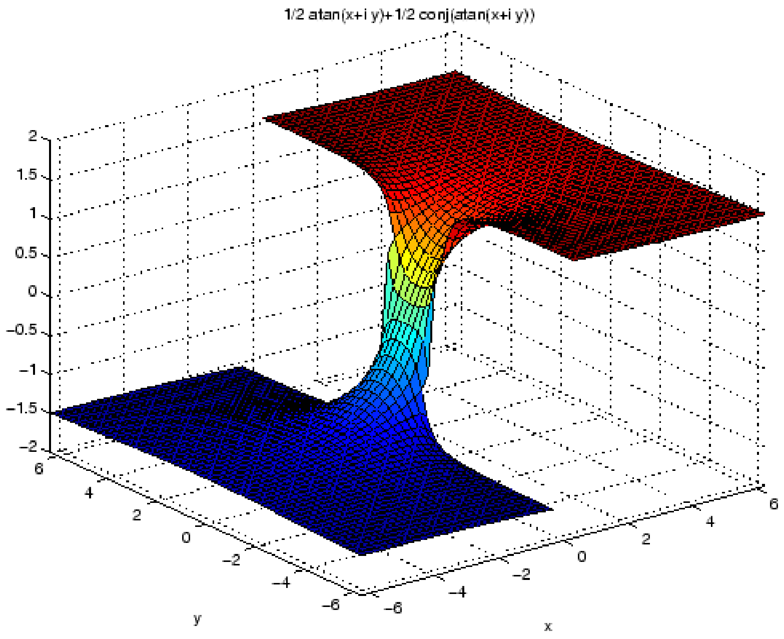
`ezsurf` does not graph points where the mathematical function is not defined (these data points are set to `NaN`s, which MATLAB does not plot).

This example illustrates this filtering of singularities/discontinuous points by graphing the function,

$$f(x,y) = real(atan(x + iy))$$

over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$

```
syms x y
ezsurf(real(atan(x+i*y)))
```



Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

See Also

`ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezpolar`, `ezsurfc`, `surf`

ezsurfc

Purpose

Combined surface and contour plotter

Syntax

```
ezsurfc(f)
ezsurfc(f,domain)
ezsurfc(x,y,z)
ezsurfc(x,y,z,[smin,smax,tmin,tmax])
ezsurfc(x,y,z,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
```

Description

`ezsurfc(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezsurfc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [xmin, xmax, ymin, ymax] or a 2-by-1 vector [min, max] (where, min < x < max, min < y < max).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints umin, umax, vmin, and vmax are sorted alphabetically. Thus, `ezsurfc(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezsurfc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurfc(x,y,z,[smin,smax,tmin,tmax])` or `ezsurfc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots f over a disk centered on the domain.

Examples

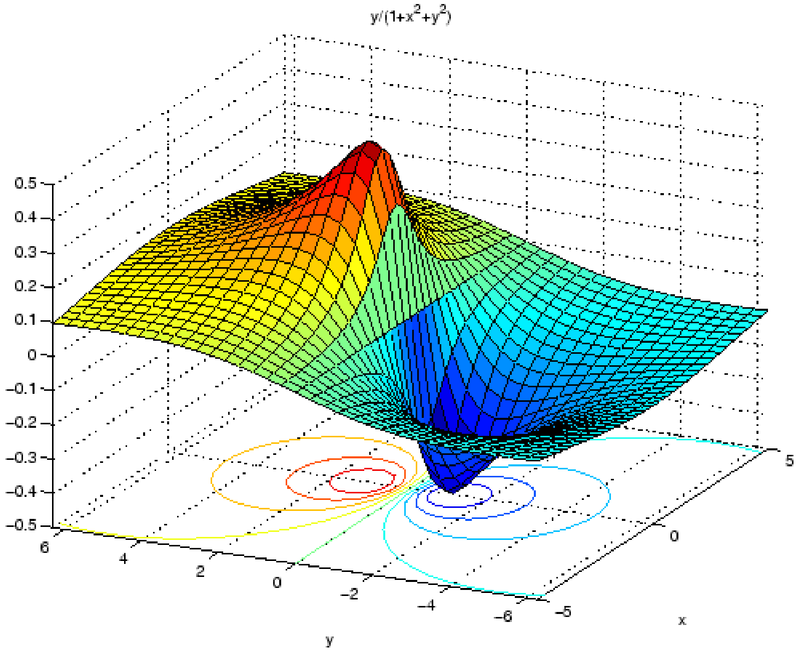
Create a surface/contour plot of the expression,

$$f(x, y) = \frac{y}{1 + x^2 + y^2}$$

over the domain -5 < x < 5, -2*pi < y < 2*pi, with a computational grid of size 35-by-35

```
syms x y
ezsurfc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi],35)
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65 and elevation = 26).



See Also

`ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezpolar`, `ezsurf`, `surfc`

factor

Purpose

Factorization

Syntax

```
factor(X)
```

Description

`factor(X)` can take a positive integer, an array of symbolic expressions, or an array of symbolic integers as an argument. If `N` is a positive integer, `factor(N)` returns the prime factorization of `N`.

If `S` is a matrix of polynomials or integers, `factor(S)` factors each element. If any element of an integer array has more than 16 digits, you must use `sym` to create that element, for example, `sym('N')`.

Examples

`factor(x^3-y^3)` returns

```
(x-y)*(x^2+x*y+y^2)
```

`factor([a^2-b^2, a^3+b^3])` returns

```
[(a-b)*(a+b), (a+b)*(a^2-a*b+b^2)]
```

`factor(sym('12345678901234567890'))` returns

```
(2)*(3)^2*(5)*(101)*(3803)*(3607)*(27961)*(3541)
```

See Also

`collect`, `expand`, `horner`, `simplify`, `simple`

findsym

Purpose Determine variables in symbolic expression or matrix

Syntax

```
findsym(S)
findsym(S,n)
```

Description `findsym(S)` for a symbolic expression or matrix `S`, returns all symbolic variables in `S` in lexicographical order, separated by commas. If `S` does not contain any variables, `findsym` returns an empty string.

`findsym(S,n)` returns the `n` variables alphabetically closest to `x`. For single-letter variables that are equally close to `x` in the alphabet, `findsym` treats the letter that comes later in the alphabet as being "closer."

For example, `findsym(S,1)` returns the variable closest to `x`. This is the default symbolic variable, when none is specified, for the `subs` and `diff` commands.

Note A symbolic variable is an alphanumeric name, other than `i` or `j`, that begins with an alphabetic character.

Examples

```
syms a x y z t
findsym(sin(pi*t))
ans = t
findsym(x+i*y-j*z)
ans = x, y, z
findsym(a+y,1)
returns y.
```

See Also `compose`, `diff`, `int`, `limit`, `taylor`

finverse

Purpose Functional inverse

Syntax

```
g = finverse(f)
g = finverse(f,v)
```

Description `g = finverse(f)` returns the functional inverse of `f`. `f` is a scalar `sym` representing a function of one symbolic variable, say `x`. Then `g` is a scalar `sym` that satisfies `g(f(x)) = x`. That is, `finverse(f)` returns f^{-1}, provided f^{-1} exists.

`g = finverse(f,v)` uses the symbolic variable `v`, where `v` is a `sym`, as the independent variable. Then `g` is a scalar `sym` that satisfies `g(f(v)) = v`. Use this form when `f` contains more than one symbolic variable.

Examples `finverse(1/tan(x))` returns

```
atan(1/x)
```

`finverse(exp(u-2*v),u)` returns

```
2*v+log(u)
```

See Also `compose`, `syms`

Purpose Round toward zero

Syntax `fix(X)`

Description `fix(X)` is the matrix of the integer parts of `X`.

`fix(X) = floor(X)` if `X` is positive and `ceil(X)` if `X` is negative.

See Also `round`, `ceil`, `floor`, `frac`

floor

Purpose

Round symbolic matrix toward negative infinity

Syntax

`floor(X)`

Description

`floor(X)` is the matrix of the greatest integers less than or equal to `X`.

Example

```
x = sym(-5/2)
[fix(x) floor(x) round(x) ceil(x) frac(x)]
= [ -2, -3, -3, -2, -1/2]
```

See Also

`round`, `ceil`, `fix`, `frac`

Purpose

Fortran representation of symbolic expression

Syntax

```
fortran(S)
```

Description

`fortran(S)` returns the Fortran code equivalent to the expression `S`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
fortran(f)
```

return

```
t0 = x-x**2/2+x**3/3-x**4/4+x**5/5
```

The statements

```
H = sym(hilb(3));
fortran(H)
```

return

```
H(1,1) = 1           H(1,2) = 1.E0/2.E0    H(1,3) = 1.E0/3.E0
H(2,1) = 1.E0/2.E0   H(2,2) = 1.E0/3.E0    H(2,3) = 1.E0/4.E0
H(3,1) = 1.E0/3.E0   H(3,2) = 1.E0/4.E0    H(3,3) = 1.E0/5.E0
```

See Also

`ccode`, `latex`, `pretty`

fourier

Purpose

Fourier integral transform

Syntax

```
F = fourier(f)
F = fourier(f,v)
F = fourier(f,u,v)
```

Description

`F = fourier(f)` is the Fourier transform of the symbolic scalar `f` with default independent variable `x`. The default return is a function of `w`. The Fourier transform is applied to a function of `x` and returns a function of `w`.

$$f = f(x) \Rightarrow F = F(w)$$

If `f = f(w)`, `fourier` returns a function of `t`.

$$F = F(t)$$

By definition

$$F(w) = \int_{-\infty}^{\infty} f(x)e^{-iwx}dx$$

where `x` is the symbolic variable in `f` as determined by `findsym`.

`F = fourier(f,v)` makes `F` a function of the symbol `v` instead of the default `w`.

$$F(v) = \int_{-\infty}^{\infty} f(x)e^{-ivx}dx$$

`F = fourier(f,u,v)` makes `f` a function of `u` and `F` a function of `v` instead of the default variables `x and w`, respectively.

$$F(v) = \int_{-\infty}^{\infty} f(u)e^{-ivu}du$$

Examples

Fourier Transform	MATLAB Command

$f(x) = e^{-x^2}$
$F[f](w) = \int_{-\infty}^{\infty} f(x)e^{-ixw}dx$
$= \sqrt{\pi}e^{-w^2/4}$
	`f = exp(-x^2)`
	`fourier(f)`
	returns
	`pi^(1/2)*exp(-1/4*w^2)`

$g(w) = e^{-\|w\|}$
$F[g](t) = \int_{-\infty}^{\infty} g(w)e^{-itw}dw$
$= \frac{2}{1+t^2}$
	`g = exp(-abs(w))`
	`fourier(g)`
	returns
	`2/(1+t^2)`

Fourier Transform	MATLAB Command

$f(x) = xe^{-\|x\|}$
 $F[f](u) = \int_{-\infty}^{\infty} f(x)e^{-ixu}dx$
 $= -\frac{4i}{(1+u^2)^{2u}}$
	`f = x*exp(-abs(x))`
	 `fourier(f,u)`
	 returns
	 `-4*i/(1+u^2)^2*u`

$f(x,v) = e^{-x^2\frac{\|v\|\sin v}{v}, x\, real}$
 $F[f(v)](u) = \int_{-\infty}^{\infty} f(x,v)e^{-ivu}dv$
 $= -a\tan\frac{u-1}{x^2} = a\tan\frac{u+1}{x^2}$
	`syms x real`
	 `f = exp(-x^2*abs(v))*sin(v)/v`
	 `fourier(f,v,u)`
	 returns
	 `-atan((u-1)/x^2)+atan((u+1)/x^2)`

See Also `ifourier`, `laplace`, `ztrans`

Purpose

Symbolic matrix elementwise fractional parts

Syntax

`frac(X)`

Description

`frac(X)` is the matrix of the fractional parts of the elements of `X`.

```
frac(X) = X - fix(X)
```

Example

```
x = sym(-5/2)
[fix(x) floor(x) round(x) ceil(x) frac(x)]
= [ -2, -3, -3, -2, -1/2]
```

See Also

`round`, `ceil`, `floor`, `fix`

funtool

Purpose

Function calculator

Syntax

```
funtool
```

Description

`funtool` is a visual function calculator that manipulates and displays functions of one variable. At the click of a button, for example, `funtool` draws a graph representing the sum, product, difference, or ratio of two functions that you specify. `funtool` includes a function memory that allows you to store functions for later retrieval.

At startup, `funtool` displays graphs of a pair of functions, `f(x) = x` and `g(x) = 1`. The graphs plot the functions over the domain `[-2*pi, 2*pi]`. `funtool` also displays a control panel that lets you save, retrieve, redefine, combine, and transform `f` and `g`.

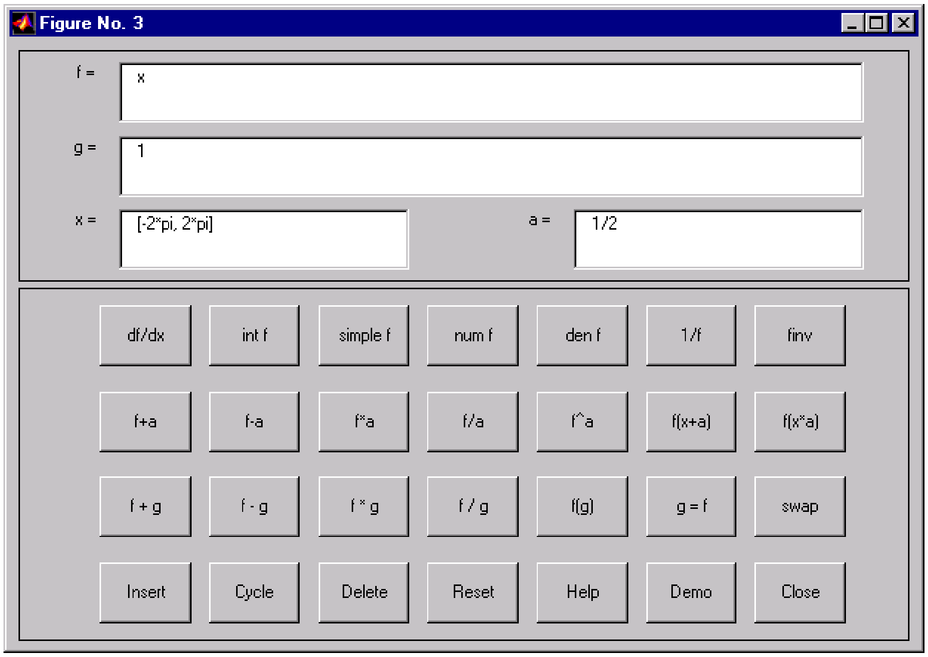


Figure No. 3

f = x

g = 1

x = [-2*pi, 2*pi]

a = 1/2

Text Fields

The top of the control panel contains a group of editable text fields.

f=	Displays a symbolic expression representing `f`. Edit this field to redefine `f`.
g=	Displays a symbolic expression representing `g`. Edit this field to redefine `g`.
x=	Displays the domain used to plot `f` and `g`. Edit this field to specify a different domain.
a=	Displays a constant factor used to modify `f` (see button descriptions in the next section). Edit this field to change the value of the constant factor.

`funtool` redraws `f` and `g` to reflect any changes you make to the contents of the control panel's text fields.

Control Buttons

The bottom part of the control panel contains an array of buttons that transform `f` and perform other operations.

The first row of control buttons replaces `f` with various transformations of `f`.

df/dx	Derivative of `f`
int f	Integral of `f`
simple f	Simplified form of `f`, if possible
num f	Numerator of `f`
den f	Denominator of `f`
1/f	Reciprocal of `f`
finv	Inverse of `f`

The operators **intf** and **finv** may fail if the corresponding symbolic expressions do not exist in closed form.

The second row of buttons translates and scales f and the domain of f by a constant factor. To specify the factor, enter its value in the field labeled **a=** on the calculator control panel. The operations are

f+a	Replaces `f(x)` by `f(x) + a`.
f-a	Replaces `f(x)` by `f(x) - a`.
f*a	Replaces `f(x)` by `f(x) * a`.
f/a	Replaces `f(x)` by `f(x) / a`.
f^a	Replaces `f(x)` by `f(x) ^ a`.
f(x+a)	Replaces `f(x)` by `f(x + a)`.
f(x*a)	Replaces `f(x)` by `f(x * a)`.

The first four buttons of the third row replace f with a combination of f and g.

f+g	Replaces `f(x)` by `f(x) + g(x)`.
f-g	Replaces `f(x)` by `f(x)-g(x)`.
f*g	Replaces `f(x)` by `f(x) * g(x)`.
f/g	Replaces `f(x)` by `f(x) / g(x)`.

The remaining buttons on the third row interchange f and g.

g=f	Replaces g with f.
swap	Replaces f with g and g with f.

The first three buttons in the fourth row allow you to store and retrieve functions from the calculator's function memory.

Insert	Adds f to the end of the list of stored functions.
Cycle	Replaces f with the next item on the function list.
Delete	Deletes f from the list of stored functions.

The other four buttons on the fourth row perform miscellaneous functions:

Reset	Resets the calculator to its initial state.
Help	Displays the online help for the calculator.
Demo	Runs a short demo of the calculator.
Close	Closes the calculator's windows.

See Also

`ezplot`, `syms`

heaviside

Purpose Compute Heaviside step function

Syntax `heaviside(x)`

Description `heaviside(x)` has the value `0` for `x < 0`, `1` for `x > 0`, and `NaN` for `x == 0`. `heaviside` is not a function in the strict sense.

See Also `dirac`

Purpose

Horner nested polynomial representation

Syntax

```
horner(P)
```

Description

Suppose `P` is a matrix of symbolic polynomials. `horner(P)` transforms each element of `P` into its Horner, or nested, representation.

Examples

`horner(x^3-6*x^2+11*x-6)` returns

```
-6+(11+(-6+x)*x)*x
```

`horner([x^2+x;y^3-2*y])` returns

```
[    (1+x)*x]
[(-2+y^2)*y]
```

See Also

`expand`, `factor`, `simple`, `simplify`, `syms`

Purpose

Generalized hypergeometric

Syntax

```
hypergeom(n,d,z)
```

Description

`hypergeom(n,d,z)` is the generalized hypergeometric function $F(n, d, z)$, also known as the Barnes extended hypergeometric function and denoted by ${}_jF_k$ where `j = length(n)` and `k = length(d)`. For scalar `a`, `b`, and `c`, `hypergeom([a,b],c,z)` is the Gauss hypergeometric function ${}_2F_1(a,b;c;z)$.

The definition by a formal power series is

$$F(n,d,z) = \sum_{k=0}^{\infty} \frac{C_{n,k}}{C_{d,k}} \cdot \frac{z^k}{k!}$$

where

$$C_{v,k} = \prod_{j=1}^{v} \frac{\Gamma(v_j + k)}{\Gamma(v_j)}$$

Either of the first two arguments may be a vector providing the coefficient parameters for a single function evaluation. If the third argument is a vector, the function is evaluated pointwise. The result is numeric if all the arguments are numeric and symbolic if any of the arguments is symbolic.

See Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 15.

Examples

```
syms a z
```

`hypergeom([],[],z)` returns `exp(z)`

`hypergeom(1,[],z)` returns `-1/(-1+z)`

`hypergeom(1,2,'z')` returns `(exp(z)-1)/z`

`hypergeom([1,2],[2,3],'z')` returns `2*(exp(z)-1-z)/z^2` `hypergeom(a,[],z)` returns `(1-z)^(-a)`

`hypergeom([],1,-z^2/4)` returns `besselj(0,z)`

Purpose

Inverse Fourier integral transform

Syntax

```
f = ifourier(F)
f = ifourier(F,u)
f = ifourier(F,v,u)
```

Description

`f = ifourier(F)` is the inverse Fourier transform of the scalar symbolic object `F` with default independent variable `w`. The default return is a function of `x`. The inverse Fourier transform is applied to a function of `w` and returns a function of `x`.

$$F = F(w) \Rightarrow f = f(x)$$

If `F = F(x)`, `ifourier` returns a function of `t`.

$$f = f(t)$$

By definition

$$f(x) = 1/(2\pi)\int_{-\infty}^{\infty} F(w)e^{iwx}dw$$

`f = ifourier(F,u)` makes `f` a function of `u` instead of the default `x`.

$$f(u) = 1/(2\pi)\int_{-\infty}^{\infty} F(w)e^{iwu}dw$$

Here `u` is a scalar symbolic object.

`f = ifourier(F,v,u)` takes `F` to be a function of `v` and `f` to be a function of `u` instead of the default `w` and `x`, respectively.

$$f(u) = 1/(2\pi)\int_{-\infty}^{\infty} F(v)e^{ivu}dv$$

Examples

Inverse Fourier Transform	MATLAB Command

$f(w) = e^{-w2/(4a^2)}$
 	
$F^{-1}[f](x) = \int_{-\infty}^{\infty} f(w)e^{ixw}dw$
 	
$= \frac{a}{\sqrt{\pi}} e^{-(ax)^2}$
	`syms a w real`
	`f = exp(-w^2/(4*a^2))`
	`F = ifourier(f)`
	`F = simple(F)`
	returns
	`a*exp(-x^2*a^2)/pi^(1/2)`

$g(x) = e^{-\|x\|}$
 	
$F^{-1}[g](t) = \int_{-\infty}^{\infty} g(x)e^{itx}dx$
 	
$= \frac{\pi}{1+t^2}$
	`syms x real`
	`g = exp(-abs(x))`
	`ifourier(g)`
	returns
	`1/(1+t^2)/pi`

Inverse Fourier Transform	MATLAB Command

$f(w) = 2e^{-\|w\|} - 1$
$F^{-1}[f](t) = \int_{-\infty}^{\infty} f(w)e^{itw}dw$
$= diract(t) + \frac{2}{\pi(1+t^2)}$
	`syms w t real`
	`f = 2*exp(-abs(w)) - 1`
	`simple(ifourier(f,t))`
	returns
	`-dirac(t)+2/pi/(1+t^2)`

$f(w,v) = e^{-w^2\|v\|}\frac{\sin v}{v}, w\,\text{real}$
$F^{-1}[f(v)](t) = \int_{-\infty}^{\infty} f(w,v)e^{ivt}dv$
$= \frac{1}{2\pi}\left(a\tan\frac{t+1}{w^2} - a\tan\frac{t-1}{w^2}\right)$
	`syms w v t real`
	`f = exp(-w^2*abs(v))*sin(v)/v`
	`ifourier(f,v,t)`
	returns
	`-1/2*(-atan((t+1)/w^2)`
	`+atan((-1+t)/w^2))/pi`

See Also

`fourier`, `ilaplace`, `iztrans`

ilaplace

Purpose Inverse Laplace transform

Syntax

```
F = ilaplace(L)
F = ilaplace(L,y)
F = ilaplace(L,y,x)
```

Description

`F = ilaplace(L)` is the inverse Laplace transform of the scalar symbolic object `L` with default independent variable `s`. The default return is a function of `t`. The inverse Laplace transform is applied to a function of `s` and returns a function of `t`.

$$L = L(s) \Rightarrow F = F(t)$$

If `L = L(t)`, `ilaplace` returns a function of `x`.

$$F = F(x)$$

By definition

$$F(t) = \int_{c-i\infty}^{c+i\infty} L(s)e^{st}\,ds$$

where `c` is a real number selected so that all singularities of `L(s)` are to the left of the line `s = c, i`.

`F = ilaplace(L,y)` makes `F` a function of `y` instead of the default `t`.

$$F(y) = \int_{c-i\infty}^{c+i\infty} L(y)e^{sy}\,ds$$

Here `y` is a scalar symbolic object.

`F = ilaplace(L,y,x)` takes `F` to be a function of `x` and `L` a function of `y` instead of the default variables `t` and `s`, respectively.

$$F(x) = \int_{c-i\infty}^{c+i\infty} L(y)e^{xy}dy$$

Examples

Inverse Laplace Transform	MATLAB Command

$f(s) = \frac{1}{s^2}$
$L^{-1}[f] = \frac{1}{2\pi i}\int_{c-ivo}^{c+ivo} f(s)e^{st}ds$
$= t$
	`f = 1/s^2`
	`ilaplace(f)`
	returns
	`t`

$g(t) = \frac{1}{(t-a)^2}$
$L^{-1}[g] = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} g(t)e^{xt}dt$
$= xe^{ax}$
	`g = 1/(t-a)^2`
	`ilaplace(g)`
	returns
	`x*exp(a*x)`

$f(u) = \frac{1}{u^2 - a^2}$
$L^{-1}[f] = \frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} g(u)e^{xu}du$
$= \frac{\sinh(x\|a\|)}{\|a\|}$
	`syms x u`
	`syms a real`
	`f = 1/(u^2-a^2)`
	`simplify(ilaplace(f,x))`
	returns
	`sinh(x*abs(a))/abs(a)`

See Also

`ifourier`, `iztrans`, `laplace`

imag

Purpose Imaginary part of complex number

Syntax `imag(Z)`

Description `imag(Z)` is the imaginary part of a symbolic `Z`.

See Also `conj`, `real`

Purpose

Integrate

Syntax

```
int(S)
int(S,v)
int(S,a,b)
int(S,v,a,b)
```

Description

`int(S)` returns the indefinite integral of `S` with respect to its symbolic variable as defined by `findsym`.

`int(S,v)` returns the indefinite integral of `S` with respect to the symbolic scalar variable `v`.

`int(S,a,b)` returns the definite integral from `a` to `b` of each element of `S` with respect to each element's default symbolic variable. `a` and `b` are symbolic or double scalars.

`int(S,v,a,b)` returns the definite integral of `S` with respect to `v` from `a` to `b`.

Examples

`int(-2*x/(1+x^2)^2)` returns

```
1/(1+x^2)
```

`int(x/(1+z^2),z)` returns

```
x*atan(z)
```

`int(x*log(1+x),0,1)` returns

```
1/4
```

`int(2*x, sin(t), 1)` returns

```
1-sin(t)^2
```

`int([exp(t),exp(alpha*t)])` returns

```
[exp(t), 1/alpha*exp(alpha*t)]
```

See Also `diff`, `symsum`

int8, int16, int32, int64

Purpose

Convert symbolic matrix to signed integers

Syntax

```
int8(S)
int16(S)
int32(S)
int64(S)
```

Description

`int8(S)` converts a symbolic matrix `S` to a matrix of signed 8-bit integers.

`int16(S)` converts `S` to a matrix of signed 16-bit integers.

`int32(S)` converts `S` to a matrix of signed 32-bit integers.

`int64(S)` converts `S` to a matrix of signed 64-bit integers.

Note The output of `int8`, `int16`, `int32`, and `int64` does not have data type `symbolic`.

The following table summarizes the output of these four functions.

Function	Output Range	Output Type	Bytes per Element	Output Class
`int8`	-128 to 127	Signed 8-bit integer	1	`int8`
`int16`	-32,768 to 32,767	Signed 16-bit integer	2	`int16`
`int32`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	`int32`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

See Also

`sym`, `vpa`, `single`, `double`, `uint8`, `uint16`, `uint32`, `uint64`

Purpose

Symbolic matrix inverse

Syntax

```
R = inv(A)
```

Description

`R = inv(A)` returns inverse of the symbolic matrix `A`.

Examples

The statements

```
A = sym([2,-1,0;-1,2,-1;0,-1,2]);
inv(A)
```

return

```
[ 3/4, 1/2, 1/4]
[ 1/2,   1, 1/2]
[ 1/4, 1/2, 3/4]
```

The statements

```
syms a b c d
A = [a b; c d]
inv(A)
```

return

```
[  d/(a*d-b*c), -b/(a*d-b*c)]
[ -c/(a*d-b*c),  a/(a*d-b*c)]
```

Suppose you have created the following M-file.

```
%% Generate a symbolic N-by-N Hilbert matrix.
function A = genhilb(N)
syms t;
for i = 1:N
        for j = 1:N
        A(i,j) = 1/(i + j - t);
        end
end
```

Then, the following statement

```
inv(genhilb(2))
```

returns

```
[     -(-3+t)^2*(-2+t), (-3+t)*(-2+t)*(-4+t)]
[(-3+t)*(-2+t)*(-4+t),      -(-3+t)^2*(-4+t)]
```

the symbolic inverse of the 2-by-2 Hilbert matrix.

See Also

`vpa`, `Arithmetic Operations`

Purpose

Inverse *z*-transform

Syntax

```
f = iztrans(F)
f = iztrans(F,k)
f = iztrans(F,w,k)
```

Description

`f = iztrans(F)` is the inverse *z*-transform of the scalar symbolic object `F` with default independent variable `z`. The default return is a function of `n`.

$$f(n) = \frac{1}{2\pi i} \oint_{|z|=R} F(z) z^{n-1} dz, n = 1,2..$$

where R is a positive number chosen so that the function $F(z)$ is analytic on and outside the circle $|z| = R$.

If `F = F(n)`, `iztrans` returns a function of `k`.

$$f = f(k)$$

`f = iztrans(F,k)` makes `f` a function of `k` instead of the default `n`. Here `k` is a scalar symbolic object.

`f = iztrans(F,w,k)` takes `F` to be a function of `w` instead of the default `findsym(F)` and returns a function of `k`.

$$F = F(w) \Rightarrow f = f(k)$$

Examples

Inverse Z-Transform	MATLAB Operation

$f(z) = \frac{2z}{(z-2)^2}$
 $Z^{-1}[f] = \frac{1}{2\pi i} \oint_{\lvert z \rvert = R} f(s) z^{n-1} dz$
 $= n2^n$
	`f = 2*z/(z-2)^2`
	 `iztrans(f)`
	 returns
	 `2^n*n`

$g(n) = \frac{n(n+1)}{n^2+2n+1}$
 $Z^{-1}\lvert g \rvert = \frac{1}{2\pi i} \oint_{\lvert n \rvert = R} g(n) n^{k-1} dn$
 $= -1^k$
	`g = n*(n+1)/(n^2+2*n+1)`
	 `iztrans(g)`
	 returns
	 `(-1)^k`

$f(z) = \frac{z}{z-a}$
 $Z^{-1}[f] - \frac{1}{2\pi i} \oint_{\lvert z \rvert = R} f(z) z^{k-1} dz$
 $= a^k$
	`f = z/(z-a)`
	 `iztrans(f,k)`
	 returns
	 `a^k`

See Also

`ifourier`, `ilaplace`, `ztrans`

Purpose

Jacobian matrix

Syntax

```
jacobian(f,v)
```

Description

`jacobian(f,v)` computes the Jacobian of the scalar or vector `f` with respect to `v`. The `(i,j)`-th entry of the result is $\partial w(i) / \partial v(j)$. Note that when `f` is scalar, the Jacobian of `f` is the gradient of `f`. Also, note that `v` can be a scalar, although in that case the result is the same as `diff(f,v)`.

Examples

The statements

```
f = [x*y*z; y; x+z];
v = [x,y,z];
R = jacobian(f,v)
b = jacobian(x+z, v)
```

return

```
R =
[y*z, x*z, x*y]
[  0,   1,   0]
[  1,   0,   1]

b =
[1, 0, 1]
```

See Also

`diff`

jordan

Purpose Jordan canonical form

Syntax

```
J = jordan(A)
[V,J] = jordan(A)
```

Description `J = jordan(A)` computes the Jordan canonical (normal) form of `A`, where `A` is a symbolic or numeric matrix. The matrix must be known exactly. Thus, its elements must be integers or ratios of small integers. Any errors in the input matrix may completely change the Jordan canonical form.

`[V,J] = jordan(A)` computes both `J`, the Jordan canonical form, and the similarity transform, `V`, whose columns are the generalized eigenvectors. Moreover, `V\A*V=J`.

Examples The statements

```
A = [1 -3 -2; -1  1 -1; 2 4 5]
[V,J] = jordan(A)
```

return

```
A =
      1     -3     -2
     -1      1     -1
      2      4      5
V =
     -1     -1      1
      0     -1      0
      1      2      0
J =
      3      0      0
      0      2      1
      0      0      2
```

Then the statement

```
V\A*V
```

returns

```
ans =
     3     0     0
     0     2     1
     0     0     2
```

See Also `eig`, `poly`

lambertw

Purpose Lambert's *W* function

Syntax

```
W = lambertw(X)
W = lambertw(K,X)
```

Description `W = lambertw(X)` evaluates Lambert's *W* function at the elements of `X`, a numeric matrix or a symbolic matrix. Lambert's *W* solves the equation

$$we^w = x$$

for `w` as a function of `x`.

`W = lambertw(K,X)` is the K-th branch of this multi-valued function.

Examples `lambertw([0 -exp(-1); pi 1])` returns

```
          0   -1.0000
     1.0737    0.5671
```

The statements

```
syms x y
lambertw([0 x;1 y])
```

return

```
[           0, lambertw(x)]
[ lambertw(1), lambertw(y)]
```

References

[1] Corless, R.M, G.H. Gonnet, D.E.G. Hare, and D.J. Jeffrey, *Lambert's W Function in Maple*, Technical Report, Dept. of Applied Math., Univ. of Western Ontario, London, Ontario, Canada.

[2] Corless, R.M, Gonnet, G.H. Gonnet, D.E.G. Hare, and D.J. Jeffrey, *On Lambert's W Function*, Technical Report, Dept. of Applied Math., Univ. of Western Ontario, London, Ontario, Canada.

Both papers are available by anonymous FTP from `cs-archive.uwaterloo.ca`

laplace

Purpose Laplace transform

Syntax

```
laplace(F)
laplace(F,t)
laplace(F,w,z)
```

Description `L = laplace(F)` is the Laplace transform of the scalar symbol `F` with default independent variable `t`. The default return is a function of `s`. The Laplace transform is applied to a function of `t` and returns a function of `s`.

$$F = F(t) \Rightarrow L = L(s)$$

If `F = F(s)`, `laplace` returns a function of `t`.

$$L = L(t)$$

By definition

$$L(s) = \int_0^\infty F(t)e^{-st}dt$$

where `t` is the symbolic variable in `F` as determined by `findsym`.

`L = laplace(F,t)` makes `L` a function of `t` instead of the default `s`.

$$L(t) = \int_0^\infty F(x)e^{-tx}dx$$

Here `L` is returned as a scalar symbol.

`L = laplace(F,w,z)` makes `L` a function of `z` and `F` a function of `w` instead of the default variables `s` and `t`, respectively.

$$L(z) = \int_0^\infty F(w)e^{-zw}dw$$

Examples

Laplace Transform	MATLAB Command

$f(t) = t^4$
 $L[f] = \int_0^\infty f(t)e^{-ts}dt$
 $= \frac{24}{s^5}$
	`f = t^4`
	 `laplace(f)`
	 returns
	 `24/s^5`

$g(s) = \frac{1}{\sqrt{s}}$
 $L[g](t) = \int_0^\infty g(s)e^{-st}ds$
 $= \sqrt{\frac{\pi}{t}}$
	`g = 1/sqrt(s)`
	 `laplace(g)`
	 returns
	 `(pi/t)^(1/2)`

$f(t) = e^{-at}$
 $L[f](x) = \int_0^\infty f(t)e^{-tx}dt$
 $= \frac{1}{x+a}$
	`f = exp(-a*t)`
	 `laplace(f,x)`
	 returns
	 `1/(x + a)`

See Also

`fourier`, `ilaplace`, `ztrans`

latex

Purpose

LaTeX representation of symbolic expression

Syntax

```
latex(S)
```

Description

`latex(S)` returns the LaTeX representation of the symbolic expression `S`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
latex(f)
```

return

```
x-1/2\,{x}^{2}+1/3\,{x}^{3}-1/4\,{x}^{4}+1/5\,{x}^{5}
```

The statements

```
H = sym(hilb(3));
latex(H)
```

return

```
\left [\begin {array}{ccc} 1&1/2&1/3\\\noalign{\medskip}1/2&1/
3&1/4
\\\noalign{\medskip}1/3&1/4&1/5\end {array}\right ]
```

The statements

```
syms alpha t
A = [alpha t alpha*t];
latex(A)
```

return

```
\left [\begin {array}{ccc} \alpha&t&\alpha\,t\end {array}\right ]
```

See Also

`pretty`, `ccode`, `fortran`

Purpose

Limit of symbolic expression

Syntax

```
limit(F,x,a)
limit(F,x,a)
limit(F,a)
limit(F)
limit(F,x,a,'right')
limit(F,x,a,'left')
```

Description

`limit(F,x,a)` takes the limit of the symbolic expression `F` as `x -> a`.

`limit(F,a)` uses `findsym(F)` as the independent variable.

`limit(F)` uses `a = 0` as the limit point.

`limit(F,x,a,'right')` or `limit(F,x,a,'left')` specify the direction of a one-sided limit.

Examples

Assume

```
syms x a t h;
```

Then

```
limit(sin(x)/x)                 => 1
limit(1/x,x,0,'right')          => inf
limit(1/x,x,0,'left')           => -inf
limit((sin(x+h)-sin(x))/h,h,0)  => cos(x)
v = [(1 + a/x)^x, exp(-x)];
limit(v,x,inf,'left')           => [exp(a),  0]
```

See Also

`pretty`, `ccode`, `fortran`

log10

Purpose

Logarithm base 10 of entries of symbolic matrix

Syntax

```
Y = log10(X)
```

Description

`Y = log10(X)` returns the logarithm to the base `10` of `X`. If `X` is a matrix, `Y` is a matrix of the same size, each entry of which is the logarithm of the corresponding entry of `X`.

See Also

`log2`

Purpose

Logarithm base 2 of entries of symbolic matrix

Syntax

```
Y = log2(X)
```

Description

`Y = log2(X)` returns the logarithm to the base `2` of `X`. If `X` is a matrix, `Y` is a matrix of the same size, each entry of which is the logarithm of the corresponding entry of `X`.

See Also

`log10`

maple

Purpose

Access Maple kernel

Syntax

```
r = maple('statement')
maple('function',arg1,arg2,...)
[r,status] = maple(...)
maple('traceon')
maple trace on
maple('traceoff')
maple trace off
```

Description

`r = maple('statement')` sends `statement` to the Maple kernel and returns the result. A semicolon for the Maple syntax is appended to `statement` if necessary.

`maple('function',arg1,arg2,...)` accepts the quoted name of any Maple function and associated input arguments. The arguments are converted to symbolic expressions if necessary, and `function` is then called with the given arguments. If the input arguments are `sym`s, then `maple` returns a `sym`. Otherwise, it returns a result of class `char`.

`[r,status] = maple(...)` is an option that returns the warning/error status. When the statement execution is successful, `r` is the result and `status` is `0`. If the execution fails, `r` is the corresponding warning/error message, and `status` is a positive integer.

`maple('traceon')` (or `maple trace on`) causes all subsequent Maple statements and results to be printed. `maple('traceoff')` (or `maple trace off`) turns this feature off.

Examples

Each of the following statements evaluate π to 100 digits.

```
maple('evalf(Pi,100)')
maple evalf Pi 100
maple('evalf','Pi',100)
```

The statement

```
[result,status] = maple('BesselK',4.3)
```

returns the following output because Maple's `BesselK` function needs two input arguments.

```
result =
Error, (in BesselK) expecting 2 arguments, got 1
status =
2
```

The `traceon` command shows how Symbolic Math Toolbox commands interact with Maple. For example, the statements

```
syms x
v = [x^2-1;x^2-4]
maple traceon % or maple trace on
w = factor(v)
```

return

```
v =
[ x^2-1]
[ x^2-4]

statement:
   map(ifactor,array([[x^2-1],[x^2-4]]));
result:
   Error, (in ifactor) invalid arguments
statement:
   map(factor,array([[x^2-1],[x^2-4]]));
result:
   matrix([[(x-1)*(x+1)], [(x-2)*(x+2)]])

w =

[ (x-1)*(x+1)]
[ (x-2)*(x+2)]
```

This example reveals that the `factor` statement first invokes Maple's integer factor (`ifactor`) statement to determine whether the argument is a factorable integer. If Maple's integer factor statement returns an error, the Symbolic Math Toolbox `factor` statement then invokes Maple's expression factoring statement.

See Also `mhelp`, `procread`

Purpose Initialize Maple kernel

Syntax

```
mapleinit
```

Description `mapleinit` determines the path to the directory containing the Maple Library, loads the Maple linear algebra and integral transform packages, initializes `digits`, and establishes several aliases. `mapleinit` is called by the MEX-file interface to Maple.

You can edit the `mapleinit` M-file to change the pathname to the Maple library. You do this by changing the `initstring` variable in `mapleinit.m` to the full pathname of the Maple library, as described below.

UNIX

Suppose you already have a copy of the Library for Maple in the UNIX directory `/usr/local/Maple/lib`. You can edit `mapleinit.m` to contain

```
maplelib = '/usr/local/Maple/lib'
```

and then delete the copy of the Maple Library that is distributed with MATLAB.

Microsoft-Windows

Suppose you already have a copy of the Library for Maple in the directory `C:\MAPLE\LIB`. You can edit `mapleinit.m` to contain

```
maplelib = 'C:\MAPLE\LIB'
```

and then delete the copy of the Maple Library that is distributed with Symbolic Math Toolboxes.

Purpose

Numeric evaluation of Maple function

Syntax

`mfun('function',par1,par2,par3,*par4*)`

Description

`mfun('function',par1,par2,par3,par4)` numerically evaluates one of the special mathematical functions known to Maple. Each `par` argument is a numeric quantity corresponding to a Maple parameter for `function`. You can use up to four parameters. The last parameter specified can be a matrix, usually corresponding to `X`. The dimensions of all other parameters depend on the Maple specifications for `function`. You can access parameter information for Maple functions using one of the following commands:

```
help mfunlist
mhelp function
```

Maple evaluates `function` using 16 digit accuracy. Each element of the result is a MATLAB numeric quantity. Any singularity in `function` is returned as `NaN`.

Examples

`mfun('FresnelC',0:5)` returns

```
0    0.7799    0.4883    0.6057    0.4984    0.5636
```

`mfun('Chi',[3*i 0])` returns

```
0.1196 + 1.5708i   NaN
```

See Also

`mfunlist`, `mhelp`

Purpose List special functions for use with `mfun`

Syntax

```
mfunlist
```

Description `mfunlist` lists the special mathematical functions for use with the `mfun` function. The following tables describe these special functions.

You can access more detailed descriptions by typing

```
mhelp function
```

Limitations In general, the accuracy of a function will be lower near its roots and when its arguments are relatively large.

Run-time depends on the specific function and its parameters. In general, calculations are slower than standard MATLAB calculations.

See Also `mfun`, `mhelp`

References [1] Abramowitz, M. and I.A., Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965.

Table Conventions The following conventions are used in the following table, unless otherwise indicated in the **Arguments** column.

`x`, `y`	real argument
`z`, `z1`, `z2`	complex argument
`m`, `n`	integer argument

MFUN Special Functions

Function Name	Definition	mfun Name	Arguments
Bernoulli Numbers and Polynomials	Generating functions: $\frac{e^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \cdot \frac{t^{n-1}}{n!}$	`bernoulli(n)` `bernoulli(n,t)`	$n \geq 0$ $0 < \lvert t \rvert < 2\pi$
Bessel Functions	`BesselI`, `BesselJ`—Bessel functions of the first kind. `BesselK`, `BesselY`—Bessel functions of the second kind.	`BesselJ(v,x)` `BesselY(v,x)` `BesselI(v,x)` `BesselK(v,x)`	`v` is real.
Beta Function	$B(x, y) = \frac{\Gamma(x) \cdot \Gamma(y)}{\Gamma(x+y)}$	`Beta(x,y)`	
Binomial Coefficients	$\binom{m}{n} = \frac{m!}{n!(m-n)!}$ $= \frac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$	`binomial(m,n)`	
Complete Elliptic Integrals	Legendre's complete elliptic integrals of the first, second, and third kind.	`EllipticK(k)` `EllipticE(k)` `EllipticPi(a,k)`	`a` is real $-\infty < a < \infty$ `k` is real $0 < k < 1$
Complete Elliptic Integrals with Complementary Modulus	Associated complete elliptic integrals of the first, second, and third kind using complementary modulus.	`EllipticCK(k)` `EllipticCE(k)` `EllipticCPi(a,k)`	`a` is real $-\infty < a < \infty$ `k` is real $0 < k < 1$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments

Complementary Error Function and Its Iterated Integrals	$erfc(z) = \frac{2}{\sqrt{\pi}} \cdot \int_z^\infty e^{-t^2} dt = 1 - erf(z)$
	$erfc(-1, z) = \frac{2}{\sqrt{\pi}} \cdot e^{-z^2}$
	$erfc(n, z) = \int_z^\infty erfc(n-1, z) dt$
		`erfc(z)`
		`erfc(n,z)`
			$n > 0$

Dawson's Integral	$F(x) = e^{-x^2} \cdot \int_0^x e^{-t^2} dt$	`dawson(x)`	
Digamma Function	$\Psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	
Dilogarithm Integral	$f(x) = \int_1^x \frac{\ln(t)}{1-t} dt$	`dilog(x)`	$x > 1$
Error Function	$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	`erf(z)`	

Euler Numbers and Polynomials	Generating function for Euler numbers:
	$\frac{1}{ch(t)} = \sum_{n=0}^{\infty} E_n \frac{t^n}{n!}$
		`euler(n)`
		`euler(n,z)`
			$n \geq 0$
			$\|t\| < \frac{\pi}{2}$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments

Exponential Integrals	$Ei(n,z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt$
	 $Ei(x) = PV - \int_{-\infty}^{x} \frac{e^t}{t}$
		`Ei(n,z)`
		 `Ei(x)`
			$n \geq 0$
			 $Real(z) > 0$

Fresnel Sine and Cosine Integrals	$C(x) = \int_0^x \cos\left(\frac{\pi}{2} \cdot t^2\right) dt$
	 $S(x) = \int_0^x \sin\left(\frac{\pi}{2} \cdot t^2\right) dt$
		`FresnelC(x)`
		 `FresnelS(x)`	

Gamma Function	$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$	`GAMMA(z)`	
Harmonic Function	$h(n) = \sum_{k=1}^{n} \frac{1}{k} = \Psi(n+1) + \gamma$	`harmonic(n)`	$n > 0$

Hyperbolic Sine and Cosine Integrals	$Shi(z) = \int_0^z \frac{\sinh(t)}{t} dt$
	 $Chi(z) = \gamma + \ln(z) + \int_0^z \frac{\cosh(t) - 1}{t} dt$
		`Shi(z)`
		 `Chi(z)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
(Generalized) Hypergeometric Function	$F(n,d,z) = \sum_{k=0}^{\infty} \frac{\prod_{i=1}^{j} \frac{\Gamma(n_i+k)}{\Gamma(n_i)} \cdot z^k}{\prod_{i=1}^{m} \frac{\Gamma(d_1+k)}{\Gamma(d_i)} \cdot k!}$ where `j` and `m` are the number of terms in `n` and `d`, respectively.	`hypergeom(n,d,x)` where `n = [n1,n2,...]` `d = [d1,d2,...]`	`n1,n2,...` are real. `d1,d2,...` are real and nonnegative.
Incomplete Elliptic Integrals	Legendre's incomplete elliptic integrals of the first, second, and third kind.	`EllipticF(x,k)` `EllipticE(x,k)` `EllipticPi(x,a,k)`	$0 < x \le \infty$ `a` is real $-\infty < a < \infty$ `k` is real $0 < k < 1$
Incomplete Gamma Function	$\Gamma(a,z) = \int_z^{\infty} e^{-t} \cdot t^{a-1} dt$	`GAMMA(z1,z2)`	
Logarithm of the Gamma Function	$\ln\Gamma(z) = \ln(\Gamma(z))$	`lnGAMMA(z)`	
Logarithmic Integral	$Li(x) = PV\left\{\int_0^x \frac{dt}{\ln t}\right\} = Ei(\ln x)$	`Li(x)`	$x > 1$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Polygamma Function	$\Psi^{(n)}(z) = \frac{d^n}{dz}\Psi(z)$ where $\Psi(z)$ is the Digamma function.	`Psi(n,z)`	$n \geq 0$
Shifted Sine Integral	$Ssi(z) = Si(z) - \frac{\pi}{2}$	`Ssi(z)`	

Orthogonal Polynomials

The following functions require the Maple Orthogonal Polynomial Package. They are available only with Extended Symbolic Math Toolbox. Before using these functions, you must first initialize the Orthogonal Polynomial Package by typing

```
maple('with','orthopoly')
```

Note that in all cases, `n` is a non-negative integer and `x` is real.

Orthogonal Polynomials

Polynomial	Maple Name	Arguments
Gegenbauer	`G(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1/2`.
Hermite	`H(n,x)`	
Laguerre	`L(n,x)`	

Orthogonal Polynomials (Continued)

Polynomial	Maple Name	Arguments
Generalized Laguerre	`L(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	
Jacobi	`P(n,a,b,x)`	`a`, `b` are nonrational algebraic expressions or rational numbers greater than `-1`.

Chebyshev of the First and Second Kind	`T(n,x)`
	`U(n,x)`	

mhelp

Purpose

Maple help

Syntax

```
mhelp topic
mhelp('topic')
```

Description

`mhelp topic` and `mhelp('topic')` both return Maple's online documentation for the specified Maple `topic`.

Examples

`mhelp BesselI` and `mhelp('BesselI')` both return Maple's online documentation for the Maple `BesselI` function.

See Also

`maple`

Purpose

Symbolic matrix elementwise modulus

Syntax

```
C = mod(A, B)
```

Description

`C = mod(A, B)` for symbolic matrices `A` and `B` with integer elements is the positive remainder in the element-wise division of `A` by `B`. For matrices with polynomial entries, `mod(A, B)` is applied to the individual coefficients.

Examples

```
ten = sym('10');
mod(2^ten,ten^3)
24

syms x
mod(x^3-2*x+999,10)
x^3+8*x+9
```

See Also

`quorem`

null

Purpose

Basis for null space

Syntax

```
Z = null(A)
```

Description

The columns of `Z = null(A)` form a basis for the null space of `A`.

`size(Z,2)` is the nullity of `A`.

`A*Z` is zero.

If `A` has full rank, `Z` is empty.

Examples

The statements

```
  A = sym(magic(4));
  Z = null(A)
  A*Z
```

return

```
  [ -1]
  [ -3]
  [  3]
  [  1]

  [ 0]
  [ 0]
  [ 0]
  [ 0]
```

See Also

`Arithmetic Operations`, `colspace`, `rank`, `rref`, `svd`, `null` in the online MATLAB Function Reference.

Purpose

Numerator and denominator

Syntax

```
[N,D] = numden(A)
```

Description

`[N,D] = numden(A)` converts each element of `A` to a rational form where the numerator and denominator are relatively prime polynomials with integer coefficients. `A` is a symbolic or a numeric matrix. `N` is the symbolic matrix of numerators, and `D` is the symbolic matrix of denominators.

Examples

`[n,d] = numden(sym(4/5))` returns `n = 4` and `d = 5`.

`[n,d] = numden(x/y + y/x)` returns

```
n =
x^2+y^2

d =
y*x
```

The statements

```
A = [a, 1/b]
[n,d] = numden(A)
```

return

```
A =
[a, 1/b]

n =
[a, 1]

d =
[1, b]
```

Purpose

Characteristic polynomial of matrix

Syntax

```
p = poly(A)
p = poly(A, v)
```

Description

If `A` is a numeric array, `p = poly(A)` returns the coefficients of the characteristic polynomial of `A`. If `A` is symbolic, `poly(A)` returns the characteristic polynomial of `A` in terms of the default variable `x`.

Note that if `A` is numeric, `poly(sym(A))` approximately equals `poly2sym(poly(A))`. The approximation is due to roundoff error.

`p = poly(A, v)` specifies to use the second input argument, `v`, in place of the default variablex.

Examples

The statements

```
syms z
A = gallery(3)
p = poly(A)
q = poly(sym(A))
s = poly(sym(A),z)
```

return

```
A =
  -149   -50  -154
   537   180   546
   -27    -9   -25

p =
1.0000   -6.0000   11.0000   -6.0000

q=
x^3-6*x^2+11*x-6

s =
z^3-6*z^2+11*z-6
```

See Also poly2sym, jordan, eig, solve

poly2sym

Purpose

Polynomial coefficient vector to symbolic polynomial

Syntax

```
r = poly2sym(c)
r = poly2sym(c, v)
```

Description

`r = poly2sym(c)` returns a symbolic representation of the polynomial whose coefficients are in the numeric vector `c`. The default symbolic variable is `x`. The variable `v` can be specified as a second input argument. If `c = [c1 c2 ...  cn]`, `r = poly2sym(c)` has the form

$$c_1x^{n-1} + c_2x^{n-2} + \ldots + c_n$$

`poly2sym` uses `sym`'s default (rational) conversion mode to convert the numeric coefficients to symbolic constants. This mode expresses the symbolic coefficient approximately as a ratio of integers, if `sym` can find a simple ratio that approximates the numeric value, otherwise as an integer multiplied by a power of 2.

`r = poly2sym(c, v)` s a polynomial in the symbolic variable `v` with coefficients from the vector `c`. If `v` has a numeric value and `sym` expresses the elements of `c` exactly, `eval(poly2sym(c))` returns the same value as `polyval(c, v)`.

Examples

`poly2sym([1 3 2])` returns

```
x^2 + 3*x + 2
```

`poly2sym([.694228, .333, 6.2832])` returns

```
6253049924220329/9007199254740992*x^2+333/1000*x+3927/625
```

`poly2sym([1 0 1 -1 2], y)` returns

```
y^4+y^2-y+2
```

See Also

`sym`, `sym2poly`, `polyval` in the online MATLAB Function Reference

Purpose

Pretty-print symbolic expressions

Syntax

```
pretty(S)
```

Description

The `pretty` function prints symbolic output in a format that resembles typeset mathematics.

`pretty(S)` prettyprints the symbolic matrix `S` using the default line width of `79`.

Examples

The following statements

```
A = sym(pascal(2))
B = eig(A)
pretty(B)
```

return

```
A =
[1, 1]
[1, 2]

B =
[3/2+1/2*5^(1/2)]
[3/2-1/2*5^(1/2)]

                           [            1/2 ]
                           [  3/2 + 1/2 5    ]
                           [                 ]
                           [            1/2 ]
                           [  3/2 - 1/2 5    ]
```

procread

Purpose

Install Maple procedure

Syntax

```
procread('filename')
```

Description

`procread('filename')` reads the specified file, which should contain the source text for a Maple procedure. It deletes any comments and newline characters, then sends the resulting string to Maple.

Extended Symbolic Math Toolbox is required.

Examples

Suppose the file `ident.src` contains the following source text for a Maple procedure.

```
ident := proc(A)
#  ident(A) computes A*inverse(A)
   local X;
   X := inverse(A);
   evalm(A &* X);
end;
```

Then the statement

```
procread('ident.src')
```

installs the procedure. It can be accessed with

```
maple('ident',magic(3))
```

or

```
maple('ident',vpa(magic(3)))
```

See Also

`maple`

Purpose

Symbolic matrix elementwise quotient and remainder

Syntax

```
[Q,R] = quorem(A,B)
```

Description

`[Q,R] = quorem(A,B)` for symbolic matrices `A` and `B` with integer or polynomial elements does element-wise division of `A` by `B` and returns quotient `Q` and remainder `R` so that `A = Q.*B+R`. For polynomials, `quorem(A,B,x)` uses variable `x` instead of `findsym(A,1)` or `findsym(B,1)`.

Example

```
syms x
p = x^3-2*x+5
[q,r] = quorem(x^5,p)
q = x^2+2
r = -5*x^2-10+4*x
[q,r] = quorem(10^5,subs(p,'10'))
q = 101
r = 515
```

See Also

`mod`

rank

Purpose Symbolic matrix rank

Syntax `rank(A)`

Description `rank(A)` is the rank of the symbolic matrix `A`.

Examples `rank([a b;c d])` is 2.

`rank(sym(magic(4)))` is `3`.

Purpose Real part of imaginary number

Syntax `real(Z)`

Description `real(Z)` is the real part of a symbolic `Z`.

See Also `conj`, `imag`

Purpose

Symbolic matrix elementwise round

Syntax

```
Y = round(X)
```

Description

`Y = round(X)` rounds the elements of `X` to the nearest integers. Values halfway between two integers are rounded away from zero.

Example

```
x = sym(-5/2)
[fix(x) floor(x) round(x) ceil(x) frac(x)]
= [ -2, -3, -3, -2, -1/2]
```

See Also

`floor`, `ceil`, `fix`, `frac`

Purpose

Reduced row echelon form

Syntax

`rref(A)`

Description

`rref(A)` is the reduced row echelon form of the symbolic matrix `A`.

Note The Maple kernel assumes the symbolic variable is never zero in the reduction process. Matrices whose elements are free symbolic variables are regarded as nonzero. For matrices whose elements are rational numbers expressed as the ratio of two arbitrary precision integers, Maple does exact rational arithmetic without encountering floating point roundoff errors.

Examples

`rref(sym(magic(4)))` returns

```
[  1,  0,  0,  1]
[  0,  1,  0,  3]
[  0,  0,  1, -3]
[  0,  0,  0,  0]
```

Purpose

Interactive evaluation of Riemann sums

Syntax

```
rsums(f)
rsums(f,a,b)
rsums(f,[a,b])
```

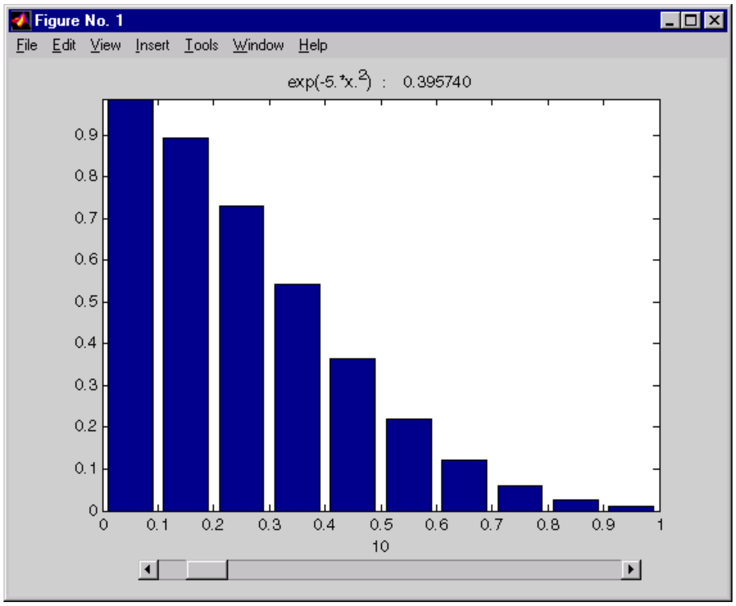
Description

`rsums(f)` interactively approximates the integral of $f(x)$ by Riemann sums from 0 to 1. `rsums(f)` displays a graph of $f(x)$. You can then adjust the number of terms taken in the Riemann sum by using the slider below the graph. The number of terms available ranges from 2 to 128. `f` can be a string or a symbolic expression.

`rsums(f,a,b)` and `rsums(f,[a,b])` approximates the integral from `a` to `b`.

Examples

Either `rsums('exp(-5*x^2)')` or `rsums exp(-5*x^2)` creates the following plot.



Purpose

Search for simplest form of symbolic expression

Syntax

```
r = simple(S)
[r,how] = simple(S)
```

Description

`r = simple(S)` tries several different algebraic simplifications of the symbolic expression `S`, displays any that shorten the length of `S`'s representation, and returns the shortest. `S` is a `sym`. If `S` is a matrix, the result represents the shortest representation of the entire matrix, which is not necessarily the shortest representation of each individual element. If no return output is given, `simple(S)` displays all possible representations and returns the shortest.

`[r,how] = simple(S)` does not display intermediate simplifications, but returns the shortest found, as well as a string describing the particular simplification. `r` is a `sym`. `how` is a string.

Examples

Expression	Simplification	Simplification Method
`cos(x)^2+sin(x)^2`	`1`	`simplify`
`2*cos(x)^2-sin(x)^2`	`3*cos(x)^2-1`	`simplify`
`cos(x)^2-sin(x)^2`	`cos(2*x)`	`combine(trig)`
`cos(x)+ (-sin(x)^2)^(1/2)`	`cos(x)+i*sin(x)`	`radsimp`
`cos(x)+i*sin(x)`	`exp(i*x)`	`convert(exp)`
`(x+1)*x*(x-1)`	`x^3-x`	`collect(x)`
`x^3+3*x^2+3*x+1`	`(x+1)^3`	`factor`
`cos(3*acos(x))`	`4*x^3-3*x`	`expand`

See Also

`collect`, `expand`, `factor`, `horner`, `simplify`

simplify

Purpose Symbolic simplification

Syntax

```
R = simplify(S)
```

Description `R = simplify(S)` simplifies each element of the symbolic matrix `S` using Maple simplification rules.

Examples `simplify(sin(x)^2 + cos(x)^2)` returns

```
1
```

`simplify(exp(c*log(sqrt(a+b))))` returns

```
(a+b)^(1/2*c)
```

The statements

```
S = [(x^2+5*x+6)/(x+2),sqrt(16)];
R = simplify(S)
```

return

```
R = [x+3,4]
```

See Also `collect`, `expand`, `factor`, `horner`, `simple`

Purpose Convert symbolic matrix to single precision

Syntax

```
single(S)
```

Description `single(S)` converts the symbolic matrix `S` to a matrix of single-precision floating-point numbers. `S` must not contain any symbolic variables, except `'eps'`.

See Also `sym`, `vpa`, `double`

sinint

Purpose

Sine integral

Syntax

```
Y = sinint(X)
```

Description

`Y = sinint(X)` evaluates the sine integral function at the elements of `X`, a numeric matrix, or a symbolic matrix. The result is a numeric matrix. The sine integral function is defined by

$$Si(x) = \int_0^x \frac{\sin t}{t} dt$$

Examples

`sinint([pi 0;-2.2 exp(3)])` returns

```
      1.8519         0
     -1.6876    1.5522
```

`sinint(1.2)` returns `1.1080`.

`diff(sinint(x))` returns `sin(x)/x`.

See Also

`cosint`

Purpose

Symbolic matrix dimensions

Syntax

```
d = size(A)
[m,n] = size(A)
d = size(A,n)
```

Description

Suppose `A` is an `m`-by-`n` symbolic or numeric matrix. The statement `d = size(A)` returns a numeric vector with two integer components, `d = [m,n]`.

The multiple assignment statement `[m,n] = size(A)` returns the two integers in two separate variables.

The statement `d = size(A,n)` returns the length of the dimension specified by the scalar n. For example, `size(A,1)` is the number of rows of `A` and `size(A,2)` is the number of columns of `A`.

Examples

The statements

```
syms a b c d
A = [a b c ; a b d; d c b; c b a];
d = size(A)
r = size(A, 2)
```

return

```
d =
     4     3

r =
     3
```

See Also

`length`, `ndims` in the online MATLAB Function Reference

solve

Purpose Symbolic solution of algebraic equations

Syntax

```
solve(eq)
solve(eq,var)
solve(eq1,eq2,...,eqn)
g = solve(eq1,eq2,...,eqn,var1,var2,...,varn)
```

Description

Single Equation/Expression2

The input to `solve` can be either symbolic expressions or strings. If `eq` is a symbolic expression (`x^2-2*x+1`) or a string that does not contain an equal sign (`'x^2-2*x+1'`), then `solve(eq)` solves the equation `eq=0` for its default variable (as determined by `findsym`).

`solve(eq,var)` solves the equation `eq` (or `eq=0` in the two cases cited above) for the variable `var`.

System of Equations

The inputs are either symbolic expressions or strings specifying equations. `solve(eq1,eq2,...,eqn)` or solves the system of equations implied by `eq1,eq2,...,eqn` in the `n` variables determined by applying `findsym` to the system.

`g = solve(eq1,eq2,...,eqn,var1,var2,...,varn)` finds the zeros for the system of equations for the variables specified as inputs.

Three different types of output are possible. For one equation and one output, the resulting solution is returned with multiple solutions for a nonlinear equation. For a system of equations and an equal number of outputs, the results are sorted alphabetically and assigned to the outputs. For a system of equations and a single output, a structure containing the solutions is returned.

For both a single equation and a system of equations, numeric solutions are returned if symbolic solutions cannot be determined.

Examples `solve('a*x^2 + b*x + c')` returns

```
[ 1/2/a*(-b+(b^2-4*a*c)^(1/2)),
   1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

`solve('a*x^2 + b*x + c','b')` returns

```
   -(a*x^2+c)/x
```

`S = solve('x + y = 1','x - 11*y = 5')` returns a structure S with

```
    S.y = -1/3, S.x = 4/3
```

`A = solve('a*u^2 + v^2', 'u - v = 1', 'a^2 - 5*a + 6')`

returns

```
   A =

       a: [4x1 sym]
       u: [4x1 sym]
       v: [4x1 sym]
```

where

```
   A.a =
    [ 2]
    [ 2]
    [ 3]
    [ 3]
   A.u =
    [ 1/3+1/3*i*2^(1/2)]
    [ 1/3-1/3*i*2^(1/2)]
    [ 1/4+1/4*i*3^(1/2)]
    [ 1/4-1/4*i*3^(1/2)]
   A.v =
    [ -2/3+1/3*i*2^(1/2)]
    [ -2/3-1/3*i*2^(1/2)]
    [ -3/4+1/4*i*3^(1/2)]
    [ -3/4-1/4*i*3^(1/2)]
```

See Also

`Arithmetic Operations`, `dsolve`, `findsym`

Purpose

Sort symbolic vectors or polynomials

Syntax

```
Y = sort(v)
Y = sort(p)
```

Description

`Y = sort(v)` sorts the elements of a symbolic vector `v` in numerical or lexicographic order.

`Y = sort(p)` sorts the terms in a polynomial `p` in order of decreasing powers.

Examples

```
  syms a b c d e x
  sort(sym(magic(3))) = [1,2,3,4,5,6,7,8,9]
  sort([a c e b d]) = [a b c d e]
  sort([a c e b d]*x.^(0:4).') =
  d*x^4 + b*x^3 + e*x^2 + c*x + a
```

See Also

`sym2poly`, `coeffs`

Purpose

Rewrite symbolic expression in terms of common subexpressions

Syntax

```
[Y,SIGMA] = subexpr(X,SIGMA)
[Y,SIGMA] = subexpr(X,'SIGMA')
```

Description

`[Y,SIGMA] = subexpr(X,SIGMA)` or `[Y,SIGMA] = subexpr(X,'SIGMA')` rewrites the symbolic expression `X` in terms of its common subexpressions. These are the subexpressions that are written as `%1`, `%2`, etc. by `pretty(S)`.

Examples

The statements

```
t = solve('a*x^3+b*x^2+c*x+d = 0');
[r,s] = subexpr(t,'s');
```

return the rewritten expression for `t` in `r` in terms of a common subexpression, which is returned in `s`.

See Also

`pretty`, `simple`, `subs`

subs

Purpose

Symbolic substitution in symbolic expression or matrix

Syntax

```
R = subs(S)
R = subs(S, new)
R = subs(S,old,new)
```

Description

`R = subs(S)` replaces all occurrences of variables in the symbolic expression `S` with values obtained from the calling function, or the MATLAB workspace.

`R = subs(S, new)` replaces the default symbolic variable in `S` with `new`.

`R = subs(S,old,new)` replaces `old` with `new` in the symbolic expression `S`. `old` is a symbolic variable or a string representing a variable name. `new` is a symbolic or numeric variable or expression.

If `old` and `new` are cell arrays of the same size, each element of `old` is replaced by the corresponding element of `new`. If `S` and `old` are scalars and `new` is an array or cell array, the scalars are expanded to produce an array result. If `new` is a cell array of numeric matrices, the substitutions are performed elementwise (i.e., `subs(x*y,{x,y},{A,B})` returns `A.*B` when `A` and `B` are numeric).

If `subs(s,old,new)` does not change `s`, `subs(s,new,old)` is tried. This provides backwards compatibility with previous versions and eliminates the need to remember the order of the arguments. `subs(s,old,new,0)` does not switch the arguments if `s` does not change.

Note If `A` is a matrix, the command `subs(S, x, A)` replaces all occurrences of the variable `x` in the symbolic expression `S` with the matrix `A`, and replaces the constant term in `S` with the constant times a matrix of all ones. To evaluate `S` in the matrix sense, use the command `polyvalm(sym2poly(S), A)`, which replaces the constant term with the constant times an identity matrix.

Examples

Single Input

Suppose a = 980 and C1 = 3 exist in the workspace.

The statement

```
y = dsolve('Dy = -a*y')
```

produces

```
y = C1*exp(-a*t)
```

Then the statement

```
subs(y)
```

produces

```
ans = 3*exp(-980*t)
```

Single Substitution

`subs(a+b,a,4)` returns `4+b`.

Multiple Substitutions

`subs(cos(a)+sin(b),{a,b},{sym('alpha'),2})` returns

```
cos(alpha)+sin(2)
```

Scalar Expansion Case

`subs(exp(a*t),'a',-magic(2))` returns

```
[   exp(-t), exp(-3*t)]
[ exp(-4*t), exp(-2*t)]
```

Multiple Scalar Expansion

`subs(x*y,{x,y},{[0 1;-1 0],[1 -1;-2 1]})` returns

```
     0    -1
     2     0
```

See Also `simplify`, `subexpr`

Purpose

Symbolic singular value decomposition

Syntax

```
sigma = svd(A)
sigma = svd(vpa(A))
[U,S,V] = svd(A)
[U,S,V] = svd(vpa(A))
```

Description

`sigma = svd(A)` is a symbolic vector containing the singular values of a symbolic matrix `A`.

`sigma = svd(vpa(A))` computes numeric singular values, using variable precision arithmetic.

`[U,S,V] = svd(A)` and `[U,S,V] = svd(vpa(A))` return numeric unitary matrices `U` and `V` whose columns are the singular vectors and a diagonal matrix `S` containing the singular values. Together, they satisfy `A = U*S*V'`.

Symbolic singular vectors are not available.

Note With symbolic inputs and multiple outputs, the `svd` function does not accept complex values as inputs.

Examples

The statements

```
digits(3)
A = sym(magic(4));
svd(A)
svd(vpa(A))
[U,S,V] = svd(A)
```

return

```
[         0]
[        34]
[ 2*5^(1/2)]
[ 8*5^(1/2)]

[ .311e-6*i]
[      4.47]
[      17.9]
[      34.1]
U =

[ -.500,  .671,  .500, -.224]
[ -.500, -.224, -.500, -.671]
[ -.500,  .224, -.500,  .671]
[ -.500, -.671,  .500,  .224]


S =

[     34.0,        0,        0,        0]
[        0,     17.9,        0,        0]
[        0,        0,     4.47,        0]
[        0,        0,        0, .835e-15]


V =

[ -.500,  .500,  .671, -.224]
[ -.500, -.500, -.224, -.671]
[ -.500, -.500,  .224,  .671]
[ -.500,  .500, -.671,  .224]
```

See Also

digits, eig, vpa

Purpose

Symbolic numbers, variables, and objects

Syntax

```
S = sym(A)
x = sym('x')
x = sym('x','real')
x = sym('x','unreal')
S = sym(A,flag)
```

Description

`S = sym(A)` constructs an object `S`, of class `'sym'`, from `A`. If the input argument is a string, the result is a symbolic number or variable. If the input argument is a numeric scalar or matrix, the result is a symbolic representation of the given numeric values.

`x = sym('x')` creates the symbolic variable with name `'x'` and stores the result in `x`. `x = sym('x','real')` also assumes that `x` is real, so that `conj(x)` is equal to `x`. `alpha = sym('alpha')` and `r = sym('Rho','real')` are other examples. Similarly, `k = sym('k','positive')` makes `k` a positive (real) variable. `x = sym('x','unreal')` makes `x` a purely formal variable with no additional properties (i.e., ensures that `x` is neither real nor positive). See also the reference pages on `syms`.

Statements like `pi = sym('pi')` and `delta = sym('1/10')` create symbolic numbers that avoid the floating-point approximations inherent in the values of `pi` and `1/10`. The `pi` created in this way temporarily replaces the built-in numeric function with the same name.

`S = sym(A,flag)` where `flag` is one of `'r'`, `'d'`, `'e'`, or `'f'`, converts a numeric scalar or matrix to symbolic form. The technique for converting floating-point numbers is specified by the optional second argument, which can be `'f'`, `'r'`, `'e'` or `'d'`. The default is `'r'`.

`'f'` stands for "floating-point." All values are represented in the form `'1.F'*2^(e)` or `'-1.F'*2^(e)` where `F` is a string of 13 hexadecimal digits and `e` is an integer. This captures the floating-point values exactly, but may not be convenient for subsequent manipulation. For example, `sym(1/10,'f')` is `'1.999999999999a'*2^(-4)` because `1/10` cannot be represented exactly in floating-point.

`'r'` stands for “rational.” Floating-point numbers obtained by evaluating expressions of the form `p/q`, `p*pi/q`, `sqrt(p)`, `2^q`, and `10^q` for modest sized integers `p` and `q` are converted to the corresponding symbolic form. This effectively compensates for the roundoff error involved in the original evaluation, but may not represent the floating-point value precisely. If no simple rational approximation can be found, an expression of the form `p*2^q` with large integers `p` and `q` reproduces the floating-point value exactly. For example, `sym(4/3,'r')` is `'4/3'`, but `sym(1+sqrt(5),'r')` is `7286977268806824*2^(-51)`.

`'e'` stands for “estimate error.” The `'r'` form is supplemented by a term involving the variable `'eps'`, which estimates the difference between the theoretical rational expression and its actual floating-point value. For example, `sym(3*pi/4)` is `3*pi/4-103*eps/249`.

`'d'` stands for “decimal.” The number of digits is taken from the current setting of `digits` used by `vpa`. Fewer than 16 digits loses some accuracy, while more than 16 digits may not be warranted. For example, with `digits(10)`, `sym(4/3,'d')` is `1.333333333`, while with digits `digits(20)`, `sym(4/3,'d')` is `1.3333333333333332593`, which does not end in a string of `3`s, but is an accurate decimal representation of the floating-point number nearest to `4/3`.

See Also

`digits`, `double`, `syms`

`eps` in the online MATLAB Function Reference

Purpose

Shortcut for constructing symbolic objects

Syntax

```
syms arg1 arg2 ...
syms arg1 arg2 ... real
syms arg1 arg2 ... unreal
syms arg1 arg2 ... positive
```

Description

`syms arg1 arg2 ...` is short-hand notation for

```
arg1 = sym('arg1');
arg2 = sym('arg2'); ...
```

`syms arg1 arg2 ...  real` is short-hand notation for

```
arg1 = sym('arg1','real');
arg2 = sym('arg2','real'); ...
```

`syms arg1 arg2 ...  unreal` is short-hand notation for

```
arg1 = sym('arg1','unreal');
arg2 = sym('arg2','unreal'); ...
```

`syms arg1 arg2 ...  positive` is short-hand notation for

```
arg1 = sym('arg1','positive');
arg2 = sym('arg2','positive'); ...
```

Each input argument must begin with a letter and can contain only alphanumeric characters.

Examples

`syms x beta real` is equivalent to

```
x = sym('x','real');
beta = sym('beta','real');
```

To clear the symbolic objects `x` and `beta` of `'real'` status, type

```
syms x beta unreal
```

Note that `clear x` will *not* clear the symbolic object of its `'real'` status. You can achieve this using

- `syms x unreal` to remove the `'real'` status from `x` without affecting any other symbolic variables.
- `clear maplemex` (or `clear mex` or `clear all`) to clear the Maple kernel.
- `maple restart` to clear all symbolic variables from the Maple workspace and reinitializes the Maple kernel. The `restart` option is less efficient if you continue using Symbolic Toolbox functions, and it is not available on the Macintosh.

See Also

`sym`

Purpose

Symbolic-to-numeric polynomial conversion

Syntax

```
c = sym2poly(s)
```

Description

`c = sym2poly(s)` returns a row vector containing the numeric coefficients of a symbolic polynomial. The coefficients are ordered in descending powers of the polynomial's independent variable. In other words, the vector's first entry contains the coefficient of the polynomial's highest term; the second entry, the coefficient of the second highest term; and so on.

Examples

The commands

```
syms x u v;
sym2poly(x^3 - 2*x - 5)
```

return

```
1     0    -2    -5
```

while `sym2poly(u^4 - 3 + 5*u^2)` returns

```
1     0     5     0    -3
```

and `sym2poly(sin(pi/6)*v + exp(1)*v^2)` returns

```
2.7183    0.5000         0
```

See Also

`poly2sym`, `polyval` in the online MATLAB Function Reference

Purpose

Symbolic summation of series

Syntax

```
r = symsum(s)
r = symsum(s,v)
r = symsum(s,a,b)
r = symsum(s,v,a,b)
```

Description

`r = symsum(s)` is the summation of the symbolic expression `s` with respect to its symbolic variable `k` as determined by `findsym` from `0` to `k-1`.

`r = symsum(s,v)` is the summation of the symbolic expression `s` with respect to the symbolic variable `v` from `0` to `v-1`.

`r = symsum(s,a,b)` and `r = symsum(s,v,a,b)` are the definite summations of the symbolic expression from `v=a` to `v=b`.

Examples

The commands

```
syms k n x
symsum(k^2)
```

return

```
1/3*k^3-1/2*k^2+1/6*k
```

`symsum(k)` returns

```
1/2*k^2-1/2*k
```

`symsum(sin(k*pi)/k,0,n)` returns

```
-1/2*sin(k*(n+1))/k+1/2*sin(k)/k/(cos(k)-1)*cos(k*(n+1))-
1/2*sin(k)/k/(cos(k)-1)
```

`symsum(k^2,0,10)` returns

```
385
```

`symsum(x^k/sym('k!'), k, 0,inf)` returns

```
exp(x)
```

Note The preceding example uses `sym` to create the symbolic expression `k!` in order to bypass the MATLAB expression parser, which does not recognize `!` as a factorial operator.

See Also

`findsym`, `int`, `syms`

taylor

Purpose

Taylor series expansion

Syntax

```
taylor(f)
taylor(f,n,v)
taylor(f,n,v,a)
```

Description

`taylor(f)` is the fifth order Maclaurin polynomial approximation to `f`.

`taylor(f,n,v)` returns the (`n-1`)-order Maclaurin polynomial approximation to `f`, where `f` is a symbolic expression representing a function and `v` specifies the independent variable in the expression. `v` can be a string or symbolic variable.

`taylor(f,n,v,a)` returns the Taylor series approximation to `f` about `a`. The argument `a` can be a numeric value, a symbol, or a string representing a numeric value or an unknown.

You can supply the arguments `n`, `v`, and `a` in any order. `taylor` determines the purpose of the arguments from their position and type.

You can also omit any of the arguments `n`, `v`, and `a`. If you do not specify `v`, `taylor` uses find`sym` to determine the function's independent variable. `n` defaults to `6`.

The Taylor series for an analytic function $f(x)$ about the basepoint $x=a$ is given below.

$$f(x) = \sum_{n=0}^{\infty} (x-a)^n \cdot \frac{f^{(n)}(a)}{n!}$$

Examples

This table describes the various uses of the `taylor` command and its relation to Taylor and MacLaurin series.

Mathematical Operation	MATLAB

$\sum_{n=0}^{5} x^n \cdot \frac{f^{(n)}(0)}{n!}$	`syms x`
	
	`taylor(f)`

$\sum_{n=0}^{m} x^n \cdot \frac{f^{(n)}(0)}{n!}$ m is a positive integer	`taylor(f,m)` m is a positive integer
$\sum_{n=0}^{5} (x-a)^n \cdot \frac{f^{(n)}(a)}{n!}$ a is a real number	`taylor(f,a)` a is a real number
$\sum_{n=0}^{m_1} (x-m_2)^n \cdot \frac{f^{(n)}(m^2)}{n!}$ m_1, m_2 are positive integers	`taylor(f,m1,m2)` m_1, m_2 are positive integers
$\sum_{n=0}^{m} (x-a)^n \cdot \frac{f^{(n)}(a)}{n!}$ a is real and m is a positive integer	`taylor(f,m,a)` a is real and m is a positive integer

In the case where `f` is a function of two or more variables (`f=f(x,y,...)`), there is a fourth parameter that allows you to select the variable for the Taylor expansion. Look at this table for illustrations of this feature.

Mathematical Operation	MATLAB
$\sum_{n=0}^{5} \frac{y^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x, y=0)$	`taylor(f,y)`
$\sum_{n=0}^{m} \frac{y^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x, y=0)$ m is a positive integer	`taylor(f,y,m)` or `taylor(f,m,y)` m is a positive integer
$\sum_{n=0}^{m} \frac{(y-a)^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x, y=a)$ a is real and m is a positive integer	`taylor(f,m,y,a)` a is real and m is a positive integer
$\sum_{n=0}^{5} \frac{(y-a)^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x, y=a)$ a is real	`taylor(f,y,a)` a is real

See Also `findsym`

Purpose

Taylor series calculator

Syntax

```
taylortool
taylortool('f')
```

Description

`taylortool` initiates a GUI that graphs a function against the Nth partial sum of its Taylor series about a basepoint `x = a`. The default function, value of N, basepoint, and interval of computation for `taylortool` are `f = x*cos(x)`, `N = 7`, `a = 0`, and `[-2*pi,2*pi]`, respectively.

`taylortool('f')` initiates the GUI for the given expression `f`.

Examples

```
taylortool('exp(x*sin(x))')

taylortool('sin(tan(x)) - tan(sin(x))')
```

See Also

`funtool`, `rsums`

Purpose

Symbolic lower triangle

Syntax

```
tril(X)
tril(X,K)
```

Description

`tril(X)` is the lower triangular part of `X`.

`tril(X,K)` returns a lower triangular matrix that retains the elements of `X` on and below the `k`-th diagonal and sets the remaining elements to 0. The values `k=0`, `k>0`, and `k<0` correspond to the main, superdiagonals, and subdiagonals, respectively.

Examples

Suppose

```
A =
[   a,   b,   c ]
[   1,   2,   3 ]
[ a+1, b+2, c+3 ]
```

Then `tril(A)` returns

```
[   a,   0,   0 ]
[   1,   2,   0 ]
[ a+1, b+2, c+3 ]
```

`tril(A,1)` returns

```
[   a,   b,   0 ]
[   1,   2,   3 ]
[ a+1, b+2, c+3 ]
```

`tril(A,-1)` returns

```
[   0,   0,   0 ]
[   1,   0,   0 ]
[ a+1, b+2,   0 ]
```

See Also

`diag`, `triu`

Purpose

Symbolic upper triangle

Syntax

```
triu(X)
triu(X, K)
```

Description

`triu(X)` is the upper triangular part of `X`.

`triu(X, K)` returns an upper triangular matrix that retains the elements of `X` on and above the `k`-th diagonal and sets the remaining elements to 0. The values `k=0`, `k>0`, and `k<0` correspond to the main, superdiagonals, and subdiagonals, respectively.

Examples

Suppose

```
A =
[    a,   b,   c ]
[    1,   2,   3 ]
[  a+1, b+2, c+3 ]
```

Then `triu(A)` returns

```
[    a,   b,   c ]
[    0,   2,   3 ]
[    0,   0, c+3 ]
```

`triu(A,1)` returns

```
[ 0, b, c ]
[ 0, 0, 3 ]
[ 0, 0, 0 ]
```

`triu(A,-1)` returns

```
[    a,   b,   c ]
[    1,   2,   3 ]
[    0, b+2, c+3 ]
```

See Also

`diag`, `tril`

uint8, uint16, uint32, uint64

Purpose

Convert symbolic matrix to unsigned integers

Syntax

```
uint8(S)
uint16(S)
uint32(S)
uint64(S)
```

Description

`uint8(S)` converts a symbolic matrix `S` to a matrix of unsigned 8-bit integers.

`uint16(S)` converts `S` to a matrix of unsigned 16-bit integers.

`uint32(S)` converts `S` to a matrix of unsigned 32-bit integers.

`uint64(S)` converts `S` to a matrix of unsigned 64-bit integers.

Note The output of `uint8`, `uint16`, `uint32`, and `uint64` does not have type `symbolic`.

The following table summarizes the output of these four functions.

Function	Output Range	Output Type	Bytes per Element	Output Class
`uint8`	0 to 255	Unsigned 8-bit integer	1	`uint8`
`uint16`	0 to 65,535	Unsigned 16-bit integer	2	`uint16`
`uint32`	0 to 4,294,967,295	Unsigned 32-bit integer	4	`uint32`
`uint64`	0 to 18,446,744,073,709, 551,615	Unsigned 64-bit integer	8	`uint64`

See Also

`sym`, `vpa`, `single`, `double`, `int8`, `int16`, `int32`, `int64`

Purpose

Variable precision arithmetic

Syntax

```
R = vpa(A)
R = vpa(A,d)
```

Description

`R = vpa(A)` uses variable-precision arithmetic (VPA) to compute each element of `A` to `d` decimal digits of accuracy, where `d` is the current setting of `digits`. Each element of the result is a symbolic expression.

`R = vpa(A,d)` uses `d` digits, instead of the current setting of `digits`.

Examples

The statements

```
digits(25)
q = vpa(sin(sym('pi')/6))
p = vpa(pi)
w = vpa('(1+sqrt(5))/2')
```

return

```
q =
.5000000000000000000000000

p = 3.141592653589793238462643

w =
1.618033988749894848204587
```

`vpa pi 75` computes π to 75 digits.

`The statements`

```
A = vpa(hilb(2),25)
B = vpa(hilb(2),5)
```

return

```
A =
[                        1., .5000000000000000000000000]
```

```
[.50000000000000000000000000, .33333333333333333333333333]

B =
[    1., .50000]
[.50000, .33333]
```

See Also `digits`, `double`

Purpose

Riemann Zeta

Syntax

```
Y = zeta(X)
Y = zeta(n, X)
```

Description

`Y = zeta(X)` evaluates the Zeta function at the elements of `X`, a numeric matrix, or a symbolic matrix. The Zeta function is defined by

$$\zeta(w) = \sum_{k=1}^{\infty} \frac{1}{k^w}$$

`Y = zeta(n, X)` returns the `n`-th derivative of `zeta(X)`.

Examples

`zeta(1.5)` returns `2.6124`.

`zeta(1.2:0.1:2.1)` returns

```
  Columns 1 through 7

    5.5916    3.9319    3.1055    2.6124    2.2858    2.0543    1.8822

  Columns 8 through 10

    1.7497    1.6449    1.5602
```

`zeta([x 2;4 x+y])` returns

```
  [   zeta(x),  1/6*pi^2]
  [ 1/90*pi^4, zeta(x+y)]
```

`diff(zeta(x),x,3)` returns `zeta(3,x)`.

ztrans

Purpose

z-transform

Syntax

```
F = ztrans(f)
F = ztrans(f,w)
F = ztrans(f,k,w)
```

Description

`F = ztrans(f)` is the z-transform of the scalar symbol `f` with default independent variable `n`. The default return is a function of `z`.

$$f = f(n) \Rightarrow F = F(z)$$

The z-transform of `f` is defined as

$$F(z) = \sum_{0}^{\infty} \frac{f(n)}{z^n}$$

where `n` is `f`'s symbolic variable as determined by `findsym`. If `f = f(z)`, then `ztrans(f)` returns a function of `w`.

$$F = F(w)$$

`F = ztrans(f,w)` makes `F` a function of the symbol `w` instead of the default `z`.

$$F(w) = \sum_{0}^{\infty} \frac{f(n)}{w^n}$$

`F = ztrans(f,k,w)` takes `f` to be a function of the symbolic variable `k`.

$$F(w) = \sum_{0}^{\infty} \frac{f(k)}{w^k}$$

Examples

Z-Transform	MATLAB Operation

$f(n) = n^4$
 $Z[f] = \sum_{n=0}^{\infty} f(n)z^{-n}$
 $= \frac{z(z^3+11z^2+11z+1)}{(z-1)^5}$
	`f = n^4`
	 `ztrans(f)`
	 returns
	 `z*(z^3+11*z^2+11*z+1)/(z-1)^5`

$g(z) = a^z$
 $Z[g] = \sum_{z=0}^{\infty} g(z)w^{-z}$
 $= \frac{w}{a-w}$
	`g = a^z`
	 `simplify(ztrans(g))`
	 returns
	 `-w/(-w+a)`

$f(n) = \sin an$
 $Z[f] = \sum_{n=0}^{\infty} f(n)w^{-n}$
 $= \frac{w\sin a}{1-2w\cos a+w^2}$
	`f = sin(a*n)`
	 `ztrans(f,w)`
	 returns
	 `w*sin(a)/(w^2-2*w*cos(a)+1)`

See Also

`fourier`, `iztrans`, `laplace`

Index