Symbolic Math Toolbox™ 5
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science
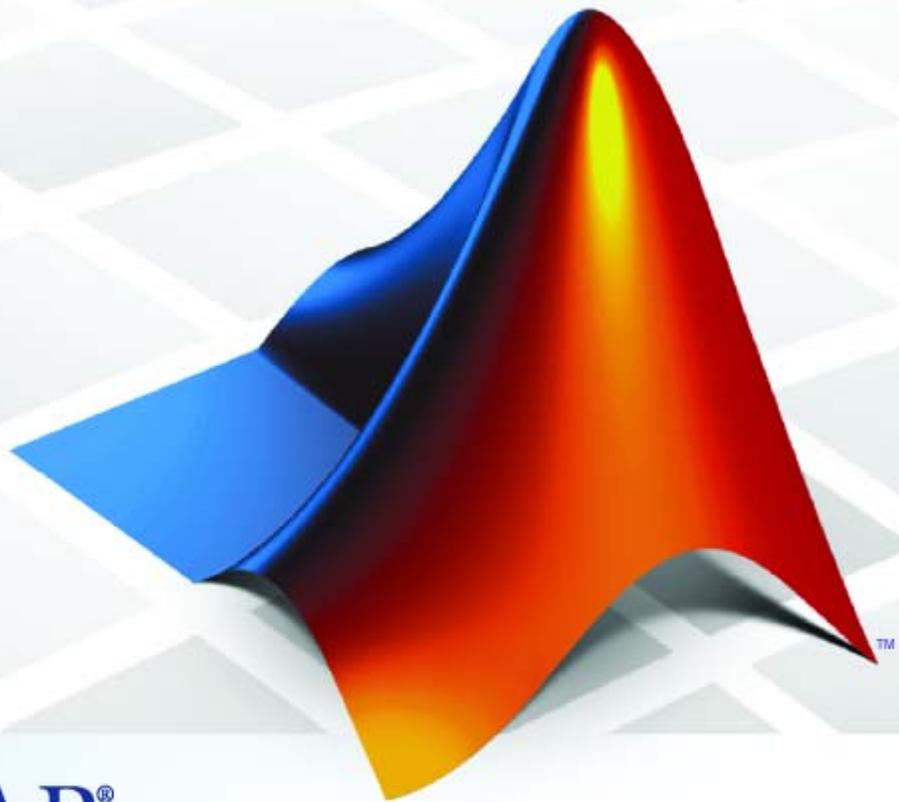
How to Contact The MathWorks

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 1993–2008 by The MathWorks, Inc.

Trademarks

Patents

Contents

3
MuPAD in Symbolic Math Toolbox

4
Function Reference

Functions — Alphabetical List

5

Index

Getting Started

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

- "Product Overview" on page 1-2
- "Symbolic Objects" on page 1-4
- "Creating Symbolic Variables and Expressions" on page 1-6
- "Substituting for Symbolic Variables" on page 1-10
- "Symbolic and Numeric Conversions" on page 1-12
- "Creating Symbolic Math Functions" on page 1-18

Product Overview

Symbolic Math Toolbox software incorporates symbolic computation into the numeric environment of MATLAB® software. It supplements MATLAB numeric and graphical capabilities with several other types of mathematical computation, 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 toolbox is the kernel of MuPAD® software.

There are two ways to interact with the MuPAD symbolic math engine:

- The Symbolic Math Toolbox software is a collection of more than 100 MATLAB functions that provide access to the MuPAD kernel using a syntax and style that is a natural extension of the MATLAB language.

- The MuPAD notebook interface is a Graphical User Interface (GUI) for performing computations in the MuPAD language. For more information

about this interface and language see Chapter 3, "MuPAD in Symbolic Math Toolbox".

If you have a version of Maple™ software consistent with the version of Symbolic Math Toolbox software that you are using, you can use that instead of MuPAD software. See "Choosing a Maple or MuPAD Engine" on page 3-35.

Symbolic Objects

Symbolic Math Toolbox software defines a new MATLAB data type called a *symbolic object*. (See "Classes (Data Types)" in the MATLAB Programming Fundamentals documentation for an introduction to MATLAB classes and objects.) Symbolic Math Toolbox software uses symbolic objects to represent symbolic variables, expressions, and matrices. The actual computations involving symbolic objects are performed primarily by the MuPAD symbolic math engine.

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 software 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 Software".

Creating Symbolic Variables and Expressions

In this section...
"sym and syms Commands" on page 1-6
"findsym Command" on page 1-8

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

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

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 =

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

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 software 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 software 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 MuPAD workspace. See "Clearing Variables and Assumptions in the MuPAD Workspace" on page 1-15 and "Clearing Assumptions and Resetting the Symbolic Engine" on page 3-30 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" on page 1-10
"Default Symbolic Variable" on page 1-11

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,

```
syms x
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*3^(1/2)*x
```

Default Symbolic Variable

If you do not specify a variable to substitute for, MATLAB software 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" on page 1-12
"Rational Symbolic Expressions" on page 1-12
"Decimal Symbolic Expressions" on page 1-13
"Converting Numeric Matrices to Symbolic Form" on page 1-13
"Constructing Real and Complex Variables" on page 1-14
"Creating Abstract Functions" on page 1-15

Floating-Point Symbolic Expressions

Consider the ordinary quantity in MATLAB software

```
t = 0.1
```

The sym function has four options for returning a symbolic representation of the numeric value stored in t. The `f' option to sym converts a double-precision floating point number to a sum of two binary numbers, in the form $-(2^e + N*2^{(52+e)})$. Here e and N are integers, and N is nonnegative.

For example,

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

returns the symbolic floating-point representation

```
(2^-4+2702159776422298*2^-56)
```

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 =
0.1
```

Converting Numeric Matrices to Symbolic 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 and Assumptions in the MuPAD 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 real variable in the MuPAD engine workspace. In other words, theMuPAD engine workspace has an assumption that x is real. If you later want to remove the real property from x in theMuPAD engine workspace, enter

```
syms x clear
```

If you want to clear all variable definitions in the MuPAD engine workspace, enter

```
reset(symengine)
```

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 MuPAD engine workspace, MATLAB still treats x as a real number. For more information on clearing symbolic variables, see "Clearing Assumptions and Resetting the Symbolic Engine" on page 3-30.

Creating Abstract Functions

If you want to create an abstract (i.e., indeterminate) 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(h + x) - f(x))/h
```

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

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

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 "Classes (Data Types)" in the MATLAB Programming Fundamentals documentation for a more detailed discussion on object-oriented programming.

2

Using Symbolic Math Toolbox Software

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

- "Calculus" on page 2-2
- "Simplifications and Substitutions" on page 2-41
- "Variable-Precision Arithmetic" on page 2-56
- "Linear Algebra" on page 2-61
- "Solving Equations" on page 2-86
- "Integral Transforms and Z-Transforms" on page 2-95
- "Special Functions of Applied Mathematics" on page 2-113
- "Generating Code from Symbolic Expressions" on page 2-122

Calculus

Differentiation

To illustrate how to take derivatives using Symbolic Math Toolbox software, 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 software 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 =
s*cos(s*t)
```

To differentiate f with respect to the variable s, enter

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

which returns:

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

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	n*x^(n - 1)
sin(a*t+b)	a*cos(b + a*t)
exp(i*theta)	i*exp(i*theta)

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 =
(nu*besselj(nu, z))/z - besselj(nu + 1, 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 =
[ -a*sin(a*x),  a*cos(a*x)]
[ -a*cos(a*x), -a*sin(a*x)]
```

You can also perform differentiation of a vector function with respect to a vector argument. Consider the transformation from Euclidean (x, y, z) to spherical (r, λ, φ) coordinates as given by $x = r \cos \lambda \cos \varphi$, $y = r \cos \lambda \sin \phi$, and $z = r \sin \lambda$. Note that λ corresponds to elevation or latitude while φ denotes azimuth or longitude.
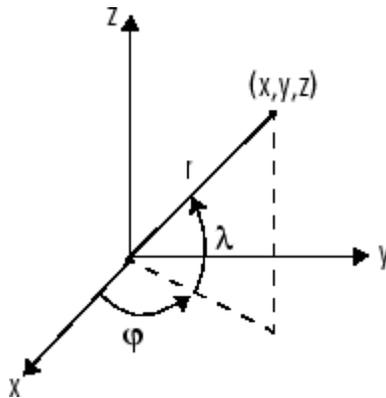
To calculate the Jacobian matrix, J, of this transformation, use the `jacobian` function. The mathematical notation for J is

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

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

and the command

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

returns

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

The arguments of the jacobian function can be column or row vectors. 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 2-41 discusses simplification in more detail.

A table summarizing diff and jacobian follows.

Mathematical Operator	MATLAB Command
$\dfrac{df}{dx}$	diff(f) or diff(f,x)
$\dfrac{df}{da}$	diff(f,a)

Mathematical Operator	MATLAB Command
$\dfrac{d^2f}{db^2}$	`diff(f,b,2)`
$J = \dfrac{\partial(r,t)}{\partial(u,v)}$	`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

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

provided this limit exists. Symbolic Math Toolbox software 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 software. 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.
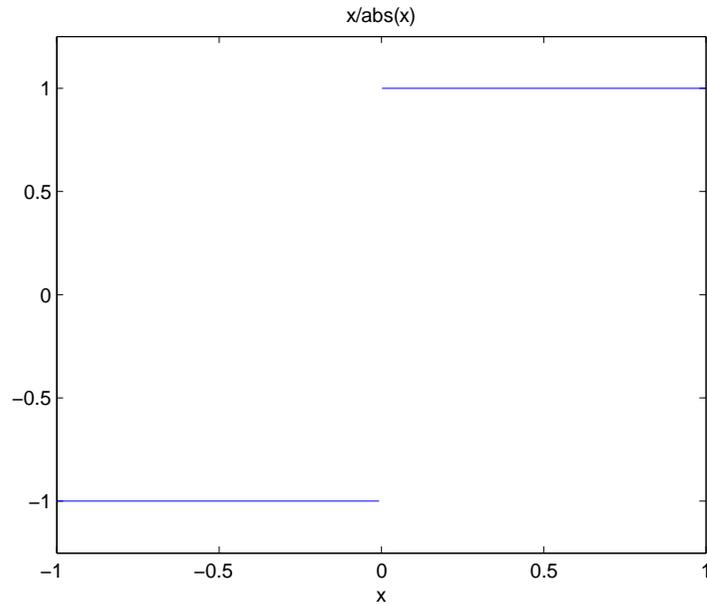
x/abs(x)

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)`

Mathematical Operation	MATLAB Command
$\displaystyle\lim_{x \to a^-} f(x)$	`limit(f,x,a,'left')`
$\displaystyle\lim_{x \to a^+} f(x)$	`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
$\displaystyle\int x^n dx = \begin{cases} \log(x) & \text{if } x = -1 \\ \dfrac{x^{n+1}}{n+1} & \text{otherwise.} \end{cases}$	`int(x^n)` or `int(x^n,x)`
$\displaystyle\int\limits_{0}^{\pi/2} \sin(2x)dx = 1$	`int(sin(2*x),0,pi/2)` or `int(sin(2*x),x,0,pi/2)`
$g = \cos(at + b)$ $\displaystyle\int g(t)dt = \sin(at+b)/a$	`g = cos(a*t + b) int(g)` or `int(g,t)`
$\displaystyle\int J_1(z)dz = -J_0(z)$	`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	piecewise([n = -1, log(x)], [n <> -1, x^(n + 1)/(n + 1)])
y^(-1)	log(y)
n^x	n^x/log(n)
sin(a*theta+b)	-cos(b + a*theta)/a
1/(1+u^2)	atan(u)
exp(-x^2)	(pi^(1/2)*erf(x))/2

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	pi^(1/2)/2
besselj(1,z)^2	0, 1	hypergeom([3/2,3/2],[2,5/2, 3],-1)/12

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

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)
```
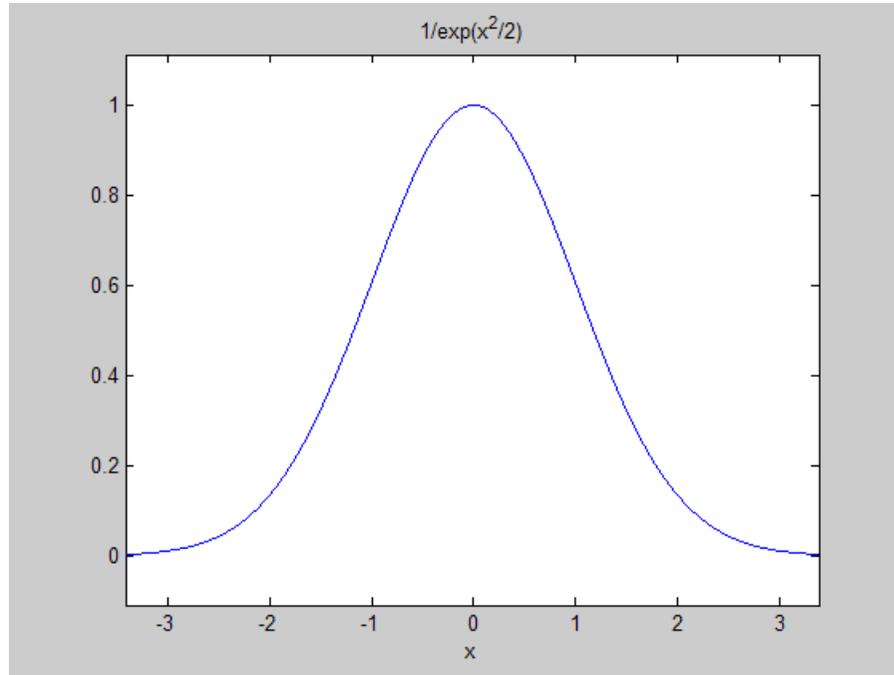
However, if you try to calculate the integral

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

without assigning a value to a, MATLAB assumes that a represents a complex number, and therefore returns a piecewise answer that depends on the argument of a. 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 =
pi^(1/2)/a^(1/2)
```

Integration with Complex Parameters

To calculate the integral

$$\int\limits_{-\infty}^{\infty} \frac{1}{a^2 + x^2} dx$$

for complex values of a, enter

```
syms a x clear
f = 1/(a^2 + x^2);
F = int(f, x, -inf, inf)
```

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

The preceding commands produce the complex output

```
F =
(pi*signIm(i/(a^2)^(1/2)))/(a^2)^(1/2)
```

The function signIm is defined as:

$$\text{signIm}(z) = \begin{cases} 1 & \text{if } \text{Im}(z) > 0, \text{ or } \text{Im}(z) = 0 \text{ and } z < 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{otherwise.} \end{cases}$$
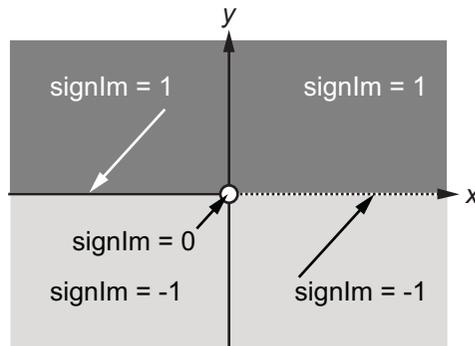
To evaluate F at `a = 1 + i`, enter

```
g = subs(F,1+i)

g =
pi/(2*i)^(1/2)

double(g)

ans =
   1.5708 - 1.5708i
```

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 $|x| < 1$. These summations are demonstrated below:

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

```
s2 = symsum(x^k,k,0,inf)

s1 =
pi^2/6

s2 =
piecewise([1 <= x, Inf], [abs(x) < 1, -1/(x - 1)])
```

Taylor Series

The statements

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

return

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

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 base point is a = 0.

The command

```
pretty(T)
```

prints T in a format resembling typeset mathematics:

```
      6        4       2
  49 x     5 x     2 x
 ------ + ---- + ---- + 1/9
 131220   1458    81
```

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.

t is a large expression; enter

```
size(char(t))

ans =
          1       99791
```

to find that t has nearly 100,000 characters in its printed form. In order to proceed with using t, first simplify its presentation:

```
t = simplify(t);
size(char(t))

ans =
          1       12137
```

To simplify t even further, use the simple function:

```
t = simple(t);
size(char(t))

ans =
          1        6988
```

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')
```

Taylor approximation vs. actual function
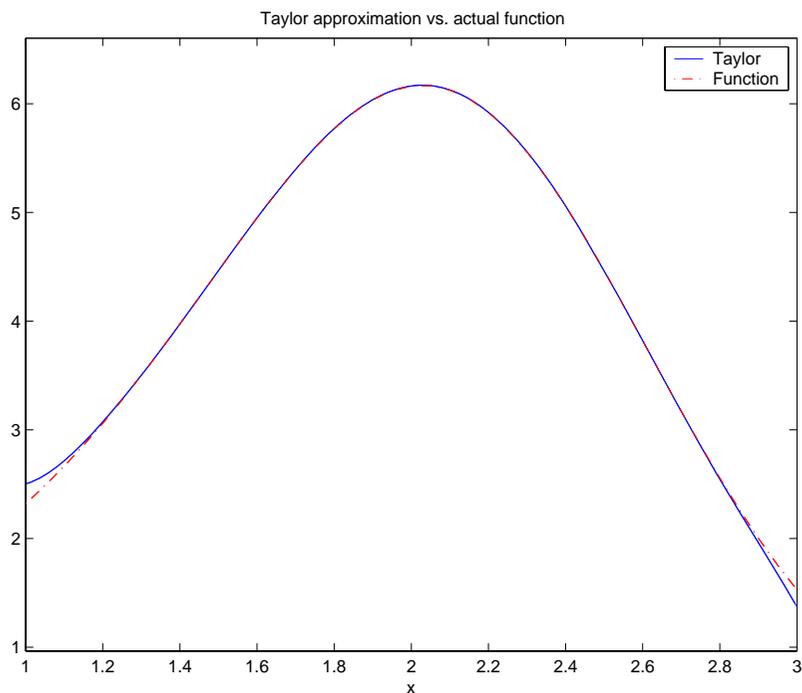
Special thanks to Professor Gunnar Bäckstrøm of UMEA in Sweden for this example.

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

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.

$$(3 x^2+6 x-1)/(x^2+x-3)$$
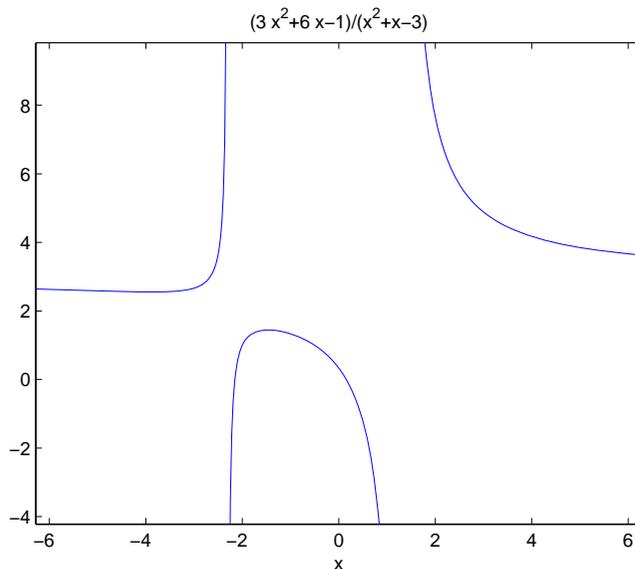
Finding the Asymptotes

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 =
```

```
   13^(1/2)/2 - 1/2
 - 13^(1/2)/2 - 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.

Horizontal and Vertical Asymptotes
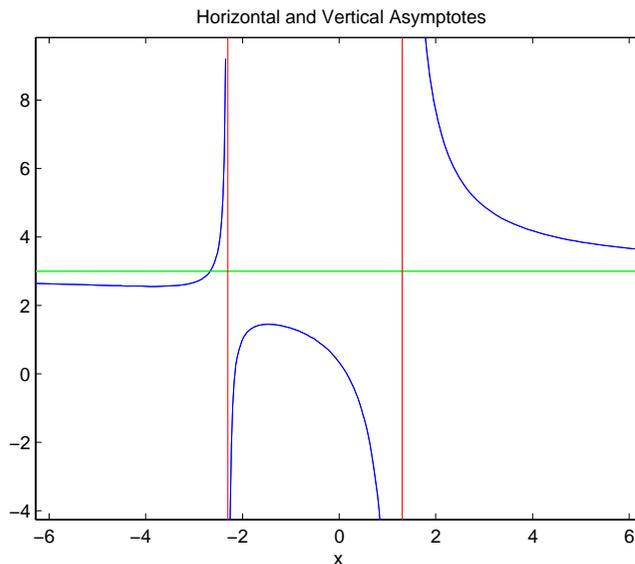
To recover the graph of f without the asymptotes, enter

```
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 $x = 0$, and might have a local minimum between $x = -6$ 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) - ((2*x + 1)*(3*x^2 + 6*x - 1))/(x^2 + x - 3)^2
```

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

```
crit_pts =

    13^(1/2)/3 - 8/3
  - 13^(1/2)/3 - 8/3
```

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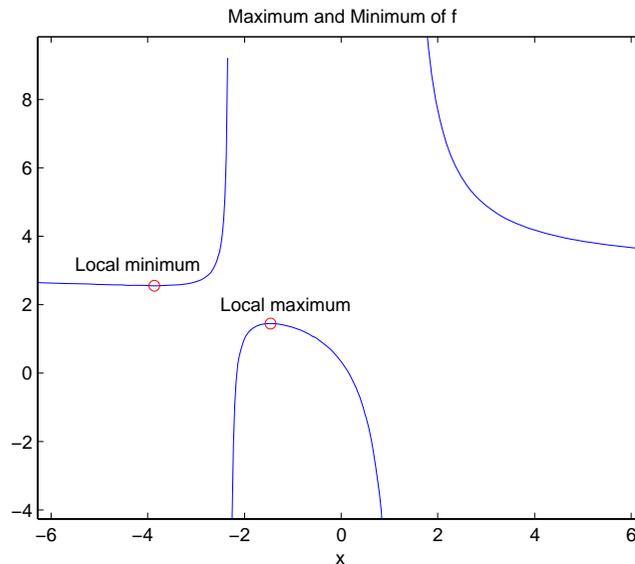
Finding the Inflection Point
To find the inflection point of f, set the second derivative equal to 0 and solve.

```
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 \1/3
                    13               |        2197     |
    - ------------------------- -  | 169/54 - ------- |    - 8/3
       /                1/2 \1/3   \           18      /
       |        2197     |
    9 | 169/54 - ------- |
       \           18      /
```

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.
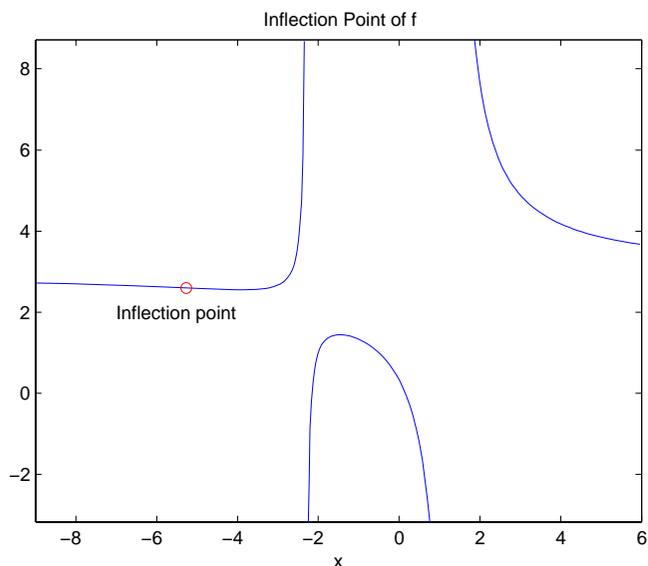
Extended Calculus Example
========================

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 2-28
- "Finding the Zeros of f3" on page 2-30
- "Finding the Maxima and Minima of f2" on page 2-33
- "Integrating" on page 2-35

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/(4*cos(x) + 5)
```

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

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')
```

Graph of f2
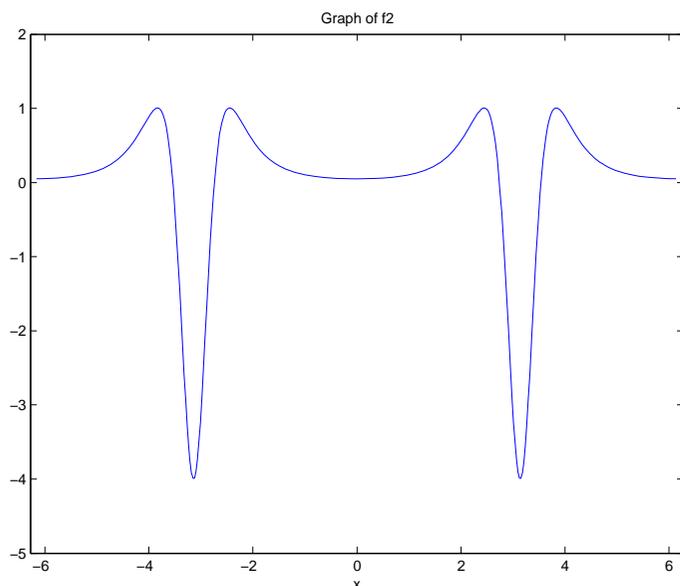
From the graph, it appears that the maximum value of $f''(x)$ is 1 and the minimum value is -4. As you will see, this is not quite true. To find the exact values of the maximum and minimum, you only need to find the maximum and minimum on the interval $(-\pi, \pi]$. This is true because $f''(x)$ is periodic 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
   384 sin(x)              4 sin(x)        96 cos(x) sin(x)
  --------------- - --------------- + ----------------
              4                2                        3
  (4 cos(x) + 5)    (4 cos(x) + 5)       (4 cos(x) + 5)
```

You can simplify this expression using the statements

```
f3 = simple(f3);
pretty(f3)
```

```
                        2
  4 sin(x) (- 16 cos(x)  + 80 cos(x) + 71)
  ----------------------------------------
                        4
              (4 cos(x) + 5)
```

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

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

This returns a 5-by-1 symbolic matrix

```
zeros =
 acos(5/2 - (3*19^(1/2))/4)
 acos((3*19^(1/2))/4 + 5/2)
                          0
  -acos(5/2 - 3/4*19^(1/2))
  -acos(3/4*19^(1/2) + 5/2)
```

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 =
   2.4483
        0 + 2.4381i
        0
```

```
      -2.4483
           0 - 2.4381i
```

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')
```
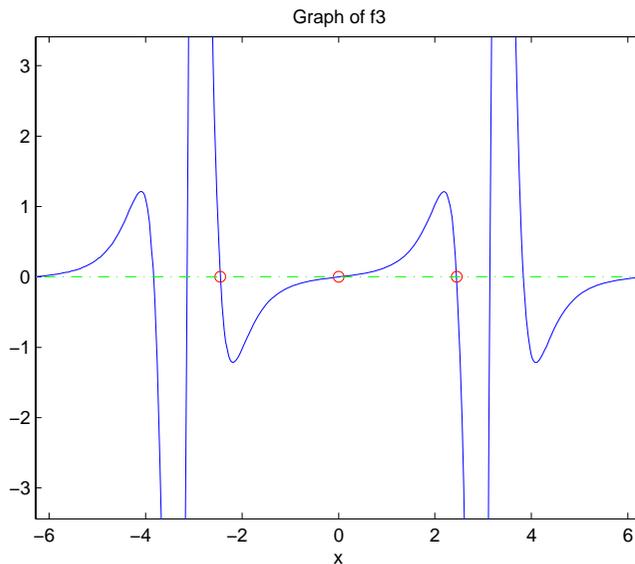

The red circles in the graph correspond to `zerosd(1)`, `zerosd(3)`, and `zerosd(4)`. As you can see in the graph, there are also zeros at $\pm\pi$. The additional zeros occur because $f'''(x)$ contains a factor of $\sin(x)$, which is zero at integer multiples of π. The function, `solve(sin(x))`, however, 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(3) zerosd(4) 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;
```
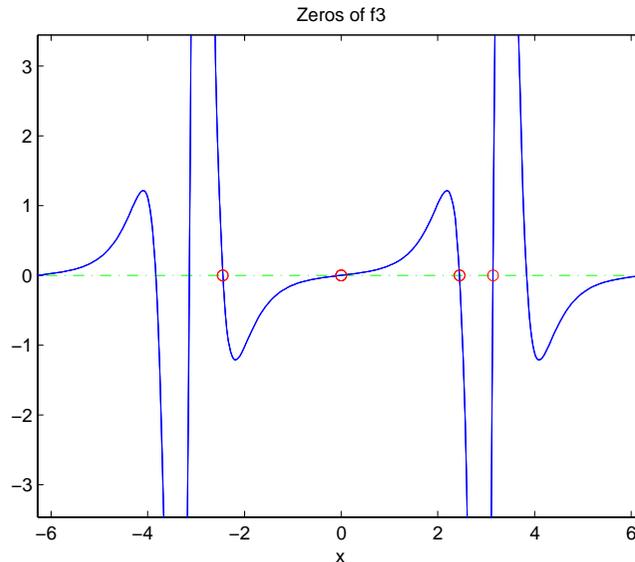

Zeros of f3

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 =
```

```
     2.4483          0    -2.4483     3.1416
     1.0051     0.0494     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 $x = 0$, 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(zerosd, subs(f2,zerosd), '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.

Maxima and Minima of f2
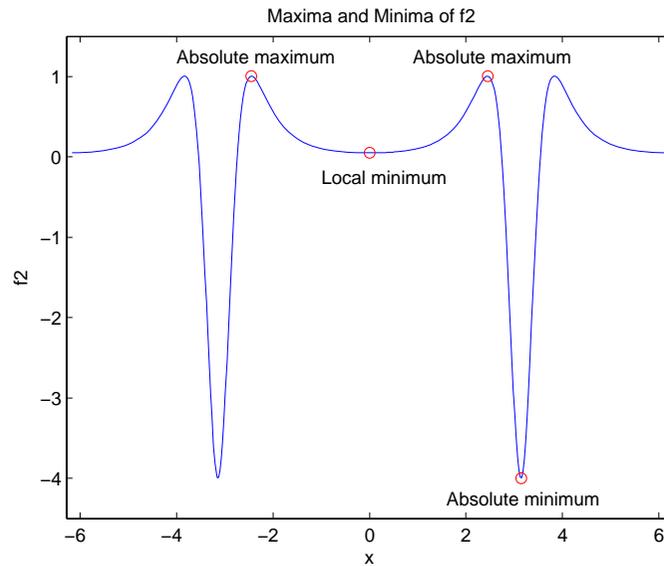
The preceding analysis shows that the actual range of $f''(x)$ is [–4, 1.0051].

Integrating

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

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

which returns

```
g =
-8/(tan(x/2)^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)
```

```
        8                1
   ------------- + ------------
      / x \2         4 cos(x) + 5
   tan| - |  + 9
      \ 2 /
```

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*atan(tan(x/2)/3))/3
```

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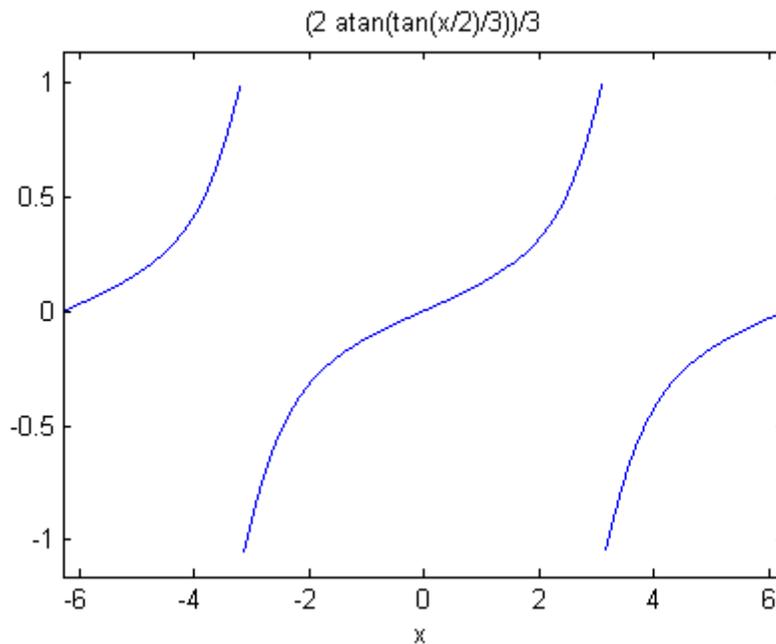
To change $F(x)$ into a true antiderivative of $f(x)$ that is differentiable everywhere, you can add a step function to $F(x)$. The height of the steps is the height of the gaps in the graph of $F(x)$. You can determine the height of the gaps by taking the limits of $F(x)$ as x approaches π from the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

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

On the other hand, the limit from the right is

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

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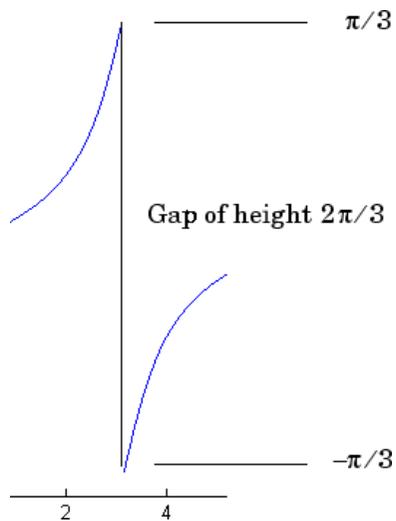
You can create the step function using the round function, which rounds numbers to the nearest integer, as follows:

```
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, generated with

```
ezplot(J,[-2*pi 2*pi])
```

(2 π round(x/(2 π)))/3
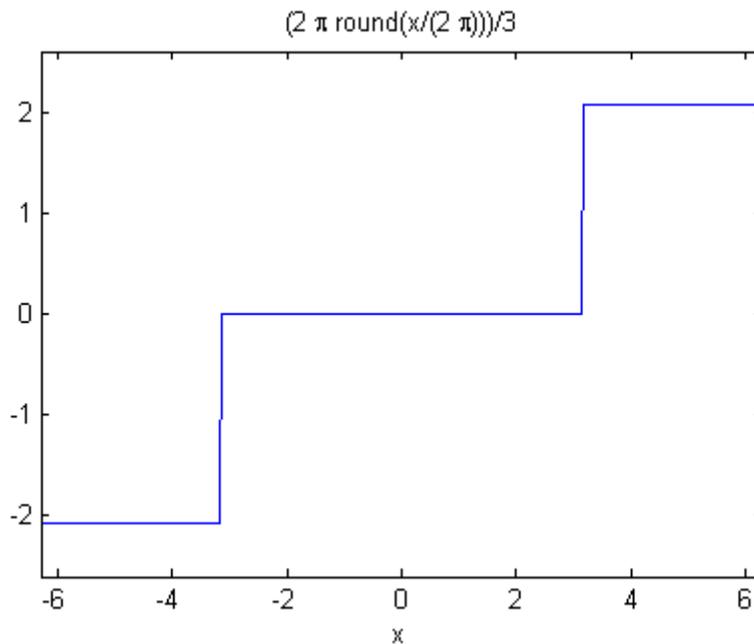
Next, add the step function *J*(*x*) to *F*(*x*) with the following code:

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

Adding the step function raises the section of the graph of *F*(*x*) on the interval [*π*, 3*π*) up by 2*π*/3, lowers the section on the interval (–3*π*, –*π*] down by 2*π*/3, and so on, as shown in the following figure.
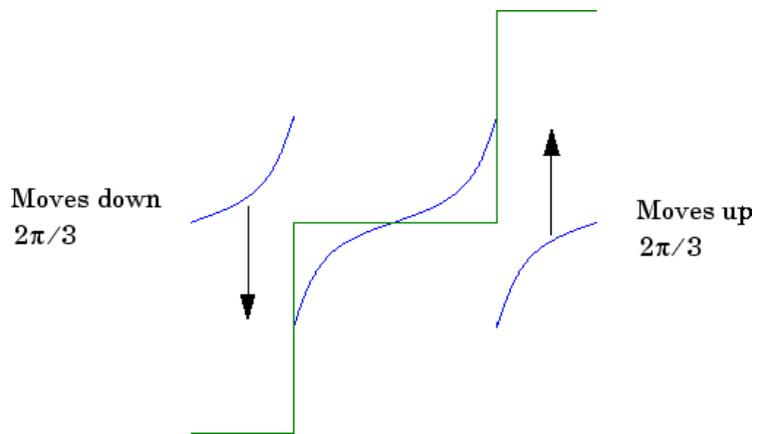
When you plot the result by entering

```
ezplot(F1)
```

you see that this representation does have a continuous graph.
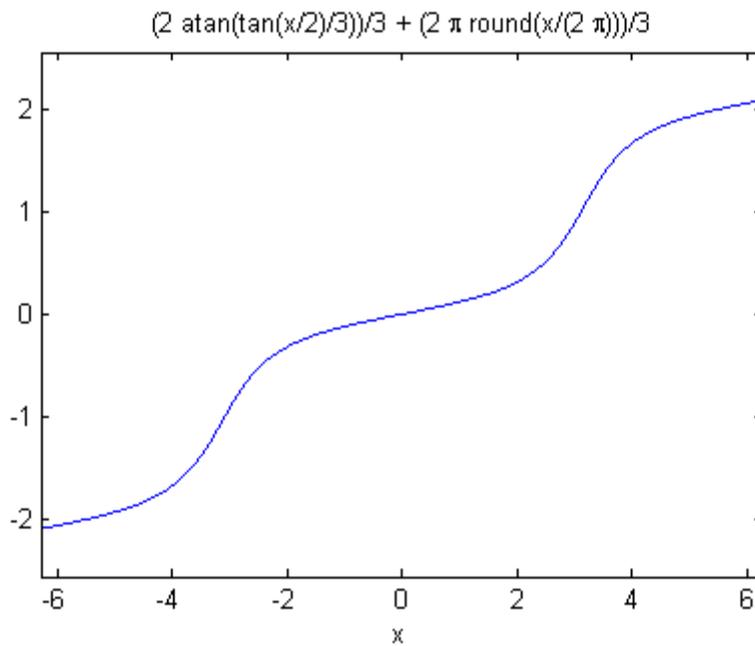
Simplifications and Substitutions

In this section...
"Simplifications" on page 2-41
"Substitutions" on page 2-50

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)

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

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`	`t + 2*t*x`

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))	3*x*(x^2 - 1) + x^3
3*x*(x^2 - 1) + x^3	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	x*(x*(x - 6) + 11) - 6
1.1+2.2*x+3.3*x^2	x*((33*x)/10 + 11/5) + 11/10

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 - 3)*(x - 1)*(x - 2)
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)*(x^6 - x^5 + x^4 - x^3 + x^2 - x + 1)]
[  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,   21649*513239]
```

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 - 1)*(x - 2)*(x - 3)`
`(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)`
`exp(x) * exp(y)`	`exp(x + y)`
`besselj(2,x) + besselj(0,x)`	`(2*besselj(1, x))/x`
`gamma(x+1)-x*gamma(x)`	`0`
`cos(x)^2 + sin(x)^2`	`1`

You can also use the syntax

```
simplify(f,n)
```

where n is a positive integer that controls how many steps `simplify` takes. The default, when you don't provide an argument n, is 50 steps.

For example,

```
z = diff(x/cos(x),3)

z =
3/cos(x) + (6*sin(x)^2)/cos(x)^3 + (6*x*sin(x)^3)/cos(x)^4 + (5*x*sin(x))/cos(x)^2
```

```
simplify(z)

ans =
(6*cos(x) - x*sin(x)*cos(x)^2 - 3*cos(x)^3 + 6*x*sin(x))/cos(x)^4

simplify(z,200)

ans =
(6*cos(x) - 3*cos(x)^3 + sin(x)*(6*x - x*cos(x)^2))/cos(x)^4
```

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

simplify(100):
1

combine(sincos):
1

combine(sinhcosh):
cos(x)^2 + sin(x)^2

combine(ln):
cos(x)^2 + sin(x)^2

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

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

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

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

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

rewrite(sinhcosh):
cosh(-i*x)^2 - sinh(-i*x)^2

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

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

mwcos2sin:
1
```

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 =
simplify
```

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 as shown:

f	simplify(f)	simple(f)
`syms a positive` `(1/a^3+6/a^2+12/a+8)^(1/3)`	`((2*a +` `1)^3)^(1/3)/a`	`1/a + 2`
`syms x` `cos(x) + i*sin(x)`	`cos(x) + i*sin(x)`	`exp(i*x)`

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

```
z = diff(x/cos(x),3)

z =
3/cos(x) + (6*sin(x)^2)/cos(x)^3 + (6*x*sin(x)^3)/cos(x)^4 + (5*x*sin(x))/cos(x)^2

simple(z)

ans =
(6*cos(x) - 3*cos(x)^3 + sin(x)*(6*x - x*cos(x)^2))/(sin(x)^2 - 1)^2

simple(simple(z))

ans =
(6*cos(x) - 3*cos(x)^3 + sin(x)*(6*x - x*cos(x)^2))/cos(x)^4
```

The `simple` function is particularly effective on expressions involving trigonometric functions:

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)+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 =

((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3) - a/(3*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3))

 a/(6*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)) - ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)/2...
 - (3^(1/2)*i*(a/(3*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)) + ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)))/2

 a/(6*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)) - ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)/2...
 + (3^(1/2)*i*(a/(3*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)) + ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3)))/2
```

This long expression has many repeated pieces, or subexpressions. 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 =
(a^3/27 + 1/4)^(1/2) - 1/2


r =

                                          sigma^(1/3) - a/(3*sigma^(1/3))
 a/(6*sigma^(1/3)) - sigma^(1/3)/2 - (3^(1/2)*i*(a/(3*sigma^(1/3)) + sigma^(1/3)))/2
 a/(6*sigma^(1/3)) - sigma^(1/3)/2 + (3^(1/2)*i*(a/(3*sigma^(1/3)) + sigma^(1/3)))/2
```

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 =
(a^3/27 + 1/4)^(1/2) - 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 = simplify(v)

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

E
E =

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

> **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

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

throughout v and E. First, use subexpr

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

which returns

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

E =
[ S,          0,  0]
[ 0, a + b + c,  0]
[ 0,          0, -S]
```

Next, substitute the symbol S into v with

```
v = simplify(subs(v,S,'S'))
v =
[ - (S*b - b^2 + a*c)/(S*c - c^2 + a*b) - 1, 1, - (b^2 + S*b - a*c)/(c^2 + S*c - a*b) - 1]
[       (S*b - b^2 + a*c)/(S*c - c^2 + a*b), 1,       (b^2 + S*b - a*c)/(c^2 + S*c - a*b)]
[                                        1, 1,                                         1]
```

Now suppose you want to evaluate v at a = 10. Use the subs command:

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

This replaces all occurrences of a in v with 10:

```
[ - (S*b - b^2 + 10*c)/(S*c - c^2 + 10*b) - 1, 1, - (b^2 + S*b - 10*c)/(c^2 + S*c - 10*b) - 1]
[       (S*b - b^2 + 10*c)/(S*c - c^2 + 10*b), 1,       (b^2 + S*b - 10*c)/(c^2 + S*c - 10*b)]
[                                          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	Attributes
A	3x3	622	sym	
E	3x3	1144	sym	
S	1x1	184	sym	
a	1x1	8	double	
ans	1x1	8	double	
b	1x1	8	double	
c	1x1	8	double	
v	3x3	1144	sym	

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.

$(1-t^2)\ \text{sech}(t)$
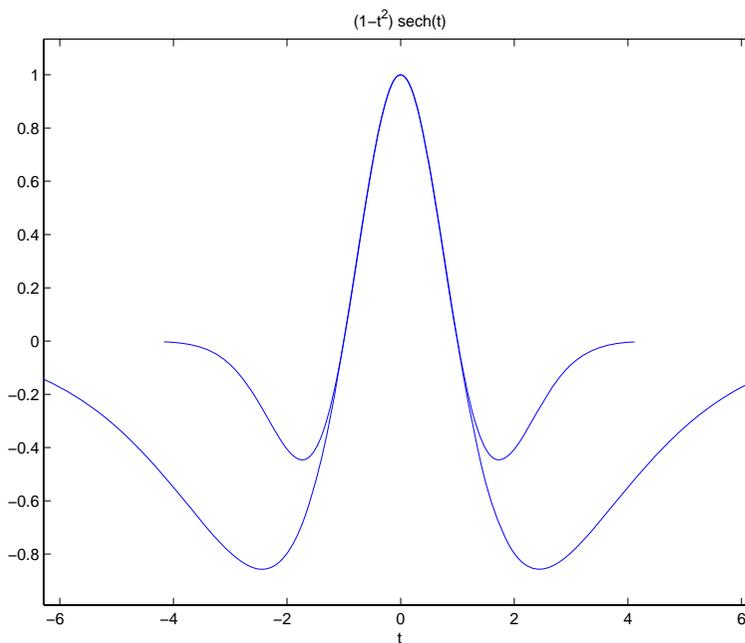
Instead, combine subs, double, and plot:

```
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`.
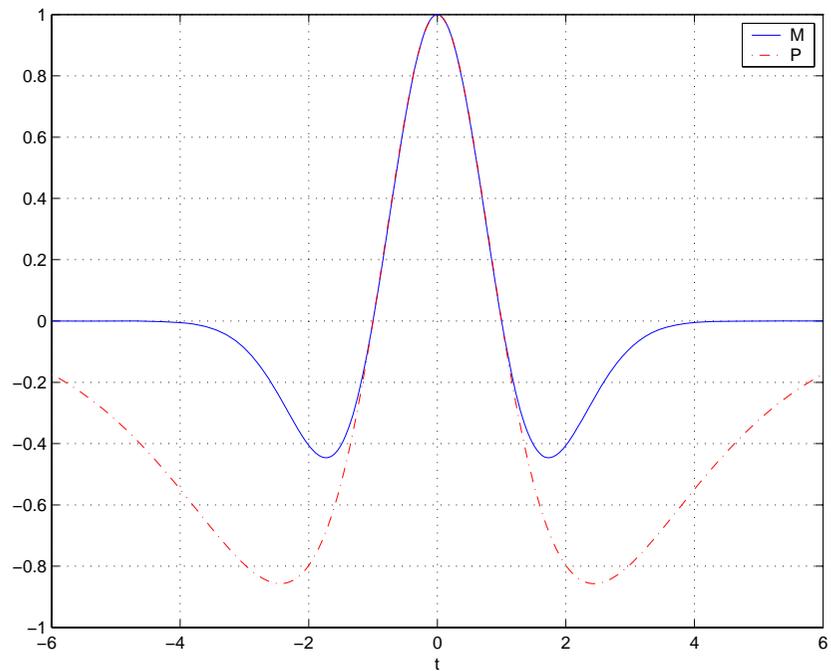
Finally the use of `subs` with strings greatly facilitates the solution of problems involving the Fourier, Laplace, or *z*-transforms. See the section "Integral Transforms and Z-Transforms" on page 2-95 for complete details.

Variable-Precision Arithmetic

Overview

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

Numeric	MATLAB floating-point arithmetic
Rational	MuPAD software exact symbolic arithmetic
VPA	MuPAD software variable-precision arithmetic

For example, the MATLAB statements

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

use numeric computation to produce

```
    0.833333333333333
```

With Symbolic Math Toolbox software, 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

```
0.833333333333333333333333
```

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.

Example: Using the Different Kinds of Arithmetic

Rational Arithmetic

By default, Symbolic Math Toolbox software uses rational arithmetic operations, i.e., MuPAD software'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 = [1.1,1.2,1.3;2.1,2.2,2.3;3.1,3.2,3.3]

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 =
    2.7183    1.6180
    1.0986    0.8147
```

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), 7338378580900475*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 =
[ 1.1, 1.2, 1.3]
[ 2.1, 2.2, 2.3]
[ 3.1, 3.2, 3.3]
```

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,  1.618033988749894902525739]
[  1.098612288668109560063613, 0.8147236863931789363491021]
```

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

```
262537412640768744.0
```

and, too,

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

returns

```
262537412640768744.0
```

You might suspect that f actually has 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)

```
262537412640768743.9999999999992500725972
```

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 software.

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

```
V =
[               1.0,               0.5, 0.3333333333333333]
[               0.5, 0.3333333333333333,               0.25]
[ 0.3333333333333333,              0.25,                0.2]
```

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 =
[   9.000000000000061,  -36.00000000000032,   30.0000000000003]
[  -36.00000000000032,   192.0000000000017, -180.0000000000015]
[    30.0000000000003, -180.0000000000015,  180.0000000000014]
```

shows the loss of two digits. So does

```
1/det(V)
```

which gives

```
2160.000000000018
```

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 =
s/240 - 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 the message

```
ans =
 FAIL
```

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

```
Z =
3/10
 -6/5
    1
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

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

results in 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 =
[ 3/10, 218/285 - (4*12589^(1/2))/285, (4*12589^(1/2))/285 + 218/285]
[ -6/5,     292/285 - 12589^(1/2)/285,    12589^(1/2)/285 + 292/285]
[    1,                             1,                             1]
```

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

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

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 =
     0.3000    -0.8098     2.3397
    -1.2000     0.6309     1.4182
     1.0000     1.0000     1.0000

Ed =
          0          0          0
          0     0.0878          0
          0          0     1.3344
```

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 - 12589^(1/2)/180
 12589^(1/2)/180 + 32/45
```

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(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  - 1040500) (x  - 1020 x + 100) (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
                 1000
                 1000
                 1020
    510 - 100*26^(1/2)
```

```
    100*26^(1/2) + 510
     (-10)*10405^(1/2)
        10*10405^(1/2)
```

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 =
    1020.4201882015047278185457449884
    1019.9935501291629257348091808173
    1019.5243552632016358324933278291
    1000.1206982933841335712817075454
    999.94691786044276755320289228602
    0.21803980548301606860857564424981
   -0.17053529728768998575200874607757
     -1020.0532142558915165931894252 6
```

Notice that these values are close to the eigenvalues of the original Rosser matrix. Further, the numerical values of F are a result of MuPAD software'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

```
G =
[    exp(i*t)/2 + 1/(2*exp(i*t)),i/(2*exp(i*t)) - (i*exp(i*t))/2]
[(i*exp(i*t))/2 - i/(2*exp(i*t)),    exp(i*t)/2 + 1/(2*exp(i*t))]
```

You can simplify this expression with the `simple` command:

```
[G,how] = simple(G)

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

how =
simplify(100)
```

Next, the command

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

produces

```
g =
 cos(t) - i*sin(t)
 cos(t) + i*sin(t)
```

You can use `simple` to simplify this form of g:

```
[g,how] = simple(g)

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

how =
rewrite(exp)
```

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 roundoff 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

```
n = 5;
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 =
 3.14159265358979323846255035973
 3.14159265358979323843066846713
 3.14159265358979323325290142782
 3.14159265358979270342635559052
  3.14159265358975439206849990722
 3.14159265358763617123926123382
 3.14159265349961053143856838564
 3.14159265052654880815569479613
 3.14159256925492306470284863101
 3.14159075458605848728982577118
  3.14155754359918083691050658226
 3.14106044663470063805218371923
 3.13504054399744654843898901261
 3.07790297231119748658424727353
 2.69162158686066606774782763593
 1.20968137605668985332455685355
```

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

```
9.3024e-023           0
3.1975e-020           0
5.2097e-018           0
5.3504e-016   4.4409e-016
3.8846e-014   3.8636e-014
2.1196e-012   2.1196e-012
9.0183e-011   9.0183e-011
3.0632e-009   3.0632e-009
8.4335e-008   8.4335e-008
1.8990e-006   1.8990e-006
3.5110e-005   3.5110e-005
5.3221e-004   5.3221e-004
6.5521e-003   6.5521e-003
6.3690e-002   6.3690e-002
4.4997e-001   4.4997e-001
1.9319e+000   1.9319e+000
```

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 $t : A \rightarrow A + tE$. As

t increases from 0 to 10^{-6}, the eigenvalues $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$ change to

$\lambda_1{}' = 1.5596 + 0.2726i$, $\lambda_2{}' = 1.5596 - 0.2726i$, $\lambda_3{}' = 2.8808$.
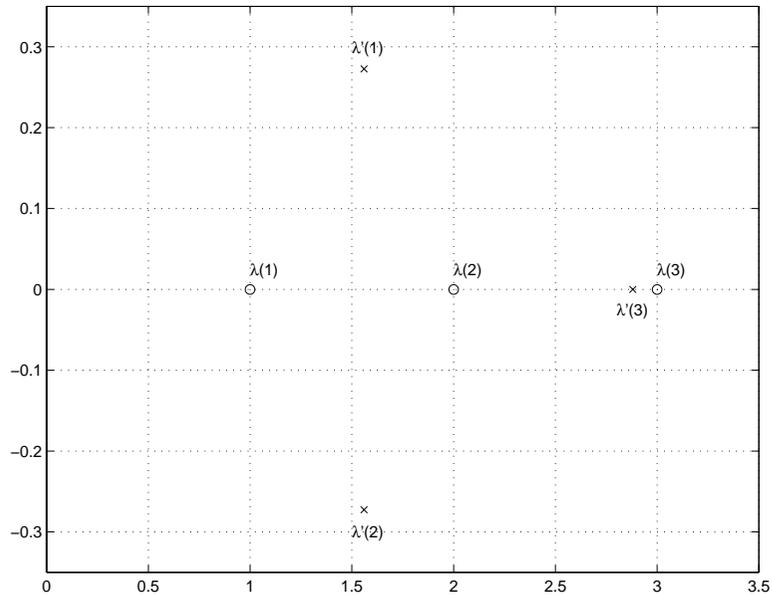
This, in turn, means that for some value of $t = \tau$, $0 < \tau < 10^{-6}$, the perturbed matrix $A(t) = A + tE$ has a double eigenvalue $\lambda_1 = \lambda_2$. The example shows how to find the value of t, called τ, where this happens.

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 =
   1.000000000010722
   1.999999999991790
   2.999999999997399
```

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

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 the perturbation matrix:

```
E = [130,-390,0;43,-129,0;133,-399,0]
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 =
[130*t - 149, - 390*t - 50, -154]
[ 43*t + 537,  180 - 129*t,  546]
[ 133*t - 27,  - 399*t - 9,  -25]
```

Computing the characteristic polynomial of this new A

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

gives

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

p is a cubic in x whose coefficients vary linearly with 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
```
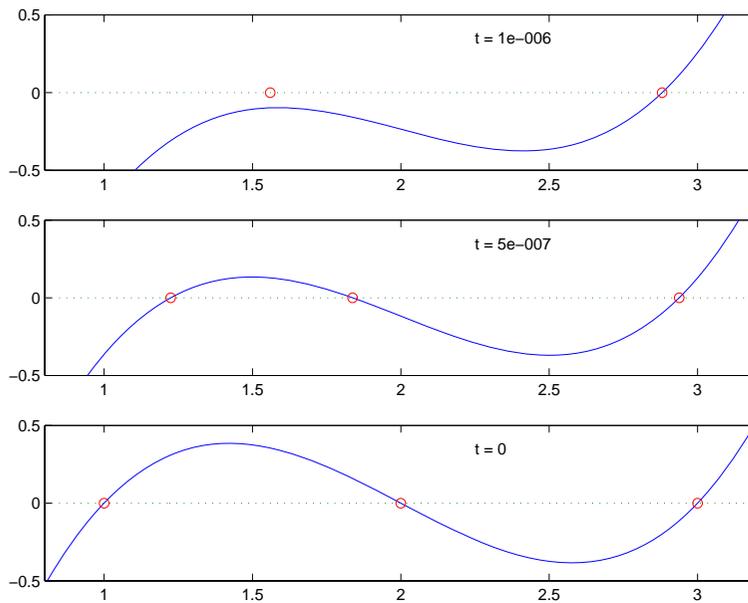

The bottom subplot shows the unperturbed polynomial, with its three roots at 1, 2, and 3. The middle subplot shows the first two roots approaching each other. In the top subplot, these two roots have become complex and only one real root remains.

The next statements compute and display the actual eigenvalues

```
e = eig(A);
ee = subexpr(e);

sigma =
(354377*t)/3 + ((39821416990007660*t^3)/9 - (20213598755*t^4)/9 -...
 (350943215806*t^2)/27 + (492508*t)/9 - 1/27)^(1/2) - (246254*t^2)/3 + t^3/27


pretty(ee)
```

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

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

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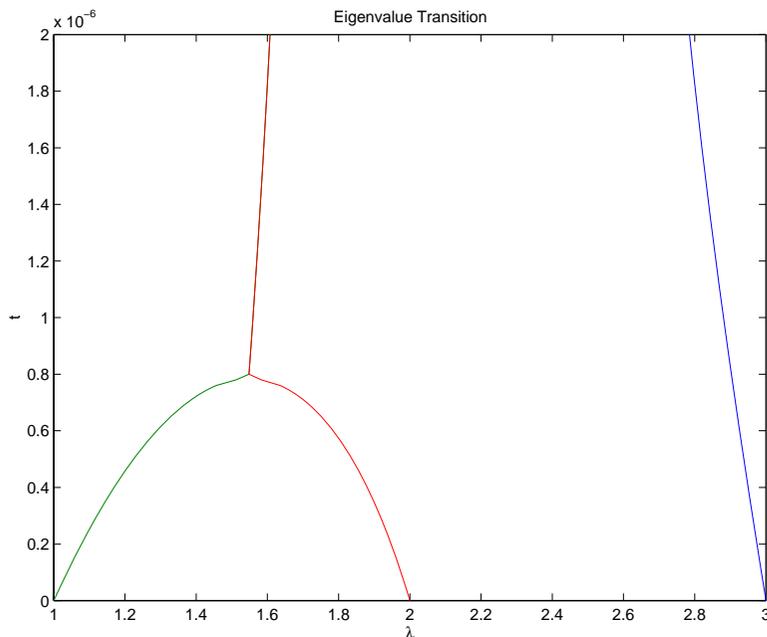
Above t = 0.8e^{-6}, the graphs of two of the eigenvalues intersect, while below t = 0.8e^{-6}, two real roots become a complex conjugate pair. What is the precise value of t that marks this transition? Let τ denote this value of t.

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 the MuPAD language. The statement

```
doc(symengine,'discrim')
```

gives theMuPAD help for the function.

MuPAD documentation

Provide feedback about this page

polylib::discrim – discriminant of a polynomial

`polylib::discrim(p, x)` returns the discriminant of the polynomial `p` with respect to the variable `x`.

→ **Examples**

Call:

`polylib::discrim(p, x)`

Parameters:

This shows that the `discrim` function is in the `polylib` library. Use these commands

```
syms a b c x
evalin(symengine,'polylib::discrim(a*x^2+b*x+c, x)')
```

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

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

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

```
p = sym(char(p));
discrim = feval(symengine,'polylib::discrim',p,x)
```

which produces

```
discrim =
242563185060*t^4 - 477857003880091920*t^3 +...
 1403772863224*t^2 - 5910096*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 =


 0.00000107692481604921513380768 - 0.0000030854463652460611912112*i
 0.0000030854463652460611912112*i + 0.00000107692481604921513380768
                              0.00000078379249059679401048084
                                  1970031.04061804553618914
```

Of the four solutions, you know that

```
tau = tau(3)
```

is the transition point

```
tau =
0.00000078379249059679401048084
```

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(4))
```

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

```
tau =
   7.8379e-007 +7.6609e-062i
```

Therefore, the fourth element of `sol.x`

```
sigma = double(sol.x(4))
```

is the double eigenvalue

```
sigma =
   1.5476 + 0.0000i
```

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 - 0.0000i
```

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

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

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 three solutions.

```
s =
          0
       pi/6
   (5*pi)/6
```

There are also solutions at each of these results plus $k\pi$ for integer k, as you can see in the MuPAD solution:

$$\left[\begin{array}{l} \texttt{solve(cos(2*x)+sin(x)=1)} \\[4pt] (x) \in \left\{ \left(\pi\, k \right) \,\middle|\, k \in \mathbb{Z} \right\} \cup \left\{ \left(\tfrac{\pi}{6} + 2\,\pi\, k \right) \,\middle|\, k \in \mathbb{Z} \right\} \cup \left\{ \left(\tfrac{5\,\pi}{6} + 2\,\pi\, k \right) \,\middle|\, k \in \mathbb{Z} \right\} \end{array}\right.$$

Several Algebraic Equations

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

$$x^2 y^2 = 0$$
$$x - \frac{y}{2} = \alpha,$$

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 =
 alpha
     0

y =
           0
 (-2)*alpha
```

Modify the first equation to $x^2 y^2 = 1$ and there are more solutions.

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

produces four distinct solutions:

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

y =
    (alpha^2 + 2)^(1/2) - alpha
    (alpha^2 - 2)^(1/2) - alpha
 - 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 =
 -1
  3
```

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 =
[  3,  5, -4]
```

The following statement

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

creates the solution matrix M

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

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 =

 (2*v)/3 - (5*u)/3
     (4*u)/3 - v/3

z =
 (2*v)/3 - (5*u)/3
     (4*u)/3 - v/3
```

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 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=t*y')
```

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

The output of this command is

```
ans =
C2*exp(t^2/2)
```

`y = C*exp(t^2/2)` is a solution to the equation for any constant `C`.

To specify an initial condition, use

```
y = dsolve('Dy=t*y', 'y(0)=2')
```

This produces

```
y =
2*exp(t^2/2)
```

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, enter `syms t`.

Example 2

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

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

results in

```
x =
 1/exp(t) - 1
 1 - 1/exp(t)
```

Example 3

Here is a second-order differential equation with two initial conditions, and the default independent variable changed to x. The commands

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

produce

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

Example 4

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

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

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

with x as the independent variable, 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 $u''(0)$.

```
u =
(pi*exp(x))/3 - (cos((3^(1/2)*x)/2)*(pi/3 - 1))/exp(x/2) ...
- (3^(1/2)*sin((3^(1/2)*x)/2)*(pi + 1))/(3*exp(x/2))
```

Further ODE Examples

This table shows a few more examples of differential equations and their Symbolic Math Toolbox syntax. The final entry in the table is the Airy differential equation, whose solution is referred to as the Airy function.

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

$\dfrac{d^2y}{dx^2} = xy(x)$
 	
$y(0) = 0,\ y(3) = \dfrac{1}{\pi}K_{1/3}(2\sqrt{3})$
 	
(The Airy equation)
	```y = dsolve('D2y = x*y','y(0) = 0',```
	 ```'y(3) = besselk(1/3, 2*sqrt(3))/pi',```
	 ```'x')```

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 =
(C2*i)/exp(t*(4*i - 3)) - C1*i*exp(t*(4*i + 3))

g = S.g
g =
C1*exp(t*(4*i + 3)) + C2/exp(t*(4*i - 3))
```

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 =
i/(2*exp(t*(4*i - 3))) - (i*exp(t*(4*i + 3)))/2

g =
exp(t*(4*i + 3))/2 + 1/(2*exp(t*(4*i - 3)))

f = simplify(f)

f =
sin(4*t)*exp(3*t)

g = simplify(g)

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

Integral Transforms and Z-Transforms

The Fourier and Inverse Fourier Transforms

The Fourier transform of a function $f(x)$ is defined as

$$F[f](w) = \int_{-\infty}^{\infty} f(x)e^{-iwx}dx,$$

and the inverse Fourier transform (IFT) as

$$F^{-1}[f](x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(w)e^{iwx}du.$$

We refer to this formulation as the Fourier transform of f with respect to x as a function of w. Or, more concisely, the Fourier transform of f with respect to x at w. Mathematicians often use the notation $F[f]$ to denote the Fourier transform of f. In this setting, the transform is taken with respect to the independent variable of f (if $f = f(t)$, then t is the independent variable; $f = f(x)$ implies that x is the independent variable, etc.) at the default variable w. We refer to $F[f]$ as the Fourier transform of f at w and $F^{-1}[f]$ is the IFT of f at x. See fourier and ifourier in the reference pages for tables that show the Symbolic Math Toolbox commands equivalent to various mathematical representations of the Fourier and inverse Fourier transforms.

For example, consider the Fourier transform of the Cauchy density function, $(\pi(1 + x^2))^{-1}$:

```
syms x
cauchy = 1/(pi*(1+x^2));
fcauchy = fourier(cauchy)
```

```
fcauchy =
((pi*heaviside(w))/exp(w) + pi*heaviside(-w)*exp(w))/pi

fcauchy = expand(fcauchy)

fcauchy =
heaviside(w)/exp(w) + heaviside(-w)*exp(w)

ezplot(fcauchy)
```
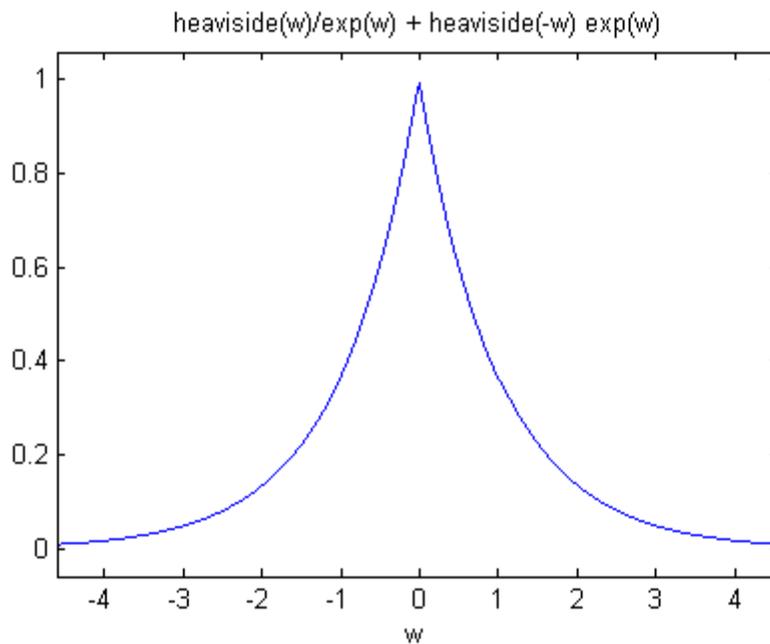

The Fourier transform is symmetric, since the original Cauchy density function is symmetric.

To recover the Cauchy density function from the Fourier transform, call `ifourier`:

```
finvfcauchy = ifourier(fcauchy)
```

```
finvfcauchy =
-(1/(i*x - 1) - 1/(i*x + 1))/(2*pi)

simplify(finvfcauchy)

ans =
1/(pi*(x^2 + 1))
```

An application of the Fourier transform is the solution of ordinary and partial differential equations over the real line. Consider the deformation of an infinitely long beam resting on an elastic foundation with a shock applied to it at a point. A "real world" analogy to this phenomenon is a set of railroad tracks atop a road bed.
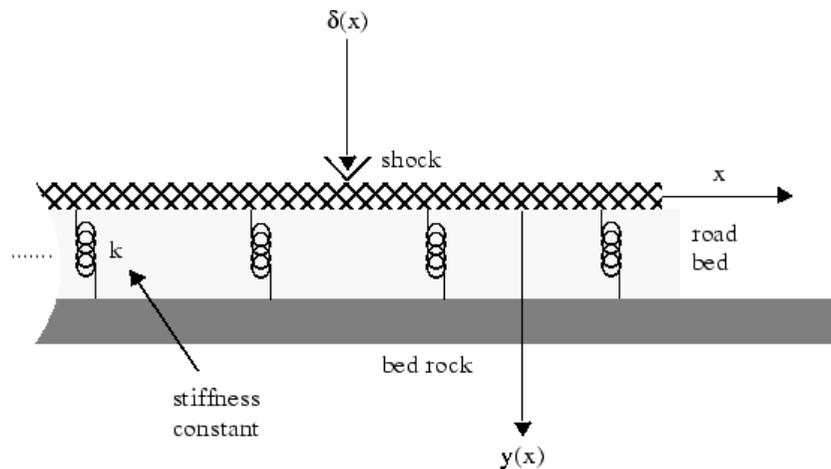
The shock could be induced by a pneumatic hammer blow (or a swing from the mighty John Henry!).

The differential equation idealizing this physical setting is

$$\frac{d^4y}{dx^4} + \frac{k}{EI}y = \frac{1}{EI}\delta(x), \quad -\infty < x < \infty.$$

Here, E represents elasticity of the beam (rail road track), I is the "beam constant," and k is the spring (road bed) stiffness. The shock force on the

right hand side of the differential equation is modeled by the Dirac Delta function $\delta(x)$. If you are unfamiliar with $\delta(x)$, you may be surprised to learn that (despite its name), it is not a function at all. Rather, $\delta(x)$ is an example of what mathematicians call a *distribution*. The Dirac Delta function (named after the physicist Paul Dirac) has the following important property

$$\int_{-\infty}^{\infty} f(x-y)\delta(y)dy = f(x).$$

A definition of the Dirac Delta function is

$$\delta(x) = \lim_{n\to\infty} n\chi_{(-1/2n,1/2n)}(x),$$

where

$$\chi_{(-1/2n,1/2n)}(x) = \begin{cases} 1 & \text{for } -\dfrac{1}{2n} < x < \dfrac{1}{2n} \\ 0 & \text{otherwise.} \end{cases}$$

You can evaluate the Dirac Delta function at a point (say) $x = 3$, using the commands

```
syms x
del = sym('dirac(x)');
vpa(subs(del,x,3))
```

which return

```
ans =
0.0
```

Returning to the differential equation, let $Y(w) = F[y(x)](w)$ and $\Delta(w) = F[\delta(x)](w)$. Indeed, try the command `fourier(del,x,w)`. The Fourier transform turns differentiation into exponentiation, and, in particular,

$$F\left[\frac{d^4y}{dx^4}\right](w) = w^4 Y(w).$$

To see a demonstration of this property, try this

```
syms w x
fourier(diff(sym('y(x)'),x,4),x,w)
```

which returns

```
w^4*transform::fourier(y(x), x, -w)
```

Note that you can call the `fourier` command with one, two, or three inputs (see the reference pages for `fourier`). With a single input argument, `fourier(f)` returns a function of the default variable `w`. If the input argument is a function of `w`, `fourier(f)` returns a function of `t`. All inputs to `fourier` must be symbolic objects.

We now see that applying the Fourier transform to the differential equation above yields the algebraic equation

$$\left(w^4 + \frac{k}{EI} \right) Y(w) = \Delta(w),$$

or

$$Y(w) \; = \; \Delta(w)G(w),$$

where

$$G(w) = \frac{1}{w^4 + \dfrac{k}{EI}} = F\big[g(x)\big](w)$$

for some function $g(x)$. That is, g is the inverse Fourier transform of G:

$$g(x) \; = \; F^{-1}[G(w)](x)$$

The Symbolic Math Toolbox counterpart to the IFT is `ifourier`. This behavior of `ifourier` parallels `fourier` with one, two, or three input arguments (see the reference pages for `ifourier`).

Continuing with the solution of our differential equation, we observe that the ratio

$$\frac{K}{EI}$$

is a relatively "large" number since the road bed has a high stiffness constant k and a rail road track has a low elasticity E and beam constant I. We make the simplifying assumption that

$$\frac{K}{EI} = 1024.$$

This is done to ease the computation of $F^{-1}[G(w)](x)$. Proceeding, we type

```
G = 1/(w^4 + 1024);
g = ifourier(G,w,x);
g = simplify(g);
pretty(g)
```

and see

```
    1/2    / PI      \                          1/2                / PI      \
   2    sin| -- + 4 x | exp(-4 x) heaviside(x)  2    heaviside(-x) sin| -- - 4 x | exp(4 x)
        \  4        /                                                \  4        /
 ----------------------------------------- + -----------------------------------------
                    512                                          512
```

Notice that g contains the Heaviside distribution

$$H(x) = \begin{cases} 1 & \text{for} \quad x > 0 \\ 0 & \text{for} \quad x < 0 \\ 1/2 & \text{for} \quad x = 0. \end{cases}$$

Since Y is the product of Fourier transforms, y is the convolution of the transformed functions. That is, $F[y] = Y(w) = \Delta(w)\ G(w) = F[\delta]\ F[g]$ implies

$$y(x) = (\delta * g)(x) = \int\limits_{-\infty}^{\infty} g(x - y)\delta(y)dy = g(x).$$

by the special property of the Dirac Delta function. To plot this function, we must substitute the domain of x into $y(x)$, using the subs command.

```
XX = -3:0.05:3;
YY = double(subs(g,x,XX));
plot(XX,YY)
title('Beam Deflection for a Point Shock')
xlabel('x'); ylabel('y(x)');
```

The resulting graph
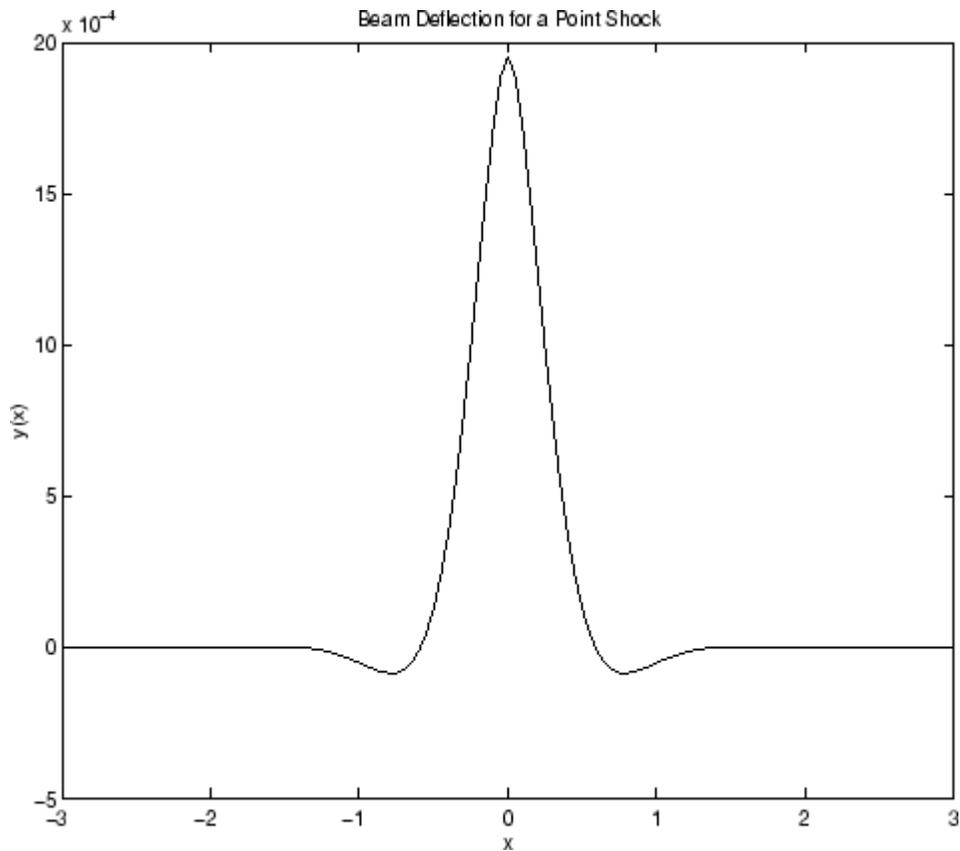
shows that the impact of a blow on a beam is highly localized; the greatest deflection occurs at the point of impact and falls off sharply from there. This is the behavior we expect from experience.

The Laplace and Inverse Laplace Transforms

The Laplace transform of a function $f(t)$ is defined as

$$L[f](s) = \int_0^\infty f(t)e^{-ts}dt,$$

while the inverse Laplace transform (ILT) of $f(s)$ is

$$L^{-1}[f](t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} f(s)e^{st}ds,$$

where c is a real number selected so that all singularities of $f(s)$ are to the left of the line $s = c$. The notation $L[f]$ denotes the Laplace transform of f at s. Similarly, $L^{-1}[f]$ is the ILT of f at t.

The Laplace transform has many applications including the solution of ordinary differential equations/initial value problems. Consider the resistance-inductor-capacitor (RLC) circuit below.
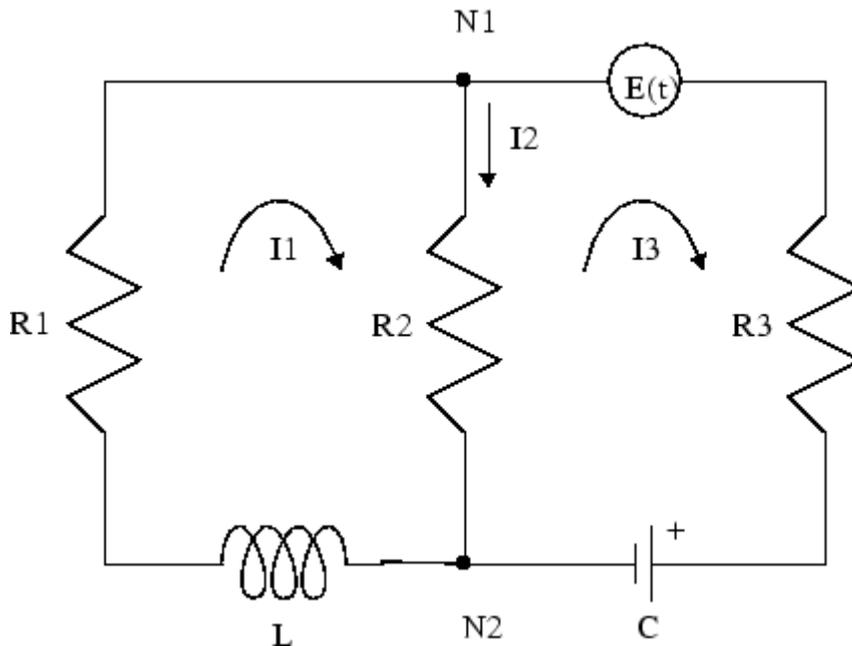
Let Rj and Ij, $j = 1, 2, 3$ be resistances (measured in ohms) and currents (amps), respectively; L be inductance (henrys), and C be capacitance (farads); $E(t)$ be the electromotive force, and $Q(t)$ be the charge.

By applying Kirchhoff's voltage and current laws, Ohm's Law, Faraday's Law, and Henry's Law, you can arrive at the following system of simultaneous ordinary differential equations.

$$\frac{dI_1}{dt} + \frac{R_2}{L}\frac{dQ}{dt} = \frac{R_1 - R_2}{L}I_1, \ I_1(0) = I_0.$$

$$\frac{dQ}{dt} = \frac{1}{R_3 + R_2}\left(E(t) - \frac{1}{C}Q(t)\right) + \frac{R_2}{R_3 + R_2}I_1, \ Q(0) = Q_0.$$

Let's solve this system of differential equations using `laplace`. We will first treat the R_j, L, and C as (unknown) real constants and then supply values later on in the computation.

```
syms R1 R2 R3 L C real
dI1 = sym('diff(I1(t),t)'); dQ = sym('diff(Q(t),t)');
I1 = sym('I1(t)'); Q = sym('Q(t)');
syms t s
E = sin(t);  % Voltage
eq1 = dI1 + R2*dQ/L - (R2 - R1)*I1/L;
eq2 = dQ - (E - Q/C)/(R2 + R3) - R2*I1/(R2 + R3);
```

At this point, we have constructed the equations in the MATLAB workspace. An approach to solving the differential equations is to apply the Laplace transform, which we will apply to `eq1` and `eq2`. Transforming `eq1` and `eq2`

```
L1 = laplace(eq1,t,s)
L2 = laplace(eq2,t,s)
```

returns

```
L1 =
s*laplace(I1(t), t, s) - I1(0)
+ ((R1 - R2)*laplace(I1(t), t, s))/L
- (R2*(Q(0) - s*laplace(Q(t), t, s)))/L

L2 =
s*laplace(Q(t), t, s) - Q(0)
- (R2*laplace(I1(t), t, s))/(R2 + R3) - (C/(s^2 + 1)
- laplace(Q(t), t, s))/(C*(R2 + R3))
```

Now we need to solve the system of equations L1 = 0, L2 = 0 for
`laplace(I1(t),t,s)` and `laplace(Q(t),t,s)`, the Laplace transforms of I_1
and Q, respectively. To do this, we need to make a series of substitutions. For
the purposes of this example, use the quantities $R1 = 4\ \Omega$ (ohms), $R2 = 2\ \Omega$,
$R3 = 3\ \Omega$, $C = 1/4$ farads, $L = 1.6$ H (henrys), $I1(0) = 15$ amps, and $Q(0) = 2$
amps/sec. Substituting these values in L1

```
syms LI1 LQ
NI1 = subs(L1,{R1,R2,R3,L,C,'I1(0)','Q(0)'}, ...
      {4,2,3,1.6,1/4,15,2})
```

returns

```
NI1 =
s*laplace(I1(t), t, s) + (5*s*laplace(Q(t), t, s))/4
 + (5*laplace(I1(t), t, s))/4 - 35/2
```

The substitution

```
NQ = subs(L2,{R1,R2,R3,L,C,'I1(0)','Q(0)'},{4,2,3,1.6,1/4,15,2})
```

returns

```
NQ =
s*laplace(Q(t), t, s) - 1/(5*(s^2 + 1))
+ (4*laplace(Q(t), t, s))/5 - (2*laplace(I1(t), t, s))/5 - 2
```

To solve for `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)`, we make a final
pair of substitutions. First, replace the strings `'laplace(I1(t),t,s)'` and
`'laplace(Q(t),t,s)'` by the syms LI1 and LQ, using

```
NI1 =
subs(NI1,{'laplace(I1(t),t,s)','laplace(Q(t),t,s)'},{LI1,LQ})
```

to obtain

```
NI1 = ...
(5*LI1)/4 + LI1*s + (5*LQ*s)/4 - 35/2
```

Collecting terms

```
NI1 = collect(NI1,LI1)
```

gives

```
NI1 =
(5*LQ*s)/4 + LI1*(s + 5/4) - 35/2
```

A similar string substitution

```
NQ = ...
subs(NQ,{'laplace(I1(t),t,s)','laplace(Q(t),t,s)'},{LI1,LQ})
```

yields

```
NQ =
(4*LQ)/5 - (2*LI1)/5 + LQ*s - 1/(5*(s^2 + 1)) - 2
```

which, after collecting terms,

```
NQ = collect(NQ,LQ)
```

gives

```
NQ =
LQ*(s + 4/5) - (2*LI1)/5 - 1/(5*(s^2 + 1)) - 2
```

Now, solving for LI1 and LQ

```
[LI1,LQ] = solve(NI1,NQ,LI1,LQ)
```

we obtain

```
LI1 =
(300*s^3 + 280*s^2 + 295*s + 280)/(20*s^4 + 51*s^3 + 40*s^2 + 51*s + 20)

LQ =
(40*s^3 + 190*s^2 + 44*s + 195)/(20*s^4 + 51*s^3 + 40*s^2 + 51*s + 20)
```

To recover I1 and Q we need to compute the inverse Laplace transform of LI1 and LQ. Inverting LI1

```
I1 = ilaplace(LI1,s,t)
```

produces

```
I1 =
(15*(cosh((1001^(1/2)*t)/40)
- (293*1001^(1/2)*sinh((1001^(1/2)*t)/40))/21879))/exp((51*t)/40)
- (5*sin(t))/51
```

Inverting `LQ`

```
Q = ilaplace(LQ,s,t)
```

yields

```
Q =
(4*sin(t))/51 - (5*cos(t))/51 + (107*(cosh((1001^(1/2)*t)/40)
+ (2039*1001^(1/2)*sinh((1001^(1/2)*t)/40))/15301))/(51*exp((51*t)/40))
```

Now let's plot the current `I1(t)` and charge `Q(t)` in two different time domains, $0 \leq t \leq 10$ and $5 \leq t \leq 25$. The statements

```
subplot(2,2,1); ezplot(I1,[0,10]);
title('Current'); ylabel('I1(t)'); grid
subplot(2,2,2); ezplot(Q,[0,10]);
title('Charge'); ylabel('Q(t)'); grid
subplot(2,2,3); ezplot(I1,[5,25]);
title('Current'); ylabel('I1(t)'); grid
text(7,0.25,'Transient'); text(16,0.125,'Steady State');
subplot(2,2,4); ezplot(Q,[5,25]);
title('Charge'); ylabel('Q(t)'); grid
text(7,0.25,'Transient'); text(15,0.16,'Steady State');
```

generate the desired plots
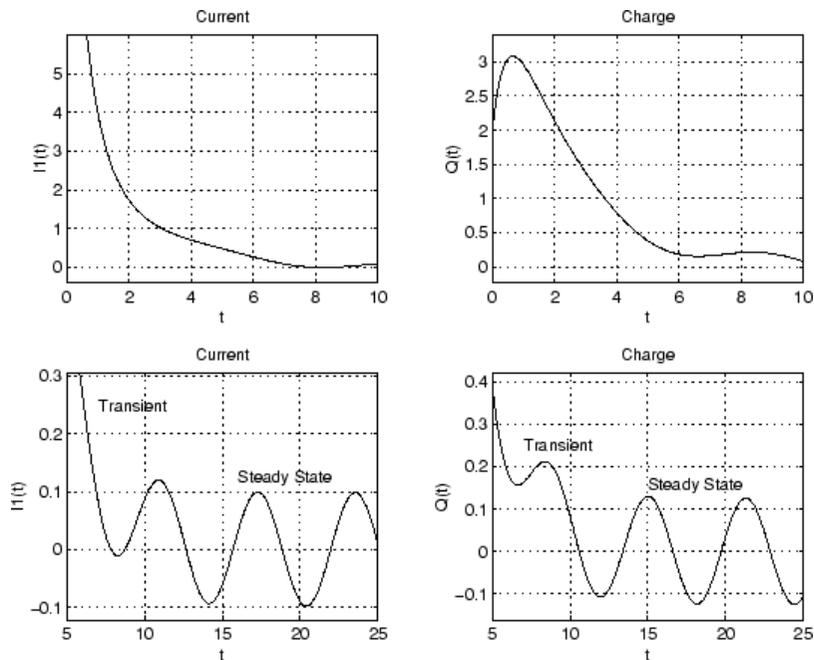
Note that the circuit's behavior, which appears to be exponential decay in the short term, turns out to be oscillatory in the long term. The apparent discrepancy arises because the circuit's behavior actually has two components: an exponential part that decays rapidly (the "transient" component) and an oscillatory part that persists (the "steady-state" component).

The Z– and Inverse Z–transforms

The (one-sided) z-transform of a function $f(n)$ is defined as

$$Z[f](z) = \sum_{n=0}^{\infty} f(n)z^{-n}.$$

The notation $Z[f]$ refers to the z-transform of f at z. Let R be a positive number so that the function $g(z)$ is analytic on and outside the circle $|z| = R$. Then the inverse z-transform (IZT) of g at n is defined as

$$Z^{-1}[g](n) = \frac{1}{2\pi i} \oint_{|z|=R} g(z)z^{n-1}dz, \ n = 1,2,...$$

The notation $Z^{-1}[f]$ means the IZT of f at n. The Symbolic Math Toolbox commands `ztrans` and `iztrans` apply the z-transform and IZT to symbolic expressions, respectively. See `ztrans` and `iztrans` for tables showing various mathematical representations of the z-transform and inverse z-transform and their Symbolic Math Toolbox counterparts.

The z-transform is often used to solve difference equations. In particular, consider the famous "Rabbit Problem." That is, suppose that rabbits reproduce only on odd birthdays (1, 3, 5, 7, ...). If $p(n)$ is the rabbit population at year n, then p obeys the difference equation

$p(n+2) = p(n+1) + p(n)$, $p(0) = 1$, $p(1) = 2$.

We can use `ztrans` to find the population each year $p(n)$. First, we apply `ztrans` to the equations

```
pn = sym('p(n)');
pn1 = sym('p(n+1)');
pn2 = sym('p(n+2)');
syms n z
eq = pn2 - pn1 - pn;
Zeq = ztrans(eq, n, z)
```

to obtain

```
Zeq =
z*p(0) - z*ztrans(p(n), n, z) - z*p(1) + z^2*ztrans(p(n), n, z)
    - z^2*p(0) - ztrans(p(n), n, z)
```

Next, replace 'ztrans(p(n),n,z)' with `Pz` and insert the initial conditions for `p(0)` and `p(1)`.

```
syms Pz
Zeq = subs(Zeq,{'ztrans(p(n),n,z)','p(0)','p(1)'},{Pz,1,2})
```

to obtain

```
Zeq =
Pz*z^2 - z - Pz*z - Pz - z^2
```

Collecting terms

```
eq = collect(Zeq,Pz)
```

yields

```
eq =
- z - Pz*(z - z^2 + 1) - z^2
```

Now solve for `Pz`

```
P = solve(eq,Pz)
```

to obtain

```
P =
-(z^2 + z)/(z - z^2 + 1)
```

To recover $p(n)$, we take the inverse z-transform of P.

```
p = iztrans(P,z,n);
p = simple(p)
```

The result is a bit complicated, but explicit:

```
(3*5^(1/2)*(1/2 - 5^(1/2)/2)^(n - 1))/5 -
    (3*5^(1/2)*(5^(1/2)/2 + 1/2)^(n - 1))/5 +
    (4*(-1)^n*cos(n*(pi/2 + i*asinh(1/2))))/i^n
```

This result can be used as is. But to simplify this result even further, use the MuPAD `rewrite` function:

```
p = feval(symengine,'rewrite',p ,'exp');
p = simple(p)

p =
(3*5^(1/2)*(1/2 - 5^(1/2)/2)^(n - 1))/5 - ...
  (3*5^(1/2)*(5^(1/2)/2 + 1/2)^(n - 1))/5 + ...
  2*exp(pi*i*n)*(2/(5^(1/2) + 1))^n + 2*(5^(1/2)/2 + 1/2)^n
```

Replace the expression `exp(pi*i*n)` with `(-1)^n`:

```
p = subs(p,exp(pi*i*n),(-1)^n);
p = simple(p)

p =
((5^(1/2)/2 + 1/2)^n*(3*5^(1/2) + 5) - ...
   (1/2 - 5^(1/2)/2)^n*(3*5^(1/2) + 15) + ...
   20*(-1)^n*(2/(5^(1/2) + 1))^n)/10

pretty(p)

  /  1/2      \n              /       1/2 \n
 | 5         |      1/2     | 5    |      1/2            n /    2    \n
 | ---- + 1/2 |  (3 5    + 5) - | 1/2 - ---- |  (3 5    + 15) + 20 (-1) | -------- |
 \  2        /              \       2  /         \        |  1/2    |
                                                           \ 5    + 1 /

 -------------------------------------------------------------------------------
                                    10
```

Finally, plot `p`:

```
m = 1:10;
y = double(subs(p,n,m));
plot(m,y,'rO')
title('Rabbit Population');
xlabel('years'); ylabel('p'); grid on
```

to show the growth in rabbit population over time.
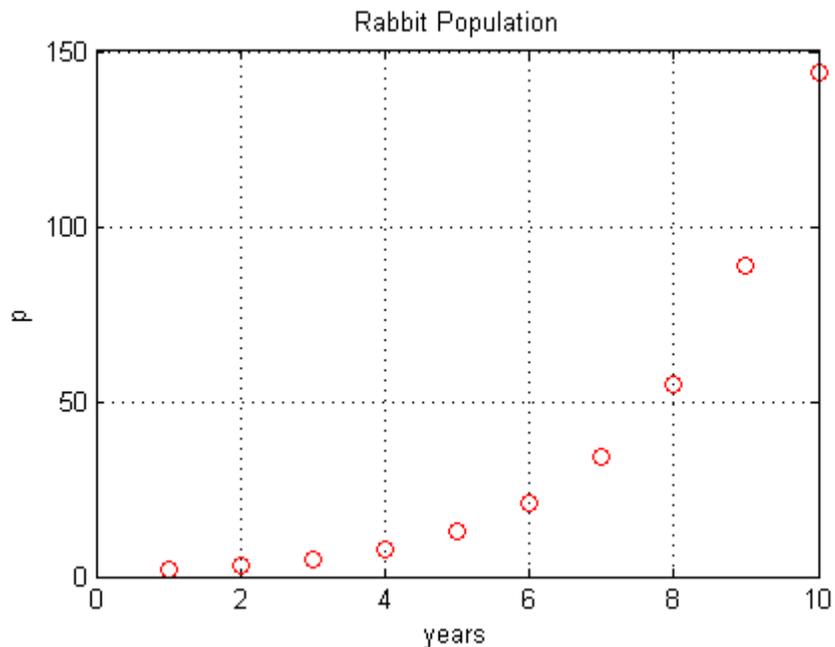
References

Andrews, L.C., B.K. Shivamoggi, *Integral Transforms for Engineers and Applied Mathematicians*, Macmillan Publishing Company, New York, 1986

Crandall, R.E., *Projects in Scientific Computation*, Springer-Verlag Publishers, New York, 1994

Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, MA, 1986

Special Functions of Applied Mathematics

In this section...

Numerical Evaluation of Special Functions Using mfun

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 special functions for the specified parameters. This allows you to evaluate functions that are not available in standard MATLAB software, such as the Fresnel cosine integral. In addition, you can evaluate several MATLAB special functions in the complex plane, such as the error function `erf`.

For example, suppose you want to evaluate the hyperbolic cosine integral at the points $2 + i$, 0, and 4.5. Look in the tables in "Syntax and Definitions of mfun Special Functions" on page 2-114 to find the available functions and their syntax. You can also enter the command

```
mfunlist
```

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

Type

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

which returns

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

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

Note mfun functions perform numerical, not symbolic, calculations. The input parameters should be scalars, vectors, or matrices of type double, or complex doubles, not symbolic variables.

Syntax and Definitions of mfun Special Functions

The following conventions are used in the next 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: $\dfrac{e^{xt}}{e^t - 1} = \sum\limits_{n=0}^{\infty} B_n(x) \cdot \dfrac{t^{n-1}}{n!}$	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	$B(x,y) = \dfrac{\Gamma(x) \cdot \Gamma(y)}{\Gamma(x+y)}$	Beta(x,y)	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Binomial coefficients	$\left(\dfrac{m}{n}\right) = \dfrac{m!}{n!(m-n)!}$ $= \dfrac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$	`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, $0 < k < 1$.

Complementary error function and its iterated integrals	$erfc(z) = \dfrac{2}{\sqrt{\pi}} \cdot \displaystyle\int_{z}^{\infty} e^{-t^2}\,dt = 1 - erf(z)$ $erfc(-1,z) = \dfrac{2}{\sqrt{\pi}} \cdot e^{-z^2}$ $erfc(n,z) = \displaystyle\int_{z}^{\infty} erfc(n-1,t)\,dt$	`erfc(z)` `erfc(n,z)`	$n > 0$
Dawson's integral	$F(x) = e^{-x^2} \cdot \displaystyle\int_{0}^{x} e^{t^2}\,dt$	`dawson(x)`	
Digamma function	$\Psi(x) = \dfrac{d}{dx}\ln(\Gamma(x)) = \dfrac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Dilogarithm integral	$f(x) = \int_1^x \dfrac{\ln(t)}{1-t} dt$	`dilog(x)`	$x > 1$
Error function	$erf(z) = \dfrac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	`erf(z)`	
Euler numbers and polynomials	Generating function for Euler numbers: $\dfrac{1}{\cosh(t)} = \sum_{n=0}^{\infty} E_n \dfrac{t^n}{n!}$	`euler(n)` `euler(n,z)`	$n \geq 0$ $\|t\| < \dfrac{\pi}{2}$
Exponential integrals	$Ei(n,z) = \int_1^{\infty} \dfrac{e^{-zt}}{t^n} dt$ $Ei(x) = PV\left(-\int_{-\infty}^x \dfrac{e^t}{t}\right)$	`Ei(n,z)` `Ei(x)`	$n \geq 0$ $\text{Real}(z) > 0$
Fresnel sine and cosine integrals	$C(x) = \int_0^x \cos\left(\dfrac{\pi}{2}t^2\right) dt$ $S(x) = \int_0^x \sin\left(\dfrac{\pi}{2}t^2\right) dt$	`FresnelC(x)` `FresnelS(x)`	
Gamma function	$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$	`GAMMA(z)`	
Harmonic function	$h(n) = \sum_{k=1}^n \dfrac{1}{k} = \Psi(n+1) + \gamma$	`harmonic(n)`	$n > 0$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
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)`	
(Generalized) hypergeometric function	$$F(n,d,z) = \sum_{k=0}^{\infty} \frac{\prod_{i=1}^{j} \frac{\Gamma(n_i + k)}{\Gamma(n_i)} \cdot z^k}{\prod_{i=1}^{m} \frac{\Gamma(d_i + k)}{\Gamma(d_i)} \cdot k!}$$ 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)`	$0 < x \leq \infty$. 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)` `z1 = a` `z2 = z`	
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	$\Psi^{(n)}(z) = \dfrac{d^n}{dz}\Psi(z)$ where $\Psi(z)$ is the Digamma function.	`Psi(n,z)`	$n \geq 0$
Shifted sine integral	$Ssi(z) = Si(z) - \dfrac{\pi}{2}$	`Ssi(z)`	

The following orthogonal polynomials are available using `mfun`. In all cases, `n` is a nonnegative integer and `x` is real.

Orthogonal Polynomials

Polynomial	mfun Name	Arguments
Chebyshev of the first and second kind	`T(n,x)` `U(n,x)`	
Gegenbauer	`G(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1/2`.
Hermite	`H(n,x)`	
Jacobi	`P(n,a,b,x)`	`a`, `b` are nonrational algebraic expressions or rational numbers greater than `-1`.
Laguerre	`L(n,x)`	
Generalized Laguerre	`L(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	

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 $z_0 > 0$.

Incident plane wave

Diffraction screen

Line of observation
$(x_0, 0, z_0)$
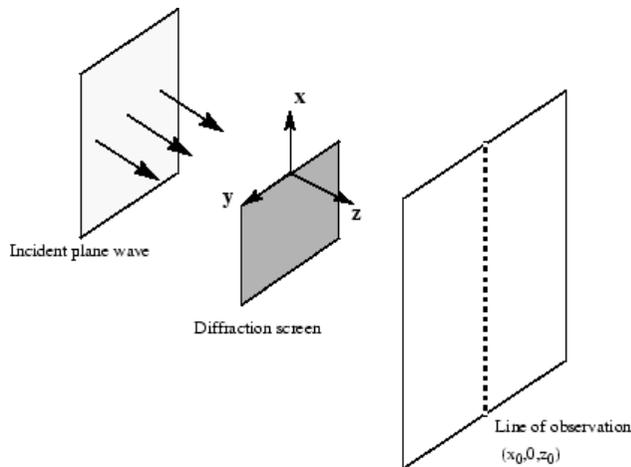
The intensity of the diffracted wave is given by

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

where

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

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

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

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

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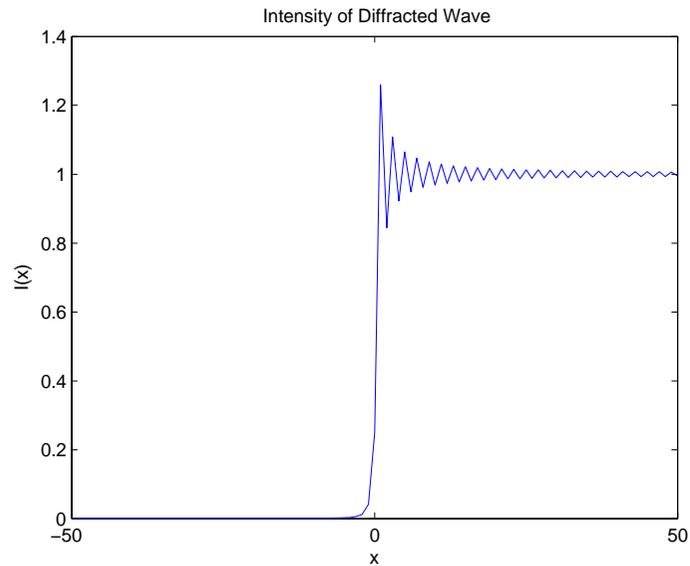
You see from the graph that the diffraction effect is most prominent near the edge of the diffraction screen (`x = 0`), as you expect.

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*.

Generating Code from Symbolic Expressions

In this section...

Generating C or Fortran Code

You can generate C or Fortran code fragments from a symbolic expression, or generate files containing code fragments, using the `ccode` and `fortran` functions. These code fragments calculate numerical values as if substituting numbers for variables in the symbolic expression.

To generate code from a symbolic expression g, enter either `ccode(g)` or `fortran(g)`.

For example:

```
syms x y
z = 30*x^4/(x*y^2+10) - x^3*(y^2 + 1)^2;
fortran(z)

ans =
      t0 = (x**4*30)/(x*y**2+10)-x**3*(y**2+1)**2

ccode(z)

ans =
  t0 = ((x*x*x*x)*3.0E1)/(x*(y*y)+1.0E1)-(x*x*x)*pow(y*y+1.0,2.0);
```

To generate a file containing code, either enter `ccode(g,'file','`*filename*`')` or `fortran(g,'file','`*filename*`')`. For the example above,

```
fortran(z,'file','C:\Work\fortrantest')
```

generates a file named `fortrantest` in the `C:\Work` directory. `fortrantest` consists of the following:

```
        t12 = x**2
```

```
        t13 = y**2
        t14 = t13+1
        t0 = (t12**2*30)/(t13*x+10)-t12*t14**2*x
```

Similarly, the command

```
  ccode(z,'file','C:\Work\ccodetest')
```

generates a file named `ccodetest` that consists of the lines

```
    t16 = x*x;
    t17 = y*y;
    t18 = t17+1.0;
    t0 = ((t16*t16)*3.0E1)/(t17*x+1.0E1)-t16*(t18*t18)*x;
```

`ccode` and `fortran` generate many intermediate variables. This is called *optimized* code. Intermediate variables can make the resulting code more efficient by reusing intermediate expressions (such as `t12` in `fortrantest`, and `t16` in `ccodetest`). They can also make the code easier to read by keeping expressions short.

Generating MATLAB Function Handles and M-Files

Using `matlabFunction`, you can generate a MATLAB function handle that calculates numerical values as if substituting numbers for variables in a symbolic expression. You can also use `matlabFunction` to generate an M-file for calculating these numerical values. The generated M-file is available for use in any MATLAB calculation, whether or not the computer running the file has a license for Symbolic Math Toolbox functions.

Generating a Function Handle

`matlabFunction` can generate a function handle from any symbolic expression. For example:

```
  syms x y
  r = sqrt(x^2 + y^2);
  ht = matlabFunction(tanh(r))

  ht =
      @(x,y)tanh((x.^2+y.^2).^(1./2))
```

You can use this function handle to calculate numerically:

```
ht(.5,.5)

ans =
    0.6089
```

This is correct, since tanh(1/sqrt(2)) = 0.6089....

You can pass the usual MATLAB double-precision numbers or matrices to the function handle. For example:

```
cc = [.5,3];dd = [-.5,.5];
ht(cc,dd)

ans =
    0.6089    0.9954
```

Controlling the Order of Variables

`matlabFunction` generates input variables in alphabetical order from a symbolic expression. That is why the function handle in "Generating a Function Handle" on page 2-123 has `x` before `y`:

```
ht = @(x,y)tanh((x.^2+y.^2).^(1./2))
```

You can specify the order of input variables in the function handle using the `vars` option. You specify the order by passing a cell array of strings, or a vector of symbolic variables. For example,

```
syms x y
r = sqrt(x^2 + 3*y^2);
ht = matlabFunction(tanh(r),'vars',[y x])

ht =
    @(y,x)tanh((x.^2+3.*y.^2).^(1./2))

ht2 = matlabFunction(tanh(r),'vars',{'x','y'})

ht2 =
    @(x,y)tanh((x.^2+3.*y.^2).^(1./2))
```

Generating an M-File

You can generate an M-file from a symbolic expression, in addition to a function handle. Specify the file name using the `file` option. Pass a string containing the path to the file.

This example generates an M-file that calculates the value of the symbolic matrix F for double-precision inputs `t`, `x`, and `y`:

```
syms x y t
z = (x^3-tan(y))/(x^3+tan(y));
w = z/(1+t^2);
F = [w,(1+t^2)*x/y;(1+t^2)*x/y,3*z-1];
matlabFunction(F,'file','C:\Work\testMatrix.m')
```

The file `testMatrix.m` contains the following code:

```
function RESULT = testMatrix(t,x,y)
%TESTMATRIX
%    RESULT = TESTMATRIX(T,X,Y)

%    This function was generated by the Symbolic Math Toolbox version 5.0.
%    17-Jul-2008 15:54:43

t4 = x.^2;
t5 = tan(y);
t6 = t4.*x;
t7 = t.^2;
t8 = t7 + 1;
t9 = 1./y;
t10 = t8.*t9.*x;
t11 = t5 - t6;
t12 = t5 + t6;
t13 = 1./t12;
RESULT = [-(t11.*t13)./t8,t10;t10,-3.*t11.*t13-1];
```

`matlabFunction` generates many intermediate variables. This is called *optimized* code. Intermediate variables can make the resulting code more efficient by reusing intermediate expressions (such as `t19`, `t20`, and `t22` in the calculation of `RESULT`). They can make the code easier to read by keeping expressions short.

If you don't want the default alphabetical order of input variables, use the vars option to control the order. Continuing the example,

```
matlabFunction(F,'file','C:\Work\testMatrix.m','vars',[x y t])
```

generates an equivalent file to the one above, with a different order of inputs:

```
function RESULT = testMatrix(x,y,t)
...
```

MuPAD in Symbolic Math Toolbox

Understanding MuPAD

In this section...
"Introduction to MuPAD" on page 3-2
"The MATLAB Workspace and MuPAD Engines" on page 3-2
"Introductory Example Using a MuPAD Notebook from MATLAB" on page 3-3

Introduction to MuPAD

Version 5 of Symbolic Math Toolbox is powered by the MuPAD symbolic engine.

- Nearly all Symbolic Math Toolbox functions work the same way as in previous versions. To read about the differences with the new engine, see the transition Release Notes.

- MuPAD notebooks provide a new interface for performing symbolic calculations, variable-precision calculations, plotting, and animations. "Introductory Example Using a MuPAD Notebook from MATLAB" on page 3-3 contains an introductory example of how to use this interface.

- Symbolic Math Toolbox functions allow you to copy variables and expressions between the MATLAB workspace and MuPAD notebooks. For more information, see "Copying Variables and Expressions Between the MATLAB Workspace and MuPAD Notebooks" on page 3-25.

- You can call MuPAD functions and procedures from the MATLAB environment. For more information, see "Calling MuPAD Functions at the MATLAB Command Line" on page 3-28.
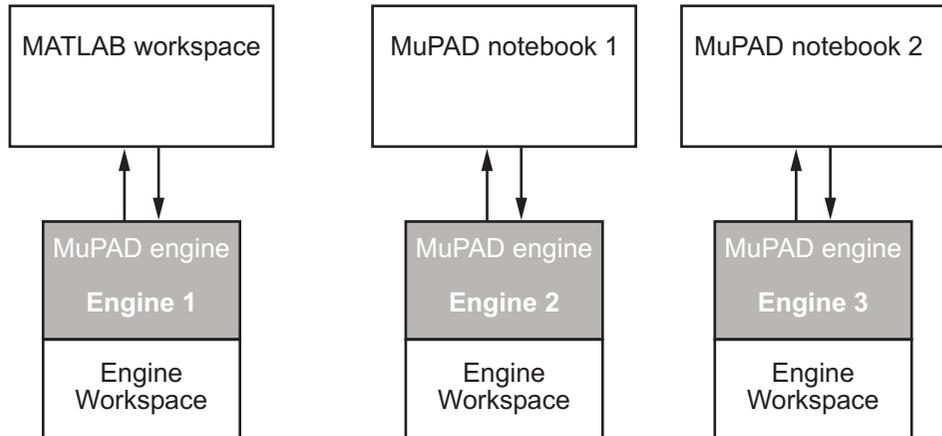
The MATLAB Workspace and MuPAD Engines

A MuPAD engine is a separate process that runs on your computer in addition to a MATLAB process. A MuPAD engine starts when you first call a function that needs a symbolic engine, such as `syms`. Symbolic Math Toolbox functions that use the symbolic engine use standard MATLAB syntax, such as `y = int(x^2)`.

Conceptually, each MuPAD notebook has its own symbolic engine, with associated workspace. You can have any number of MuPAD notebooks open simultaneously.

One engine exists for use by Symbolic Math Toolbox.

Each MuPAD notebook also has its own engine.

| MATLAB workspace | MuPAD notebook 1 | MuPAD notebook 2 |

| MuPAD engine **Engine 1** | MuPAD engine **Engine 2** | MuPAD engine **Engine 3** |

| Engine Workspace | Engine Workspace | Engine Workspace |

The engine workspace associated with the MATLAB workspace is generally empty, except for assumptions you make about variables. For more information, see "Clearing Assumptions and Resetting the Symbolic Engine" on page 3-30.

Introductory Example Using a MuPAD Notebook from MATLAB

This example shows how to use a MuPAD notebook to calculate symbolically the mean and variance of a normal random variable that is restricted to be positive. For more information on using a MuPAD notebook, see "Calculating in a MuPAD Notebook" on page 3-15.
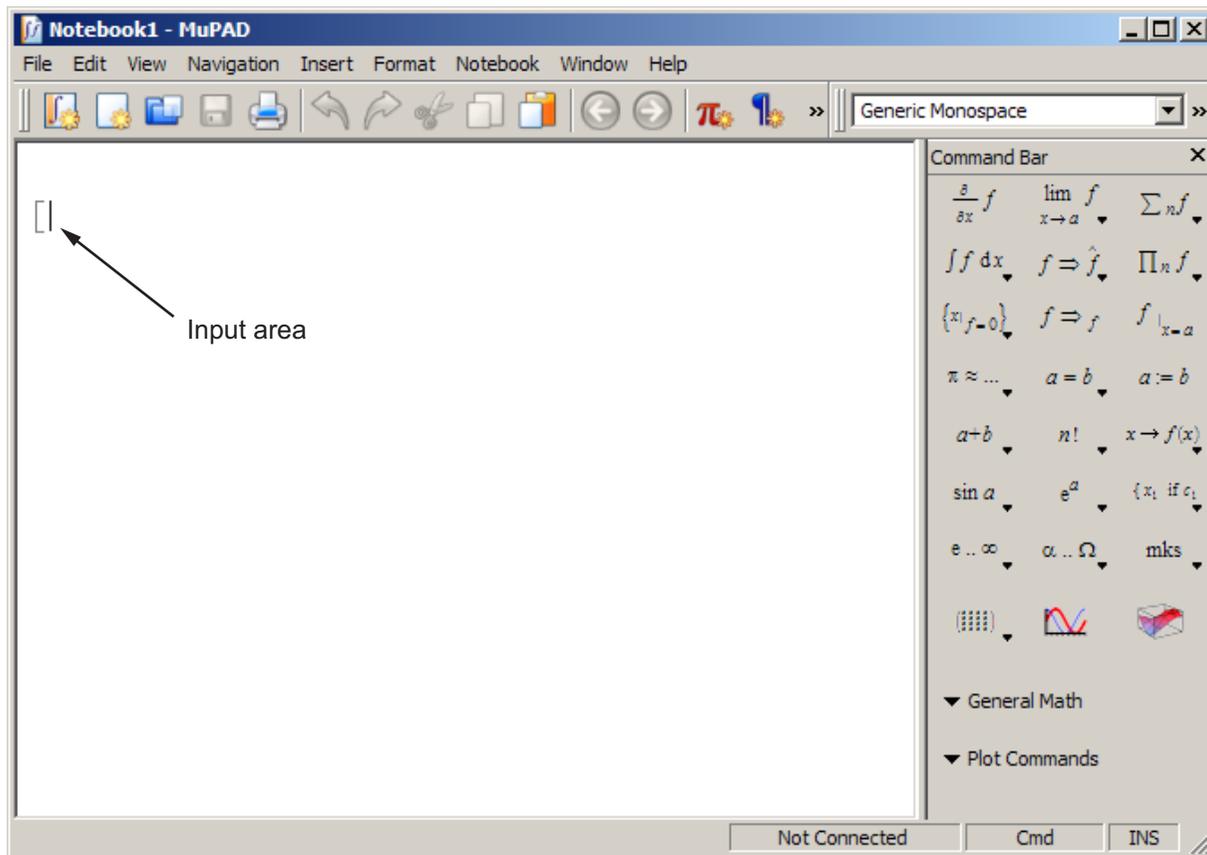
The density function of the normal and positive random variable is

$$f(x) = \begin{cases} e^{-x^2/2}\sqrt{2/\pi} & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

1 At the MATLAB command line, enter the command

```
mupad
```

2 A blank MuPAD notebook opens. You perform calculations by typing in the input area, demarcated by a left bracket.


3 In the input area, type

```
f := exp(-x^2/2)*sqrt(2/PI)
```

and press **Enter**.

> **Note** Assignment in a MuPAD notebook uses `:=`, not the MATLAB syntax `=`. Also, the MuPAD syntax for the mathematical constant π is `PI`, not the MATLAB syntax `pi`. For more information on common syntax differences, see "Differences Between MATLAB and MuPAD Syntax" on page 3-20.

The MuPAD notebook displays results in real math notation.
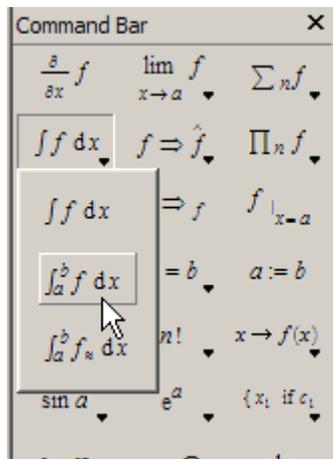
Your notebook appears as follows.
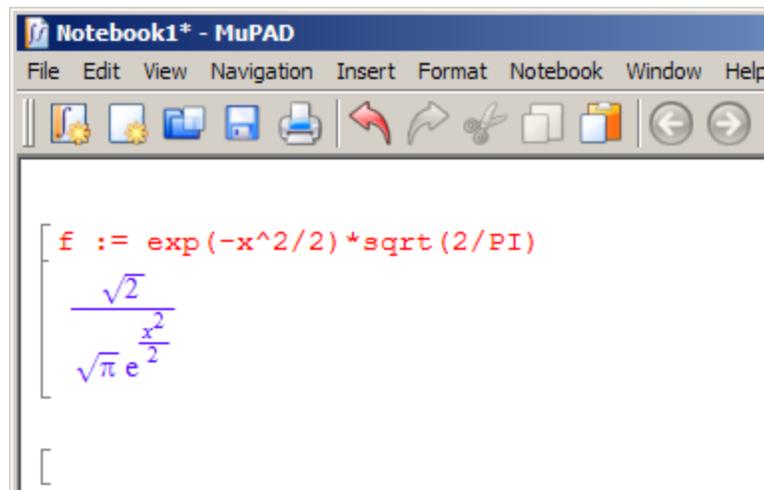
4 The mean of the random variable is

$$\text{mean} = \int_{0}^{\infty} x \cdot f \, dx.$$

To calculate the mean of the random variable, type

```
mean :=
```

a To place an integral in the correct syntax, click the integral button in the right-hand command bar, and select definite limits as shown.

b The correct syntax for integration appears in the input area.

```
mean := int(#f, #x=#a..#b)
```

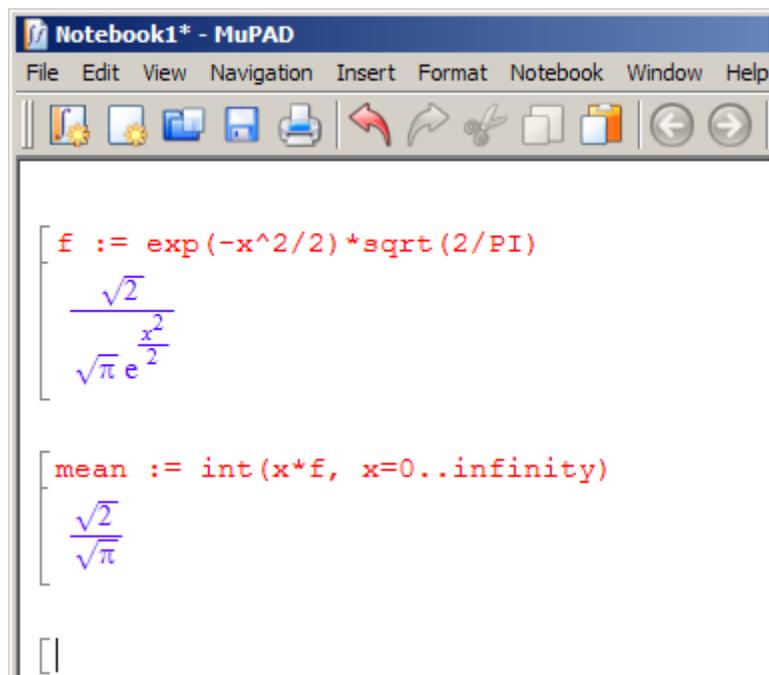**c** Replace #f with x*f, #x with x, #a with 0, and #b with infinity.

- Use the **Tab** key to select the replaceable fields #f, #x, etc.

- Use **Ctrl+space** to autocomplete inputs; e.g., enter infi followed by **Ctrl+space** to enter infinity.

Once your input area reads

```
mean := int(x*f, x=0..infinity)
```

press **Enter**.

Note The syntax for integration, and for infinity, differ from the MATLAB versions.

---

5 The variance of the random variable is

$$\text{variance} = \int_0^\infty (x - \text{mean})^2 \cdot f \, dx.$$

To calculate the variance of the random variable, type

```
variance := int((x-mean)^2*f, x=0..infinity)
```

and press **Enter**.

```
variance := int((x-mean)^2*f, x=0..infinity)
```

$$\int_0^\infty \frac{\sqrt{2}\left(x - \frac{\sqrt{2}}{\sqrt{\pi}}\right)^2}{\sqrt{\pi}\,e^{\frac{x^2}{2}}}\,\mathrm{d}x$$

6 The result of evaluating `variance` is a complicated expression. Try to simplify it with the `simplify` command. Type

```
simplify(variance)
```

and press **Enter**. The result is indeed simpler.

```
simplify(variance)
```

$$\frac{\pi - 2}{\pi}$$

7 Another expression for the variance of the random variable is

$$\text{variance} = \int_0^\infty x^2 \cdot f\,dx - \text{mean}^2.$$

To calculate the variance of the random variable using this definition, type
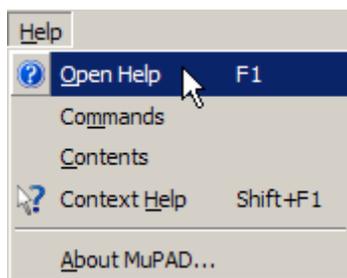
```
variance2 := int(x^2*f, x=0..infinity) - mean^2
```

and press **Enter**.

```
variance2 := int(x^2*f, x=0..infinity) - mean^2
```

$$1 - \frac{2}{\pi}$$

The two expressions for variance, `variance` and `variance2`, are obviously equivalent.

For more information on working in MuPAD notebooks, select **Help > Open Help**, or press **F1** to launch the MuPAD Help viewer.

Within the MuPAD Help viewer, both the "Getting Started" and "The MuPAD Notebook Interface" sections can help you understand and use MuPAD notebooks.

MuPAD for MATLAB Users

In this section...

Getting Help for MuPAD

There is extensive online help available for MuPAD. You can access the help browser from the MATLAB workspace in a variety of ways:

- Enter `doc(symengine)` at the MATLAB command line to display MuPAD Help.
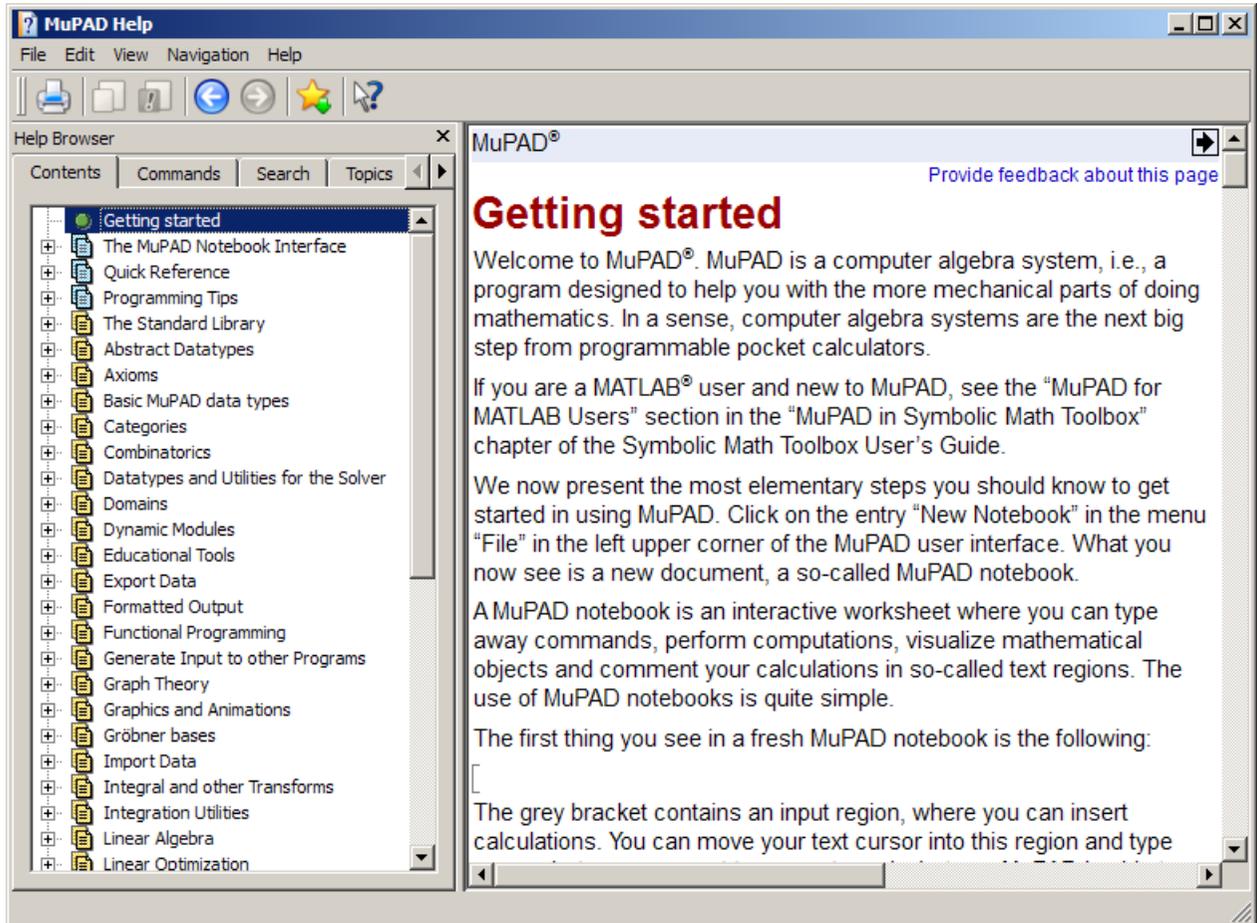
MuPAD Help contains complete documentation of the MuPAD language. It also explains how to use the various MuPAD interfaces, such as notebooks and the editor.

- For help on a specific MuPAD function, enter `doc(symengine,'functionName')` at the MATLAB command line to display MuPAD Help at the `functionName` function.

- There is also a MuPAD Tutorial PDF file available at `http://www.mathworks.com/access/helpdesk/...help/pdf_doc/symbolic/mupad_tutorial.pdf` .

Launching, Opening, and Saving MuPAD Notebooks

To open a new MuPAD notebook from the MATLAB command line, enter

```
nb = mupad
```

(You can use any variable name you like instead of nb.) This opens a blank MuPAD notebook.

The variable nb is a handle to the notebook. This handle is used only for communication between the MATLAB workspace and the MuPAD notebook. It can be used as described in "Copying Variables and Expressions Between the MATLAB Workspace and MuPAD Notebooks" on page 3-25.

You can also open an existing MuPAD notebook file named *file_name* from the MATLAB command line by entering the command

```
nb2 = mupad('file_name')
```

This command is useful in case you lose the handle to a notebook; save the notebook file, and reopen it with a fresh handle.

Warning **You can lose data when saving a MuPAD notebook. A notebook saves its inputs and outputs, but not the state of its engine. In particular, variables copied into a notebook using setVar(nb,...) are not saved with the notebook.**

You can open and save MuPAD notebook files using the usual file system commands, and by using the MATLAB or MuPAD **File** menu. However, to use a handle to a notebook, you must open the notebook using the mupad command at the MATLAB command line.


Note MuPAD notebook files open in an un-evaluated state; in other words, the notebook is not synchronized with its engine when it opens. To synchronize a notebook with its engine, choose **Evaluate All** from the **Notebook** menu. For more information, see "Synchronizing a Notebook and its Engine" on page 3-19.

Opening Recent Files and Other MuPAD Interfaces

If you have no MuPAD interfaces open, the command

```
mupadwelcome
```

brings up a window for launching various MuPAD interfaces.
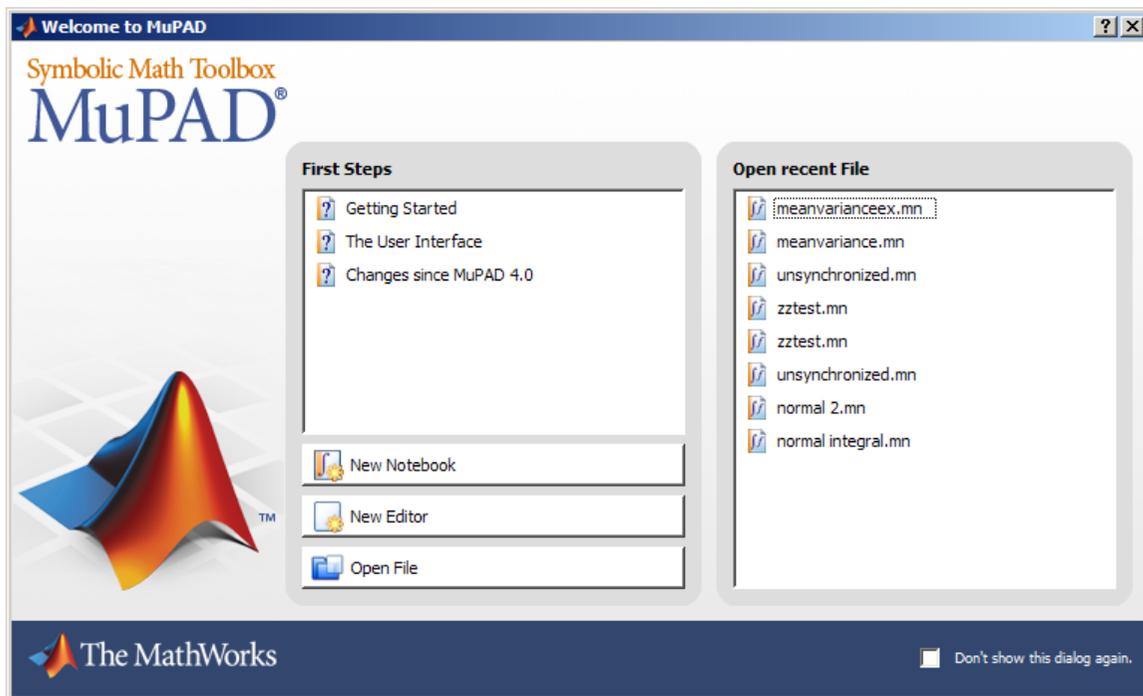
- To access MuPAD Help, click one of the three options in the **First Steps** pane.

- To launch a file in the **Open recent File** list, single-click its name.

- To launch a new notebook, click the **New Notebook** button.

- To launch a program editor, click the **New Editor** button. For information on this interface and its associated debugger, see MuPAD Help.

- To open an existing MuPAD notebook or program file, click **Open File** and navigate to the file.

> **Note** If you have a MuPAD interface open, the `mupadwelcome` command launches a new notebook.

Alternatively, you can launch the `mupadwelcome` screen from the MATLAB **Start** menu as pictured.
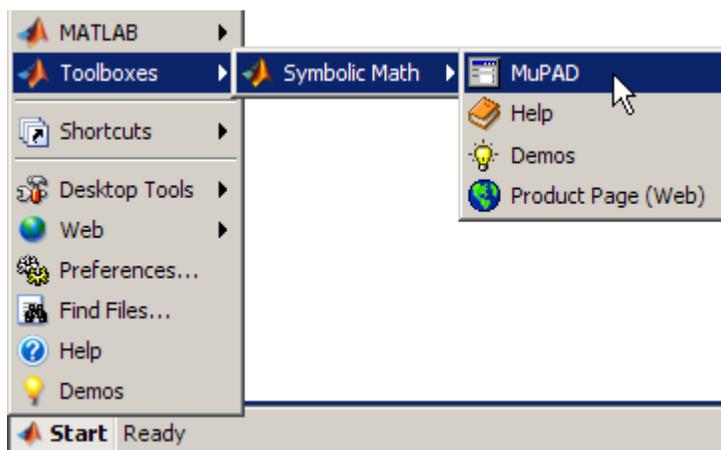
This option works the same as the `mupadwelcome` function. For example, if you have a MuPAD interface open, the **Start** menu opens a new MuPAD notebook instead of the welcome screen.

Recovering the mupadwelcome Screen

If you check **Don't show this dialog again** in the **Welcome to MuPAD** screen, the screen is no longer available. To undo this action and make the screen available:

1 Enter `doc(symengine)` to launch MuPAD Help.

2 From the **View** menu, select **Configure**.

3 From the **User Interface** menu, uncheck **Hide welcome dialog**.

Calculating in a MuPAD Notebook

The Visual Elements of a Notebook

A MuPAD notebook has the following main components.

Command Bar

Menu

Input areas

Results and Plots
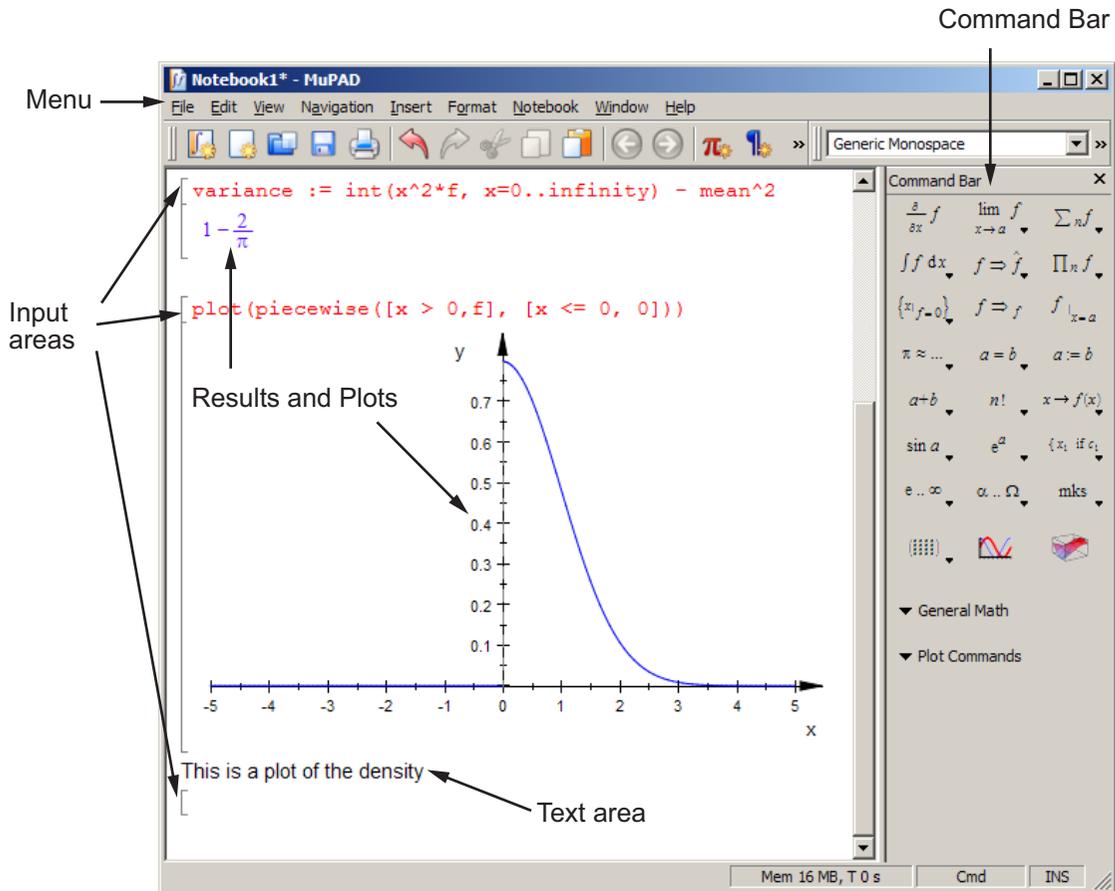
Text area

- Enter commands for execution, evaluation, or plotting in input areas.

- Enter commentary in text areas.

- Use the Command Bar to help you enter commands into input areas with the proper syntax.

- Use the **Insert** menu to add a text area (called **Text Paragraph**) or input area (called **Calculation**).

- Use the **Notebook** menu to evaluate expressions in input areas.

Working in a Notebook

The MuPAD notebook interface differs from the MATLAB interface. Here are some things to keep in mind when working in a MuPAD notebook:

- Commands typed in an input area are not evaluated until you press **Enter**.

- You can edit the commands typed in *any* input area. For example, you can change a command, correct syntax, or try different values of parameters simply by selecting the area you wish to change and typing over it. Press **Enter** to have the notebook evaluate the result.

- Results do not automatically cascade or propagate through a notebook, as described in "Cascading Calculations" on page 3-16.

- The MATLAB method of recalling a previous command by typing an **up arrow** key does not have the same effect in a MuPAD notebook. Instead, you use arrow keys for navigation in MuPAD notebooks, similar to most word processors.

Cascading Calculations

If you change a variable in a notebook, the changes do not automatically propagate throughout the notebook. For example, consider the following set of MuPAD commands.

```
z := sin(x)
```
$$\sin(x)$$

```
y := z/(1+z^2)
```
$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

```
w := simplify(y/(1-y))
```
$$-\frac{2\sin(x)}{\cos(2x) + 2\sin(x) - 3}$$

Now change the definition of z in the first line of the notebook from `sin(x)` to `cos(x)` and press **Enter**.

```
z := cos(x)
```
$$\cos(x)$$

```
y := z/(1+z^2)
```
$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

```
w := simplify(y/(1-y))
```
$$-\frac{2\sin(x)}{\cos(2x) + 2\sin(x) - 3}$$

Only the first line was reevaluated. Therefore y and z are no longer synchronized; the notebook is in an inconsistent state.

To have the changes cascade to all parts of the notebook, select **Notebook > Evaluate All**.
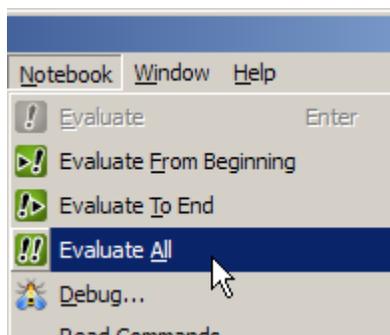
The engine evaluates all the expressions in the notebook from top to bottom, and the notebook becomes consistent.

```
z := cos(x)
```
$$\cos(x)$$

```
y := z/(1+z^2)
```
$$\frac{\cos(x)}{\cos(x)^2 + 1}$$

```
w := simplify(y/(1-y))
```
$$\frac{\cos(x)}{\cos(x)^2 - \cos(x) + 1}$$

Synchronizing a Notebook and its Engine

When you open a saved MuPAD notebook file, the notebook display is not synchronized with its engine. For example, suppose you saved the notebook pictured in the start of "Cascading Calculations" on page 3-16:

$$z := \sin(x)$$

$$\sin(x)$$

$$y := z/(1+z^2)$$

$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

$$w := \text{simplify}(y/(1-y))$$

$$-\frac{2\sin(x)}{\cos(2x) + 2\sin(x) - 3}$$

If you open that file and immediately try to work in it, without synchronizing the notebook with its engine, the expressions in the notebook display are unavailable for calculations. For example, try to calculate $u := (1+w)/w$:

```
z := sin(x)
```

$$\sin(x)$$

```
y := z/(1+z^2)
```

$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

```
w := simplify(y/(1-y))
```

$$-\frac{2\sin(x)}{\cos(2x) + 2\sin(x) - 3}$$

```
u := (1+w)/w
```

$$\frac{w+1}{w}$$

The variable w has no definition as far as the engine is concerned.

To remedy this situation, select **Notebook > Evaluate All**. The variable u changes to reflect the value of w.

```
u := (1+w)/w
```

$$\frac{\left(\dfrac{2\sin(x)}{\cos(2x) + 2\sin(x) - 3} - 1\right)\left(\cos(2x) + 2\sin(x) - 3\right)}{2\sin(x)}$$

Differences Between MATLAB and MuPAD Syntax

There are several differences between MATLAB and MuPAD syntax. Be aware of which interface you are using in order to use the correct syntax:

- Use MATLAB syntax in the MATLAB workspace, *except* for the functions `evalin(symengine,...)` and `feval(symengine,...)`, which use MuPAD syntax.

- Use MuPAD syntax in MuPAD notebooks.

You must define MATLAB variables before using them. However, every expression entered in a MuPAD notebook is assumed to be a combination of symbolic variables unless otherwise defined. This means that you have to be especially careful when working in MuPAD notebooks, since fewer of your typographic errors cause syntax errors.

This table lists common tasks, meaning commands or functions, and how they differ in MATLAB and MuPAD syntax.

Common Tasks in MATLAB and MuPAD Syntax

Task	MATLAB syntax	MuPAD syntax
Assignment	`=`	`:=`
List variables	`whos`	`anames(All, User)`
Numerical value of expression	`double(`*expression*`)`	`float(`*expression*`)`
Suppress output	`;`	`:`
Enter matrix	`[x11,x12,x13; x21,x22,x23]`	`matrix([[x11,x12,x13], [x21,x22,x23]])`
`{a,b,c}`	cell array	set
Linear algebra commands	Nothing extra needed	`linalg::` prefix, or `use(linalg)`
Autocompletion	**Tab**	**Ctrl-space**
Equality, inequality comparison	`==`, `~=`	`=`, `<>`

The next table lists the differences between MATLAB expressions and MuPAD expressions.

MATLAB vs. MuPAD Expressions

MATLAB Expression	MuPAD Expression
Inf	infinity
pi	PI
i	I
NaN	undefined
fix	trunc
log	ln
asin	arcsin
acos	arccos
atan	arctan
asinh	arcsinh
acosh	arccosh
atanh	arctanh
acsc	arccsc
asec	arcsec
acot	arccot
acsch	arccsch
asech	arcsech
acoth	arccoth
besselj	besselJ
bessely	besselY
besseli	besselI
besselk	besselK
lambertw	lambertW
sinint	Si
cosint	Ci
eulergamma	EULER

MATLAB vs. MuPAD Expressions (Continued)

MATLAB Expression	MuPAD Expression
conj	conjugate
catalan	CATALAN
laplace	transform::laplace
ilaplace	transform::invlaplace
ztrans	transform::ztrans
iztrans	transform::invztrans

The MuPAD definition of Fourier transform and inverse Fourier transform differ from their Symbolic Math Toolbox counterparts by the sign of the exponent:

	Symbolic Math Toolbox definition	MuPAD definition
Fourier transform	$$F[f](w) = \int\limits_{-\infty}^{\infty} f(x)e^{-iwx}dx$$ F = fourier(f)	$$F[f](w) = \int\limits_{-\infty}^{\infty} f(x)e^{iwx}dx$$ F = transform::fourier(f,x,w)
Inverse Fourier transform	$$F^{-1}[f](x) = \frac{1}{2\pi} \int\limits_{-\infty}^{\infty} f(w)e^{iwx}dw$$ Finv = ifourier(f)	$$F^{-1}[f](x) = \frac{1}{2\pi} \int\limits_{-\infty}^{\infty} f(w)e^{-iwx}dw$$ Finv = transform::invfourier(f,w,x)

The MuPAD definition of exponential integral differs from the Symbolic Math Toolbox counterpart:

	Symbolic Math Toolbox definition	MuPAD definition
Exponential integral	$\mathrm{expint}(x) = -\mathrm{Ei}(-x) =$ $\displaystyle\int_x^\infty \frac{\exp(-t)}{t}\,dt \text{ for } x > 0 =$ $\mathrm{Ei}(1, x).$	$\displaystyle\mathrm{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t}\,dt \text{ for } x < 0.$ $\displaystyle\mathrm{Ei}(n, t) = \int_1^\infty \frac{\exp(-xt)}{t^n}\,dt.$ The definitions of `Ei` extend to the complex plane, with a branch cut along the negative real axis.

Integration of MuPAD and MATLAB

In this section...

Copying Variables and Expressions Between the MATLAB Workspace and MuPAD Notebooks

You can copy a variable in a MuPAD notebook to a variable in the MATLAB workspace using a MATLAB command. Similarly, you can copy a variable or symbolic expression in the MATLAB workspace to a variable in a MuPAD notebook using a MATLAB command. In order to do either assignment, you need to know the handle to the MuPAD notebook you want to address.

The only way to assign variables between a MuPAD notebook and the MATLAB workspace is to start the notebook using the syntax

```
nb = mupad;
```

(you can use any variable name for the handle nb) or open an existing notebook file with

```
nb = mupad(file_name);
```

This handle is used only for communication between the MATLAB workspace and the MuPAD notebook.

- To copy a symbolic variable in the MATLAB workspace to a variable in the MuPAD notebook engine with the same name, enter

```
setVar(notebook_handle,variable)
```

at the MATLAB command line. For example, if nb is the handle to the notebook and z is the variable, enter

```
setVar(nb,z)
```

There is no indication in the MuPAD notebook that the variable z exists. Check that it exists by entering z in an input area of the notebook, or by entering the command anames(All, User) in the notebook.

- To assign a symbolic expression to a variable in a MuPAD notebook, enter

    ```
    setVar(notebook_handle,'variable',expression)
    ```

    at the MATLAB command line. For example, if nb is the handle to the notebook, exp(x) - sin(x) is the expression, and z is the variable, enter

    ```
    syms x
    setVar(nb,'z',exp(x) - sin(x))
    ```

    For this type of assignment, x must be a symbolic variable in the MATLAB workspace.

    Again, there is no indication in the MuPAD notebook that the variable z exists. Check that it exists by entering z in an input area of the notebook, or by entering the command anames(All, User) in the notebook.

- To copy a symbolic variable in a MuPAD notebook to a variable in the MATLAB workspace, enter

    ```
    MATLABvar = getVar(notebook_handle,'variable');
    ```

    at the MATLAB command line. For example, if nb is the handle to the notebook, z is the variable in the MuPAD notebook, and u is the variable in the MATLAB workspace, enter

    ```
    u = getVar(nb,'z')
    ```

    The communication between the MATLAB workspace and the MuPAD notebook takes place with the notebook's engine. Therefore, the variable z must be synchronized into the notebook's MuPAD engine before using getVar, and not merely displayed in the notebook. If you try to use getVar to copy a variable z that is undefined in the MuPAD engine, the resulting MATLAB variable u is empty. For more information on this topic, see "Synchronizing a Notebook and its Engine" on page 3-19.

> **Note** All such copying and assignment must be done from the MATLAB
> workspace, not from a MuPAD notebook.
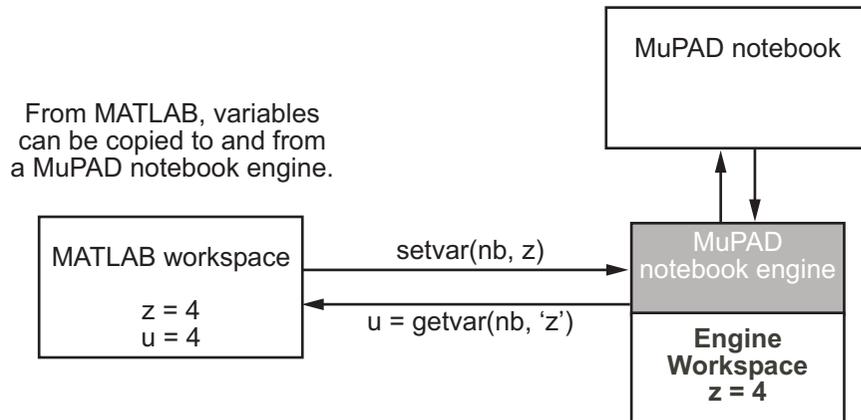
Copying and Pasting Using the System Clipboard

You can also copy and paste between notebooks and the MATLAB workspace
using standard editing commands. If you copy a result in a MuPAD notebook
to the system clipboard, you may get the text associated with the expression,
or a picture, depending on your operating system and application support.

For example, consider the following MuPAD expression.

Select the output with the mouse and copy it to the clipboard.

Paste this into the MATLAB workspace. The result is text:

```
exp(x)/(x^2 + 1)
```

If you paste it into WordPad on a Windows® system, the result is a picture.
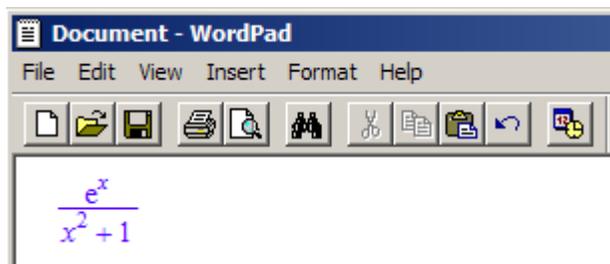
Calling MuPAD Functions at the MATLAB Command Line

To access MuPAD functions and procedures at the MATLAB command line, use the `evalin(symengine,...)` function or the `feval(symengine,...)` function. These functions are designed to work like the existing MATLAB functions `evalin` and `feval`.

evalin

For `evalin`, the syntax is

```
y = evalin(symengine,'MuPAD_Expression');
```

Use `evalin` when you want to perform computations in the MuPAD language, while working in the MATLAB workspace. You can use `evalin` for any MuPAD expression.

For example, to make a three-element symbolic vector of the `sin(kx)` function, `k = 1` to `3`, enter

```
y = evalin(symengine,'sin(k*x)$k=1..3')

y =
[ sin(x), sin(2*x), sin(3*x)]
```

feval

For evaluating a MuPAD function, you can also use the `feval` function. `feval` has a different syntax than `evalin`, so it can be simpler to use. The syntax is

```
y = feval(symengine,'MuPAD_Function',x1,...,xn);
```

MuPAD_Function represents the name of a MuPAD function. The arguments `x1,...,xn` must be symbolic variables, numbers, or strings. For example, to find the tenth element in the Fibonacci sequence, enter

```
z = feval(symengine,'numlib::fibonacci',10)

z =
55
```

The next example compares the use of a symbolic solution of an equation to the solution returned by the MuPAD numeric `fsolve` function near the point `x = 3`. (For information on this function, enter `doc(symengine,'numeric::fsolve')` at the MATLAB command line.)

```
syms x
f = sin(x^2);
solve(f)

ans =
 0
 0

feval(symengine, 'numeric::fsolve',f,'x=3')

ans =
x = 3.0699801238394654654386548746678
```

As you might expect, the answer is the numerical value of $\sqrt{3\pi}$. The setting of MATLAB `format` does not affect the display; it is the full returned value from the MuPAD `'numeric::fsolve'` function.

Usage of evalin vs. feval

The `evalin(symengine,...)` function causes the MuPAD engine to evaluate a string. Since the MuPAD engine workspace is generally empty, expressions

returned by `evalin(symengine,...)` are not simplified or evaluated according to their definitions in the MATLAB workspace.

For example:

```
syms x
y = x^2;
evalin(symengine, 'cos(y)')

ans =
cos(y)
```

The variable y is not expressed in terms of x because y is not known to the MuPAD engine workspace.

In contrast, `feval(symengine,...)` can pass symbolic variables that exist in the MATLAB workspace, and these variables are evaluated before being processed in the MuPAD engine. For example:

```
syms x
y = x^2;
feval(symengine,'cos',y)

ans =
cos(x^2)
```

Clearing Assumptions and Resetting the Symbolic Engine

The symbolic engine workspace associated with the MATLAB workspace is usually empty. The MATLAB workspace keeps track of the values of symbolic variables, and passes them to the symbolic engine for evaluation as necessary. However, the symbolic engine workspace contains all assumptions you make about symbolic variables, such as whether a variable is real or positive. These assumptions can affect solutions to equations, simplifications, and transformations, as explained in "Examples of the Effect of Assumptions" on page 3-33.

Note The commands

```
syms x
x = sym('x')
clear x
```

clear any existing value of x in the MATLAB workspace, but do not clear
assumptions about x in the symbolic engine workspace.

- If you make an assumption about the nature of a variable, e.g., by the
  commands

  ```
  syms x real
  ```

  or

  ```
  syms x positive
  ```

  clearing the variable x from the MATLAB workspace does not clear the
  assumption from the symbolic engine workspace. To clear the assumption,
  enter the command

  ```
  syms x clear
  ```

  For more detail, see "Checking a Variable's Assumptions" on page 3-32 and
  "Examples of the Effect of Assumptions" on page 3-33.

- If you reset the symbolic engine by entering the command

  ```
  reset(symengine)
  ```

  or if you change symbolic engines with the symengine command, MATLAB
  no longer recognizes any symbolic variables that exist in the MATLAB
  workspace. Clear the variables with the clear command, or renew them
  with the syms or sym commands. The symengine command is discussed in
  "Choosing a Maple or MuPAD Engine" on page 3-35.

Here is an example of how the MATLAB workspace and the symbolic engine
workspace respond to a sequence of commands:

Step	Command	MATLAB workspace	MuPAD Engine Workspace
1	`syms x positive`	x	x is positive
2	`clear x`	empty	x is positive
3	`syms x`	x	x is positive
4	`syms x clear`	x	empty

Checking a Variable's Assumptions

To check whether a variable, say x, has any assumption in the symbolic engine workspace associated with the MATLAB workspace, enter the command

```
evalin(symengine,'getprop(x)')
```

at the MATLAB command line.

- If the returned answer is C_, there are no assumptions about the variable. (C_ means it can be any complex number.)
- If the returned value is anything else, there are assumptions about the variable.

For example:

```
syms x real
evalin(symengine,'getprop(x)')

ans =
R_

syms z
evalin(symengine,'assume(z <> 0)')
evalin(symengine,'getprop(z)')

ans =
C_ minus {0}

syms z clear
evalin(symengine,'getprop(z)')
```

```
ans =
C_
```

For more information about the basic sets that can be returned as assumptions, enter

```
doc(symengine,'solvelib::BasicSet')
```

Examples of the Effect of Assumptions

Assumptions can change the answers that the `solve` function returns, and can change the results of simplifications. The only assumptions you can make using MATLAB commands are `real` or `positive`.

For example, consider what transpires when solving the equation `x^3 = 1`:

```
syms x
solve('x^3=1')

ans =
                     1
 - (3^(1/2)*i)/2 - 1/2
   (3^(1/2)*i)/2 - 1/2

syms x real
solve('x^3=1')

ans =
1
```

However, clearing `x` does not change the underlying assumption that `x` is real:

```
clear x
syms x
solve('x^3=1')

ans =
1
```

Clearing `x` with `syms x clear` clears the assumption:

```
syms x clear
solve('x^3=1')

ans =

                       1
 - (3^(1/2)*i)/2 - 1/2
   (3^(1/2)*i)/2 - 1/2
```

Using evalin or feval, you can make a variety of assumptions about an expression; see "Calling MuPAD Functions at the MATLAB Command Line" on page 3-28. All such assumptions can be cleared with the command syms x clear, as in this example:

```
evalin(symengine,'assume(a <> 0)');
evalin(symengine,'solve(a*x^2+b*x+c=0,x)')

ans =
{-(b - (b^2 - 4*a*c)^(1/2))/(2*a),...
 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)}

syms a clear
evalin(symengine,'solve(a*x^2+b*x+c=0,x)')

ans =
piecewise([a <> 0, {-(b - (b^2 - 4*a*c)^(1/2))/(2*a),...
  -(b + (b^2 - 4*a*c)^(1/2))/(2*a)}],...
  [a = 0 and b <> 0, {-c/b}], [a = 0 and b = 0 and c = 0, C_],...
  [a = 0 and b = 0 and c <> 0, {}])
```

Using Maple and MuPAD Engines

In this section...
"Choosing a Maple or MuPAD Engine" on page 3-35
"Differences in Maple and MuPAD Syntax" on page 3-36

Choosing a Maple or MuPAD Engine

You can use a Maple engine with Symbolic Math Toolbox software instead of the default MuPAD engine. You must have a compatible version of Maple software, available from Maplesoft®, installed on your computer. Contact `http://www.maplesoft.com/` to find which versions of Maple software work with Symbolic Math Toolbox software.

To choose a Maple engine:

1 Enter

    symengine

at the MATLAB command line to display the following GUI.
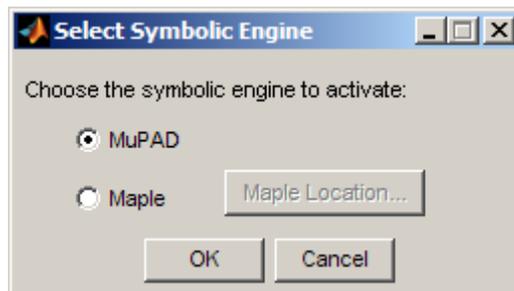
2 • To use a Maple engine, click the **Maple** button, then the **Maple Location** button, and navigate to your computer's Maple installation directory.

To determine this directory:

a Launch or switch to Maple.

b Enter the command

```
kernelopts(mapledir);
```

at the Maple prompt.

- To use a MuPAD engine, click the **MuPAD** button.

3 Clear all symbolic variables if you change engines while using Symbolic Math Toolbox software. Do this by entering the clear command, or syms *variable–list*.

Caution   Results may differ when computed with a Maple engine compared to those computed with a MuPAD engine. Also, the capabilities of the engines may differ.

Differences in Maple and MuPAD Syntax

Prior to Version 5 of Symbolic Math Toolbox, Extended Symbolic Math Toolbox™ software supported calling the Maple engine from the MATLAB workspace with the maple and procread functions. Symbolic Math Toolbox software now uses the MuPAD engine by default, and uses the evalin(symengine,...) and feval(symengine,...) functions to access the MuPAD engine from the MATLAB workspace.

Caution   Be sure to select the functions and syntax that are appropriate for your selected symbolic engine. For example, use procread and maple with a Maple engine; use evalin and feval with a MuPAD engine. The tables in the remainder of this section show the differences in syntax between the two engines.

The maple Function and MuPAD Calls

The maple function is equivalent to one of three MuPAD calls, depending on its syntax:

Maple Engine Syntax	**MuPAD Engine Syntax**
maple(*command*)	evalin(symengine,'*command*')

Maple Engine Syntax	MuPAD Engine Syntax
`maple('function',arg1,arg2,...)`	`feval(symengine, 'function', arg1, arg2,...)`
`maple restart`	`reset(symengine)`

- The content of the command and function names must be in the correct syntax, as well as the MATLAB function call. The tables in "Maple and MuPAD Syntax" on page 3-39, "Maple and MuPAD Library Calls" on page 3-40, and "Maple and MuPAD Constants" on page 3-43 describe the differences between Maple and MuPAD syntax.

- The `feval` and `evalin` functions return symbolic expressions, while the `maple` command returns strings or symbolic expressions, depending on the types of the inputs.

- To convert a symbolic expressions to a string, call the `char` method. For example, if x is a symbolic variable equal to `z^2/cos(z)`, then

      y = char(x)

  sets y to the string `'z^2/cos(z)'`.

The procread Function and evalin Calls

The `procread` function, used with a Maple engine, is equivalent to an appropriate `evalin(symengine,...)` method for a MuPAD engine. The `evalin` call should contain a command to read the code, as shown in the table below.

The following table contains an example in both Maple and MuPAD syntax.

Example of `procread` and `evalin` Calls	
procread for Maple Engine	**evalin for MuPAD Engine**

`procread('check.src')`
 	
 The `check.src` file contains the following code in Maple syntax:
 	
 ``` check := proc(A)   # check(A) computes A*inverse(A)   local X;   X := inverse(A):   evalm(A &* X); end; ```
	`evalin(symengine,'read("check.mu")')`
	
	 Note that the " in the `evalin` call are double-quote characters, not two single-quote characters.
	
	 The `check.mu` file contains the following code in MuPAD syntax:
	
	 ``` check := proc(A) // check(A) computes A*inverse(A) local X; begin X := 1/A; A * X; end_proc: ```

mhelp Calls and the doc(symengine) method

The Maple engine `mhelp` function displays help about Maple functions to the command line. The equivalent MuPAD function is the `doc` method of the object returned by `symengine`. The `doc` method opens MuPAD Help to the specified function. This table shows the differences in syntax.

Maple Syntax	MuPAD Syntax
`mhelp int`	`doc(symengine,'int')`
`mhelp convert`	`doc(symengine,'rewrite')`

The table uses the equivalence of the Maple function `convert` and the MuPAD function `rewrite`. To find the differing syntax between the two engines, see "Maple and MuPAD Syntax" on page 3-39 and "Maple and MuPAD Library Calls" on page 3-40.

Maple and MuPAD Syntax

The following table lists common tasks (meaning functions or procedures) that have differences between Maple and MuPAD syntax. Blank entries in the table mean the task is not explicitly supported. Optional arguments are enclosed in angle brackets < >.

Task	Maple Syntax	MuPAD Syntax

Comment	`#comment`	`// one-line comment`
		`/* multi-line`
		`comment */`

Previous results	`%, %%, %%%`	`%, %1, %2`

Timing	`s:=time();`
	`cmd1;...;cmdN;`
	`time()-s`
		`time(),`
		`time(cmd1,...,cmdN)`

Matrices	`<1,2;3,4>`	`matrix([[1,2],[3,4]])`
Appending	`[op(x), a, b, ...]`	`append(x, a, b, ...)`
Concatenating lists	`[op(x),op(y)]`	`x.y, [op(x),op(y)]`
Sequences	`i^2$i=1..5 or` `seq(i^2,i=1..5)`	`i^2$i=1..5`
	`seq(f, i = m..n, e)`	`f(i) $ i = m..n step e`
Types	`whattype(expr)`	`type(expr),` `domtype(expr)`
Testing types	`type(expr,type)`	`testtype(expr,type)`
Operand queries	`op(n,expr), n >= 0`	`op(expr,n), n >= 0`
Clearing variables	`x := 'x'`	`delete x`
Namespaces	`x[y], :-`	`x::y`
Employing package	`with`	`use`
String/name concatenation	`x || 10`	`x .  10`

Task	Maple Syntax	MuPAD Syntax
For loops	`for i from 6 by 2 to 100 do body end do;`	`for i from 6 to 100 step 2 do body end_for`
	`for x in obj do body; end do`	`for x in obj do body; end_for`
While loops	`while i < 100 do i := i+10; end do`	`while i < 100 do i := i+10; end_while`
Repeat		`repeat body until cond end_repeat`
If/then/else	`if b then x end if`	`if b then x end_if`
Trapping errors	`traperror(stmt)`	`traperror(stmt, <T>, <MaxSteps>)`
Try/catch	`try stmt; catch string:stmt; finally stmt end`	
Procedure definition	`f:=proc(x) local n; body; end;`	`f:=proc(x) local n; begin body; end_proc;`
Non-commuting multiplication	`&* (& is neutral-op) or A . B`	`domain overload of *`

Maple and MuPAD Library Calls

The following table lists tasks that require different library function calls in Maple syntax and MuPAD syntax. Blank entries in the table mean the call is not explicitly supported. Optional arguments are enclosed in angle brackets < >.

Task	Maple Syntax	MuPAD Syntax
Reading data	`read(filename)`	`read(filename, <Quiet>, <Plain>)`
Opening files	`open(name,mode)`	`fopen(filename | TempFile,<Read | Write | Append>, <Bin | Text | Raw>)`

Task	Maple Syntax	MuPAD Syntax
Displaying results	`print(e1, e2, ...  )`	`print(<Unquoted>, <NoNL>, <KeepOrder>, <Typeset>, object1, object2, ...)`
Substituting	`subs(x=a,f)`	`subs(f, x=a, <unsimplified>)`
Limits	`limit(f, x=a, <left | right | real | complex>)`	`limit(f, x=a, <Left | Right | Real>, <Intervals>)`
Mapping	`map(f, expr, arg1, arg2, ...)`	`map(expr, f, <arg1, arg2, ...>, <Unsimplified>)`
Logarithms	`log(x)`	`ln(x)`
Some special functions	`AiryAi(n,z)`	`airyAi(z,n)`
	`argument(z)`	`arg(z)`
	`dirac(n,x), dirac([n1..nk],[x1..xk])`	`dirac(x,n)`
	`EllipticK(z)` different norm	`EllipticK(z)`
	`EllipticPi(z,nu,k)` different norm	`EllipticPi(m,phi,n)`
	`FresnelC(x)` different norm	`fresnelC(x)`
	`GAMMA(x)`	`gamma(x)`
	`GAMMA(a,x)`	`igamma(a,x)`
	`lnGAMMA(x)`	`lngamma(x)`
	`Heaviside(t)`	`heaviside(x)`
	`JacobiAM(z,k)` different norm	`jacobiAM(u,m)`
	`Tensor[KroneckerDelta](m,n)`	`kroneckerDelta(m,n)`
	`Psi(n,x)`	`psi(x,n)`
	`Wrightomega(x)`	`wrightOmega(x)`

Task	Maple Syntax	MuPAD Syntax
Factorials	`n!` or `factorial(n)`	`n!` or `fact(n)`
Integrals	`int(f, x = a..b)`	`int(f, x = a..b, <PrincipalValue>)`
Taylor series	`taylor(f, x=a, n)`	`taylor(f, x = x0, <order>, <mode>, <NoWarning>)`
Coefficients	`coeffs(p, x, 't')`, `coeff(p, x, n)`	`coeffs(f, <vars>, <x>, n)`
Simplifying basic	`simplify(f)`	`Simplify(f, <Steps>)`, `simplify(f)`
Simplifying to a target	`simplify(f, <symbolic>, <assume=prop>, n1, n2, ...)`	`simplify(f,target)`, `target = cos, sin, exp, ln, sqrt, unit, logic, condition,` or `relation`
Simplifying special	`simplify[radical]`	`radsimp`
Combining	`combine(f, n, opt1, opt2, ...)`	`combine(f, <[target1, target2, ...]>)`
Rewriting	`convert(expr, form)`	`rewrite(expr, func)`
Collecting	`collect(f, x, <form>, <func>)`	`collect(f, x, <f>)`
Factoring	`factor(a, K)`	`factor(f, <Adjoin = adjoin>, <MaxDegree = n>)`
Finding free variables (indeterminates)	`indets(expr, <typename>)`	`freeIndets(expr)`
Solving equations	`solve(equations, variables)`	`solve(system, vars, <options>)`
Solving ODEs	`dsolve({ODE, ICs}, y(x), options)`	`solve(ODE)`
Solving recurrences	`rsolve(eqns,fcns)`	`solve(REC)`
Finding discontinuities	`discont(f,x)`	`discont(f, x = a .. b, <Undefined>)`
Making assumptions	`assume(x,real)`	`assume(x,R_)`

Task	Maple Syntax	MuPAD Syntax
Listing properties	`about(x)`	`getprop(x)`
Evaluating numerically	`evalf(expr)`	`float(expr)`
Evaluating Boolean	`evalb(expr)`	`bool(expr)`
Evaluating matrix	`evalm(a+b)`, `evalm(x.b)`	`a+b`, `a*b` matrices a,b
Parsing string	`parse(string,options)`	`text2expr(string)`
Showing code	`print(f)`	`expose(f)`
Leading terms	`lcoeff`	`lcoeff(p)`, `lterm(p)`
Partial fractions	`convert(expr, parfrac)`	`partfrac(f)`
Plotting 2-D	`plot(f,x=x0..x1)`	`plot(f1,f2,..., x=x0..x1)`, `plotfunc2d(f1,f2,..., x=x0..x1)`
Plotting 3-D	`plot3d(f, x=a..b, y=c..d)`	`plotfunc3d(f1,f2,..., x=x0..x1, y=y0..y1)`
Pseudo division	`Prem(p,q)`	`pdivide(p,q)`
Splitting objects	`selectremove`	`split(obj,f)`

Maple and MuPAD Constants

The following table lists the constants and global symbols that differ between Maple syntax and MuPAD syntax.

Symbol	Maple Syntax	MuPAD Syntax
Catalan number	`Catalan`	`CATALAN`
Digits of precision for floats	`Digits`	`DIGITS`
Euler's gamma constant	`gamma`	`EULER`
Library locations	`libname`	`LIBPATH, READPATH, PACKAGEPATH`

Symbol	Maple Syntax	MuPAD Syntax
Empty sequence	`NULL`	`null()`
Default number of terms	`Order`	`ORDER`
$\pi = 3.14159...$	`Pi`	`PI`
Three-state Boolean logic values	`True/False/FAIL`	`TRUE/FALSE/UNKNOWN`

4

Function Reference

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
mfun	Numeric evaluation of special mathematical function
mfunlist	List special functions for use with `mfun`
sinint	Sine integral
zeta	Riemann Zeta

MuPAD

doc	Get help for MuPAD functions
evalin	Evaluate MuPAD expressions
feval	Evaluate MuPAD expressions
getVar	Get variable from MuPAD notebook
mupad	Start MuPAD notebook
mupadwelcome	Launch MuPAD interfaces
openmn	Open MuPAD notebook
openmu	Open MuPAD program file
reset	Close MuPAD engine
setVar	Assign variable in MuPAD notebook
symengine	Choose symbolic engine

Pedagogical and Graphical Applications

ezcontour	Contour plotter
ezcontourf	Filled contour plotter

ezmesh	3-D mesh plotter
ezmeshc	Combined mesh and contour plotter
ezplot	Function 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

ccode	C code representation of symbolic expression
double	Convert symbolic matrix to MATLAB numeric form
fortran	Fortran representation of symbolic expression
int8, int16, int32, int64	Convert symbolic matrix to signed integers
latex	LaTeX representation of symbolic expression
matlabFunction	Convert symbolic expression to function handle or file
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

`ceil`	Round symbolic matrix toward positive infinity
`conj`	Symbolic complex conjugate
`eq`	Symbolic equality test
`findsym`	Determine variables in symbolic expression or matrix
`fix`	Round toward zero
`floor`	Round symbolic matrix toward negative infinity
`frac`	Symbolic matrix elementwise fractional parts
`imag`	Imaginary part of complex number
`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
`symvar`	Find symbolic variables in symbolic expression or matrix

Integral and Z-Transforms

`fourier`	Fourier integral transform
`ifourier`	Inverse Fourier integral transform
`ilaplace`	Inverse Laplace transform
`iztrans`	Inverse z-transform
`laplace`	Laplace transform
`ztrans`	z-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

```
syms 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 =
(a22*b1 - a21*b2)/(a11*a22 - a12*a21)

x2 =
-(a12*b1 - a11*b2)/(a11*a22 - a12*a21)
```

See Also   null, solve

Purpose	C code representation of symbolic expression

Syntax

```
ccode(s)
ccode(s,'file',fileName)
```

Description   ccode(s) returns a fragment of C that evaluates the symbolic expression s.

ccode(s,'file',fileName) writes an "optimized" C code fragment that evaluates the symbolic expression s to the file named fileName. "Optimized" means intermediate variables are automatically generated in order to simplify the code.

Examples   The statements

```
syms x
f = taylor(log(1+x));
ccode(f)
```

return

```
t0 = x-(x*x)*(1.0/2.0)+(x*x*x)*(1.0/3.0)-(x*x*x*x)*(1.0/4.0)+(x*x*x*x*x)*(1.0/5.0);
```

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;
```

The statements

```
syms x
z = exp(-exp(-x));
ccode(diff(z,3),'file','C:\ccodetest');
```

return a file named ccodetest containing the following:

```
t7 = exp(-x);
t8 = exp(-t7);
t0 = t8*exp(x*-2.0)*-3.0+t8*exp(x*-3.0)+t7*t8;
```

See Also   fortran, latex, matlabFunction, pretty

"Generating Code from Symbolic Expressions" on page 2-122

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

coeffs

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) = [ 5, 3]
coeffs(z,x) = [ 5*y^3, 3*y^2]
[c,t] = coeffs(z,y) returns c = [5*x, 3*x^2], t = [ y^3, y^2]
```

See Also   sym2poly

collect

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 =
2*exp(x) + x*(exp(x) + 2) + x^2

R2 =
y^3 + x*y^2 + (x^2 + 1)*y + x*(x^2 + 1)

R3 =
[ y + x*(y + 1) + 1, x + y]
```

See Also expand, factor, simple, simplify, syms

colspace

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`

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/(sin(y)^2 + 1)
                    compose(f,g,t)   -> 1/(sin(t)^2 + 1)
                    compose(h,g,x,z)   -> sin(z)^t
                    compose(h,g,t,z)   -> x^sin(z)
                    compose(h,p,x,y,z) -> (1/exp(z/u))^t
                    compose(h,p,t,u,z) -> x^(1/exp(y/z))

See Also   finverse, subs, syms

conj

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 6

        -Inf    -1.7279    -1.0422    -0.6492    -0.3788    -0.1778

    Columns 7 through 11

     -0.0223    0.1005    0.1983    0.2761    0.3374
```

The statements

```
syms x;
f = cosint(x);
diff(f)
```

return

```
cos(x)/x
```

See Also   sinint

det

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
```

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

```
syms a b c x y z
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]
```

diag

Suppose

```
A = sym([a,b,c;1,2,3;x,y,z])

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*x*cos(x^2)
                   ```

                   and

                   ```
                   diff(t^6,6)
                   ```

                   returns

                   ```
                   720
                   ```

See Also   `int`, `jacobian`, `findsym`

digits

Purpose	Variable precision accuracy
| **Syntax** | ```
digits
digits(d)
d = digits
``` |

Description `digits` specifies the number of significant decimal digits that MuPAD software 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 =
1.0
```

Whereas

```
digits(15)
y = vpa(x*z+1)
```

used 15-digit decimal arithmetic and returns

```
y =
1.00000000000001
```

See Also   double, vpa

dirac

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 Symbolic Math Toolbox functions 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   Get help for MuPAD functions

Syntax   doc(symengine)
doc(symengine,'MuPAD_function_name')

Description   doc(symengine) brings up the MuPAD help browser.

doc(symengine,'MuPAD_function_name') brings up the MuPAD help browser at the definition of MuPAD_function_name.

Example   doc(symengine,'simplify') brings up the following window.

See Also   "Getting Help for MuPAD" on page 3-10

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

dsolve

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

    C3/exp(a*t)

dsolve('Df = f + sin(t)') returns

    C5*exp(t) - sin(t)/2 - cos(t)/2

dsolve('(Dy)^2 + y^2 = 1','s') returns

                 -1
                  1
       sin(C12 + s)
        sin(C8 - s)

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)

z = dsolve('Dx = y', 'Dy = -x') returns

```
z =
  y: [1x1 sym]
  x: [1x1 sym]
```

Enter z.x and z.y to see the results:

```
z.x

ans =
(C23*i)/exp(i*t) - C22*i*exp(i*t)

z.y

ans =
C22*exp(i*t) + C23/exp(i*t)
```

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(rosser);
eig(R)
```

return

```
ans =

                      0
                   1000
                   1000
                   1020
    510 - 100*26^(1/2)
```

```
    100*26^(1/2) + 510
     (-10)*10405^(1/2)
        10*10405^(1/2)
```

eig(vpa(R)) returns

```
  ans =

             1020.0490184299968238463137913055
                                        1020.0
             1019.9019513592784830028224109023
                                        1000.0
                                        1000.0
          0.098048640721516997177589097721801
    3.5909362432684247788350279718981*10^(-40)
             -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 (char(A) = char(B)). eq does not expand or simplify the string expressions before making the comparison.
See Also	sym

evalin

Purpose   Evaluate MuPAD expressions

Syntax   `result = evalin(symengine,'MuPAD_expression')`
                   `[result,status] = evalin(symengine,'MuPAD_expression')`

Description   `result = evalin(symengine,'MuPAD_expression')` evaluates the
                   MuPAD expression `MuPAD_expression`, and returns `result` as a
                   symbolic object.

                   `[result,status] = evalin(symengine,'MuPAD_expression')`
                   returns the error status in `status` and the error message in `result`
                   if `status` is nonzero. If `status` is 0, `result` is a symbolic object;
                   otherwise, it is a string.

Example   `evalin(symengine,'polylib::discrim(a*x^2+b*x+c,x)')`

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

See Also   `doc`, `feval`

                   "Calling MuPAD Functions at the MATLAB Command Line" on page
                   3-28

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];
simplify(expm(t*A))
```

return

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

expand

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 often used with polynomials. It also expands trigonometric, exponential, and logarithmic functions.

Examples   `syms x y a b t;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(2*a*b)*exp(a^2)*exp(b^2)
```

`expand([sin(2*t), cos(2*t)])` returns

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

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 software 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

```
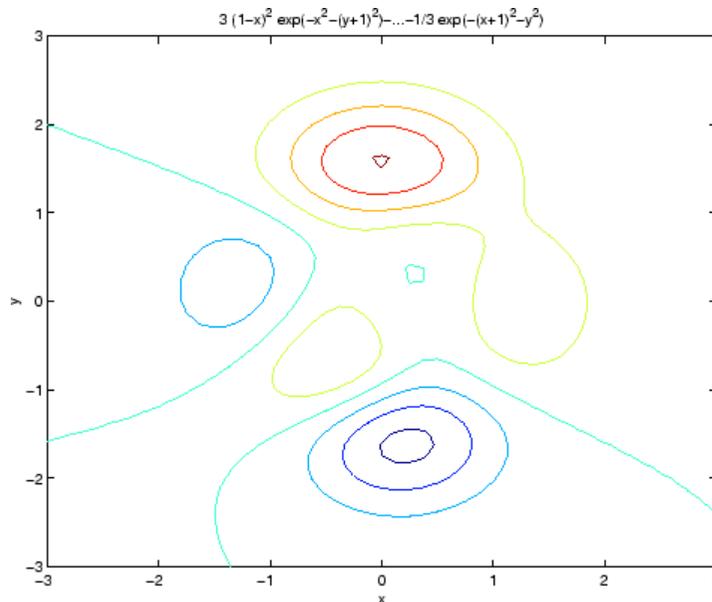syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2) ...
    - 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2) ...
    - 1/3*exp(-(x+1)^2 - y^2);
```

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 software 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

```
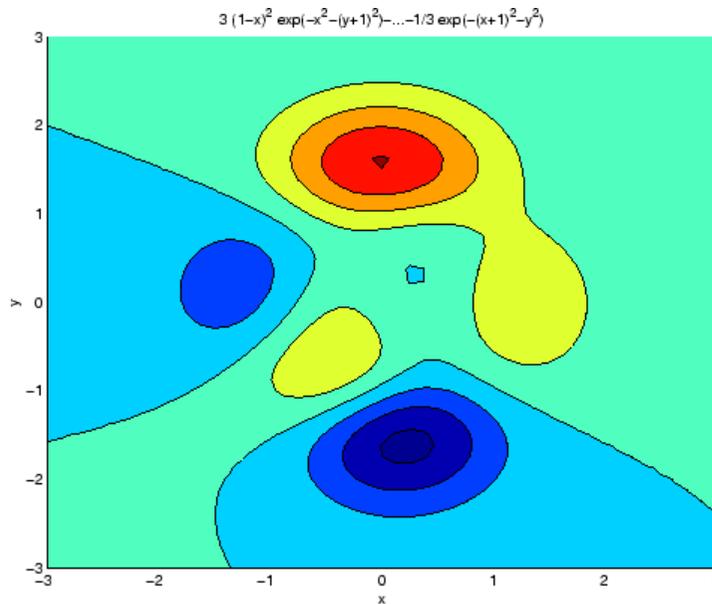syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2) ...
    - 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2) ...
    - 1/3*exp(-(x+1)^2 - y^2);
```

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)
```


$3 (1-x)^2 \exp(-x^2-(y+1)^2)-\ldots-1/3 \exp(-(x+1)^2-y^2)$

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

ezmesh

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 software 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.

```
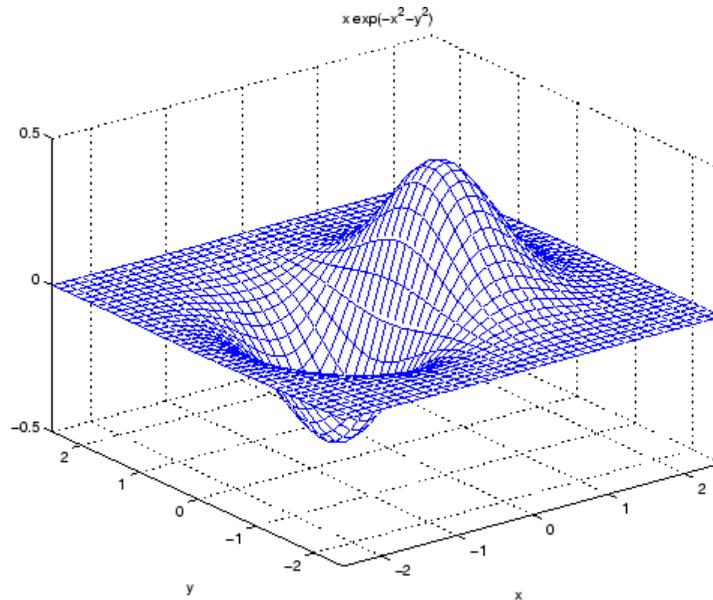syms x y
ezmesh(x*exp(-x^2-y^2),[-2.5,2.5],40)
colormap([0 0 1])
```


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 software 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

ezplot

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)
```
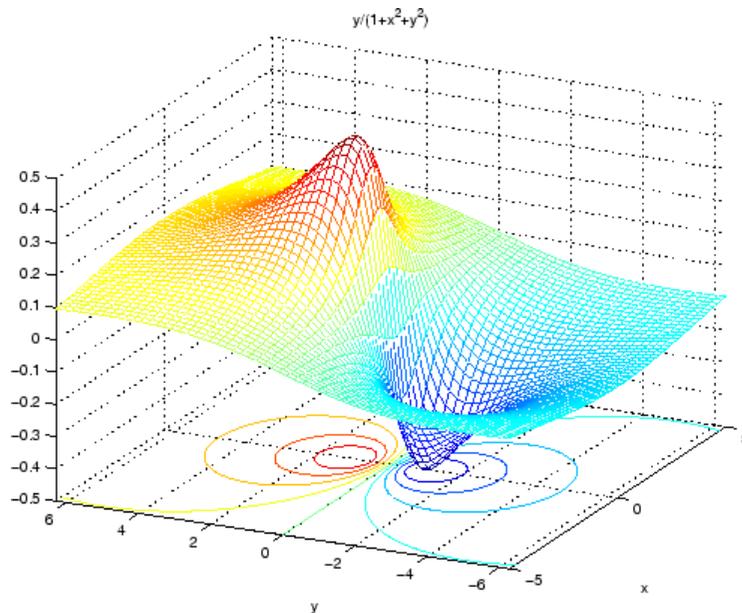
Description ezplot(f) plots the expression $f = f(x)$ over the default domain
$-2\pi < x < 2\pi$.

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.

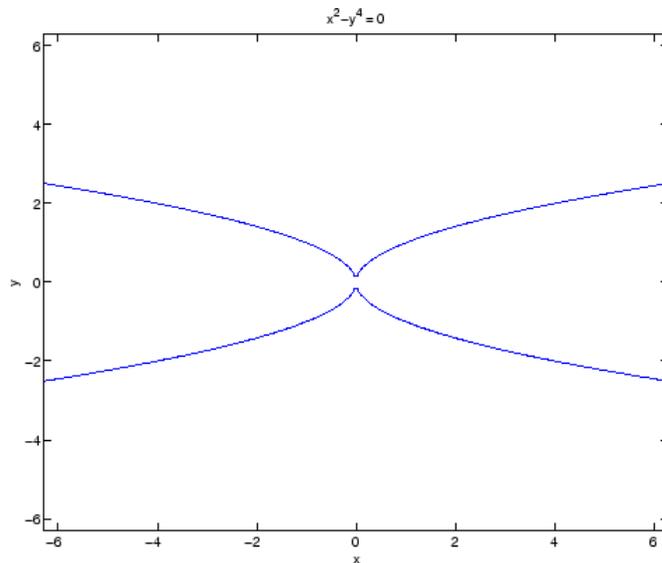**Examples**   This example plots the implicitly defined function,

$x^2 - y^4 = 0$

over the domain [-2π, 2π].

```
syms x y
ezplot(x^2-y^4)
```
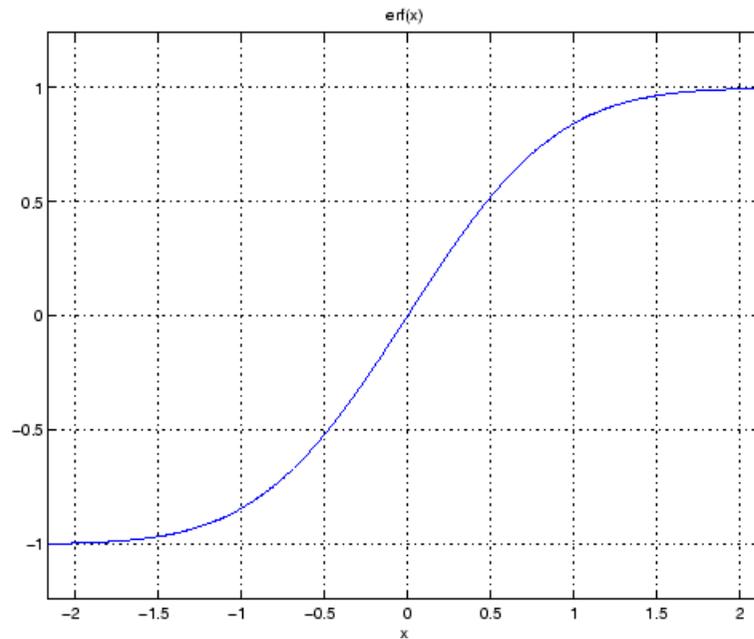

$x^2 - y^4 = 0$

ezplot

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])
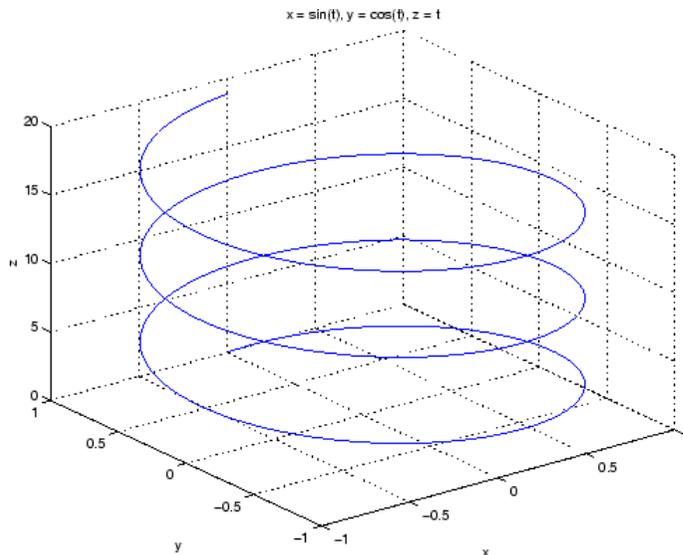
ezplot3

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 $r = f(\theta)$ over the default domain
                $0 < \theta < 2\pi$.
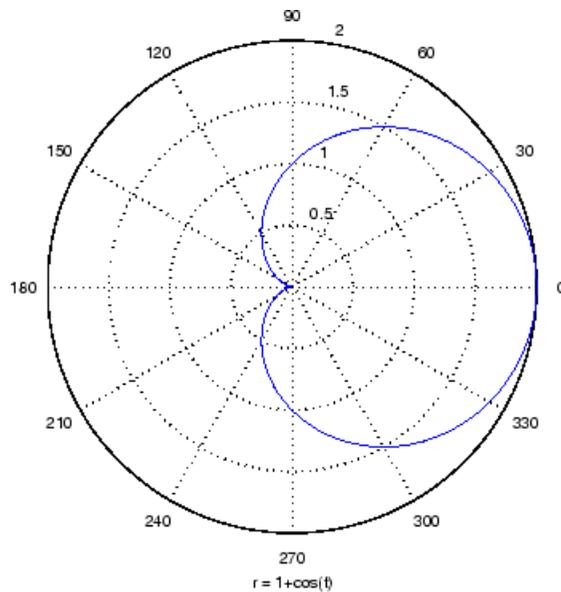
                ezpolar(f,[a,b]) plots *f* for a $< \theta <$ b.

Example   This example creates a polar plot of the function,

                $1 + \cos(t)$

                over the domain [0, 2π].

```
syms t
ezpolar(1+cos(t))
```

ezsurf

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 software 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 NaNs, which MATLAB does not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function,

$$f(x,y) = real(\text{atan}(x + iy))$$

over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

```
syms x y
ezsurf(real(atan(x+i*y)))
```


1/2 atan(x+i y)+1/2 conj(atan(x+i y))
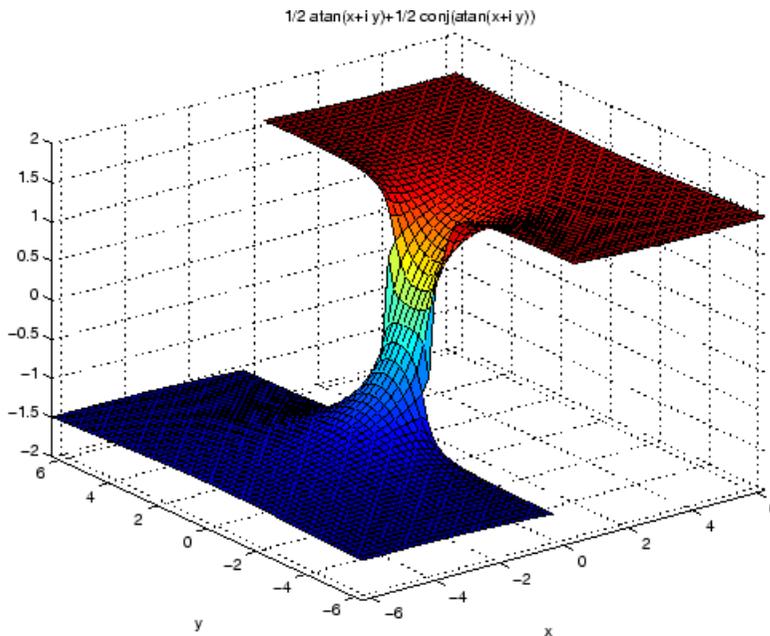
Note also that ezsurf creates graphs that have axis labels, a title, and extend to the axis limits.

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 software 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).
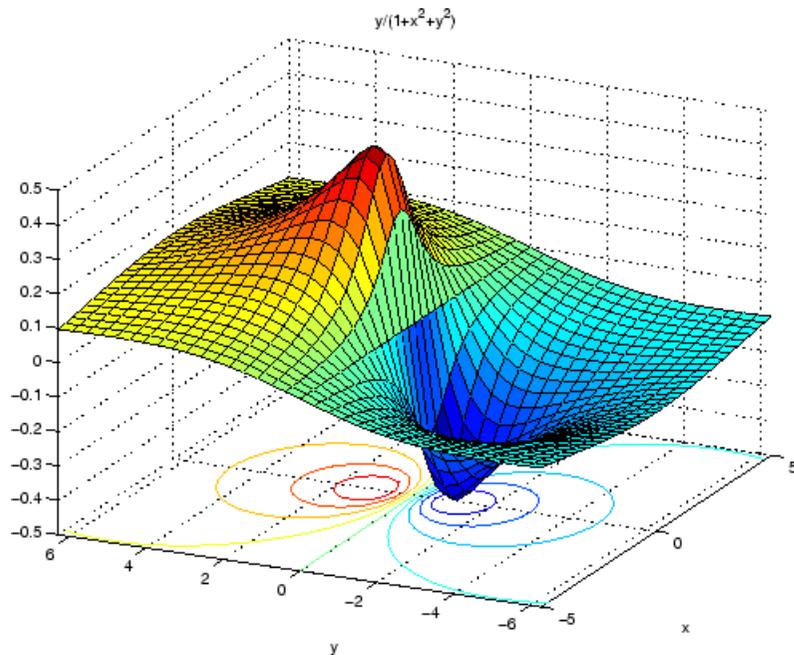
See Also   ezcontour, ezcontourf, ezmesh, ezmeshc, ezplot, ezpolar, ezsurf, surfc

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	`syms x y a b; 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*3541*3607*3803*27961`
See Also	`collect`, `expand`, `horner`, `simplify`, `simple`

feval

Purpose   Evaluate MuPAD expressions

Syntax   ```
result = feval(symengine,F,x1,...,xn)
[result,status] = feval(symengine,F,x1,...,xn)
```

**Description**   `result = feval(symengine,F,x1,...,xn)` evaluates F, which is
either a MuPAD function name or a symbolic object, with arguments
x1,...,xn, with `result` a symbolic object.

`[result,status] = feval(symengine,F,x1,...,xn)` returns the
error status in `status`, and the error message in g if `status` is nonzero.
If `status` is 0, `result` is a symbolic object; otherwise, it g is a string.

**Examples**
```
syms a b c x
p = a*x^2+b*x+c;
feval(symengine,'polylib::discrim',p,x)

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

Alternatively, the same calculation based on variables not defined in
the MATLAB workspace is:

```
feval(symengine,'polylib::discrim','a*x^2 + b*x + c', 'x')

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

See Also   doc, evalin

"Calling MuPAD Functions at the MATLAB Command Line" on page
3-28

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)
ans = y
```

See Also   compose, diff, int, limit, symvar, taylor

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   syms x u v; finverse(1/tan(x)) returns

                      atan(1/x)

                   finverse(exp(u-2*v),u) returns

                      2*v + log(u)

See Also   compose, syms

fix

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

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

fortran

Purpose	Fortran representation of symbolic expression
Syntax	`fortran(S)` `fortran(S,'file',fileName)`
Description	`fortran(S)` returns the Fortran code equivalent to the expression `S`. `fortran(S,'file',fileName)` writes an "optimized" Fortran code fragment that evaluates the symbolic expression `S` to the file named `fileName`. "Optimized" means intermediate variables are automatically generated in order to simplify the code.
Examples	The statements

```
syms x
f = taylor(log(1+x));
fortran(f)
```

return

```
t0 = x-x**2*(1.0D0/2.0D0)+x**3*(1.0D0/3.0D0)-x**4*(1.0D0/4.0D0)+x*
     ~*5*(1.0D0/5.0D0)
```

The statements

```
H = sym(hilb(3));
fortran(H)
```

return

```
        H(1,1) = 1
        H(1,2) = 1.0D0/2.0D0
        H(1,3) = 1.0D0/3.0D0
        H(2,1) = 1.0D0/2.0D0
        H(2,2) = 1.0D0/3.0D0
        H(2,3) = 1.0D0/4.0D0
        H(3,1) = 1.0D0/3.0D0
        H(3,2) = 1.0D0/4.0D0
```

```
        H(3,3) = 1.0D0/5.0D0
```

The statements

```
syms x
z = exp(-exp(-x));
fortran(diff(z,3),'file','C:\fortrantest');
```

return a file named `fortrantest` containing the following:

```
        t7 = exp(-x)
        t8 = exp(-t7)
        t0 = t8*exp(x*(-2))*(-3)+t8*exp(x*(-3))+t7*t8
```

See Also `ccode`, `latex`, `matlabFunction`, `pretty`

"Generating Code from Symbolic Expressions" on page 2-122

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,

$$F(w) = \int\limits_{-\infty}^{\infty} f(x)e^{-iwx}dx$$

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`.

$$F(v) = \int\limits_{-\infty}^{\infty} f(x)e^{-ivx}dx$$

`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.

$$F(v) = \int\limits_{-\infty}^{\infty} f(u)e^{-ivu}du$$

Examples

Fourier Transform	MATLAB Commands

$f(x) = e^{-x^2}$ $$F[f](w) = \int\limits_{-\infty}^{\infty} f(x)e^{-ixw}dx$$ $$= \sqrt{\pi}\,e^{-w^2/4}$$	`syms x`
	`f = exp(-x^2)`
	`fourier(f)`
	returns
	`pi^(1/2)/exp(w^2/4)`

$g(w) = e^{-|w|}$ $$F[g](t) = \int\limits_{-\infty}^{\infty} g(w)e^{-itw}dw$$ $$= \frac{2}{1+t^2}$$	`syms w`
	`g = exp(-abs(w))`
	`fourier(g)`
	returns
	`2/(v^2 + 1)`

fourier

Fourier Transform	MATLAB Commands

$f(x) = xe^{-\lvert x \rvert}$
 	
$F[f](u) = \int\limits_{-\infty}^{\infty} f(x)e^{-ixu}dx$
 	
$= -\dfrac{4iu}{(1+u^2)^2}$
	```matlab syms x u```
	 `f = x*exp(-abs(x))`
	 `fourier(f,u)`
	 returns
	 `-(4*i*u)/(u^2 + 1)^2`

$f(x,v) = e^{-x^2\frac{\lvert v \rvert \sin v}{v}},\ x \text{ real}$
 	
$F[f(v)](u) = \int\limits_{-\infty}^{\infty} f(x,v)e^{-ivu}dv$
 	
$= -\arctan\dfrac{u-1}{x^2} + \arctan\dfrac{u+1}{x^2}$
	`syms v u`
	 `syms x real`
	 `f = exp(-x^2*abs(v))*sin(v)/v`
	 `fourier(f,v,u)`
	 returns
	 `piecewise([x <> 0, atan((u + 1)/x^2) - atan(1/x^2*(u - 1))])`

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.

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

getVar

Purpose	Get variable from MuPAD notebook
Syntax	`y = getVar(nb,'z')`
Description	`y = getVar(nb,'z')` assigns the symbolic expression z in the MuPAD notebook nb to a symbolic variable y in the MATLAB workspace.

Example	``mpnb = mupad;``
	``(make a variable f:=x^2 in the MuPAD notebook)``
	``f = getVar(mpnb,'f')``
	
	f is a symbolic variable in the MATLAB workspace, with value x^2.

See Also	`mupad, setVar`

Purpose   Compute Heaviside step function

Syntax   `heaviside(x)`

Description `heaviside(x)` has the value 0 for `x < 0`, 1 for `x > 0`, and 0.5 for `x = 0`. `heaviside(sym(0)) = 1/2`.

See Also   `dirac`

horner

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
```
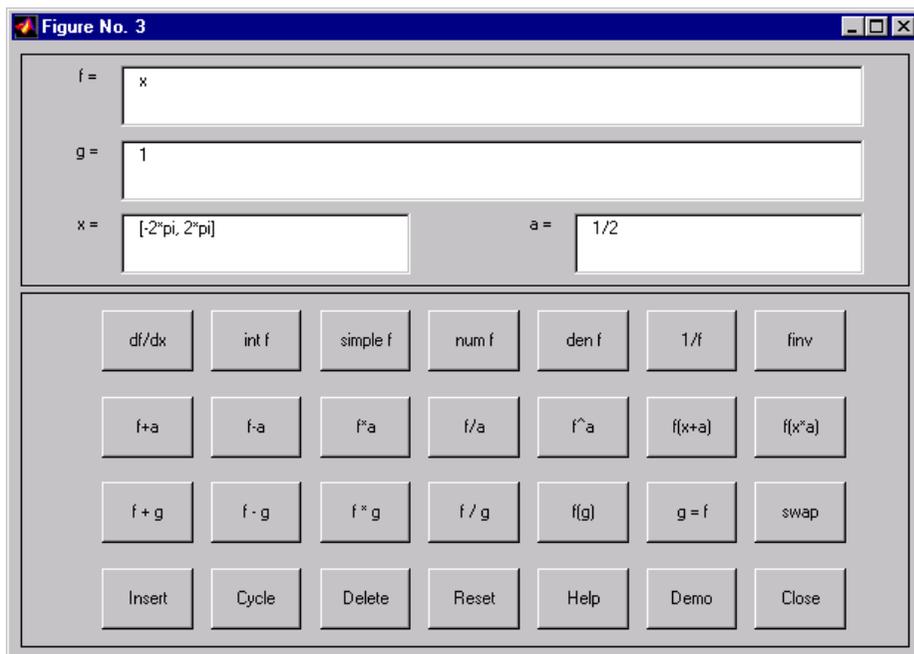syms x y
horner(x^3-6*x^2+11*x-6)
```

returns

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

horner([x^2+x;y^3-2*y]) returns

```
  x*(x + 1)
 y*(y^2 - 2)
```

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/(z - 1)

hypergeom(a,[],z) returns 1/(1 - z)^a

ifourier

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 Commands

$f(w) = e^{-w^2/(4a^2)}$
 	
 $F^{-1}[f](x) = \dfrac{1}{2\pi} \displaystyle\int_{-\infty}^{\infty} f(w)e^{ixw}dw$
 	
 $= \dfrac{\lvert a \rvert}{\sqrt{\pi}} e^{-(ax)^2}$
	`syms a w real`
	 `f = exp(-w^2/(4*a^2))`
	 `F = ifourier(f)`
	 `F = simple(F)`
	 returns
	 `abs(a)/(pi^(1/2)*exp(a^2*x^2))`

$g(x) = e^{-\lvert x \rvert}$
 	
 $F^{-1}[g](t) = \dfrac{1}{2\pi} \displaystyle\int_{-\infty}^{\infty} g(x)e^{itx}dx$
 	
 $= \dfrac{1}{\pi\left(1+t^2\right)}$
	`syms x real`
	 `g = exp(-abs(x))`
	 `ifourier(g)`
	 returns
	 `1/(pi*(t^2 + 1))`

$f(w) = 2e^{-\lvert w \rvert} - 1$
 	
 $F^{-1}[f](t) = \dfrac{1}{2\pi} \displaystyle\int_{-\infty}^{\infty} f(w)e^{itw}dw$
 	
 $= -\mathrm{dirac}(t) + \dfrac{2}{\pi(1+t^2)}$
	`syms w t real`
	 `f = 2*exp(-abs(w)) - 1`
	 `simple(ifourier(f,t))`
	 returns
	 `2/(pi*t^2 + pi) - dirac(t)`

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

$$F(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} L(s)e^{st}ds,$$

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.

$$F(y) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} L(y)e^{sy}ds$$

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.

$$F(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} L(y)e^{xy}dy$$

Examples

Inverse Laplace Transform	MATLAB Command

$f(s) = \dfrac{1}{s^2}$
 	
 $L^{-1}[f] = \dfrac{1}{2\pi i} \int_{c-ivo}^{c+ivo} f(s)e^{st}ds$
 	
 $= t$
	```syms s```
	 ```f = 1/s^2```
	 ```ilaplace(f)```
	 returns
	 ```t```

$g(t) = \dfrac{1}{(t-a)^2}$
 	
 $L^{-1}[g] = \dfrac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} g(t)e^{xt}dt$
 	
 $= xe^{ax}$
	```syms a t```
	 ```g = 1/(t-a)^2```
	 ```ilaplace(g)```
	 returns
	 ```x*exp(a*x)```

$f(u) = \dfrac{1}{u^2 - a^2}$
 	
 $L^{-1}[f] = \dfrac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} g(u)e^{xu}du$
 	
 $= \dfrac{\sinh(xa)}{a}$
	```syms x u```
	 ```syms a real```
	 ```f = 1/(u^2-a^2)```
	 ```simplify(ilaplace(f,x))```
	 returns
	 ```sinh(a*x)/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 S with respect to the 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

```
syms x t z alpha;
int(-2*x/(1+x^2)^2)
```

returns

```
1/(x^2 + 1)
```

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

```
cos(t)^2
```

int([exp(t),exp(alpha*t)]) returns

```
[ exp(t), exp(alpha*t)/alpha]
```

See Also   diff, symsum

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

inv

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

```
[      -(t - 2)*(t - 3)^2, t^3 - 9*t^2 + 26*t - 24]
[ t^3 - 9*t^2 + 26*t - 24,      -(t - 3)^2*(t - 4)]
```

the symbolic inverse of the 2-by-2 Hilbert matrix.

See Also   vpa, Arithmetic Operations

iztrans

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.

$$f(n) = \frac{1}{2\pi i} \oint_{|z|=R} F(z)z^{n-1}dz, \; n = 1, 2, \ldots$$

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

$f(z) = \dfrac{2z}{(z-2)^2}$
 	
 $Z^{-1}[f] = \dfrac{1}{2\pi i} \oint\limits_{|z|=R} f(s)z^{n-1}dz$
 	
 $= n2^n$
	`syms z`
	 `f = 2*z/(z-2)^2`
	 `iztrans(f)`
	 returns
	 `2^n + 2^n*(n - 1)`

$g(n) = \dfrac{n(n+1)}{n^2 + 2n + 1}$
 	
 $Z^{-1}|g| = \dfrac{1}{2\pi i} \oint\limits_{|n|=R} g(n)n^{k-1}dn$
 	
 $= -1^k$
	`syms n`
	 `g = n*(n+1)/(n^2+2*n+1)`
	 `iztrans(g)`
	 returns
	 `(-1)^k`

$f(z) = \dfrac{z}{z-a}$
 	
 $Z^{-1}[f] = \dfrac{1}{2\pi i} \oint\limits_{|z|=R} f(z)z^{k-1}dz$
 	
 $= a^k$ if $a \neq 0$
	`syms z a k`
	 `f = z/(z-a)`
	 `simplify(iztrans(f,k))`
	 returns
	 `piecewise([a = 0,`
	 `kroneckerDelta(k, 0)],`
	 `[a <> 0, a^k])`

See Also   `ifourier`, `ilaplace`, `ztrans`

jacobian

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

```
syms x y z
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

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
    -1     0     0
     2     0     1

J =
     2     1     0
     0     2     0
     0     0     3
```

Then the statement

```
V\A*V
```

returns

```
ans =
     2     1     0
     0     2     0
     0     0     3
```

See Also   eig, poly

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(0, x)]
[ lambertw(0, 1), lambertw(0, 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

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

$$L(s) = \int\limits_{0}^{\infty} F(t)e^{-st}dt$$

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`.

$$L(t) = \int\limits_{0}^{\infty} F(x)e^{-tx}dx$$

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.

laplace

$$L(z) = \int\limits_0^\infty F(w)e^{-zw}dw$$

Examples

Laplace Transform	MATLAB Command

$f(t) = t^4$
 	
 $L[f] = \int\limits_0^\infty f(t)e^{-ts}dt$
 	
 $= \dfrac{24}{s^5}$
	```syms t```
	```f = t^4```
	```laplace(f)```
	returns
	```24/s^5```

$g(s) = \dfrac{1}{\sqrt{s}}$
 	
 $L[g](t) = \int\limits_0^\infty g(s)e^{-st}ds$
 	
 $= \sqrt{\dfrac{\pi}{t}}$
	```syms s```
	```g = 1/sqrt(s)```
	```laplace(g)```
	returns
	```pi^(1/2)/t^(1/2)```

$f(t) = e^{-at}$
 	
 $L[f](x) = \int\limits_0^\infty f(t)e^{-tx}dt$
 	
 $= \dfrac{1}{x+a}$
	```syms t a x```
	```f = exp(-a*t)```
	```laplace(f,x)```
	returns
	```1/(a + x)```

See Also   fourier, ilaplace, ztrans

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

```
\frac{x^5}{5} - \frac{x^4}{4} + \frac{x^3}{3} - \frac{x^2}{2} + x
```

The statements

```
H = sym(hilb(3));
latex(H)
```

return

```
\left(\begin{array}{ccc} 1 & \frac{1}{2} & \frac{1}{3}\\ ...
\frac{1}{2} & \frac{1}{3} & \frac{1}{4}\\ ...
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{array}\right)
```

The statements

```
syms alpha t
A = [alpha t alpha*t];
latex(A)
```

return

```
\left(\begin{array}{ccc} \mathrm{alpha} & t & \mathrm{alpha}\, ...
 t \end{array}\right)
```

latex

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)               => [ 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`

matlabFunction

Purpose	Convert symbolic expression to function handle or file
Syntax	`g = matlabFunction(f)` `g = matlabFunction(f,param1,value1,...)`

Description   `g = matlabFunction(f)` converts the symbolic expression `f` to a function handle `g`.

`g = matlabFunction(f,param1,value1,...)` converts the symbolic expression `f` to a function handle `g`, with the following options for parameter/value pairs:

- Parameter = `'vars'` allows you to set the order of the variables in the resulting function handle or M-file. The default order is alphabetical. The `value` should be either a cell array of strings, or a vector of symbolic variables. There must be at least as many value entries as there are free variables in the symbolic expression `f`.

- Parameter = `'file'` allows you to generate an *optimized* M-file that can accept double or matrix arguments and evaluate the symbolic expression applied to the arguments. Optimized means intermediate variables are automatically generated in order to simplify or speed the code. The `value` should be a string representing the path to the M-file. If the string does not end in ".m", `matlabFunction` appends ".m"

Examples

```
syms x y
r = sqrt(x^2 + y^2);
ht = matlabFunction((sin(r)/r)*exp(-r/4))


ht =
  @(x,y)sin((x.^2+y.^2).^(1./2))./(exp((x.^2+y.^2).^(1./2)./4).*(x.^2+y.^2).^(1./2))
```

The following example generates a file:

```
syms x y z
r = x^2 + y^2 + z^2;
f=matlabFunction(log(r)+r^(-1/2),'file','C:\Work\myfile')
```

Here is the resulting optimized file, `myfile.m`, located in the `C:\Work` directory:

```
function RESULT = myfile(x,y,z)
%MYFILE
%    RESULT = MYFILE(X,Y,Z)

%    This function was generated by the Symbolic Math Toolbox version 5.0.
%    11-Jul-2008 14:26:44

t4 = x.^2;
t5 = y.^2;
t6 = z.^2;
t7 = t4 + t5 + t6;
RESULT = log(t7)+1./t7.^(1./2);
```

See Also   ccode, fortran, subs, sym2poly

"Generating Code from Symbolic Expressions" on page 2-122

mfun

Purpose	Numeric evaluation of special mathematical 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 MuPAD software. Each `par` argument is a numeric quantity corresponding to a 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 specifications for `function`. You can access parameter information for `mfun` functions in "Syntax and Definitions of mfun Special Functions" on page 5-99.

MuPAD software 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`

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.

Syntax and Definitions of mfun Special Functions

The following conventions are used in the next 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)	$n \geq 0$ $0 < \|t\| < 2\pi$
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)	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Binomial coefficients	$\left(\dfrac{m}{n}\right) = \dfrac{m!}{n!(m-n)!}$ $= \dfrac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$	`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, $0 < k < 1.$
Complementary error function and its iterated integrals	$erfc(z) = \dfrac{2}{\sqrt{\pi}} \cdot \displaystyle\int_z^\infty e^{-t^2} dt = 1 - erf(z)$ $erfc(-1,z) = \dfrac{2}{\sqrt{\pi}} \cdot e^{-z^2}$ $erfc(n,z) = \displaystyle\int_z^\infty erfc(n-1,t)dt$	`erfc(z)` `erfc(n,z)`	$n > 0$
Dawson's integral	$F(x) = e^{-x^2} \cdot \displaystyle\int_0^x e^{t^2} dt$	`dawson(x)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Digamma function	$\Psi(x) = \dfrac{d}{dx} \ln(\Gamma(x)) = \dfrac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	
Dilogarithm integral	$f(x) = \displaystyle\int_1^x \dfrac{\ln(t)}{1-t} dt$	`dilog(x)`	$x > 1$
Error function	$erf(z) = \dfrac{2}{\sqrt{\pi}} \displaystyle\int_0^z e^{-t^2} dt$	`erf(z)`	
Euler numbers and polynomials	Generating function for Euler numbers: $\dfrac{1}{\cosh(t)} = \displaystyle\sum_{n=0}^{\infty} E_n \dfrac{t^n}{n!}$	`euler(n)` `euler(n,z)`	$n \geq 0$ $\|t\| < \dfrac{\pi}{2}$
Exponential integrals	$Ei(n,z) = \displaystyle\int_1^{\infty} \dfrac{e^{-zt}}{t^n} dt$ $Ei(x) = PV\left(-\displaystyle\int_{-\infty}^{x} \dfrac{e^t}{t}\right)$	`Ei(n,z)` `Ei(x)`	$n \geq 0$ $\text{Real}(z) > 0$
Fresnel sine and cosine integrals	$C(x) = \displaystyle\int_0^x \cos\left(\dfrac{\pi}{2} t^2\right) dt$ $S(x) = \displaystyle\int_0^x \sin\left(\dfrac{\pi}{2} t^2\right) dt$	`FresnelC(x)` `FresnelS(x)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
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)`	
(Generalized) hypergeometric function	$$F(n,d,z) = \sum_{k=0}^{\infty} \frac{\prod_{i=1}^{j} \frac{\Gamma(n_i+k)}{\Gamma(n_i)} \cdot z^k}{\prod_{i=1}^{m} \frac{\Gamma(d_i+k)}{\Gamma(d_i)} \cdot k!}$$ 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, $0 < k < 1$.

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Incomplete gamma function	$\Gamma(a,z) = \int\limits_{z}^{\infty} e^{-t} \cdot t^{a-1} dt$	GAMMA(z1,z2) z1 = a z2 = z	
Logarithm of the gamma function	$\ln\text{GAMMA}(z) = \ln(\Gamma(z))$	lnGAMMA(z)	
Logarithmic integral	$Li(x) = PV \left\{ \int\limits_{0}^{x} \frac{dt}{\ln t} \right\} = Ei(\ln x)$	Li(x)	$x > 1$
Polygamma function	$\Psi^{(n)}(z) = \dfrac{d^n}{dz} \Psi(z)$ where $\Psi(z)$ is the Digamma function.	Psi(n,z)	$n \geq 0$
Shifted sine integral	$Ssi(z) = Si(z) - \dfrac{\pi}{2}$	Ssi(z)	

The following orthogonal polynomials are available using mfun. In all cases, n is a nonnegative integer and x is real.

Orthogonal Polynomials

Polynomial	mfun Name	Arguments
Chebyshev of the first and second kind	T(n,x) U(n,x)	

Orthogonal Polynomials (Continued)

Polynomial	mfun 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)	
Jacobi	P(n,a,b,x)	a, b are nonrational algebraic expressions or rational numbers greater than -1.
Laguerre	L(n,x)	
Generalized Laguerre	L(n,a,x)	a is a nonrational algebraic expression or a rational number greater than -1.
Legendre	P(n,x)	

Examples

```
mfun('H',5,10)

ans =
    3041200

mfun('dawson',3.2)

ans =
    0.1655
```

Limitations   In general, the accuracy of a function will be lower near its roots and when its arguments are relatively large.

Running time depends on the specific function and its parameters. In general, calculations are slower than standard MATLAB calculations.

See Also   `mfun`

References   [1] Abramowitz, M. and I.A., Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965.

mod

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`

Purpose	Start MuPAD notebook
Syntax	`mphandle = mupad` `mphandle = mupad(file)`
Description	`mphandle = mupad` creates a MuPAD notebook, and keeps a handle (pointer) to the notebook in the variable `mphandle`. You can use any variable name you like instead of `mphandle`. `mphandle = mupad(file)` opens the MuPAD notebook or program file named `file`, and keeps a handle (pointer) to the notebook or program file in the variable `mphandle`. This syntax has the functionality of both `openmn` and `openmu`.

Examples	To start a new notebook and define a handle `mphandle` to the notebook, enter:
	
	   `mphandle = mupad`
	
	To open an existing notebook named `notebook1.mn` located in the `C:\work` directory, and define a handle `mphandle` to the notebook, enter:
	
	   `mphandle = mupad('C:\Work\notebook1.mn');`

See Also	`getVar`, `mupadwelcome`, `openmn`, `openmu`, `setVar`

mupadwelcome

Purpose   Launch MuPAD interfaces

Syntax   `mupadwelcome`

Description   `mupadwelcome` brings up a window that enables you to launch various
                   MuPAD interfaces:

- Notebook, for performing calculations
- Editor, for writing programs and libraries
- Help, in the **First Steps** pane

It also enables you to access recent MuPAD files or browse for files.
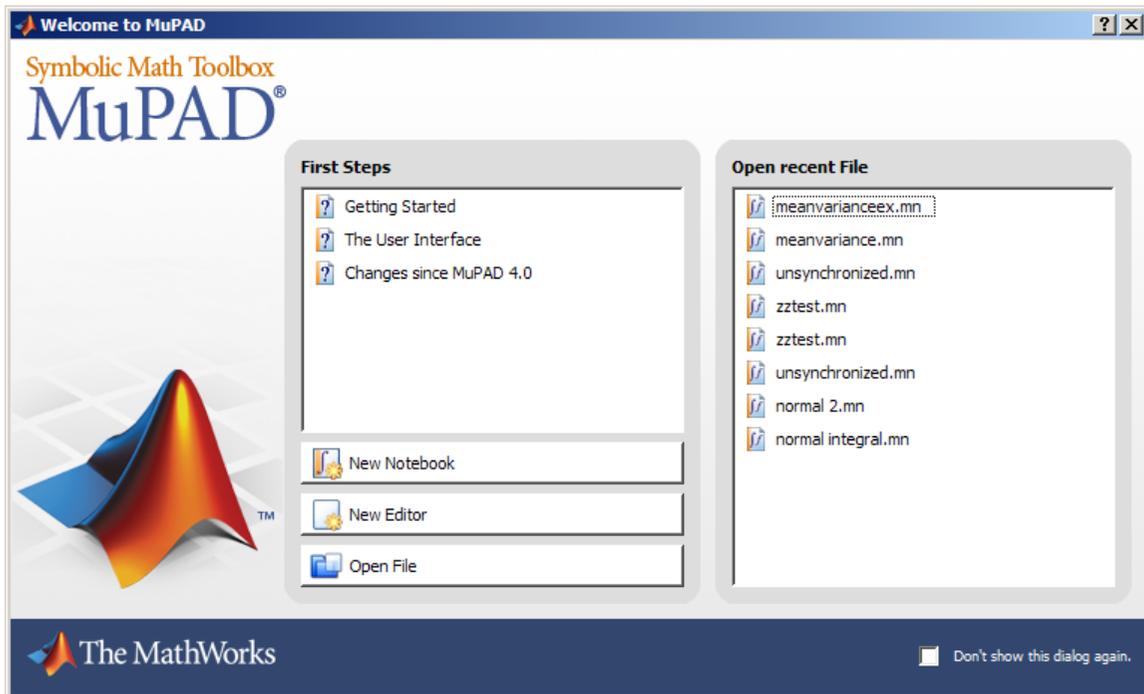
If you have disabled the `mupadwelcome` screen by checking the **Don't show this dialog again** box, you can enable it again by opening the MuPAD Help browser, and selecting the menu item **View > Configure**. From the **User Interface** menu, uncheck **Hide welcome dialog**.

If you are currently running a MuPAD notebook or editor, `mupadwelcome` brings up a new MuPAD notebook.

See Also

`mupad`

"Opening Recent Files and Other MuPAD Interfaces" on page 3-13

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`.

                   `syms x y; [n,d] = numden(x/y + y/x)` returns

```
  n =
  x^2 + y^2

  d =
  x*y
```

The statements

```
  syms a b
  A = [a, 1/b]
  [n,d] = numden(A)
```

return

```
  A =
  [a, 1/b]

  n =
  [a, 1]

  d =
  [1, b]
```

openmn

Purpose	Open MuPAD notebook
Syntax	nb = openmn(file)
Description	nb = openmn(file) opens the MuPAD notebook file, and returns a handle to the file in nb. nb = mupad(file) accomplishes the same task.
Example	To open a notebook named e-e-x.mn in the directory \Documents\Notes 2008b of drive H:, enter

```
nb = openmn('H:\Documents\Notes 2008b\e-e-x.mn');
```

See Also   mupad, openmu

Purpose	Open MuPAD program file
Syntax	`nb = openmu(file)`
Description	`nb = openmu(file)` opens the MuPAD program `file`, and returns a handle to the file in `nb`. `nb = mupad(file)` accomplishes the same task.
Example	To open a program file named `yyx.mu` in the directory `\Documents\Notes 2008b` of drive `H:`, enter
	`nb = openmu('H:\Documents\Notes 2008b\yyx.mu');`
See Also	`mupad, openmn`

poly

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

$$c_1 x^{n-1} + c_2 x^{n-2} + ... + c_n$$

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*x^2)/9007199254740992 + (333*x)/1000 + 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 - 5^(1/2)/2
 5^(1/2)/2 + 3/2


  +-            -+
  |         1/2  |
  |        5     |
  |  3/2 - ----  |
  |         2    |
  |              |
  |    1/2       |
  |   5          |
  |  ---- + 3/2  |
  |    2         |
  +-            -+
```

quorem

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 elementwise 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 = 4*x - 5*x^2 - 10
[q,r] = quorem(10^5,subs(p,'10'))
q = 101
r = 515
```

See Also   `mod`

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.

real

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	Close MuPAD engine

Syntax   reset(symengine)

Description   reset(symengine) closes the MuPAD engine associated with the
                   MATLAB workspace, and resets all its assumptions. Immediately
                   before or after executing reset(symengine) you should clear all
                   symbolic objects in the MATLAB workspace.

See Also   symengine

round

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

rref

Purpose   Reduced row echelon form

Syntax   rref(A)

Description   rref(A) is the reduced row echelon form of the symbolic matrix A.

> **Note** Matrices whose elements are free symbolic variables are
> regarded as nonzero.

Examples   rref(sym(magic(4))) returns

```
[ 1,  0,  0,  1]
[ 0,  1,  0,  3]
[ 0,  0,  1, -3]
[ 0,  0,  0,  0]
```

rsums

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.
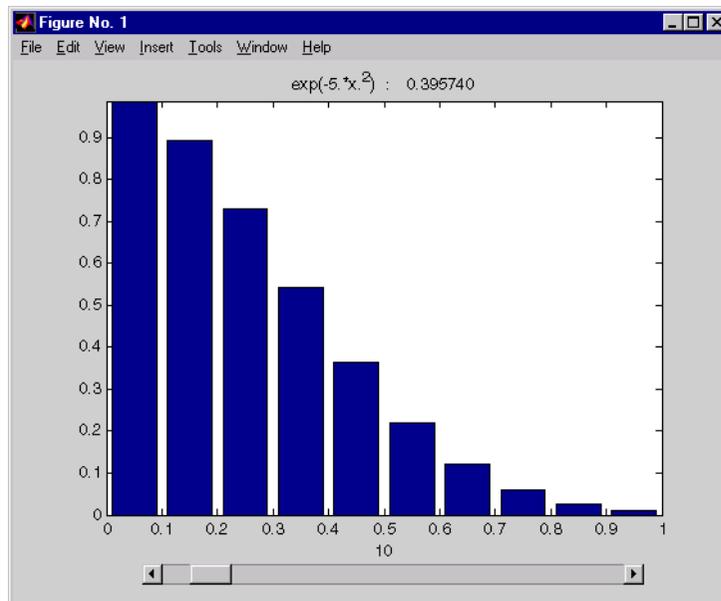
Purpose   Assign variable in MuPAD notebook

Syntax   setVar(nb,y)
                 setVar(nb,'v',y)

Description setVar(nb,y) assigns the symbolic expression y in the MATLAB
                 workspace to the variable y in the MuPAD notebook nb.

                 setVar(nb,'v',y) assigns the symbolic expression y in the MATLAB
                 workspace to the variable v in the MuPAD notebook nb.

Examples   mpnb = mupad;
                    syms x;
                    y = exp(-x);
                    setVar(mpnb,y)
                    setVar(mpnb,'z',sin(y))

                 After executing these statements, the MuPAD engine associated with
                 the mpnb notebook contains the variables y, with value exp(-x), and z,
                 with value sin(exp(-x)).

See Also   getVar, mupad

simple

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(100)
cos(x)^2-sin(x)^2	cos(2*x)	simplify
cos(x)+i*sin(x)	exp(i*x)	rewrite(exp)
(x+1)*x*(x-1)	x^3 - x	simplify(100)
x^3+3*x^2+3*x+1	(x + 1)^3	simplify
cos(3*acos(x))	4*x^3 - 3*x	simplify(100)

See Also   collect, expand, factor, horner, simplify

Purpose	Symbolic simplification

Syntax
```
R = simplify(S)
R = simplify(S, n)
```

Description   R = simplify(S) simplifies each element of the symbolic matrix S using MuPAD simplification rules.

R = simplify(S, n) uses the positive integer n to control how many steps of simplification the simplify function attempts. The default value (without n) is n = 50.

Examples   syms x a b c; simplify(sin(x)^2 + cos(x)^2) returns

```
   1
```

simplify(exp(c*log(sqrt(a+b)))) returns

```
  (a + b)^(c/2)
```

The statements

```
  S = [(x^2+5*x+6)/(x+2),sqrt(16)];
  R = simplify(S)
```

return

```
  [ x + 3, 4]
```

See Also   collect, expand, factor, horner, simple

single

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`

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

$$Si(x) = \int\limits_{0}^{x} \frac{\sin t}{t} dt$$

Examples   sinint([pi 0;-2.2 exp(3)]) returns

```
    1.8519        0
   -1.6876    1.5522
```

sinint(1.2) returns 1.1080.

syms x; diff(sinint(x)) returns sin(x)/x.

See Also   cosint

size

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

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

                 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)

```
   -(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

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 =
   3
   2
   2
   3

  A.u =
   (3^(1/2)*i)/4 + 1/4
   (2^(1/2)*i)/3 + 1/3
   1/3 - (2^(1/2)*i)/3
   1/4 - (3^(1/2)*i)/4

  A.v =
     (3^(1/2)*i)/4 - 3/4
     (2^(1/2)*i)/3 - 2/3
   - (2^(1/2)*i)/3 - 2/3
   - (3^(1/2)*i)/4 - 3/4
```

See Also   Arithmetic Operations, dsolve, findsym

sort

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.

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:

```
r =

                                 s^(1/3) - b/(3*a) - (c/(3*a) - b^2/(9*a^2))/s^(1/3)
 (c/(3*a) - b^2/(9*a^2))/(2*s^(1/3)) - s^(1/3)/2 - b/(3*a) - (3^(1/2)*i*(s^(1/3) + ...
      (c/(3*a) - b^2/(9*a^2))/s^(1/3)))/2
 (c/(3*a) - b^2/(9*a^2))/(2*s^(1/3)) - s^(1/3)/2 - b/(3*a) + (3^(1/2)*i*(s^(1/3) + ...
      (c/(3*a) - b^2/(9*a^2))/s^(1/3)))/2

s =
(c^3/(27*a^3) + d^2/(4*a^2) + (b^3*d)/(27*a^4) - (b^2*c^2)/(108*a^4) - ...
      (b*c*d)/(6*a^3))^(1/2) - d/(2*a) - b^3/(27*a^3) + (b*c)/(6*a^2)
```

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. That is, `R = subs(S,old,new)` evaluates `S` at `old = new`. The substitution is first attempted as a MATLAB expression resulting in the computation being done in double precision arithmetic if all the values in `new` are double precision. Convert the `new` values to `sym` to ensure symbolic or variable precision arithmetic.

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 C2 = 3 exist in the workspace.

The statement

```
y = dsolve('Dy = -a*y')
```

produces

```
C2/exp(a*t)
```

Then the statements

```
a = 980; C2 = 3; subs(y)
```

produces

```
3/exp(980*t)
```

Single Substitution

syms a b; subs(a+b,a,4) returns b + 4.

Multiple Substitutions

subs(cos(a)+sin(b),{a,b},{sym('alpha'),2}) returns

```
sin(2) + cos(alpha)
```

Scalar Expansion Case

syms t; subs(exp(a*t),'a',-magic(2)) returns

```
[   1/exp(t), 1/exp(3*t)]
[ 1/exp(4*t), 1/exp(2*t)]
```

Multiple Scalar Expansion

`syms x y; 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

```
       ans =
         0
   2*5^(1/2)
```

```
 8*5^(1/2)
       34

ans =
         34.0
         17.9
         4.47
 2.3*10^(-7)*i

U =
[ 0.5,  0.671,  0.5, -0.224]
[ 0.5, -0.224, -0.5, -0.671]
[ 0.5,  0.224, -0.5,  0.671]
[ 0.5, -0.671,  0.5,  0.224]

S =
[ 34.0,    0,    0,              0]
[    0, 17.9,    0,              0]
[    0,    0, 4.47,              0]
[    0,    0,    0, 8.87*10^(-16)]

V =
[ 0.5,  0.5,  0.671,  0.224]
[ 0.5, -0.5, -0.224,  0.671]
[ 0.5, -0.5,  0.224, -0.671]
[ 0.5,  0.5, -0.671, -0.224]
```

See Also   digits, eig, vpa

Purpose   Symbolic numbers, variables, and objects

Syntax   S = sym(A)
                   x = sym('x')
                   x = sym('x','real')
                   k = sym('k','positive')
                   x = sym('x','clear')
                   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','clear') 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. For compatibility with previous
                   versions of the software, x = sym('x','unreal') has exactly the same
                   effect as x = sym('x','clear').

                   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 `(2^e+N*2^(e-52))` or `-(2^e+N*2^(e-52))` , where N and e are integers, N $\geq$ 0. This captures the floating-point values exactly, but may not be convenient for subsequent manipulation. For example, `sym(1/10,'f')` is `(2^-4+2702159776422298*2^-56)` 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,'e')` is `3*pi/4*(1+3143276*eps/65)`.

'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 3s, but is an accurate decimal representation of the floating-point number nearest to 4/3.

See Also   `digits`, `double`, `findsym`, `reset`, `syms`, `symvar`

`eps` in the online MATLAB Function Reference

Purpose   Choose symbolic engine

Syntax   symengine

Description symengine brings up a GUI for choosing the MATLAB symbolic engine: a MuPAD engine or a Maple engine.

Example   symengine

brings up the following GUI:
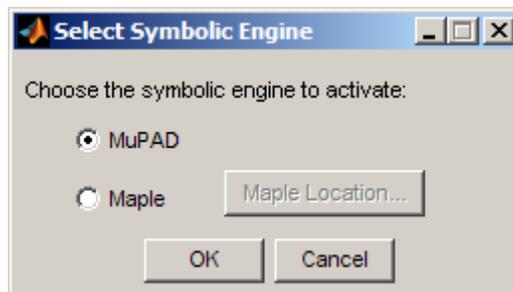
See Also   "Choosing a Maple or MuPAD Engine" on page 3-35

syms

Purpose   Shortcut for constructing symbolic objects

Syntax
```
syms arg1 arg2 ...
syms arg1 arg2 ... real
syms arg1 arg2 ... clear
syms arg1 arg2 ... positive
```

Description   syms arg1 arg2 ... is shorthand notation for

```
arg1 = sym('arg1');
arg2 = sym('arg2'); ...
```

syms arg1 arg2 ...   real is shorthand notation for

```
arg1 = sym('arg1','real');
arg2 = sym('arg2','real'); ...
```

syms arg1 arg2 ...   clear is shorthand notation for

```
arg1 = sym('arg1','clear');
arg2 = sym('arg2','clear'); ...
```

syms arg1 arg2 ...   positive is shorthand notation for

```
arg1 = sym('arg1','positive');
arg2 = sym('arg2','positive'); ...
```

Each input argument must begin with a letter and can contain only
alphanumeric characters. For compatibility with previous versions
of the software, syms arg1 arg2 ...   unreal has exactly the same
effect as syms arg1 arg2 ...   clear.

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 clear
```

Note that `clear` `x` will *not* clear the symbolic object of its `'real'` status. You can achieve this using

- `syms x clear` to remove the `'real'` status from `x` without affecting any other symbolic variables.

- `reset(symengine)` clears all symbolic variables from the MuPAD workspace and reinitializes the MuPAD engine.

See Also `findsym`, `reset`, `sym`, `symvar`

"Clearing Assumptions and Resetting the Symbolic Engine" on page 3-30

sym2poly

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`, `subs`, `sym`, `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

```
k^3/3 - k^2/2 + k/6
```

symsum(k) returns

```
k^2/2 - k/2
```

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`

Purpose   Find symbolic variables in symbolic expression or matrix

Syntax   symvar(s)
                   symvar(s,n)

Description   symvar(s) returns a vector containing all the symbolic variables in
                   s. The variables are returned in lexicographic order. If there are no
                   symbolic variables in s, then symvar returns the empty vector. symvar
                   does not consider the constants pi, i, and j to be variables.

                   symvar(s,n) returns a vector containing the n symbolic variables in s
                   that are lexicographically closest to 'x'.

Examples
```
syms x y z a b
w = x^2/(sin(3*y - b));
symvar(w)
```

returns

```
[ b, x, y]
```

Furthermore,

```
symvar(w,2)
```

returns

```
[ x, y]
```

See Also   findsym, sym, 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 `findsym` to determine the function's independent variable. `n` defaults to `6`.

The Taylor series for an analytic function $f(x)$ about the base point $x=a$ 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 Operation
$\displaystyle\sum_{n=0}^{5} x^n \cdot \frac{f^{(n)}(0)}{n!}$	`syms x` `taylor(f)`

Mathematical Operation	MATLAB Operation
$\displaystyle\sum_{n=0}^{m} x^n \cdot \frac{f^{(n)}(0)}{n!}$ m is a positive integer	`taylor(f,m)` m is a positive integer
$\displaystyle\sum_{n=0}^{5} (x-a)^n \cdot \frac{f^{(n)}(a)}{n!}$ a is a real number	`taylor(f,a)` a is a real number
$\displaystyle\sum_{n=0}^{m_1} (x-m_2)^n \cdot \frac{f^{(n)}(m_2)}{n!}$ m_1, m_2 are positive integers	`taylor(f,m1,m2)` m_1, m_2 are positive integers
$\displaystyle\sum_{n=0}^{m} (x-a)^n \cdot \frac{f^{(n)}(a)}{n!}$ 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 Operation
$\displaystyle\sum_{n=0}^{5} \frac{y^n}{n!} \cdot \left.\frac{\partial^n}{\partial y^n} f(x,y)\right|_{y=0}$	`taylor(f,y)`

taylor

Mathematical Operation	MATLAB Operation
$$\sum_{n=0}^{m} \frac{y^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x,y)\bigg\vert_{y=0}$$ m is a positive integer	`taylor(f,y,m)` or `taylor(f,m,y)` m is a positive integer
$$\sum_{n=0}^{5} \frac{(y-a)^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x,y)\bigg\vert_{y=a}$$ a is real	`taylor(f,y,a)` a is real
$$\sum_{n=0}^{m} \frac{(y-a)^n}{n!} \cdot \frac{\partial^n}{\partial y^n} f(x,y)\bigg\vert_{y=a}$$ a is real and m is a positive integer	`taylor(f,m,y,a)` a is real and m is a positive integer

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 base point x = a. The default
                   function, value of N, base point, 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

tril

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 = 0.5

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 =
[ 1.0,                          0.5]
[ 0.5, 0.3333333333333333333333333]
```

```
B =
[ 1.0,     0.5]
[ 0.5, 0.33333]
```

See Also   digits, double

Purpose   Riemann Zeta

Syntax   Y = zeta(X)
                 Y = zeta(X, n)

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(X, n) 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 6

    5.5916    3.9319    3.1055    2.6124    2.2858    2.0543

  Columns 7 through 10

    1.8822    1.7497    1.6449    1.5602
```

syms x y; zeta([x 2;4 x+y]) returns

```
  [ zeta(x),      pi^2/6]
  [ pi^4/90, zeta(x + y)]
```

diff(zeta(x),x,3) returns zeta(x, 3).

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

$f(n) = n^4$
 	
 $Z[f] = \sum_{n=0}^{\infty} f(n)z^{-n}$
 	
 $= \dfrac{z(z^3 + 11z^2 + 11z + 1)}{(z-1)^5}$
	`syms n`
	
	 `f = n^4`
	
	 `ztrans(f)`
	
	 `returns`
	
	 `(z^4 + 11*z^3 + 11*z^2 +`
	 `z)/(z - 1)^5`

$g(z) = a^z$
 	
 $Z[g] = \sum_{z=0}^{\infty} g(z)w^{-z}$
 	
 $= \dfrac{w}{w-a}$
	`g = a^z`
	
	 `ztrans(g)`
	
	 `returns`
	
	 `-w/(a - w)`

$f(n) = \sin(an)$
 	
 $Z[f] = \sum_{n=0}^{\infty} f(n)w^{-n}$
 	
 $= \dfrac{w\sin a}{1 - 2w\cos a + w^2}$
	`f = sin(a*n)`
	
	 `ztrans(f,w)`
	
	 `returns`
	
	 `(w*sin(a))/(w^2 - 2*cos(a)*w`
	 `+ 1)`

See Also   `fourier`, `iztrans`, `laplace`

Index