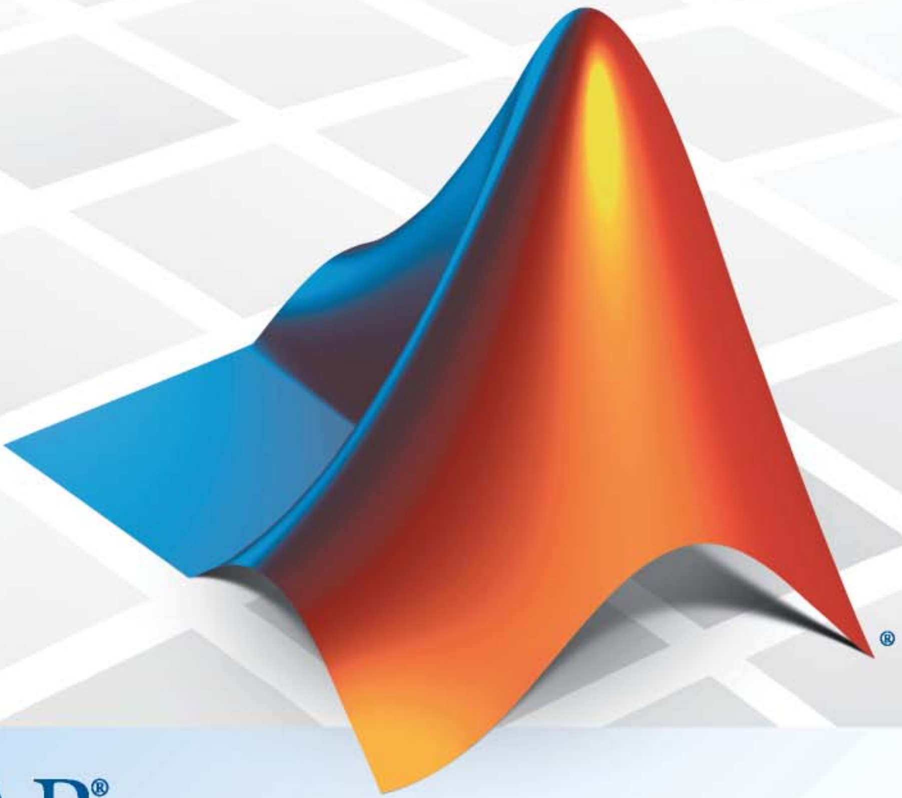
Symbolic Math Toolbox™ 5

User's Guide



MATLAB®

MathWorks®

Accelerating the pace of engineering and science

How to Contact 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

Trademarks

Patents

Revision History

August 1993	First printing	
October 1994	Second printing	
May 1997	Third printing	Revised for Version 2
May 2000	Fourth printing	Minor changes
June 2001	Fifth printing	Minor changes
July 2002	Online only	Revised for Version 2.1.3 (Release 13)
October 2002	Online only	Revised for Version 3.0.1
December 2002	Sixth printing	
June 2004	Seventh printing	Revised for Version 3.1 (Release 14)
October 2004	Online only	Revised for Version 3.1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 3.1.2 (Release 14SP2)
September 2005	Online only	Revised for Version 3.1.3 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1.4 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.5 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.2.2 (Release 2007b)
March 2008	Online only	Revised for Version 3.2.3 (Release 2008a)
October 2008	Online only	Revised for Version 5.0 (Release 2008a+)
October 2008	Online only	Revised for Version 5.1 (Release 2008b)
November 2008	Online only	Revised for Version 4.9 (Release 2007b+)
March 2009	Online only	Revised for Version 5.2 (Release 2009a)
September 2009	Online only	Revised for Version 5.3 (Release 2009b)
March 2010	Online only	Revised for Version 5.4 (Release 2010a)
September 2010	Online only	Revised for Version 5.5 (Release 2010b)

Contents

Introduction

Product Overview

Symbolic Math Toolbox™ software lets you to perform symbolic computations within the MATLAB® numeric environment. It provides tools for solving and manipulating symbolic math expressions and performing variable-precision arithmetic. The toolbox contains hundreds of symbolic functions that leverage the MuPAD® engine for a broad range of mathematical tasks such as:

- Differentiation
- Integration
- Linear algebraic operations
- Simplification
- Transforms
- Variable-precision arithmetic
- Equation solving

Symbolic Math Toolbox software also includes the MuPAD language, which is optimized for handling and operating on symbolic math expressions. In addition to covering common mathematical tasks, the libraries of MuPAD functions cover specialized areas such as number theory and combinatorics. You can extend the built-in functionality by writing custom symbolic functions and libraries in the MuPAD language.

Accessing Symbolic Math Toolbox Functionality

Key Features

Symbolic Math Toolbox software provides a complete set of tools for symbolic computing that augments the numeric capabilities of MATLAB. The toolbox includes extensive symbolic functionality that you can access directly from the MATLAB command line or from the MuPAD Notebook Interface. You can extend the functionality available in the toolbox by writing custom symbolic functions or libraries in the MuPAD language.

Working from MATLAB

You can access the Symbolic Math Toolbox functionality directly from the MATLAB Command Window. This environment lets you call functions using familiar MATLAB syntax.

The MATLAB Help browser presents the documentation that covers working from the MATLAB Command Window. To access the MATLAB Help browser, you can:

- Select **Help > Product Help** , and then select **Symbolic Math Toolbox** in the left pane
- Enter `doc` at the MATLAB command line

If you are a new user, begin with Chapter 2, "Getting Started"

Working from MuPAD

Also you can access the Symbolic Math Toolbox functionality from the MuPAD Notebook Interface using the MuPAD language. The MuPAD Notebook Interface includes a symbol palette for accessing common MuPAD functions. All results are displayed in typeset math. You also can convert the results into MathML and TeX. You can embed graphics, animations, and descriptive text within your notebook.

An editor, debugger, and other programming utilities provide tools for authoring custom symbolic functions and libraries in the MuPAD language. The MuPAD language supports multiple programming styles including imperative, functional, and object-oriented programming. The language treats variables as symbolic by default and is optimized for handling and operating on symbolic math expressions. You can call functions written in the MuPAD language from the MATLAB Command Window. For more information see "Calling MuPAD Functions at the MATLAB Command Line" on page 4-28

The MuPAD Help browser presents documentation covering the MuPAD Notebook Interface. To access the MuPAD Help browser :

- From the MuPAD Notebook Interface, select **Help > Open Help**
- From the MATLAB Command Window, enter `doc(symengine)`.

If you are a new user of the MuPAD Notebook Interface, read the Getting Started chapter of the MuPAD documentation.

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

Getting Started

Symbolic Objects

In this section...

Overview

Symbolic objects are a special MATLAB data type introduced by the Symbolic Math Toolbox software. They allow you to perform mathematical operations in the MATLAB workspace analytically, without calculating numeric values. You can use symbolic objects to perform a wide variety of analytical computations:

- Differentiation, including partial differentiation
- Definite and indefinite integration
- Taking limits, including one-sided limits
- Summation, including Taylor series
- Matrix operations
- Solving algebraic and differential equations
- Variable-precision arithmetic
- Integral transforms

Symbolic objects present symbolic variables, symbolic numbers, symbolic expressions and symbolic matrices.

Symbolic Variables

To declare variables *x* and *y* as symbolic objects use the `syms` command:

```
syms x y
```

You can manipulate the symbolic objects according to the usual rules of mathematics. For example:

```
x + x + y

ans =
 2*x + y
```

You also can create formal symbolic mathematical expressions and symbolic matrices. See "Creating Symbolic Variables and Expressions" on page 2-6 for more information.

Symbolic Numbers

Symbolic Math Toolbox software also enables you to convert numbers to symbolic objects. To create a symbolic number, use the `sym` command:

```
a = sym('2')
```

If you create a symbolic number with 10 or fewer decimal digits, you can skip the quotes:

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

The following example illustrates the difference between a standard double-precision MATLAB data and the corresponding symbolic number. The MATLAB command

```
sqrt(2)
```

returns a double-precision floating-point number:

```
ans =
    1.4142
```

On the other hand, if you calculate a square root of a symbolic number 2:

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

you get the precise symbolic result:

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

Symbolic results are not indented. Standard MATLAB double-precision results are indented. The difference in output form shows what type of data is presented as a result.

To evaluate a symbolic number numerically, use the `double` command:

```
double(a)

ans =
    1.4142
```

You also can create a rational fraction involving symbolic numbers:

```
sym(2)/sym(5)

ans =
2/5
```

or more efficiently:

```
sym(2/5)

ans =
2/5
```

MATLAB performs arithmetic on symbolic fractions differently than it does on standard numeric fractions. By default,MATLAB stores all numeric values as double-precision floating-point data. 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 in the usual procedure for adding rational numbers:

```
sym(2/5) + sym(1/3)

ans =
 11/15
```

To learn more about symbolic representation of rational and decimal fractions, see "Estimating the Precision of Numeric to Symbolic Conversions" on page 2-18.

Creating Symbolic Variables and Expressions

In this section...

Creating Symbolic Variables

The sym command creates symbolic variables and expressions. For example, the commands

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

create a symbolic variable x with the value x assigned to it in the MATLAB workspace and a symbolic variable a with the value alpha assigned to it. An alternate way to create a symbolic object is to use the syms command:

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

You can use sym or syms to create symbolic variables. The syms command:

- Does not use parentheses and quotation marks: `syms x`
- Can create multiple objects with one call
- Serves best for creating individual single and multiple symbolic variables

The sym command:

- Requires parentheses and quotation marks: `x = sym('x')`. When creating a symbolic number with 10 or fewer decimal digits, you can skip the quotation marks: `f = sym(5)`.
- Creates one symbolic object with each call.
- Serves best for creating symbolic numbers and symbolic expressions.
- Serves best for creating symbolic objects in functions and scripts.

Note In Symbolic Math Toolbox, `pi` is a reserved word.

Creating Symbolic Expressions

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
```

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 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 = a*x^2 + b*x + c;
```

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

Creating Symbolic Objects with Identical Names

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 later you enter

```
syms f;
f
```

then MATLAB removes the value `a + b` from the expression `f`:

```
f =
f
```

You can use the syms command to clear variables of definitions that you previously assigned to them in your MATLAB session. However, syms does not clear the following assumptions of the variables: complex, real, and positive. These assumptions are stored separately from the symbolic object. See "Deleting Symbolic Objects and Their Assumptions" on page 2-32 for more information.

Creating a Matrix of Symbolic Variables

Using Existing Symbolic Objects as Elements

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

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

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

Since matrix A is circulant, the sum of elements over each row and each column is the same. Find the sum of all the elements of the first row:

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

ans =
a + b + c
```

Check if the sum of the elements of the first row equals the sum of the elements of the second column:

```
sum(A(1,:)) == sum(A(:,2))
```

The sums are equal:

```
ans =
    1
```

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

Generating Elements While Creating a Matrix

The `sym` function also lets you define a symbolic matrix or vector without having to define its elements in advance. In this case, the `sym` function generates the elements of a symbolic matrix at the same time when it creates a matrix. The function presents all generated elements using the same form: the base (which must be a valid variable name), a row index, and a column index. Use the first argument of `sym` to specify the base for the names of generated elements. You can use any valid variable name as a base. To check whether the name is a valid variable name, use the `isvarname` function. By default, `sym` separates a row index and a column index by underscore. For example, create the 2-by-4 matrix `A` with the elements `A1_1, ..., A2_4`:

```
A = sym('A', [2 4])

A =
[ A1_1, A1_2, A1_3, A1_4]
[ A2_1, A2_2, A2_3, A2_4]
```

To control the format of the generated names of matrix elements, use `%d` in the first argument:

```
A = sym('A%d%d', [2 4])

A =
[ A11, A12, A13, A14]
[ A21, A22, A23, A24]
```

Creating a Matrix of Symbolic Numbers

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 precise symbolic 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]
```

For more information on numeric to symbolic conversions see "Estimating the Precision of Numeric to Symbolic Conversions" on page 2-18.

Finding Symbolic Variables in Expressions and Matrices

To determine what symbolic variables are present in an expression, use the `symvar` 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:

```
symvar(f)

ans =
[ n, x]
```

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

```
symvar(g)

ans =
[ a, b, t]
```

Performing Symbolic Computations

In this section...

Simplifying Symbolic Expressions

Symbolic Math Toolbox provides a set of simplification functions allowing you to manipulate an output of a symbolic expression. For example, the following polynomial of the golden ratio `rho`

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

and get a very short answer:

```
ans =
0
```

Symbolic simplification is not always so straightforward. There is no universal simplification function, because the meaning of a simplest representation of a symbolic expression cannot be defined clearly. Different problems require different forms of the same mathematical expression. Knowing what form is more effective for solving your particular problem, you can choose the appropriate simplification function.

For example, to show the order of a polynomial or symbolically differentiate or integrate a polynomial, use the standard polynomial form with all the parenthesis multiplied out and all the similar terms summed up. To rewrite a polynomial in the standard form, use the `expand` function:

```
syms x;
f = (x ^2- 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - x + 1);
expand(f)

ans =
x^10 - 1
```

The `factor` simplification function shows the polynomial roots. If a polynomial cannot be factored over the rational numbers, the output of the `factor` function is the standard polynomial form. For example, to factor the third-order polynomial, enter:

```
syms x;
g = x^3 + 6*x^2 + 11*x + 6;
factor(g)

ans =
(x + 3)*(x + 2)*(x + 1)
```

The nested (Horner) representation of a polynomial is the most efficient for numerical evaluations:

```
syms x;
h = x^5 + x^4 + x^3 + x^2 + x;
horner(h)

ans =
x*(x*(x*(x*(x + 1) + 1) + 1) + 1)
```

For a list of Symbolic Math Toolbox simplification functions, see "Simplifications" on page 3-42.

Substituting in Symbolic Expressions

Substituting Symbolic Variables with Numbers

You can substitute a symbolic variable with a numeric value by using the `subs` function. For example, evaluate the symbolic expression `f` at the point `x` = 2/3:

```
syms x;
f = 2*x^2 - 3*x + 1;
subs(f, 1/3)

ans =
    0.2222
```

The `subs` function does not change the original expression `f`:

```
f

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

Substituting in Multivariate Expressions

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)

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

Substituting One Symbolic Variable for Another

You also can substitute one symbolic variable for another symbolic variable. For example to replace the variable `y` with the variable `x`, enter

```
subs(f, y, x)

ans =
x^3 + 5*x^(3/2)
```

Substituting a Matrix into a Polynomial

You can also substitute a matrix into a symbolic polynomial with numeric coefficients. There are two ways to substitute a matrix into a polynomial: element by element and according to matrix multiplication rules.

Element-by-Element Substitution. To substitute a matrix at each element, use the `subs` command:

```
A = [1 2 3;4 5 6];
syms x; f = x^3 - 15*x^2 - 24*x + 350;
subs(f,A)

ans =
   312   250   170
    78   -20  -118
```

You can do element-by-element substitution for rectangular or square matrices.

Substitution in a Matrix Sense. If you want to substitute a matrix into a polynomial using standard matrix multiplication rules, a matrix must be square. For example, you can substitute the magic square `A` into a polynomial `f`:

1 Create the polynomial:

   ```
   syms x;
   f = x^3 - 15*x^2 - 24*x + 350;
   ```

2 Create the magic square matrix:

   ```
   A = magic(3)
   ```

```
A =
     8     1     6
     3     5     7
     4     9     2
```

3 Get a row vector containing the numeric coefficients of the polynomial `f`:

```
b = sym2poly(f)

b =
     1   -15   -24   350
```

4 Substitute the magic square matrix `A` into the polynomial `f`. Matrix `A` replaces all occurrences of `x` in the polynomial. The constant times the identity matrix `eye(3)` replaces the constant term of `f`:

```
A^3 - 15*A^2 - 24*A + 350*eye(3)

ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

The `polyvalm` command provides an easy way to obtain the same result:

```
polyvalm(sym2poly(f),A)

ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

Substituting the Elements of a Symbolic Matrix

To substitute a set of elements in a symbolic matrix, also use the `subs` command. Suppose you want to replace some of the elements of a symbolic circulant matrix A

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

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

To replace the (2, 1) element of `A` with `beta` and the variable `b` throughout the matrix with variable `alpha`, enter

```
alpha = sym('alpha');
beta = sym('beta');
A(2,1) = beta;
A = subs(A,b,alpha)
```

The result is the matrix:

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

For more information on the `subs` command see “Substitutions” on page 3-53.

Estimating the Precision of Numeric to Symbolic Conversions

The `sym` command converts a numeric scalar or matrix to symbolic form. By default, the `sym` command returns a rational approximation of a numeric expression. For example, you can convert the standard double-precision variable into a symbolic object:

```
t = 0.1;
sym(t)

ans =
1/10
```

The technique for converting floating-point numbers is specified by the optional second argument, which can be `'f'`, `'r'`, `'e'` or `'d'`. The default option is `'r'` that stands for rational approximation“Converting to Rational Symbolic Form” on page 2-19.

Converting to Floating-Point Symbolic Form

The `'f'` option to `sym` converts a double-precision floating-point number to a sum of two binary numbers. All values are represented as rational numbers `N*2^e`, where `e` and `N` are integers, and `N` is nonnegative. For example,

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

returns the symbolic floating-point representation:

```
ans =
3602879701896397/36028797018963968
```

Converting to Rational Symbolic Form

If you call `sym` command with the `'r'` option

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

you get the results in the rational form:

```
ans =
1/10
```

This is the default setting for the `sym` command. If you call this command without any option, you get the result in the same rational form:

```
sym(t)

ans =
1/10
```

Converting to Rational Symbolic Form with Machine Precision

If you call the `sym` command with the option `'e'`, it 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 precision):

```
sym(t, 'e')

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

Converting to Decimal Symbolic Form

If you call the `sym` command with the option `'d'`, it returns the decimal expansion of `t` up to the number of significant digits:

```
sym(t, 'd')

ans =
0.10000000000000000555111512312578
```

By default, the `sym(t,'d')` command returns a number with 32 significant digits. To change the number of significant digits, use the `digits` command:

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

ans =
0.1
```

Differentiating Symbolic Expressions

With the Symbolic Math Toolbox software, you can find

- Derivatives of single-variable expressions
- Partial derivatives
- Second and higher order derivatives
- Mixed derivatives

For in-depth information on taking symbolic derivatives see “Differentiation” on page 3-2.

Expressions with One Variable

To differentiate a symbolic expression, use the `diff` command. The following example illustrates how to take a first derivative of a symbolic expression:

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

ans =
2*cos(x)*sin(x)
```

Partial Derivatives

For multivariable expressions, you can specify the differentiation variable. If you do not specify any variable, MATLAB chooses a default variable by the proximity to the letter x:

```
syms x y;
f = sin(x)^2 + cos(y)^2;
diff(f)

ans =
2*cos(x)*sin(x)
```

For the complete set of rules MATLAB applies for choosing a default variable, see "Finding a Default Symbolic Variable" on page 2-26.

To differentiate the symbolic expression `f` with respect to a variable `y`, enter:

```
syms x y;
f = sin(x)^2 + cos(y)^2;
diff(f, y)

ans =
-2*cos(y)*sin(y)
```

Second Partial and Mixed Derivatives

To take a second derivative of the symbolic expression `f` with respect to a variable `y`, enter:

```
syms x y;
f = sin(x)^2 + cos(y)^2;
diff(f, y, 2)

ans =
2*sin(y)^2 - 2*cos(y)^2
```

You get the same result by taking derivative twice: `diff(diff(f, y))`. To take mixed derivatives, use two differentiation commands. For example:

```
syms x y;
f = sin(x)^2 + cos(y)^2;
diff(diff(f, y), x)
```

```
ans =
0
```

Integrating Symbolic Expressions

You can perform symbolic integration including:

- Indefinite and definite integration
- Integration of multivariable expressions

For in-depth information on the `int` command including integration with real and complex parameters, see "Integration" on page 3-12.

Indefinite Integrals of One-Variable Expressions

Suppose you want to integrate a symbolic expression. The first step is to create the symbolic expression:

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

To find the indefinite integral, enter

```
int(f)

ans =
x/2 - sin(2*x)/4
```

Indefinite Integrals of Multivariable Expressions

If the expression depends on multiple symbolic variables, you can designate a variable of integration. If you do not specify any variable, MATLAB chooses a default variable by the proximity to the letter `x`:

```
syms x y n;
f = x^n + y^n;
int(f)

ans =
x*y^n + (x*x^n)/(n + 1)
```

For the complete set of rules MATLAB applies for choosing a default variable, see “Finding a Default Symbolic Variable” on page 2-26.

You also can integrate the expression `f = x^n + y^n` with respect to `y`

```
syms x y n;
f = x^n + y^n;
int(f, y)

ans =
x^n*y + (y*y^n)/(n + 1)
```

If the integration variable is `n`, enter

```
syms x y n;
f = x^n + y^n;
int(f, n)

ans =
x^n/log(x) + y^n/log(y)
```

Definite Integrals

To find a definite integral, pass the limits of integration as the final two arguments of the `int` function:

```
syms x y n;
f = x^n + y^n;
int(f, 1, 10)

ans =
piecewise([n = -1, log(10) + 9/y],...
[n <> -1, (10*10^n - 1)/(n + 1) + 9*y^n])
```

If MATLAB Cannot Find a Closed Form of an Integral

If the `int` function cannot compute an integral, MATLAB issues a warning and returns an unresolved integral:

```
syms x y n;
f = exp(x)^(1/n) + exp(y)^(1/n);
```

```
int(f, n, 1, 10)

Warning: Explicit integral could not be found.

ans =
int(exp(x)^(1/n) + exp(y)^(1/n), n = 1..10)
```

Solving Equations

You can solve different types of symbolic equations including:

- Algebraic equations with one symbolic variable
- Algebraic equations with several symbolic variables
- Systems of algebraic equations

For in-depth information on solving symbolic equations including differential equations, see "Solving Equations" on page 3-93.

Algebraic Equations with One Symbolic Variable

You can find the values of variable `x` for which the following expression is equal to zero:

```
syms x;
solve(x^3 - 6*x^2 + 11*x - 6)

ans =
 1
 2
 3
```

By default, the `solve` command assumes that the right-side of the equation is equal to zero. If you want to solve an equation with a nonzero right part, use quotation marks around the equation:

```
syms x;
solve('x^3 - 6*x^2 + 11*x - 5 = 1')

ans =
 1
 2
 3
```

Algebraic Equations with Several Symbolic Variables

If an equation contains several symbolic variables, you can designate a variable for which this equation should be solved. For example, you can solve the multivariable equation:

```
syms x y;
f = 6*x^2 - 6*x^2*y + x*y^2 - x*y + y^3 - y^2;
```

with respect to a symbolic variable y:

```
solve(f, y)

ans =
    1
  2*x
 -3*x
```

If you do not specify any variable, you get the solution of an equation for the alphabetically closest to x variable. For the complete set of rules MATLAB applies for choosing a default variable see "Finding a Default Symbolic Variable" on page 2-26.

Systems of Algebraic Equations

You also can solve systems of equations. For example:

```
syms x y z;
[x, y, z] = solve('z = 4*x', 'x = y', 'z = x^2 + y^2')

x =
 0
 2

y =
 0
 2

z =
 0
```

```
8
```

Finding a Default Symbolic Variable

When performing substitution, differentiation, or integration, if you do not specify a variable to use, MATLAB uses a *default* variable. The default variable is basically the one closest alphabetically to `x`. To find which variable is chosen as a default variable, use the `symvar(`*`expression`*`, 1)` command. For example:

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

ans =
t

syms sx tx;
g = sx + tx;
symvar(g, 1)

ans =
tx
```

For more information on choosing the default symbolic variable, see the `symvar` command.

Creating Plots of Symbolic Functions

You can create different types of graphs including:

- Plots of explicit functions
- Plots of implicit functions
- 3-D parametric plots
- Surface plots

See “Pedagogical and Graphical Applications” on page 5-6 for in-depth coverage of Symbolic Math Toolbox graphics and visualization tools.

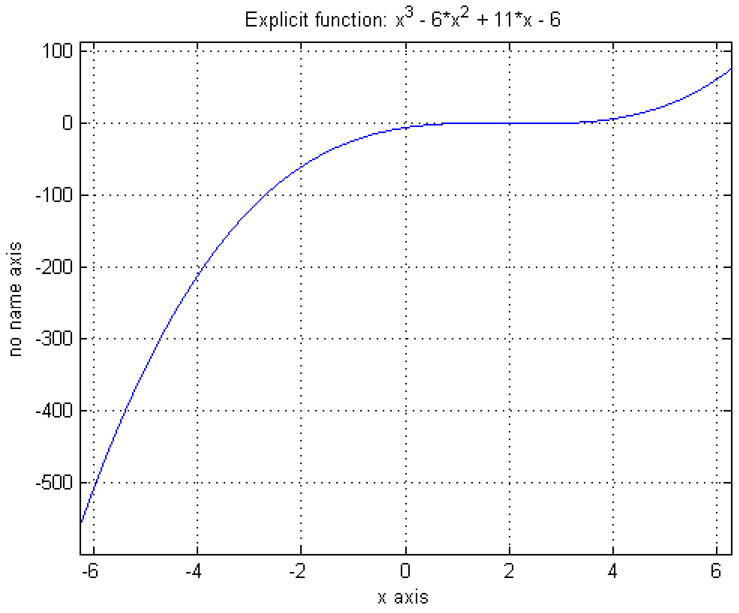
Explicit Function Plot

The simplest way to create a plot is to use the `ezplot` command:

```
syms x;
ezplot(x^3 - 6*x^2 + 11*x - 6);
hold on;
```

The `hold on` command retains the existing plot allowing you to add new elements and change the appearance of the plot. For example, now you can change the names of the axes and add a new title and grid lines. When you finish working with the current plot, enter the `hold off` command:

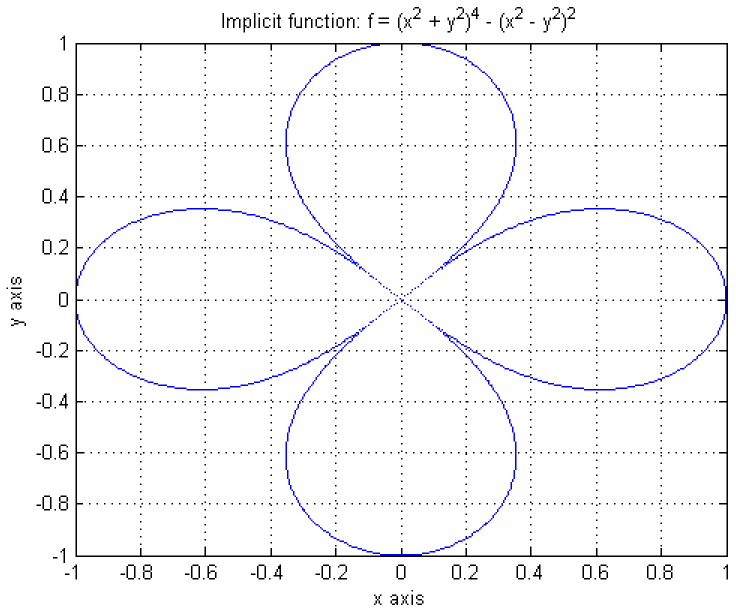
```
xlabel('x axis');
ylabel('no name axis');
title('Explicit function: x^3 - 6*x^2 + 11*x - 6');
grid on;
hold off
```



Implicit Function Plot

You can plot implicitly defined functions. For example, create a plot for the following implicit function over the domain $-1 < x < 1$:

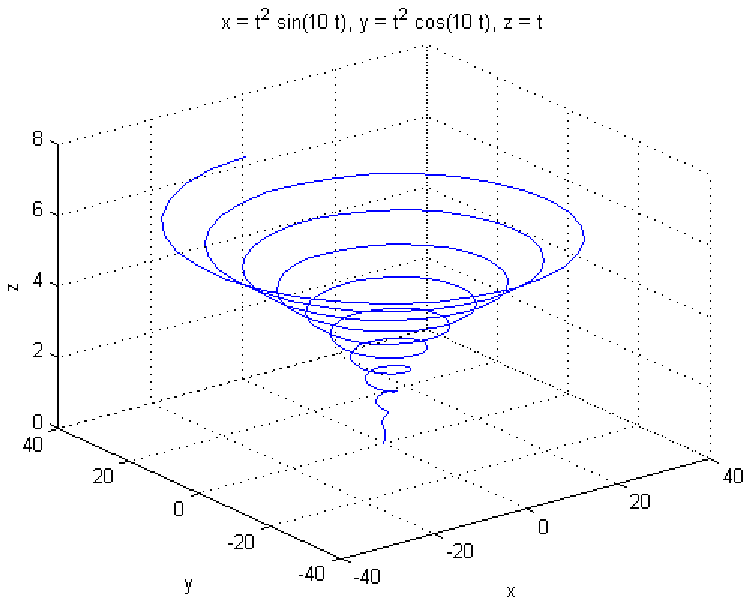
```
syms x y;
f = (x^2 + y^2)^4 - (x^2 - y^2)^2;
ezplot(f, [-1 1]);
hold on;
xlabel('x axis');
ylabel('y axis');
title('Implicit function: f = (x^2 + y^2)^4 - (x^2 - y^2)^2');
grid on;
hold off
```



3-D Plot

3-D graphics is also available in Symbolic Math Toolbox . To create a 3-D plot, use the `ezplot3` command. For example:

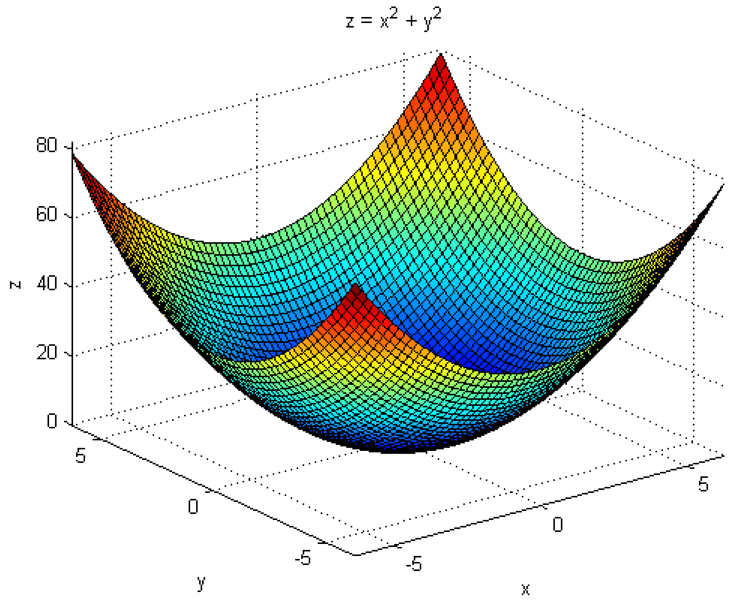
```
syms t;
ezplot3(t^2*sin(10*t), t^2*cos(10*t), t);
```



Surface Plot

If you want to create a surface plot, use the `ezsurf` command. For example, to plot a paraboloid $z = x^2 + y^2$, enter:

```
syms x y;
ezsurf(x^2 + y^2);
hold on;
zlabel('z');
title('z = x^2 + y^2');
hold off
```



Assumptions for Symbolic Objects

In this section...

Default Assumption

In Symbolic Math Toolbox, symbolic variables are single complex variables by default. For example, if you declare `z` as a symbolic variable:

```
syms z
```

MATLAB assumes `z` is a complex variable. You can always check if a symbolic variable is assumed to be complex or real by entering `conj` command. If `conj(x) == x` returns 1, `x` is a real variable:

```
z == conj(z)

ans =
     0
```

Setting Assumptions for Symbolic Variables

The `sym` and `syms` commands allow you to set up assumptions for symbolic variables. For example, create the real symbolic variables `x` and `y`, and the positive symbolic variable `z`:

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

or more efficiently

```
syms x y real;
syms z positive;
```

There are two assumptions you can assign to a symbolic object within the `sym` command: real and positive. Together with the default complex property of a symbolic variable, it gives you three choices for an assumption for a symbolic variable: complex, real, and positive.

Deleting Symbolic Objects and Their Assumptions

When you declare x to be real with the command

```
syms x real
```

you create a symbolic object x and the assumption that the object is real. Symbolic objects and their assumptions are stored separately. When you delete a symbolic object from the MATLAB workspace

```
clear x
```

the assumption that x is real stays in symbolic engine. If you declare a new symbolic variable x later, it inherits the assumption that x is real instead of getting a default assumption. If later you solve an equation and simplify an expression with the symbolic variable x, you could get incomplete results. For example, the assumption that x is real causes the polynomial x^2+1 to have no roots:

```
syms x real;
clear x;
syms x;
solve(x^2+1)

Warning: Explicit solution could not be found.
> In solve at 81

ans =
[ empty sym ]
```

The complex roots of this polynomial disappear because the symbolic variable x still has the assumption that x is real stored in the symbolic engine. To clear the assumption, enter

```
syms x clear
```

After you clear the assumption, the symbolic object stays in the MATLAB workspace. If you want to remove both the symbolic object and its assumption, use two subsequent commands:

1. To clear the assumption, enter

   ```
   syms x clear
   ```

2. To delete the symbolic object, enter

   ```
   clear x
   ```

For more information on clearing symbolic variables, see “Clearing Assumptions and Resetting the Symbolic Engine” on page 4-33.

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

In this section...

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 such simplification, see "More Examples" on page 3-5.

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, 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. Basically, the default variable is the letter closest to x in the alphabet. See the complete set of rules in "Finding a Default Symbolic Variable" on page 2-26. In the preceding example, `diff(f)` takes the derivative of `f` with respect to `t` because the letter `t` is closer to x in the alphabet than the letter `s` is. To determine the default variable that MATLAB differentiates with respect to, use the `symvar` command:

```
symvar(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)

`syms x n;`
`f = x^n;`
	`diff(f)`
	
	`ans =`
	`n*x^(n - 1)`

`syms a b t;`
`f = sin(a*t + b);`
	`diff(f)`
	
	`ans =`
	`a*cos(b + a*t)`

`syms theta;`
`f = exp(i*theta);`
	`diff(f)`
	
	`ans =`
	`exp(theta*i)*i`

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)]
```

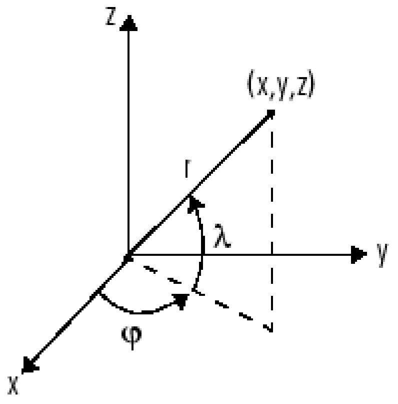
The command

```
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 3-42 discusses simplification in more detail.

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

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

Limits

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

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

provided this limit exists. Symbolic Math Toolbox 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)
```

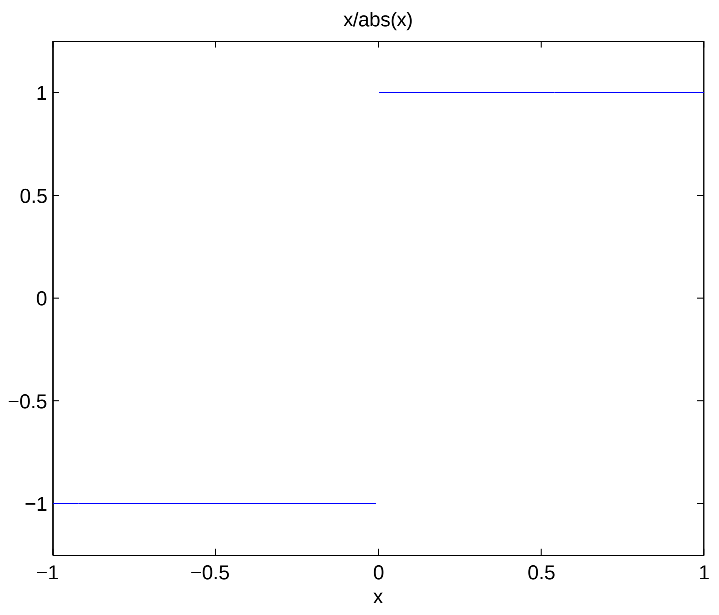
which returns

```
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.



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

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

enter

```
syms x;
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

```
syms x;
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,

```
syms x;
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**
$\lim\limits_{x \to 0} f(x)$	`limit(f)`
$\lim\limits_{x \to a} f(x)$	`limit(f, x, a)` or `limit(f, a)`
$\lim\limits_{x \to a^-} f(x)$	`limit(f, x, a, 'left')`
$\lim\limits_{x \to a^+} f(x)$	`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 `symvar`. See how `int` works by looking at this table.

Mathematical Operation	MATLAB Command
$\int x^n dx = \begin{cases} \log(x) & \text{if } n = -1 \\ \dfrac{x^{n+1}}{n+1} & \text{otherwise.} \end{cases}$	`int(x^n)` or `int(x^n,x)`
$\int_0^{\pi/2} \sin(2x)dx = 1$	`int(sin(2*x), 0, pi/2)` or `int(sin(2*x), x, 0, pi/2)`
$g = \cos(at + b)$ $\int g(t)dt = \sin(at+b)/a$	`g = cos(a*t + b)` `int(g)` or `int(g, t)`
$\int J_1(z)dz = -J_0(z)$	`int(besselj(1, z))` or `int(besselj(1, z), z)`

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

- The antiderivative, `F`, may not exist in closed form.
- The antiderivative may define an unfamiliar function.
- The antiderivative may exist, but the software can't find 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)**

`syms x n;`
`f = x^n;`
	`int(f)`
	
	`ans =`
	`piecewise([n = -1, log(x)], [n <> -1,`
	`x^(n + 1)/(n + 1)])`

`syms y;`
`f = y^(-1);`
	`int(f)`
	
	`ans =`
	`log(y)`

`syms x n;`
`f = n^x;`
	`int(f)`
	
	`ans =`
	`n^x/log(n)`

`syms a b`
`theta;`
`f =`
`sin(a*theta+b);`
	`int(f)`
	
	`ans =`
	`-cos(b + a*theta)/a`

f	int(f)

`syms u;`
`f = 1/(1+u^2);`
	`int(f)`
	
	`ans =`
	`atan(u)`

`syms x;`
`f = exp(-x^2);`
	`int(f)`
	
	`ans =`
	`(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.

Definite Integral	Command
$\int_a^b f(x)dx$	`int(f, a, b)`
$\int_a^b f(v)dv$	`int(f, v, a, b)`

Here are some additional examples.

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

`syms x;`
`f = x^7;`
	`a = 0;`
	`b = 1;`
		`int(f, a, b)`
		
		`ans =`
		`1/8`

`syms x;`
`f = 1/x;`
	`a = 1;`
	`b = 2;`
		`int(f, a, b)`
		
		`ans =`
		`log(2)`

`syms x;`
`f =`
`log(x)*sqrt(x);`
	`a = 0;`
	`b = 1;`
		`int(f, a, b)`
		
		`ans =`
		`-4/9`

`syms x;`
`f = exp(-x^2);`
	`a = 0;`
	`b = inf;`
		`int(f, a, b)`
		
		`ans =`
		`pi^(1/2)/2`

`syms z;`
`f =`
`besselj(1,z)^2;`
	`a = 0;`
	`b = 1;`
		`int(f, a, b)`
		
		`ans =`
		`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
```

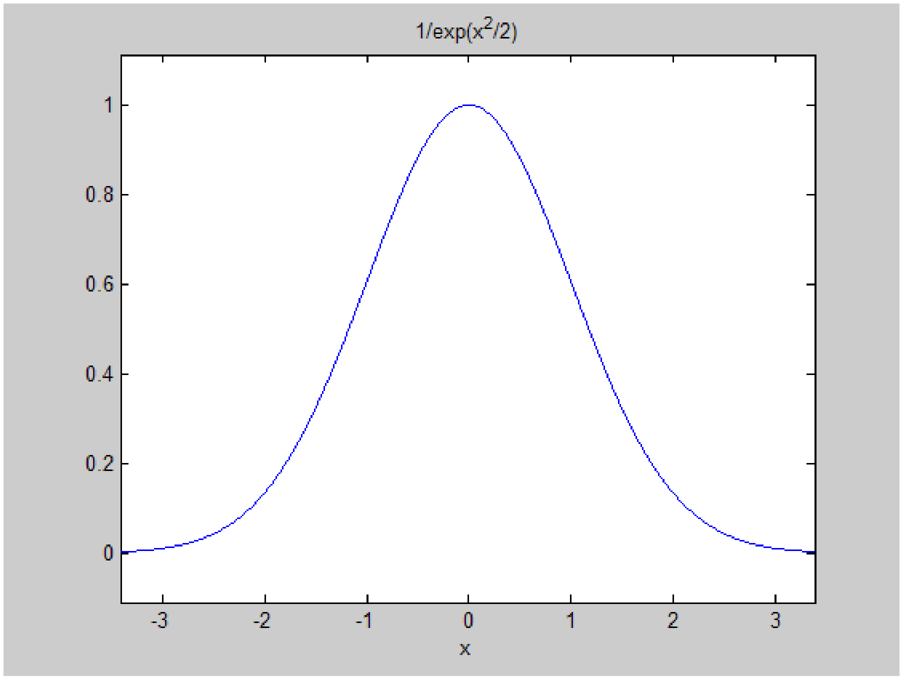
Integration with Real Parameters

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

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

for complex values of `a`, enter

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

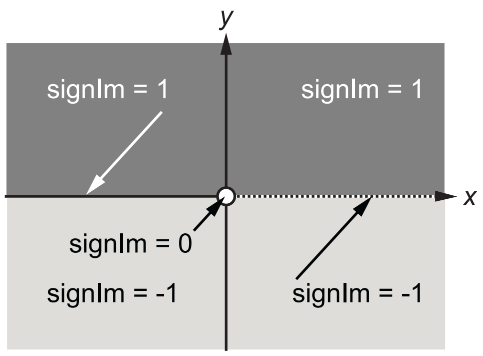
`syms` is used with the `clear` option to clear the `real` property that was assigned to `a` in the preceding example — see "Deleting Symbolic Objects and Their Assumptions" on page 2-32.

The preceding commands produce the complex output

```
F =
(pi*signIm(i/a))/a
```

The function `signIm` is defined as:

$$\operatorname{signIm}(z) = \begin{cases} 1 & \text{if } \operatorname{Im}(z) > 0, \text{ or } \operatorname{Im}(z) = 0 \text{ and } z < 0 \\ 0 & \text{if } z = 0 \\ \text{-1} & \text{otherwise.} \end{cases}$$



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
    ------ + ---- + ---- + -
    131220   1458    81    9
```

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 about 100,000 characters in its printed form. In order to proceed with using `t`, first simplify its presentation:

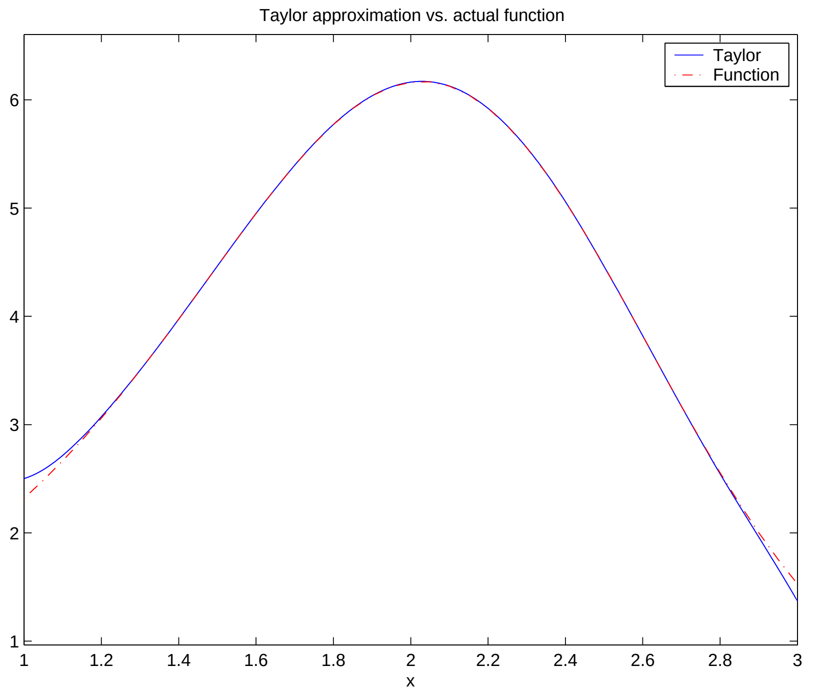
```
t = simplify(t);
size(char(t))

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')
```



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

Calculus Example

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

- "Defining the Function" on page 3-23
- "Finding the Asymptotes" on page 3-24
- "Finding the Maximum and Minimum" on page 3-26
- "Finding the Inflection Point" on page 3-28

Defining the Function

The function in this example is

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

To create the function, enter the following commands:

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

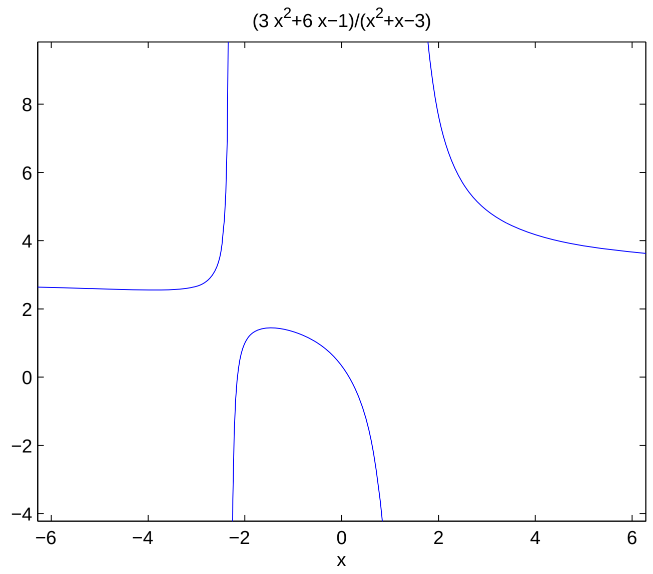
This returns

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

You can plot the graph of `f` by entering

```
ezplot(f)
```

This displays the following plot.



Finding the Asymptotes

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

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

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

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

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

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

```
roots =
  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

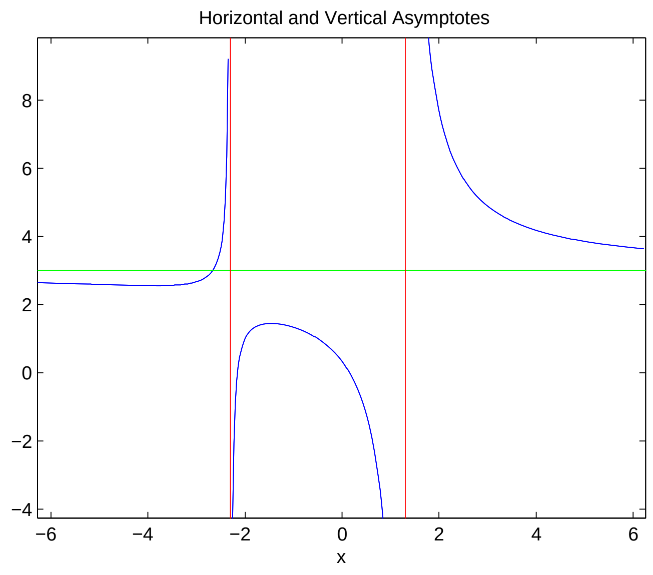
$$x = \frac{-1-\sqrt{13}}{2}.$$

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.



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

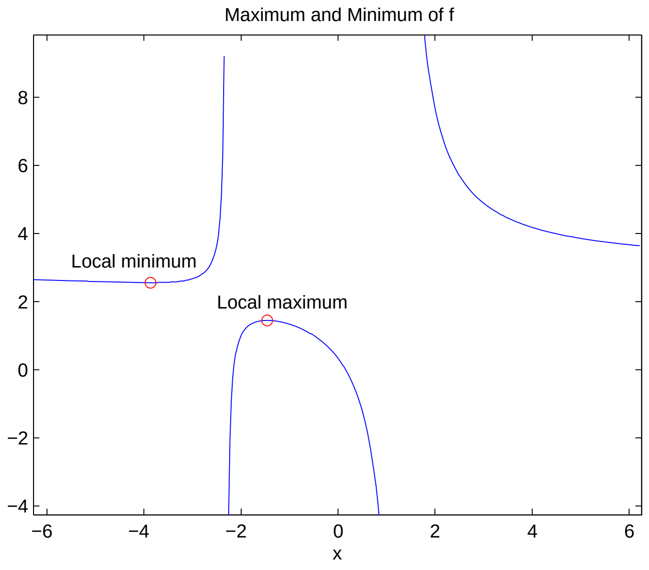
$$x_2 = \frac{-8+\sqrt{13}}{3}.$$

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

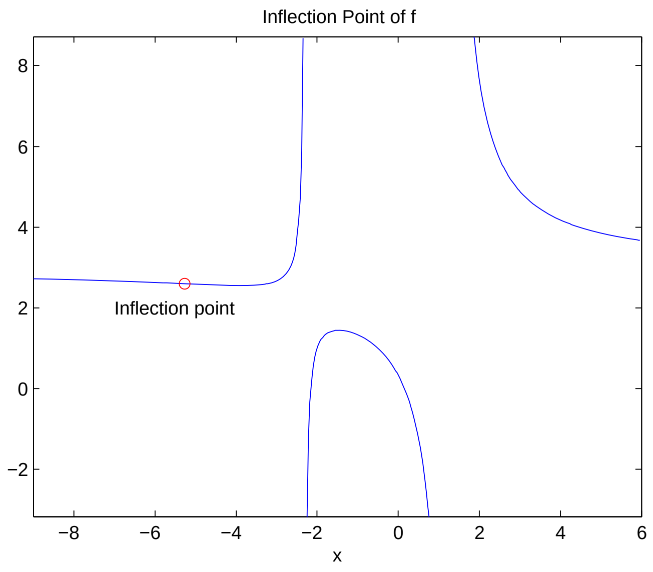
```
                           /             1/2 \1/3
         13                |         2197    |      8
  - -------------------- - | 169/54 - ------- |    - -
                      1    \            18   /      3
                      -
      /          1/2 \3
      | 169   2197    |
    9 | --- - ------- |
      \ 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 3-31
- “Finding the Zeros of f3” on page 3-32
- “Finding the Maxima and Minima of f2” on page 3-36
- “Integrating” on page 3-37

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

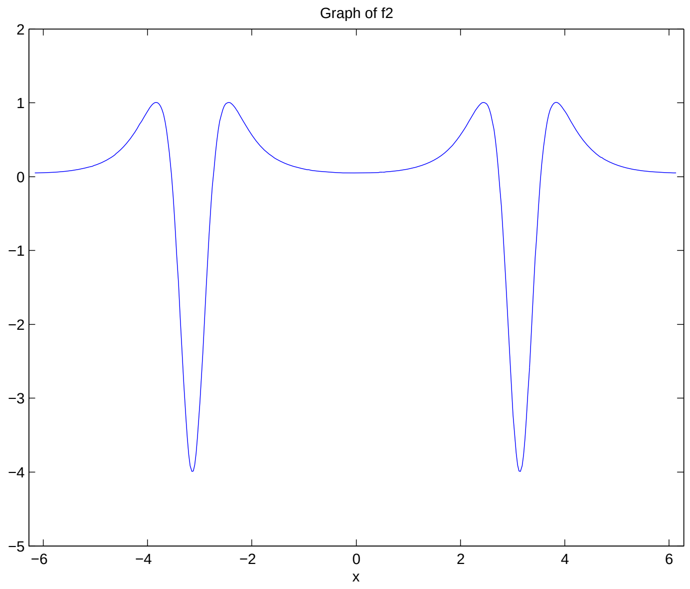
```
f2 = diff(f, 2)
```

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')
```



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

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

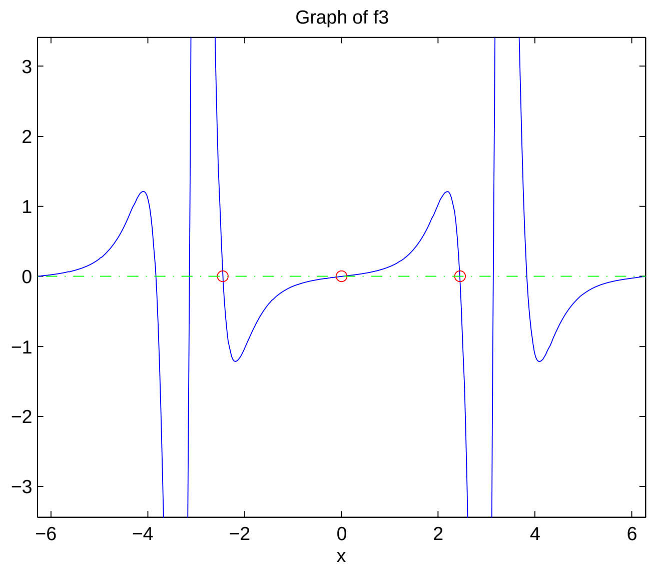
```
format;
% Default format of 5 digits
zerosd = double(zeros)
```

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')
```



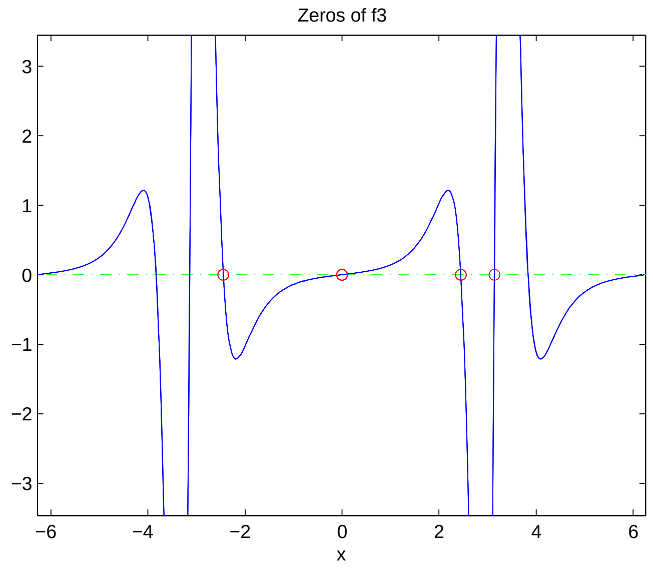
The red circles in the graph correspond to `zerosd(1)`, `zerosd(3)`, and `zerosd(4)`. As you can see in the graph, there are also zeros at $\pm\pi$. The additional zeros occur because $f'''(x)$ contains a factor of $sin(x)$, which is zero at integer multiples of π. The function, `solve(sin(x))`, however,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



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
```

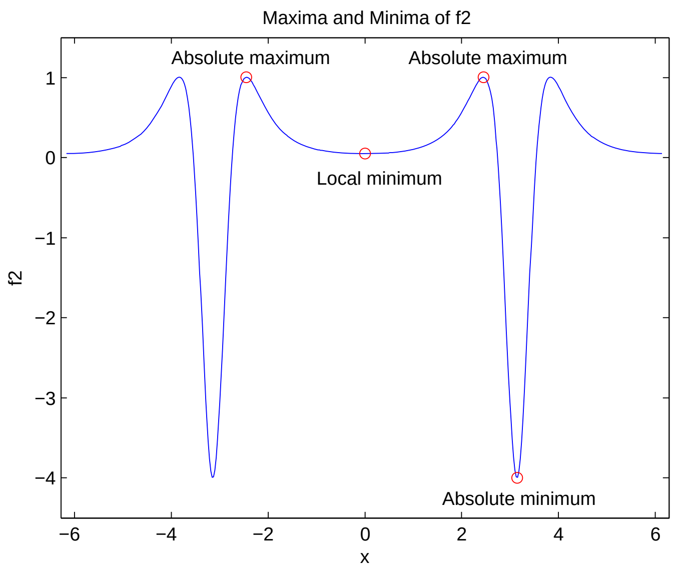
This shows the following:

- $f''(x)$ 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.



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

Integrating

Integrate $f(x)$:

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

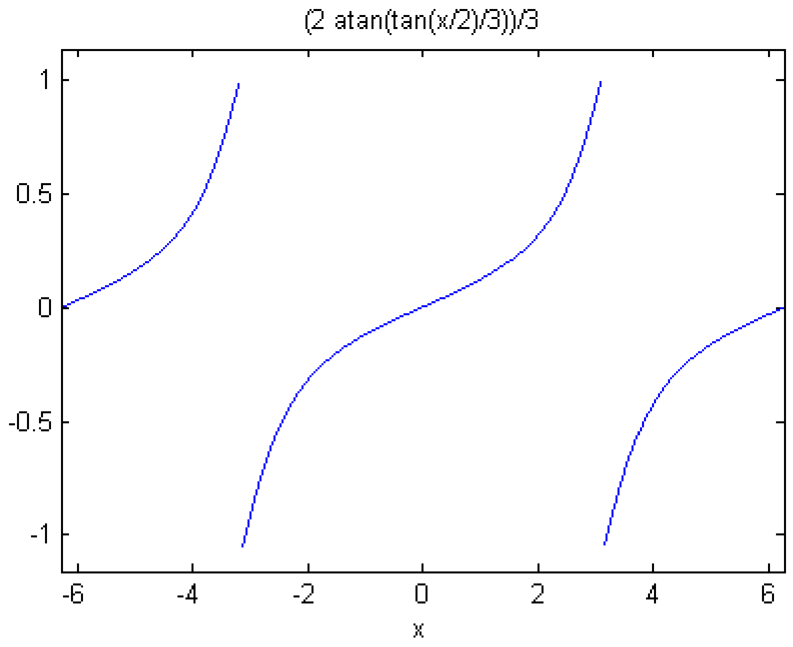
The result

```
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)
```



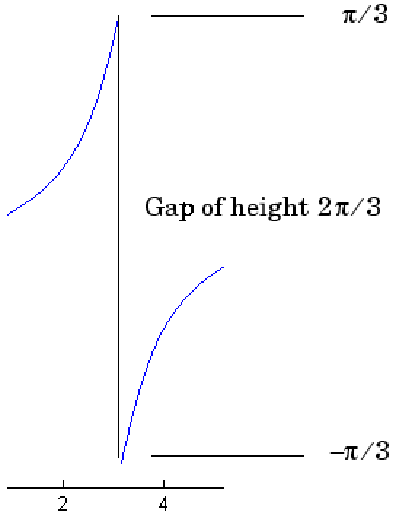
To change $F(x)$ into a true antiderivative of $f(x)$ that is differentiable everywhere, you can add a step function to $F(x)$. The height of the steps is the height of the gaps in the graph of $F(x)$. You can determine the height of the gaps by taking the limits of $F(x)$ as x approaches π from the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

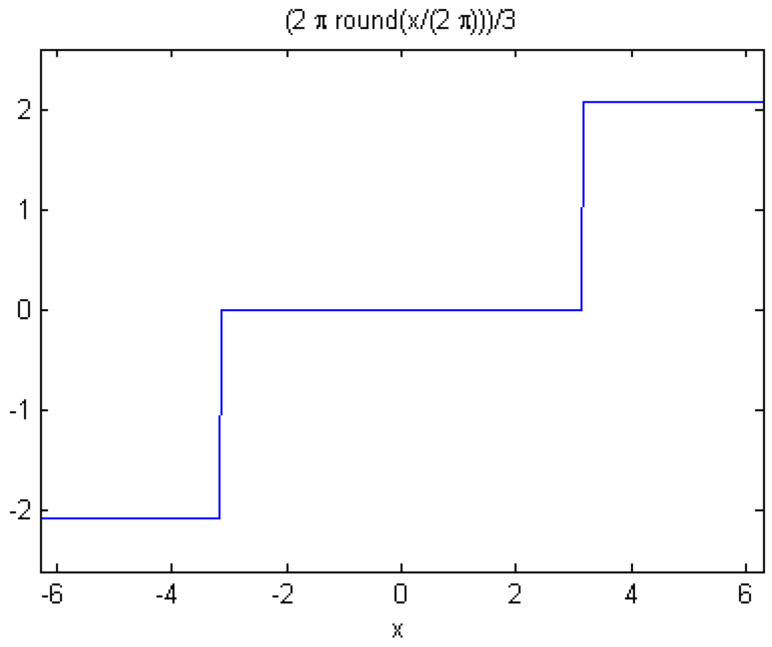


You can create the step function using the `round` function, which rounds numbers to the nearest integer, as follows:

```
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

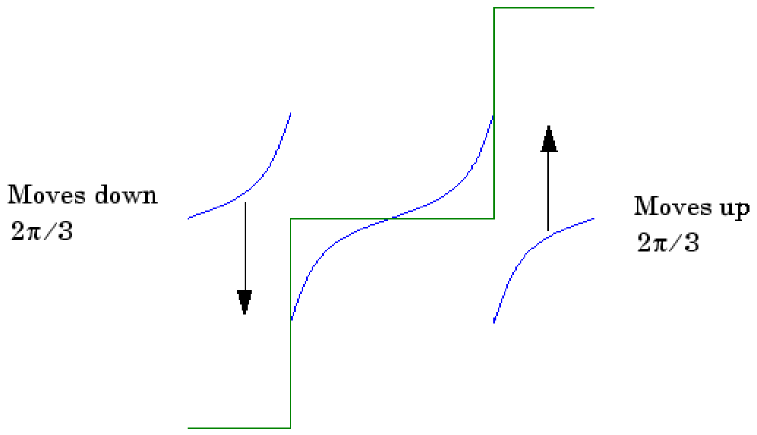
```
ezplot(J, [-2*pi 2*pi])
```



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
```

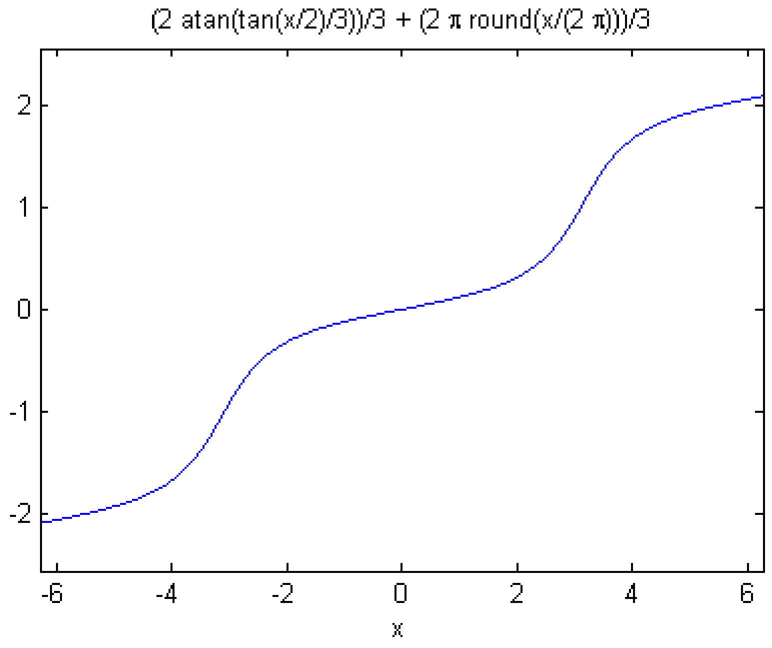
Adding the step function raises the section of the graph of *F*(*x*) on the interval [π, 3π) up by 2π/3, lowers the section on the interval (–3π, –π] down by 2π/3, and so on, as shown in the following figure.



When you plot the result by entering

```
ezplot(F1)
```

you see that this representation does have a continuous graph.



Simplifications and Substitutions

In this section...

Simplifications

Here are three different symbolic expressions.

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

Here are their prettyprinted forms, generated by

```
pretty(f);
pretty(g);
pretty(h)
```

```
   3      2
  x  - 6 x  + 11 x - 6

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

  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)**

`syms x;`
`f =`
`(x-1)*(x-2)*(x-3);`
	`collect(f)`
	
	`ans =`
	`x^3 - 6*x^2 + 11*x - 6`

`syms x;`
`f = x*(x*(x - 6) +`
`11) - 6;`
	`collect(f)`
	
	`ans =`
	`x^3 - 6*x^2 + 11*x - 6`

`syms x t;`
`f = (1+x)*t + x*t;`
	`collect(f)`
	
	`ans =`
	`(2*t)*x + t`

expand

The statement `expand(f)` distributes products over sums and applies other identities involving functions of sums as shown in the examples below.

f	expand(f)
`syms a x y;` `f = a*(x + y);`	`expand(f)` `ans =` `a*x + a*y`
`syms x;` `f = (x - 1)*(x - 2)*(x - 3);`	`expand(f)` `ans =` `x^3 - 6*x^2 + 11*x - 6`
`syms x;` `f = x*(x*(x - 6) + 11) - 6;`	`expand(f)` `ans =` `x^3 - 6*x^2 + 11*x - 6`
`syms a b;` `f = exp(a + b);`	`expand(f)` `ans =` `exp(a)*exp(b)`
`syms x y;` `f = cos(x + y);`	`expand(f)` `ans =` `cos(x)*cos(y) - sin(x)*sin(y)`

f	expand(f)

`syms x;`
`f =`
`cos(3*acos(x));`
	`expand(f)`
	
	`ans =`
	`3*x*(x^2 - 1) + x^3`

`syms x;`
`f = 3*x*(x^2 -`
`1) + x^3;`
	`expand(f)`
	
	`ans =`
	`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)

`syms x;`
`f = x^3 - 6*x^2`
`+ 11*x - 6;`
	`horner(f)`
	
	`ans =`
	`x*(x*(x - 6) + 11) - 6`

`syms x;`
`f = 1.1 + 2.2*x`
`+ 3.3*x^2;`
	`horner(f)`
	
	`ans =`
	`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)

`syms x;`
`f = x^3 - 6*x^2`
`+ 11*x - 6;`
	`factor(f)`
	
	`ans =`
	`(x - 3)*(x - 1)*(x - 2)`

`syms x;`
`f = x^3 - 6*x^2`
`+ 11*x - 5;`
	`factor(f)`
	
	`ans =`
	`x^3 - 6*x^2 + 11*x - 5`

`syms x;`
`f = x^6 + 1;`
	`factor(f)`
	
	`ans =`
	`(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.

```
ans =
[    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` folder. 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:

```
ans =
[           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)

`syms x;`
`f = x*(x*(x - 6) +`
`11) - 6;`
	`simplify(f)`
	
	`ans =`
	`(x - 1)*(x - 2)*(x - 3)`

`syms x;`
`f = (1 - x^2)/(1 - x);`
	`simplify(f)`
	
	`ans =`
	`x + 1`

`syms a;`
`f = (1/a^3 + 6/a^2 +`
`12/a + 8)^(1/3);`
	`simplify(f)`
	
	`ans =`
	`((2*a + 1)^3/a^3)^(1/3)`

`syms x y;`
`f = exp(x) * exp(y);`
	`simplify(f)`
	
	`ans =`
	`exp(x + y)`

`syms x;`
`f = besselj(2, x) +`
`besselj(0, x);`
	`simplify(f)`
	
	`ans =`
	`(2*besselj(1, x))/x`

`syms x;`
`f = gamma(x + 1) -`
`x*gamma(x);`
	`simplify(f)`
	
	`ans =`
	`0`

`syms x;`
`f = cos(x)^2 + sin(x)^2;`
	`simplify(f)`
	
	`ans =`
	`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 100 steps. For example,

```
syms x;
z = 3 - 1/sin(x)^2 - cot(x)^2

z =
3 - 1/sin(x)^2 - cot(x)^2

simplify(z)

ans =
4 - 2/sin(x)^2

simplify(z, 200)

ans =
 2 - 2*cot(x)^2
```

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

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

displays the following alternative simplifications in the MATLAB command window along with the result:

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

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

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

mwcos2sin:
1

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

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

`syms x;`
`f = cos(x) + i*sin(x);`
	`simplify(f)`
	
	`ans =`
	`cos(x) + sin(x)*i`
		`g = simple(f)`
		
		`g =`
		`exp(x*i)`

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

```
syms x;
z = exp((cos(x)^2 - sin(x)^2)/(sin(x)*cos(x)))


z =
exp((cos(x)^2 - sin(x)^2)/(cos(x)*sin(x)))


z1 = simple(z)

z1 =
exp(cot(x) - tan(x))

z2 = simple(simple(z))

z2 =
exp(2/tan(2*x))
```

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

f	simple(f)

`syms x;`
`f = cos(x)^2 +`
`sin(x)^2;`
	`f = simple(f)`
	
	`f =`
	`1`

`syms x;`
`f = 2*cos(x)^2 -`
`sin(x)^2;`
	`f = simple(f)`
	
	`f =`
	`2 - 3*sin(x)^2`

`syms x;`
`f = cos(x)^2 -`
`sin(x)^2;`
	`f = simple(f)`
	
	`f =`
	`cos(2*x)`

`syms x;`
`f = cos(x) +`
`i*sin(x);`
	`f = simple(f)`
	
	`f =`
	`exp(x*i)`

`syms x;`
`f = cos(3*acos(x));`
	`f = simple(f)`
	
	`f =`
	`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 the variable `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)*(a/(3*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3))...
+ ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3))*i)/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)*(a/(3*((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3))...
+ ((a^3/27 + 1/4)^(1/2) - 1/2)^(1/3))*i)/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)*(a/(3*sigma^(1/3)) + sigma^(1/3))*i)/2
 a/(6*sigma^(1/3)) - sigma^(1/3)/2 + (3^(1/2)*(a/(3*sigma^(1/3)) + sigma^(1/3))*i)/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 =
[   (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2)/(a - c) - (a - b)/(a - c),...
  - (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2)/(a - c) - (a - b)/(a - c),...
                                                                        1]
[ - (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2)/(a - c) - (b - c)/(a - c),...
    (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2)/(a - c) - (b - c)/(a - c),...
                                                                        1]
[ 1,                                   1,                               1]

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

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, S,         0]
[  0, 0, a + b + c]
```

Next, substitute the symbol S into v with

```
v = simplify(subs(v, S, 'S'))

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

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

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

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

ans =
     8
```

To look at the contents of the workspace, type:

```
whos
```

which gives

```
  Name      Size             Bytes  Class     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);
```

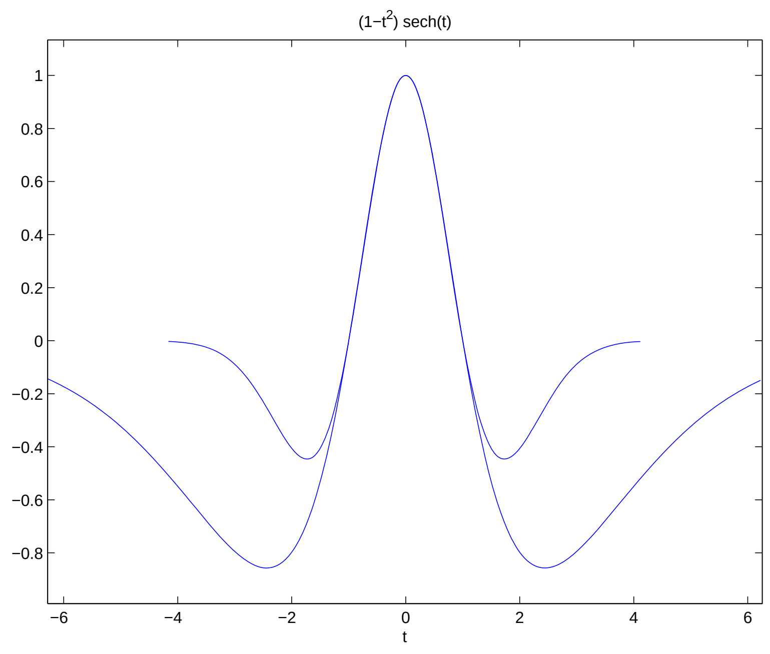
and want to see how `M` and `P` differ graphically.

One approach is to type

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

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

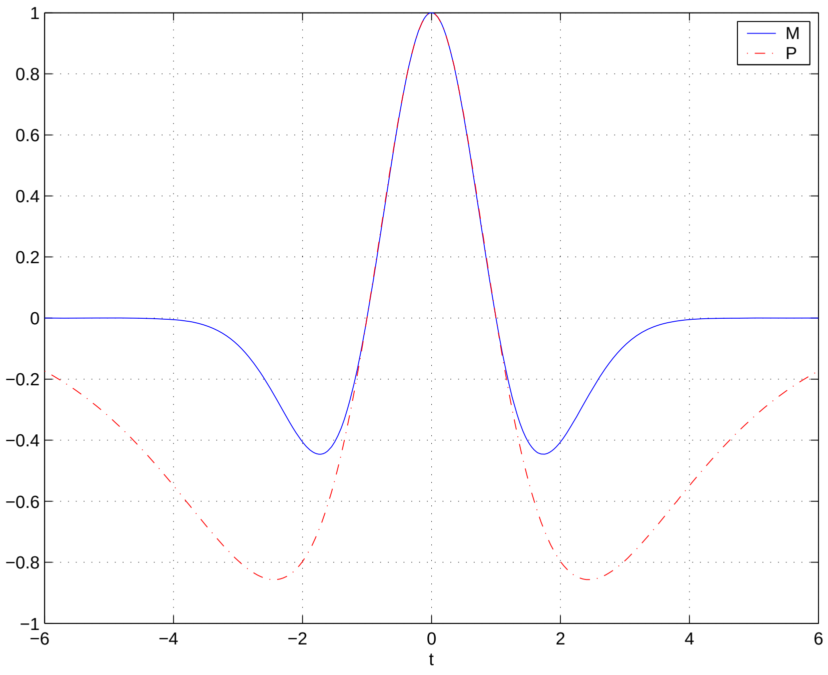
but this plot does not readily help you identify the curves.



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 3-103 for complete details.

Variable-Precision Arithmetic

In this section…

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

```
  ans =
   0.833333333333333
```

With Symbolic Math Toolbox software, the statement

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

uses symbolic computation to yield

```
ans =
5/6
```

And, also with the toolbox, the statements

```
digits(25)
```

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

use variable-precision arithmetic to return

```
ans =
0.83333333333333333333333333333333
```

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

```
format short;
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 is:

```
S = sym(A)

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    0.6324
```

whose elements are not the ratios of small integers, so

```
sym(E)
```

reproduces the floating-point representation in a symbolic form:

```
ans =
[ 3060513257434037/1125899906842624,   910872158600853/562949953421312]
[ 2473854946935173/2251799813685248, 1423946432832521/2251799813685248]
```

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

```
digits(4);
vpa(S)
```

MATLAB returns the output

```
ans =
[ 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, 0.6323592462254095103446616]
```

Restore the default `digits` setting:

```
digits(32);
```

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

```
format long;
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

```
ans =
262537412640768744.0
```

and, too,

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

returns

```
ans =
262537412640768744.0
```

You might suspect that `f` actually has an integer value. However, the 40-digit value

```
vpa(f,40)

ans =
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

```
H =
[   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

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

and

```
det(H)
```

yields

```
ans =
1/2160
```

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

```
% Solve Hx = b
b = [1; 1; 1];
x = H\b
```

produce the solution

```
 x =
  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

```
digits(16);
V = vpa(hilb(3))
```

returns

```
V =
[                1.0,                 0.5, 0.3333333333333333]
[                0.5, 0.3333333333333333,               0.25]
[ 0.3333333333333333,                0.25, 0.2000000000007276]
```

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

```
ans =
 2160.000000000018
```

and

```
V\b
```

which is

```
ans =
  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:

```
ans =
[ empty sym ]
```

Calculating the column space of `H` with

```
colspace(H)
```

returns a permutation of the identity matrix:

```
ans =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]
```

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, null space and column space are nontrivial:

```
Z = null(H)
C = colspace(H)

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) and [V,E] = eig(A).

The variable-precision counterparts areE = eig(vpa(A)) and [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 = sym([8/9 1/2 1/3; 1/2 1/3 1/4; 1/3 1/4 1/5])
```

```
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` and the diagonal elements of `E` are the eigenvalues of `H`:

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

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

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

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

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

$x^2 - \frac{64}{45}x + \frac{253}{2160}$ 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 =
     1020.420188201504727818545749884
    1019.9935501291629257348091808173
    1019.5243552632016358324933278291
    1000.1206982933841335712817075454
    999.94691786044276755320289228602
   0.21803980548301606860857564424981
 -0.17053529728768998575200874607757
     -1020.05321425589151659318942526
```

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

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

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

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

how =
simplify
```

Next, the command

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

produces

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

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

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

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

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)`. 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
```

For `n = 5`, the matrix is

```
A

A =
[   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 efficient way to generate the matrix is

```
n = 5;
[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
  3.1415926535897543920684990722
 3.14159265358767361712392612382
 3.14159265349961053143856838564
 3.14159265052654880815569479613
 3.14159256925492306470284863101
 3.14159075458605848728982577118
  3.1415575435991808369105065826
 3.14106044663470063805218371923
 3.13504054399744654843898901261
 3.07790297231119748658424727353
 2.69162158686066606774782763593
 1.20968137605668985332455685355
```

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`:

```
format long;
double(pi*ones(16,1)-S)
```

The results are

```
ans =
   0.000000000000000
   0.000000000000000
   0.000000000000000
   0.000000000000001
   0.000000000000039
   0.000000000002120
   0.000000000090183
   0.000000003063244
   0.000000084334870
   0.000001899003735
```

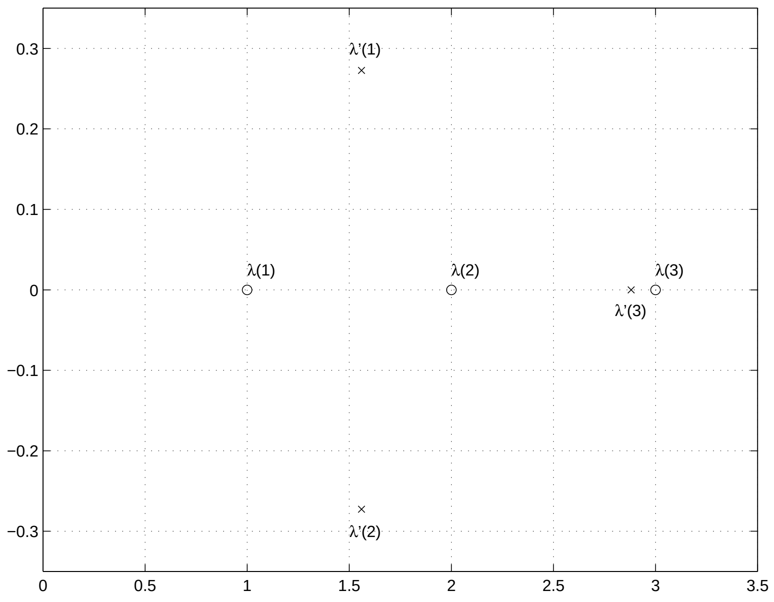
```
0.000035109990612
0.000532206955093
0.006552109592347
0.063689681278596
0.449971066729127
1.931911277533103
```

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

Eigenvalue Trajectories

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

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



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

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

```
A = gallery(3)

A =
  -149      -50     -154
   537      180      546
   -27       -9      -25
```

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

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

produce

```
e =
   1.000000000010722
   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

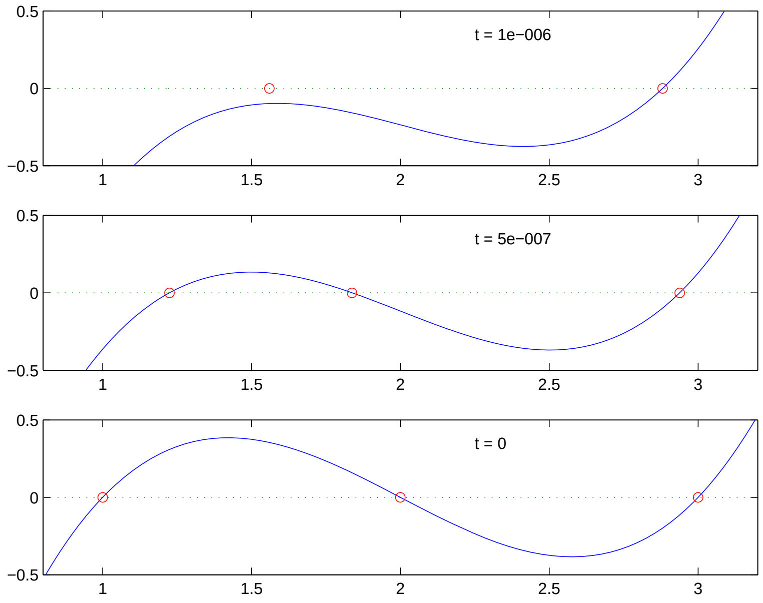
```
p =
11*x - 1221271*t - x^2*(t + 6) + 492512*t*x + x^3 - 6
```

`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 =
(1221271*t)/2 + (t + 6)^3/27 - ((492512*t + 11)*(t + 6))/6 +...
(((492512*t)/3 - (t + 6)^2/9 + 11/3)^3 + ((1221271*t)/2 +...
(t + 6)^3/27 - ((492512*t + 11)*(t + 6))/6 + 3)^2)^(1/2) + 3

pretty(ee)
```

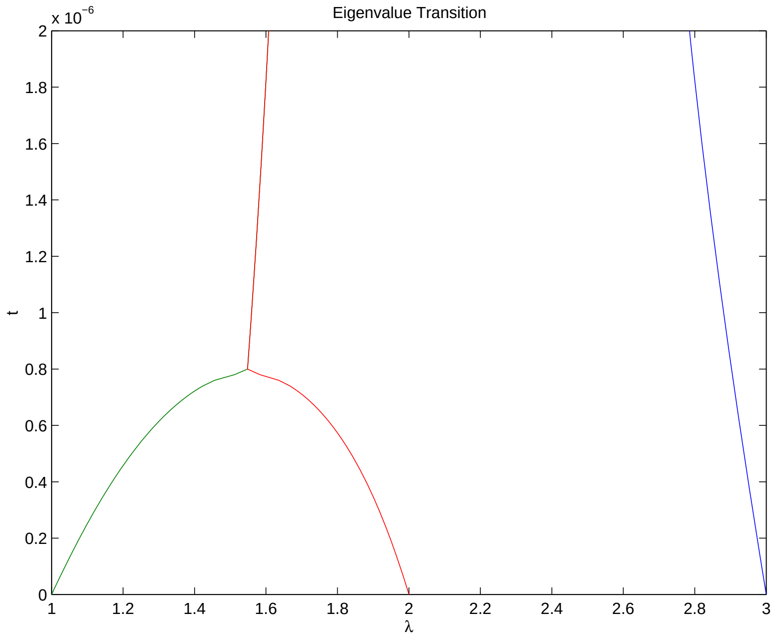
showing that `e(2)` and `e(3)` form a complex conjugate pair:

```
+-                                                                                          -+
|                                                         2                                  |
|                                   1   492512 t   (t + 6)    11                             |
|                                   -   -------- - -------- + --                             |
|                          t        3      3          9       3                              |
|                          - + sigma  - ------------------------ + 2                         |
|                          3                          1                                      |
|                                                     -                                      |
|                                                     3                                      |
|                                                sigma                                       |
|                                                                                            |
|                                                   /                            2      \    |
|                                                   |      1   492512 t   (t + 6)    11 |    |
|                                                   |      -   -------- - -------- + -- |    |
|                                               1/2 |      3      3          9       3  |    |
|                                              3    | sigma  + ------------------------ | i  |
|           1                     2                 |                        1          |    |
|           -   492512 t   (t + 6)    11            |                        -          |    |
|           3   -------- - -------- + --            |                        3          |    |
|  t   sigma       3          9       3             \                   sigma           /    |
|  - - ------ + ------------------------ + 2 - --------------------------------------------  |
|  3     2                     1                                    2                        |
|                              -                                                             |
|                              3                                                             |
|                       2 sigma                                                              |
|                                                                                            |
|                                                   /                            2      \    |
|                                                   |      1   492512 t   (t + 6)    11 |    |
|                                                   |      -   -------- - -------- + -- |    |
|                                               1/2 |      3      3          9       3  |    |
|                                              3    | sigma  + ------------------------ | i  |
|           1                     2                 |                        1          |    |
|           -   492512 t   (t + 6)    11            |                        -          |    |
|           3   -------- - -------- + --            |                        3          |    |
|  t   sigma       3          9       3             \                   sigma           /    |
|  - - ------ + ------------------------ + 2 + --------------------------------------------  |
|  3     2                     1                                    2                        |
|                              -                                                             |
|                              3                                                             |
|                       2 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.



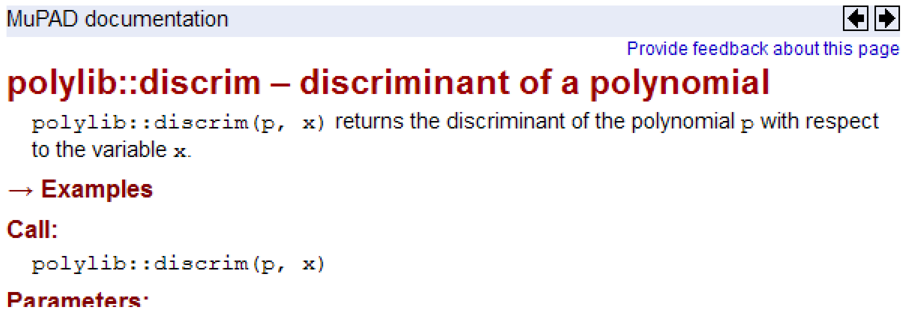
Above `t = 0.8e`$^{-6}$, the graphs of two of the eigenvalues intersect, while below `t` = 0.8e^{-6}, two real roots become a complex conjugate pair. What is the precise value of `t` that marks this transition? Let τ denote this value of `t`.

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 the MuPAD 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

```
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.

```
s = solve(discrim);
tau = vpa(s)
tau =
                                                                     1970031.04061804553618913725474883634597991201389
                                                                  0.000000783792490596794010485879469854518820556090553664
                              0.00000107692481604921513807537160160597784208236311263 - 0.00000308544636502289065492747*i
0.0000030854463650228906549274653827563618021771075729 5*i + 0.00000107692481604921513807537160160597784249167873707
```

Of the four solutions, you know that

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

is the transition point

```
tau =
0.000000783792490596794010 48084
```

because it is closest to the previous estimate.

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

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

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

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

Find τ now by

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

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

```
tau =
  7.8379e-007
```

Therefore, the second element of `sol.x`

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

is the double eigenvalue

```
sigma =
    1.5476
```

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

```
e = eig(double(subs(A, t, tau)))
```

```
e =
    1.5476
    1.5476
    2.9048
```

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

Solving Equations

In this section...

Solving Algebraic Equations

If `S` is a symbolic expression,

```
solve(S)
```

attempts to find values of the symbolic variable in `S` (as determined by `symvar`) 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
```

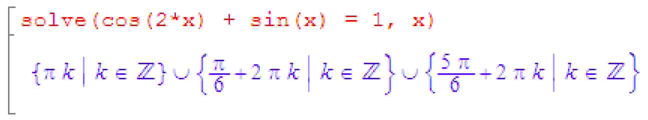
Note that these examples assume equations of the form $f(x) = 0$. If you need to solve equations of the form $f(x) = q(x)$, you must use quoted strings. In particular, the command

```
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:



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

$$\{\pi k \mid k \in \mathbb{Z}\} \cup \left\{\frac{\pi}{6} + 2\pi k \mid k \in \mathbb{Z}\right\} \cup \left\{\frac{5\pi}{6} + 2\pi k \mid k \in \mathbb{Z}\right\}$$

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 = sym('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 $x^2y^2 = 1$ 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 `symvar` 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]

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

results 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 =
1 - (8*(cos(x)/2 - 1/2)^2)/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
$\frac{dy}{dt} + 4y(t) = e^{-t}$ $y(0) = 1$	`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')`
$\frac{d^2y}{dx^2} = xy(x)$ $y(0) = 0,\ y(3) = \frac{1}{\pi}K_{1/3}(2\sqrt{3})$ (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 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 toolbox returns the computed solutions in the structure `S`. You can determine the values of `f` and `g` by typing

```
f = S.f
g = S.g

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

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

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

```
[f, g] = dsolve('Df = 3*f + 4*g, Dg = -4*f + 3*g',...
'f(0) = 0, g(0) = 1')

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

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

Now, suppose you are solving a system of differential equations in matrix form. For example, solve the system $Y' = AY + B$, where A, B, and Y represent the following matrices:

```
syms t x y;
A = [1 2; -1 1];
B = [1; t];
Y = [x; y];
sys = A*Y + B

sys =
 x + 2*y + 1
   t - x + y
```

The `dsolve` function does not accept matrices. To be able to use this solver, extract the components of the matrix and convert them to strings:

```
eq1 = char(sys(1))
eq2 = char(sys(2))

eq1 =
x + 2*y + 1

eq2 =
t - x + y
```

Use the `strcat` function to concatenate the left and right sides of the equations. Use the `dsolve` function to solve the system:

```
[x, y] = dsolve(strcat('Dx = ',eq1), strcat('Dy = ',eq2))

x =
sin(2^(1/2)*t)^2/9 + (2*t*cos(2^(1/2)*t)^2)/3 +...
(2*t*sin(2^(1/2)*t)^2)/3 + cos(2^(1/2)*t)^2/9 +...
2^(1/2)*C2*exp(t)*cos(2^(1/2)*t) +...
2^(1/2)*C1*exp(t)*sin(2^(1/2)*t)

y =
C1*exp(t)*cos(2^(1/2)*t) - (t*cos(2^(1/2)*t)^2)/3 -...
(t*sin(2^(1/2)*t)^2)/3 - (2*cos(2^(1/2)*t)^2)/9 -...
(2*sin(2^(1/2)*t)^2)/9 - C2*exp(t)*sin(2^(1/2)*t)
```

Integral Transforms and Z-Transforms

In this section...

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

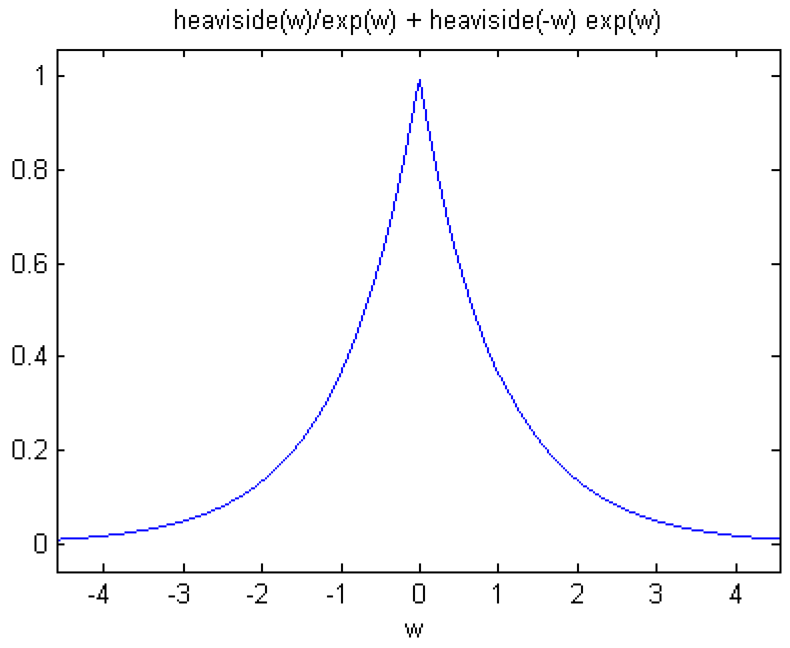
$$F^{-1}[f](x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(w)e^{iwx}dw.$$

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)
```

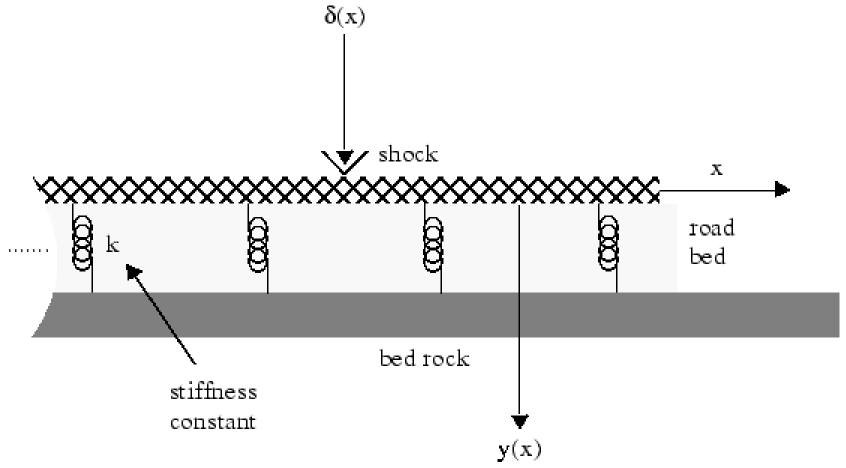


The Fourier transform is symmetric, since the original Cauchy density function is symmetric.

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

```
finvfcauchy = ifourier(fcauchy)
finvfcauchy =
-(1/(- 1 + x*i) - 1/(1 + x*i))/(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.

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 $\delta(x)$. The Dirac Delta function 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,

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

To see a demonstration of this property, try this

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

which returns

```
ans =
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

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

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

Notice that g contains the Heaviside distribution

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

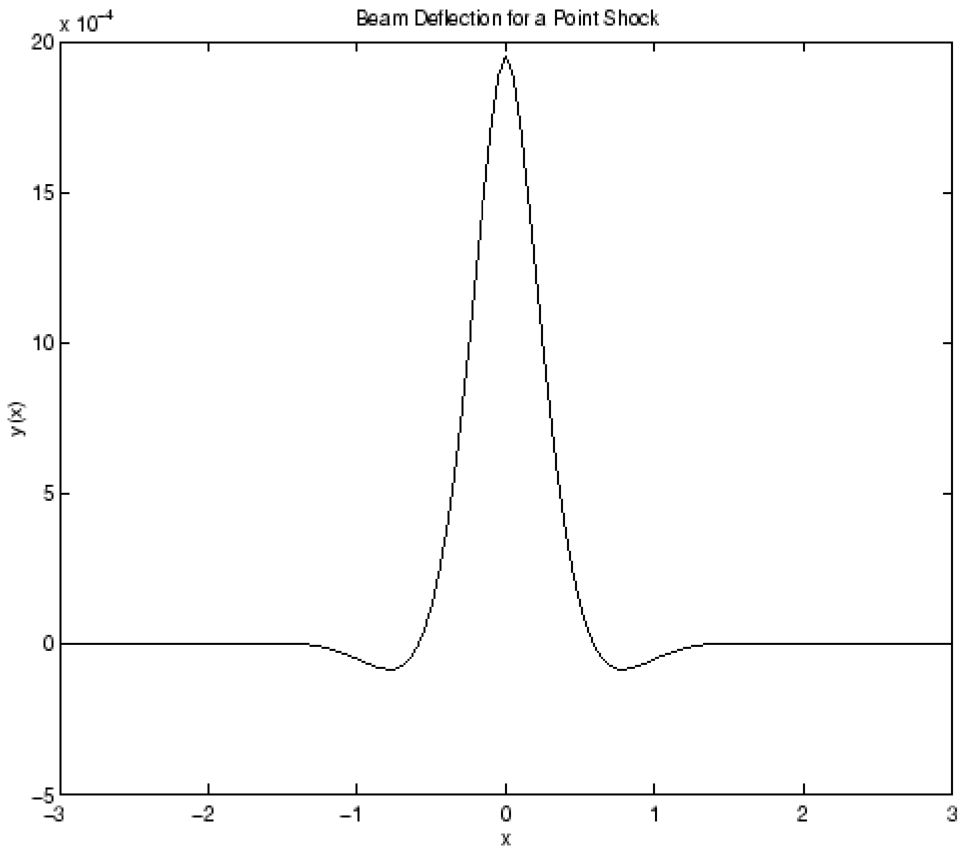
Since Y is the product of Fourier transforms, y is the convolution of the transformed functions. That is, $F[y] = Y(w) = \Delta(w)\ G(w) = F[\delta]\ F[g]$ implies

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

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

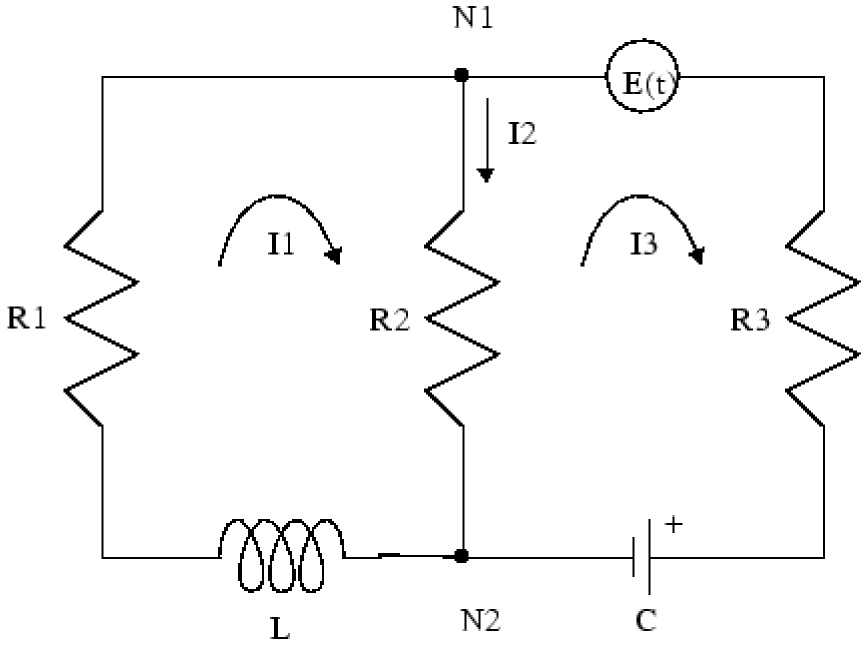
$$L[f](s) = \int_0^\infty f(t)e^{-ts}dt,$$

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 (amperes), 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, and Faraday'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 Ω (ohms), $R2$ = 2 Ω, $R3$ = 3 Ω, C = 1/4 farads, L = 1.6 H (henrys), $I1(0)$ = 15 A (amperes), and $Q(0)$ = 2 A/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 =
(s + 5/4)*LI1 + (5*LQ*s)/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 =
(s + 4/5)*LQ - (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 =
(5*(60*s^3 + 56*s^2 + 59*s + 56))/((s^2 + 1)*(20*s^2 + 51*s + 20))


LQ =
(40*s^3 + 190*s^2 + 44*s + 195)/((s^2 + 1)*(20*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)
```

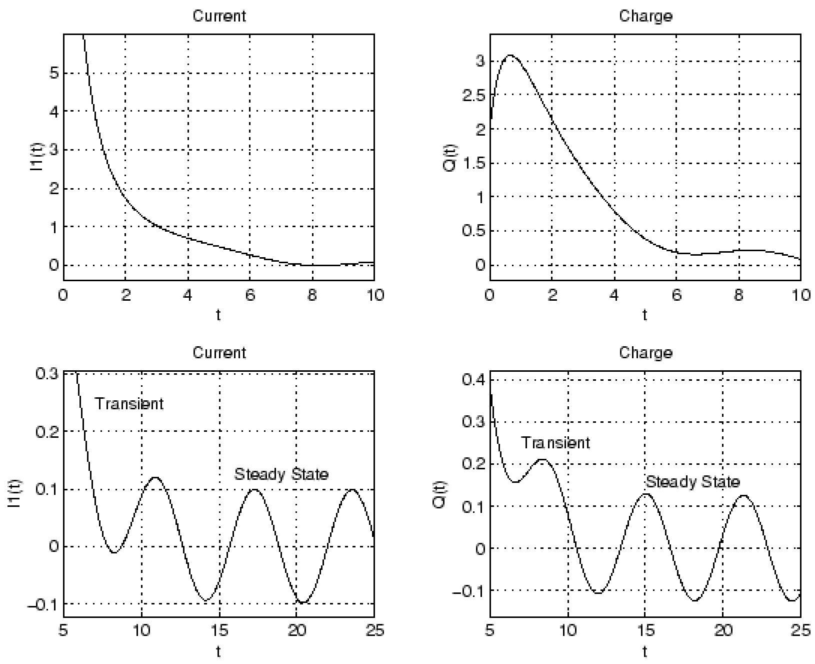
yields

```
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, $0 \le t \le 10$ and $5 \le t \le 25$. 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

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

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^2 - z - 1)*Pz - z^2 - z
```

Now solve for `Pz`

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

to obtain

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

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

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

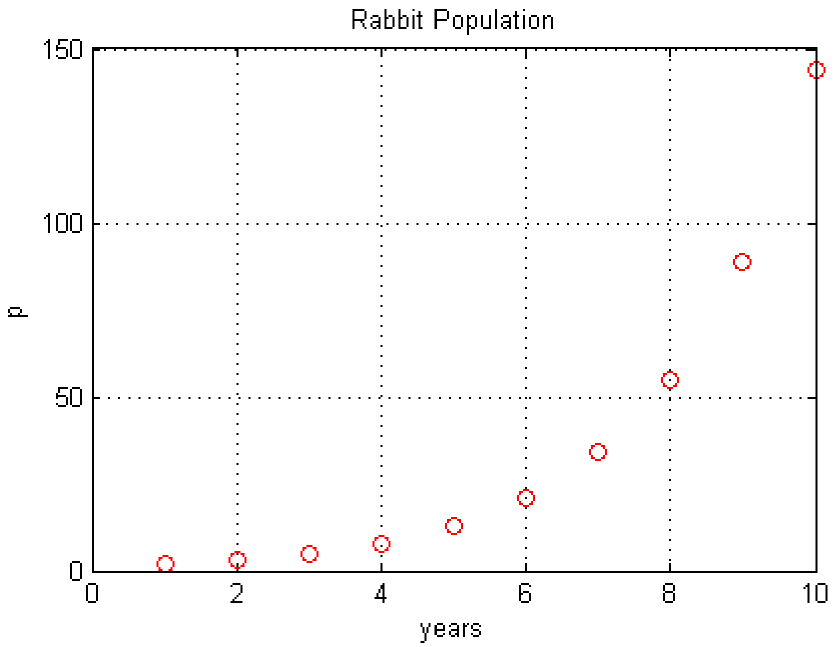
The result is a bit complicated, but explicit:

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

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., Shivamoggi, B.K., *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 3-121 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 the special value `NaN` 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: $\frac{e^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \cdot \frac{t^{n-1}}{n!}$	`bernoulli(n)` `bernoulli(n,t)`	$n \geq 0$ $0 < \lvert t \rvert < 2\pi$
Bessel functions	`BesselI, BesselJ`—Bessel functions of the first kind. `BesselK, BesselY`—Bessel functions of the second kind.	`BesselJ(v,x)` `BesselY(v,x)` `BesselI(v,x)` `BesselK(v,x)`	`v` is real.
Beta function	$B(x, y) = \frac{\Gamma(x) \cdot \Gamma(y)}{\Gamma(x + y)}$	`Beta(x,y)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Binomial coefficients	$\binom{m}{n} = \frac{m!}{n!(m-n)!}$ $= \frac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$	`binomial(m,n)`	

Complete elliptic integrals	Legendre's complete elliptic integrals of the first, second, and third kind. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.	`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. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.	`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	$erfc(z) = \frac{2}{\sqrt{\pi}} \cdot \int_z^\infty e^{-t^2} dt = 1 - erf(z)$
	 $erfc(-1,z) = \frac{2}{\sqrt{\pi}} \cdot e^{-z^2}$
	 $erfc(n,z) = \int_z^\infty erfc(n-1,t)dt$
		`erfc(z)`
		 `erfc(n,z)`
			$n > 0$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Dawson's integral	$F(x) = e^{-x^2} \cdot \int_0^x e^{t^2} dt$	`dawson(x)`	
Digamma function	$\Psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	
Dilogarithm integral	$f(x) = \int_1^x \frac{\ln(t)}{1-t} dt$	`dilog(x)`	$x > 1$
Error function	$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	`erf(z)`	
Euler numbers and polynomials	Generating function for Euler numbers: $\frac{1}{\cosh(t)} = \sum_{n=0}^{\infty} E_n \frac{t^n}{n!}$	`euler(n)` `euler(n,z)`	$n \geq 0$ $\lvert t \rvert < \frac{\pi}{2}$
Exponential integrals	$Ei(n,z) = \int_1^{\infty} \frac{e^{-zt}}{t^n} dt$ $Ei(x) = PV\left(-\int_{-\infty}^{x} \frac{e^t}{t}\right)$	`Ei(n,z)` `Ei(x)`	$n \geq 0$ $\text{Real}(z) > 0$
Fresnel sine and cosine integrals	$C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt$ $S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt$	`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. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.	`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` $= z$	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Logarithm of the gamma function	$\mathrm{lnGAMMA}(z) = \ln(\Gamma(z))$	`lnGAMMA(z)`	
Logarithmic integral	$Li(x) = PV\left\{\int_0^x \frac{dt}{\ln t}\right\} = Ei(\ln x)$	`Li(x)`	$x > 1$
Polygamma function	$\Psi^{(n)}(z) = \frac{d^n}{dz}\Psi(z)$ where $\Psi(z)$ is the Digamma function.	`Psi(n,z)`	$n \geq 0$
Shifted sine integral	$Ssi(z) = Si(z) - \frac{\pi}{2}$	`Ssi(z)`	

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)`	

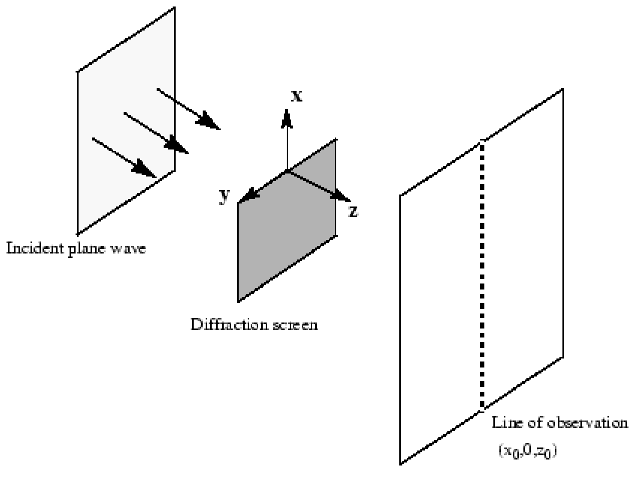
Orthogonal Polynomials (Continued)

Polynomial	mfun Name	Arguments
Generalized Laguerre	`L(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	

Diffraction Example

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

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



The intensity of the diffracted wave is given by

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

where

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

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

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

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

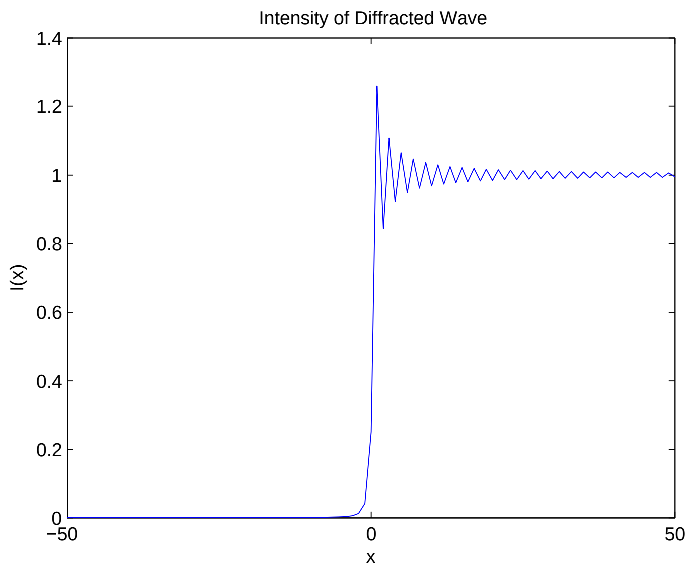
How does the intensity of the diffracted wave behave along the line of observation? Since k and z_0 are constants independent of x, you set

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

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.

Using Graphics

In this section...

Creating Plots

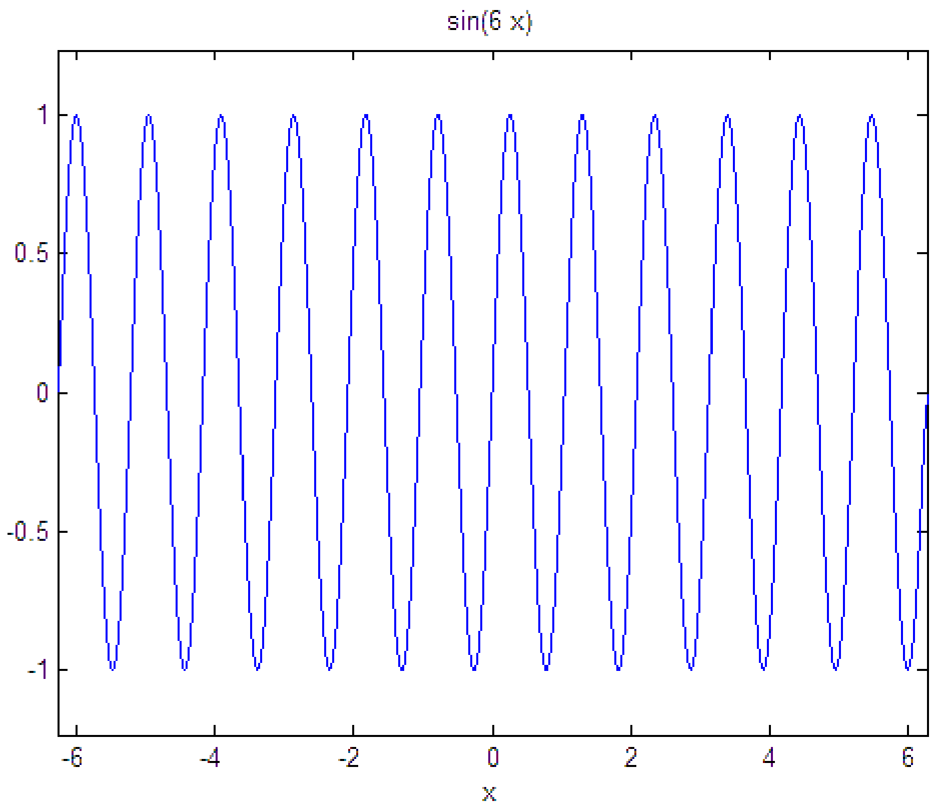
Using Symbolic Plotting Functions

MATLAB provides a wide variety of techniques for plotting numerical data. Graphical capabilities of MATLAB include plotting tools, standard plotting functions, graphic manipulation and data exploration tools, and tools for printing and exporting graphics to standard formats. The Symbolic Math Toolbox features expand the graphic capabilities of MATLAB and enable you to plot symbolic functions. The toolbox provides the following plotting functions:

- `ezplot` that creates 2-D plots of explicit and implicit symbolic functions in Cartesian coordinates.
- `ezplot3` that creates 3-D parametric function plots. The `animate` option of `ezplot3` lets you create animated function plots.
- `ezpolar` that creates symbolic function plots in polar coordinates.
- `ezsurf` that creates surface plots of symbolic functions. The `ezsurfc` plotting function creates combined surface and contour function plots.
- `ezcontour` that creates contour plots of symbolic functions. The `ezcontourf` function creates filled contour plots.
- `ezmesh` that creates mesh plots of symbolic functions. The `ezmeshc` function creates combined mesh and contour function plots.

For example, plot the symbolic function `sin(6x)` in Cartesian coordinates. By default, `ezplot` uses the range $-2\pi < x < 2\pi$:

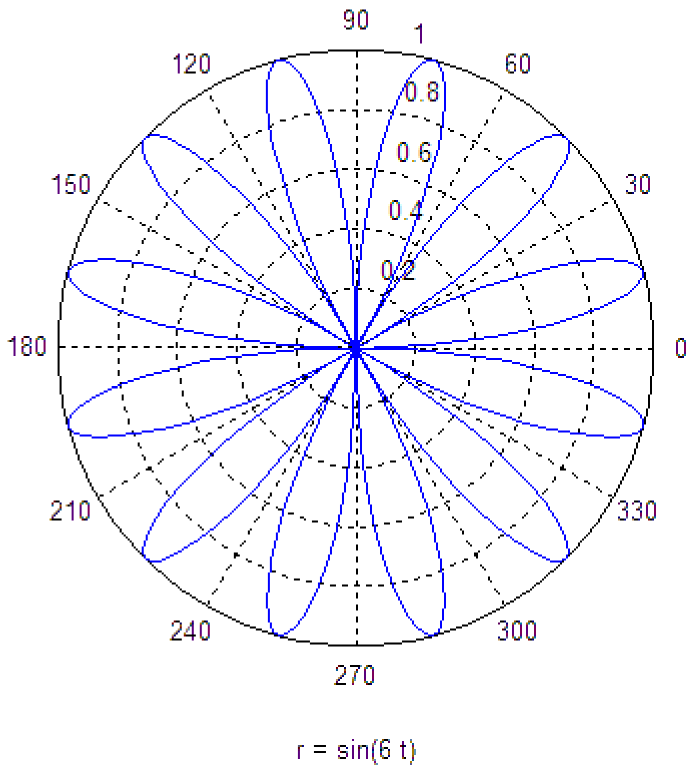
```
syms x;
ezplot(sin(6*x))
```



When plotting a symbolic function, `ezplot` uses the default 60-by-60 grid (mesh setting). The plotting function does not adapt the mesh setting around steep parts of a function plot or around singularities. (These parts are typically less smooth than the rest of a function plot.) Also, `ezplot` does not allow you to change the mesh setting.

To plot a symbolic function in polar coordinates r (radius) and θ (polar angle), use the `ezpolar` plotting function. By default, `ezpolar` plots a symbolic function over the domain $0 < \theta < 2\pi$. For example, plot the function `sin(6t)` in polar coordinates:

```
syms t;
ezpolar(sin(6*t))
```



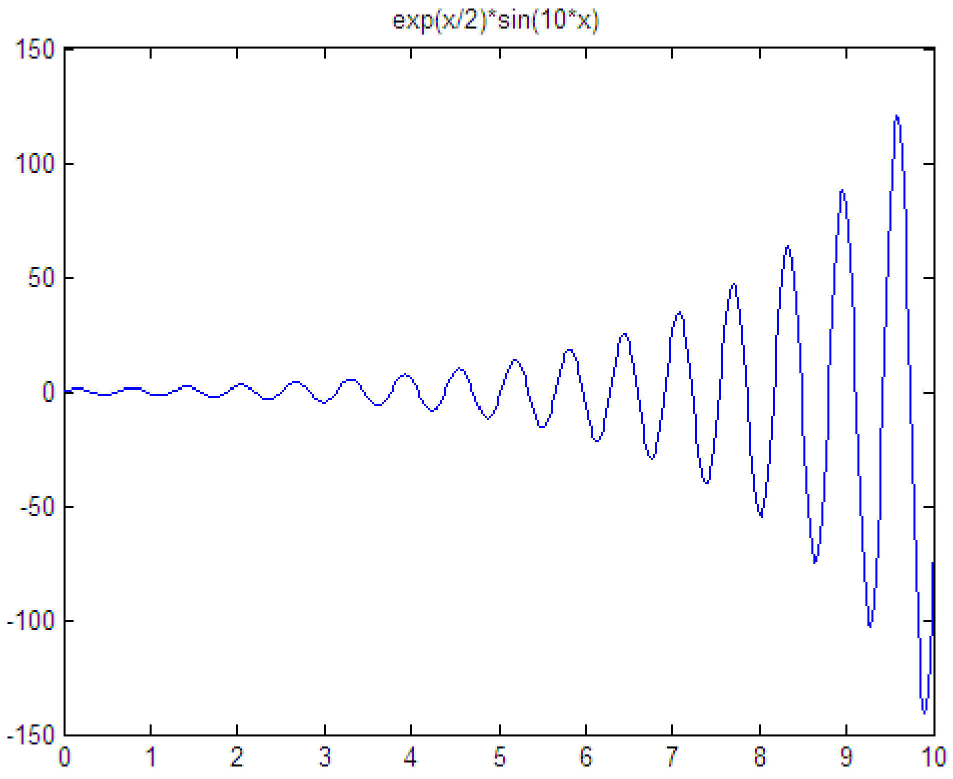
Using MATLAB Plotting Functions

When plotting a symbolic expression, you also can use the plotting functions provided by MATLAB. For example, plot the symbolic expression $e^{x/2} \sin(10x)$. First, use `matlabFunction` to convert the symbolic expression to a MATLAB function. The result is a function handle `h` that points to the resulting MATLAB function:

```
syms x;
h = matlabFunction(exp(x/2)*sin(10*x))
```

Now, plot the resulting MATLAB function by using one of the standard plotting functions that accept function handles as arguments. For example, use the `fplot` function:

```
fplot(h, [0 10]);
hold on;
title('exp(x/2)*sin(10*x)');
hold off
```



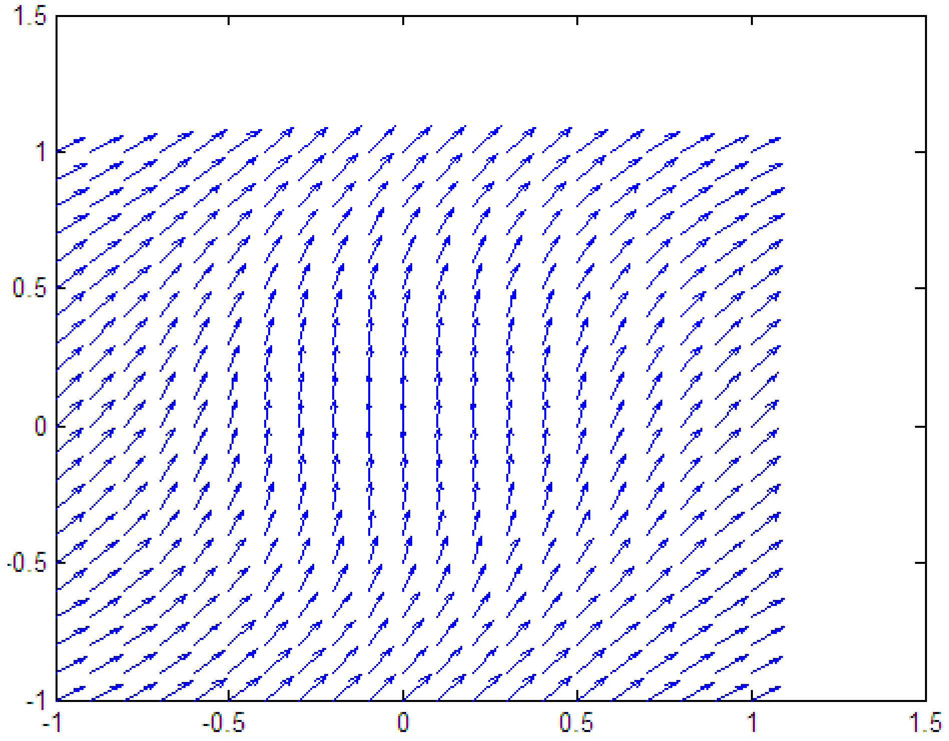
An alternative approach is to replace symbolic variables in an expression with numeric values by using the `subs` function. For example, in the following expressions *u* and *v*, substitute the symbolic variables *x* and *y* with the numeric values defined by `meshgrid`:

```
syms x y
```

```
u = sin(x^2 + y^2); v = cos(x*y);
[X, Y] = meshgrid(-1:.1:1,-1:.1:1);
U = subs(u, [x y], {X,Y}); V = subs(v, [x y], {X,Y});
```

Now, you can use standard MATLAB plotting functions to plot the expressions *U* and *V*. For example, create the plot of a vector field defined by the functions *U*(*X*, *Y*) and *V*(*X*, *Y*):

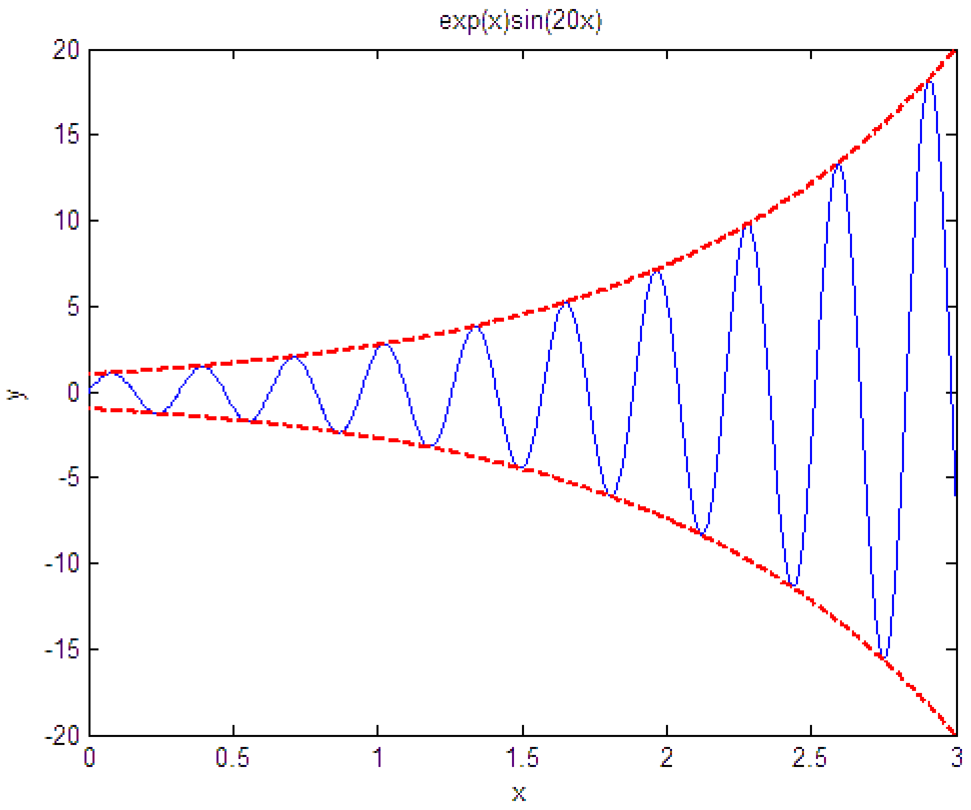
```
quiver(X, Y, U, V)
```



Plotting Multiple Symbolic Functions in One Graph

To plot several symbolic functions in one graph, add them to the graph sequentially. To be able to add a new function plot to the graph that already contains a function plot, use the `hold on` command. This command retains the first function plot in the graph. Without this command, the system replaces the existing plot with the new one. Now, add new plots. Each new plot appears on top of the existing plots. While you use the `hold on` command, you also can change the elements of the graph (such as colors, line styles, line widths, titles) or add new elements. When you finish adding new function plots to a graph and modifying the graph elements, enter the `hold off` command:

```
syms x y;
ezplot(exp(x)*sin(20*x) - y, [0, 3, -20, 20]);
hold on;
p1 = ezplot(exp(x) - y, [0, 3, -20, 20]);
set(p1,'Color','red', 'LineStyle', '--', 'LineWidth', 2);
p2 = ezplot(-exp(x) - y, [0, 3, -20, 20]);
set(p2,'Color','red', 'LineStyle', '--', 'LineWidth', 2);
title('exp(x)sin(20x)');
hold off
```



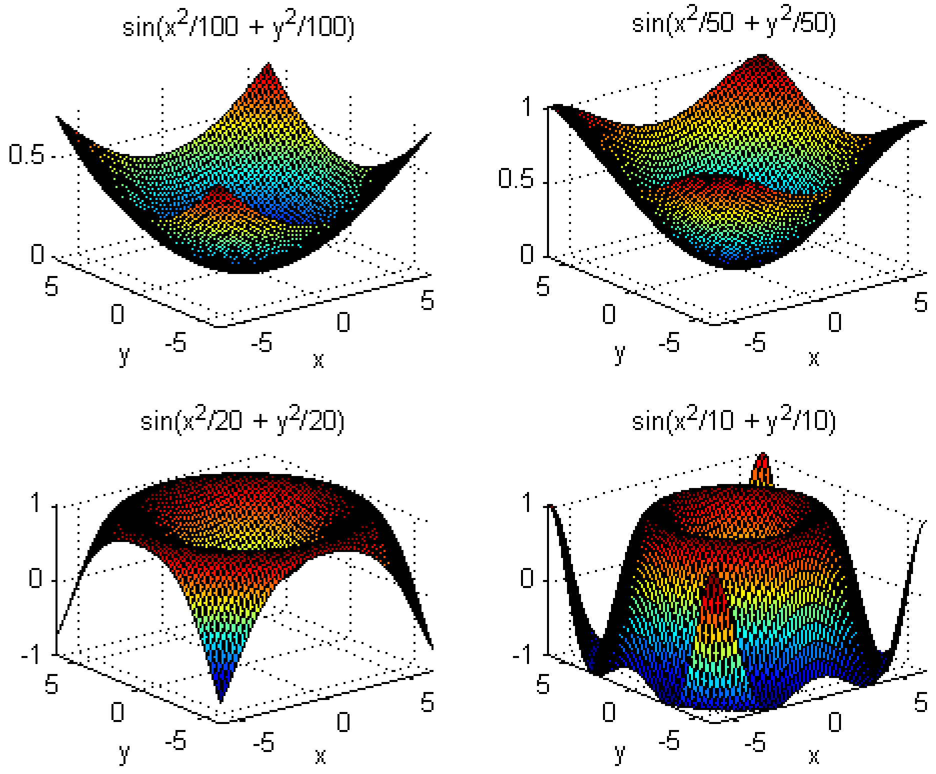
Plotting Multiple Symbolic Functions in One Figure

To display several function plots in one figure without overlapping, divide a figure window into several rectangular panes (tiles). Then, you can display each function plot in its own pane. For example, you can assign different values to symbolic parameters of a function, and plot the function for each value of a parameter. Collecting such plots in one figure can help you compare the plots. To display multiple plots in the same window, use the `subplot` command:

```
subplot(m,n,p)
```

This command partitions the figure window into an `m`-by-`n` matrix of small subplots and selects the subplot `p` for the current plot. MATLAB numbers the subplots along the first row of the figure window, then the second row, and so on. For example, plot the expression `sin(x^2 + y^2)/a` for the following four values of the symbolic parameter `a`:

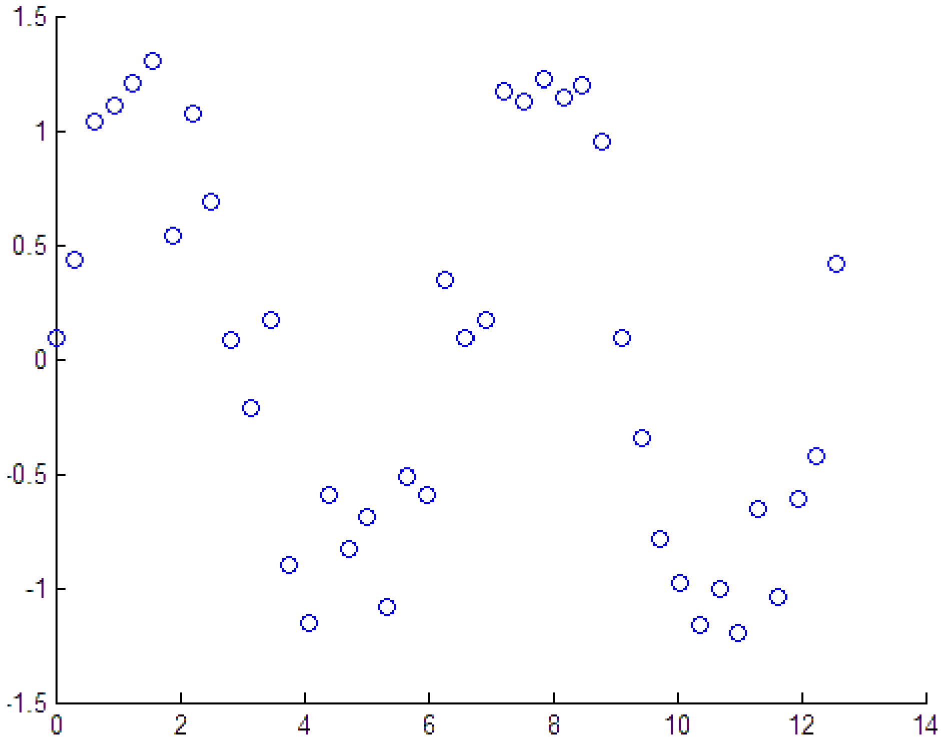
```
syms x y;
z = x^2 + y^2;
subplot(2, 2, 1); ezsurf(sin(z/100));
subplot(2, 2, 2); ezsurf(sin(z/50));
subplot(2, 2, 3); ezsurf(sin(z/20));
subplot(2, 2, 4); ezsurf(sin(z/10));
```



Combining Symbolic Function Plots and Numeric Data Plots

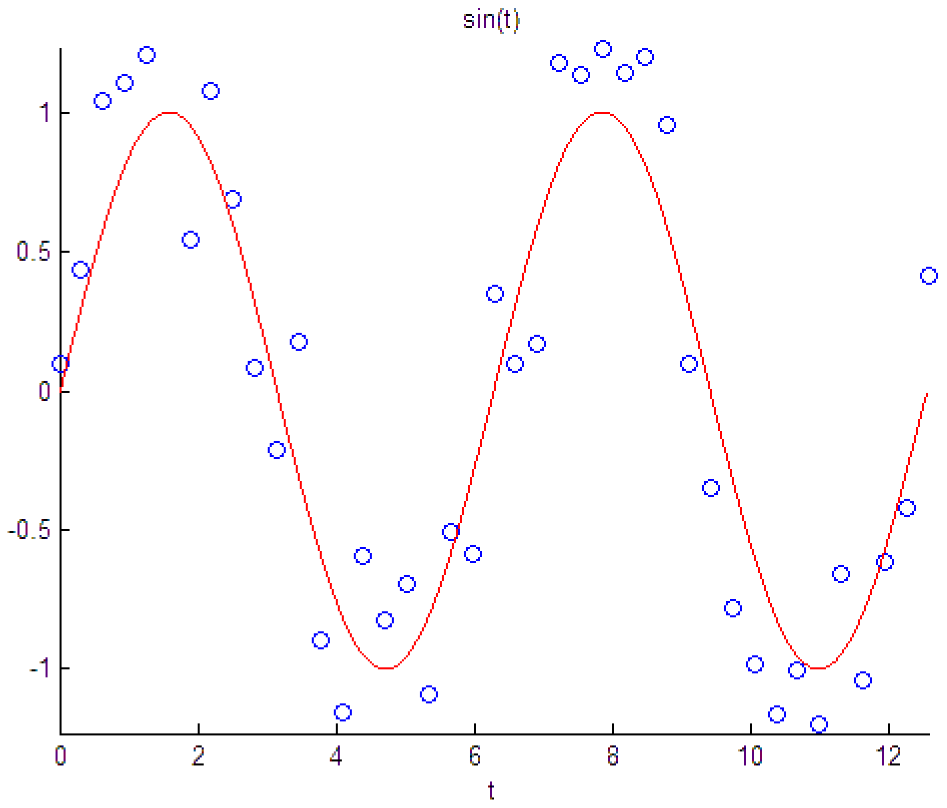
The combined graphical capabilities of MATLAB and the Symbolic Math Toolbox software let you plot numeric data and symbolic functions in one graph. Suppose, you have two discrete data sets, *x* and *y*. Use the `scatter` plotting function to plot these data sets as a collection of points with coordinates (*x1*, *y1*), (*x2*, *y2*), ..., (*x3*, *y3*):

```
x = 0:pi/10:4*pi;
y = sin(x) + (-1).^randi(10, 1, 41).*rand(1, 41)./2;
scatter(x, y)
```



Now, suppose you want to plot the sine function on top of the scatter plot in the same graph. First, use the `hold on` command to retain the current plot in the figure. (Without this command, the symbolic plot that you are about to create replaces the numeric data plot.) Then, use `ezplot` to plot the sine function. By default, MATLAB does not use a different color for a new function; the sine function appears in blue. To change the color or any other property of the plot, create the handle for the `ezplot` function call, and then use the `set` function:

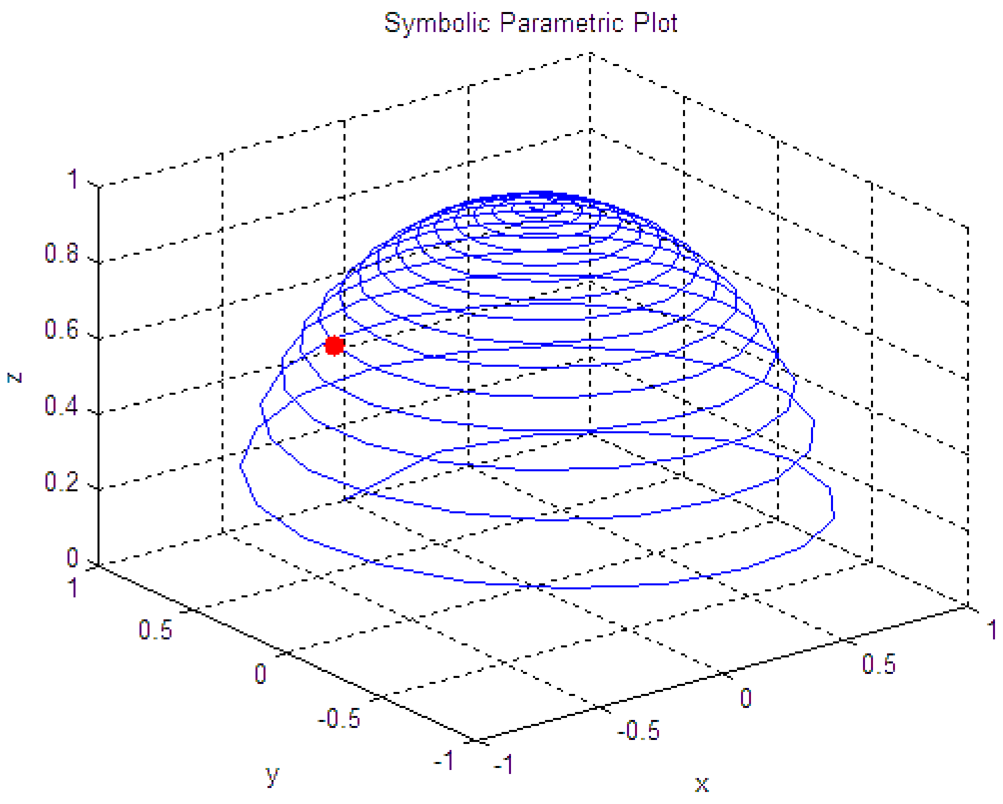
```
hold on;
syms t;
p = ezplot(sin(t), [0 4*pi]);
set(p,'Color','red');
```



MATLAB provides the plotting functions that simplify the process of generating spheres, cylinders, ellipsoids, and so on. The Symbolic Math Toolbox software lets you create a symbolic function plot in the same graph with these volumes. For example, use the following commands to generate the spiral function plot wrapped around the top hemisphere. The `animate` option switches the `ezplot3` function to animation mode. The red dot on the resulting graph moves along the spiral:

```
syms t;
```

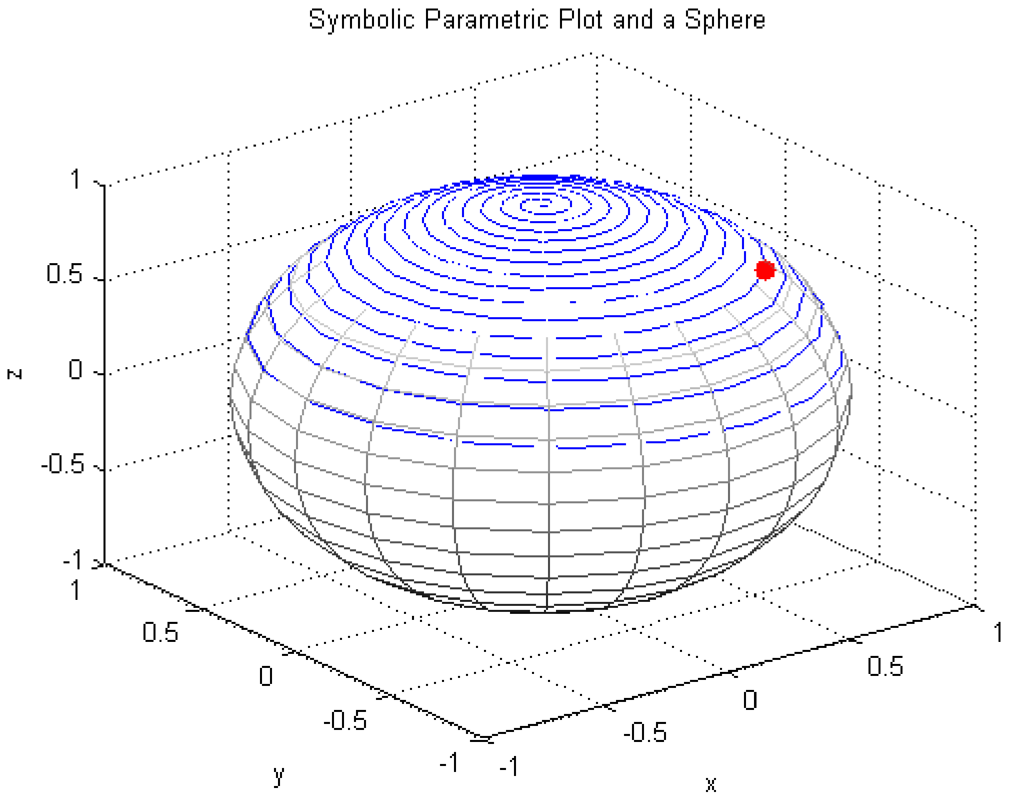
```
x = (1-t)*sin(100*t);
y = (1-t)*cos(100*t);
z = sqrt(1 - x^2 - y^2);
ezplot3(x, y, z, [0 1], 'animate');
title('Symbolic Parametric Plot');
```



Add the sphere with radius 1 and the center at (0, 0, 0) to this graph. The `sphere` function generates the required sphere, and the `mesh` function creates a mesh plot for that sphere. Combining the plots clearly shows that the symbolic parametric function plot is wrapped around the top hemisphere:

```
hold on;
[X,Y,Z] = sphere;
mesh(X, Y, Z);
```

```
colormap(gray);
title('Symbolic Parametric Plot and a Sphere');
```



Exploring Function Plots

Plotting a symbolic function can help you visualize and explore the features of the function. Graphical representation of a symbolic function can also help you communicate your ideas or results. MATLAB displays a graph in a special window called a *figure* window. This window provides interactive tools for further exploration of a function or data plot.

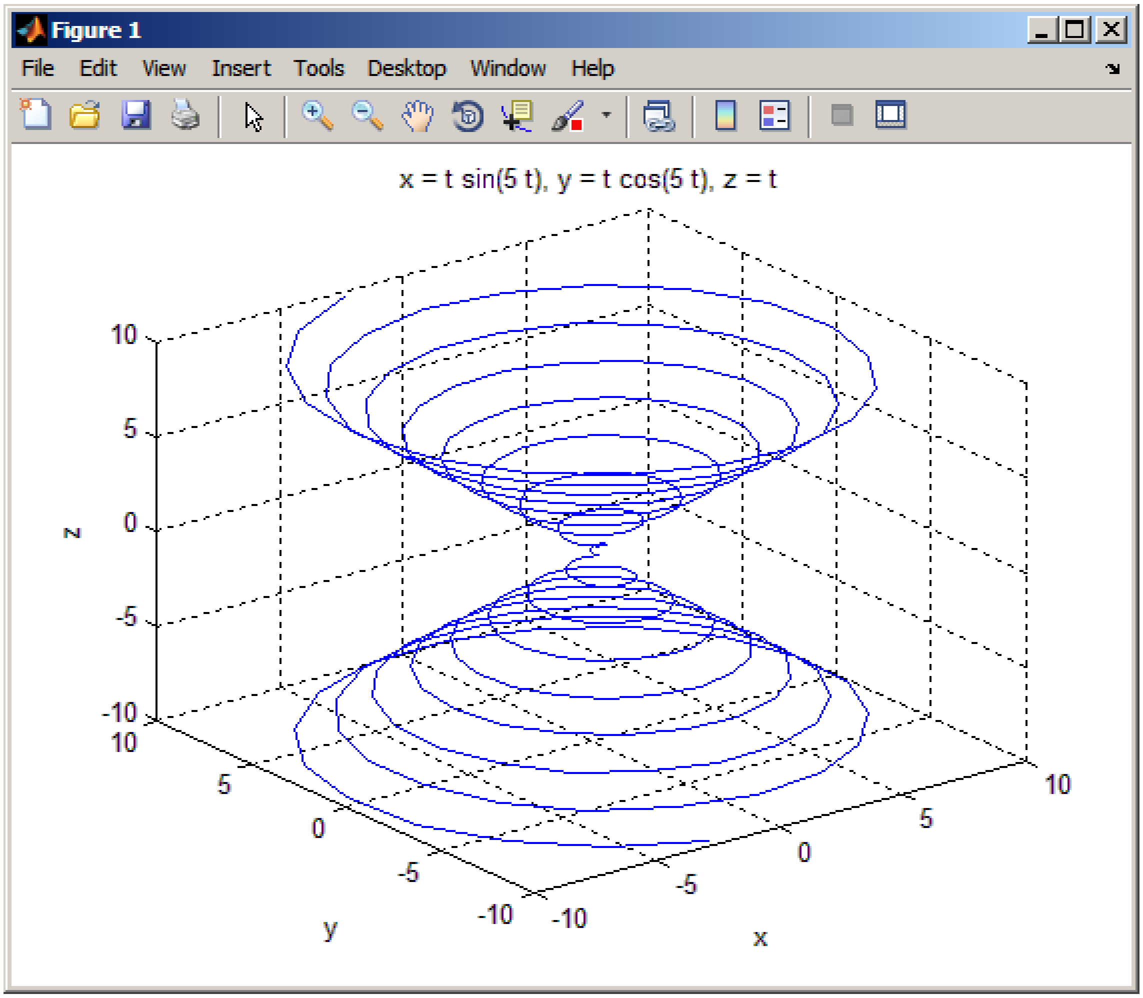


Figure 1
File Edit View Insert Tools Desktop Window Help
x = t sin(5 t), y = t cos(5 t), z = t
10
5
0
-5
-10
z
10
5
0
-5
-10
y
-10
-5
0
5
10
x

Interactive data exploration tools are available in the figure toolbar and also from the **Tools** menu. By default, a figure window displays one toolbar that provides shortcuts to the most common operations. You can enable two other toolbars from the **View** menu. When exploring symbolic function plots, use the same operations as you would for the numeric data plots. For example:

- Zoom in and out on particular parts of a graph ( ). Zooming allows you to see small features of a function plot. Zooming behaves differently for 2-D or 3-D views. For more information, see "Enlarging the View".

- Shift the view of the graph with the pan tool ( ). Panning is useful when you have zoomed in on a graph and want to move around the plot to view different portions. For more information, see "Panning — Shifting Your View of the Graph".

- Rotate 3-D graphs ( ). Rotating 3-D graphs allows you to see more features of the surface and mesh function plots. For more information, see "Rotate 3D — Interactive Rotation of 3-D Views".

- Display particular data values on a graph and export them to MATLAB workspace variables ( ). For more information, see "Data Cursor — Displaying Data Values Interactively".

For more information about data exploration tools available in MATLAB, see "Ways to Explore Graphical Data".

Editing Graphs

MATLAB supports the following two approaches for editing graphs:

- Interactive editing lets you use the mouse to select and edit objects on a graph.

- Command-line editing lets you use MATLAB commands to edit graphs.

These approaches work for graphs that display numeric data plots, symbolic function plots, or combined plots.

To enable the interactive plot editing mode in the MATLAB figure window, click the Edit Plot button ( ) or select **Tools > Edit Plot** from the main menu. If you enable plot editing mode in the MATLAB figure window, you can perform point-and-click editing of your graph. In this mode, you can modify the appearance of a graphics object by double-clicking the object and changing the values of its properties. For more information about interactive editing, see “Working in Plot Edit Mode”.

The complete collection of properties is accessible through a graphical user interface called the Property Editor. To open a graph in the Property Editor window:

1. Enable plot editing mode in the MATLAB figure window
2. Double-click any element on the graph.

For information about editing object properties in plot editing mode, see “The Property Editor”.

If you prefer to work from the MATLAB command line or if you want to create a code file, you can edit graphs by using MATLAB commands. For information about command-line graph editing, see “Understanding Handle Graphics® Objects”.

Also, you can combine the interactive and command-line editing approaches to achieve the look you want for the graphs you create.

Saving Graphs

After you create, edit, and explore a function plot, you might want to save the result. MATLAB provides three different ways to save graphs:

- Save a graph as a MATLAB FIG-file (a binary format). The FIG-file stores all information about a graph, including function plots, graph data, annotations, data tips, menus and other controls. You can open the FIG-file only with MATLAB. For more information, see “Saving a Graph in FIG-File Format”.
- Export a graph to a different file format. When saving a graph, you can choose a file format other than FIG. For example, you can export your graphs to EPS, JPEG, PNG, BMP, TIFF, PDF, and other file formats. You can open the exported file in an appropriate application. For more information, see “Saving to a Different Format — Exporting Figures”.

- Print a graph on paper or print it to file. To ensure the correct plot size, position, alignment, paper size and orientation, use Print Preview. For more information, see "Printing Figures".
- Generate a MATLAB file from a graph. You can use the generated code to reproduce the same graph or create a similar graph using different data. This approach is useful for generating MATLAB code for work that you have performed interactively with the plotting tools. For more information, see "Generating a MATLAB File to Recreate a Graph".

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*3.0D1)/(x*y**2+1.0D1)-x**3*(y**2+1.0D0)**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', 'fortrantest')
```

generates a file named `fortrantest` in the current folder. `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','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. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. 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 Functions

You can use `matlabFunction` to generate a MATLAB function handle that calculates numerical values as if you were substituting numbers for variables in a symbolic expression. Also, `matlabFunction` can create a file that accepts numeric arguments and evaluates the symbolic expression applied to the arguments. The generated 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(sqrt(x.^2+y.^2))
```

You can use this function handle to calculate numerically:

```
ht(.5,.5)

ans =
    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 3-147 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 symbolic arrays, or a vector of symbolic variables. For example:

```
syms x y z
r = sqrt(x^2 + 3*y^2 + 5*z^2);
```

```
ht1 = matlabFunction(tanh(r), 'vars', [y x z])

ht1 =
    @(y,x,z)tanh(sqrt(x.^2+y.^2.*3.0+z.^2.*5.0))

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

ht2 =
    @(x,y,z)tanh(sqrt(x.^2+y.^2.*3.0+z.^2.*5.0))

ht3 = matlabFunction(tanh(r), 'vars', {'x', [y z]})

ht3 =
    @(x,in2)tanh(sqrt(x.^2+in2(:,1).^2.*3.0+in2(:,2).^2.*5.0))
```

Generating a File

You can generate a file from a symbolic expression, in addition to a function handle. Specify the file name using the `file` option. Pass a string containing the file name or the path to the file. If you do not specify the path to the file, `matlabFunction` creates this file in the current folder.

This example generates a 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','testMatrix.m')
```

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

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

t2 = x.^2;
t3 = tan(y);
t4 = t2.*x;
t5 = t.^2;
t6 = t5 + 1;
```

```
t7 = 1./y;
t8 = t6.*t7.*x;
t9 = t3 + t4;
t10 = 1./t9;
F = [-(t10.*(t3 - t4))./t6,t8; t8,- t10.*(3.*t3 - 3.*t2.*x) - 1];
```

`matlabFunction` generates many intermediate variables. This is called *optimized* code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. Intermediate variables can make the resulting code more efficient by reusing intermediate expressions (such as `t4`, `t6`, `t8`, `t9`, and `t10` in the calculation of `F`). Using intermediate variables 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','testMatrix.m','vars',[x y t])
```

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

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

Naming Output Variables

By default, the names of the output variables coincide with the names you use calling `matlabFunction`. For example, if you call `matlabFunction` with the variable *`F`*

```
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','testMatrix.m','vars',[x y t])
```

the generated name of an output variable is also *`F`*:

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

If you call `matlabFunction` using an expression instead of individual variables

```
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(w + z + F,'file','testMatrix.m',...
'vars',[x y t])
```

the default names of output variables consist of the word `out` followed by the number, for example:

```
function out1 = testMatrix(x,y,t)
...
```

To customize the names of output variables, use the `output` option:

```
syms x y z
r = x^2 + y^2 + z^2;
q = x^2 - y^2 - z^2;
f = matlabFunction(r, q, 'file', 'new_function',...
'outputs', {'name1','name2'})
```

The generated function returns *`name1`* and *`name2`* as results:

```
function [name1,name2] = new_function(x,y,z)
...
```

Converting MuPAD Expressions

You can convert a MuPAD expression or function to a MATLAB function:

```
syms x y;
f = evalin(symengine, 'arcsin(x) + arccos(y)');
matlabFunction(f, 'file', 'new_function');
```

The created file contains the same expressions written in the MATLAB language:

```
function f = new_function(x,y)
%NEW_FUNCTION
%    F = NEW_FUNCTION(X,Y)

f = asin(x) + acos(y);
```

Note `matlabFunction` cannot correctly convert some MuPAD expressions to MATLAB functions. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the results of conversion. To verify the results, execute the resulting function.

Generating Embedded MATLAB Function Blocks

Using `emlBlock`, you can generate an Embedded MATLAB® Function block. The generated block is available for use in Simulink® models, whether or not the computer running the simulations has a license for Symbolic Math Toolbox functions.

Generating and Editing a Block

Suppose, you want to create a model involving the van der Pol equation. Before you can convert a symbolic expression to an Embedded MATLAB Function block, create an empty model or open an existing one:

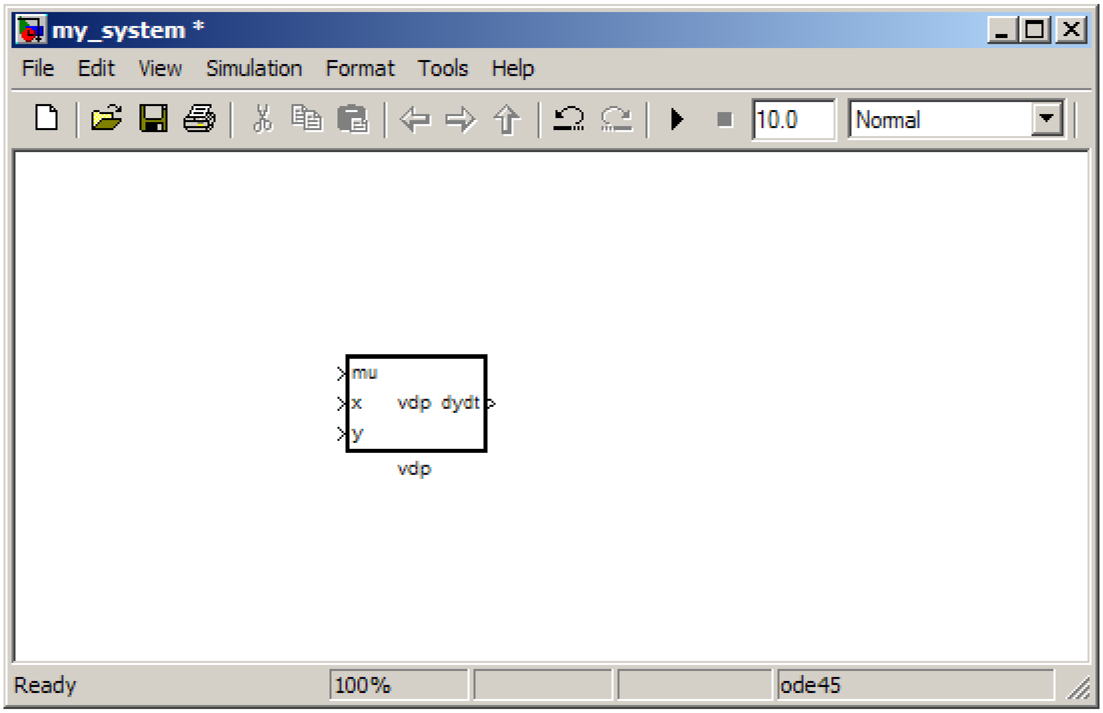
```
new_system('my_system');
open_system('my_system');
```

Create a symbolic expression and pass it to the `emlBlock` command. Also specify the block name:

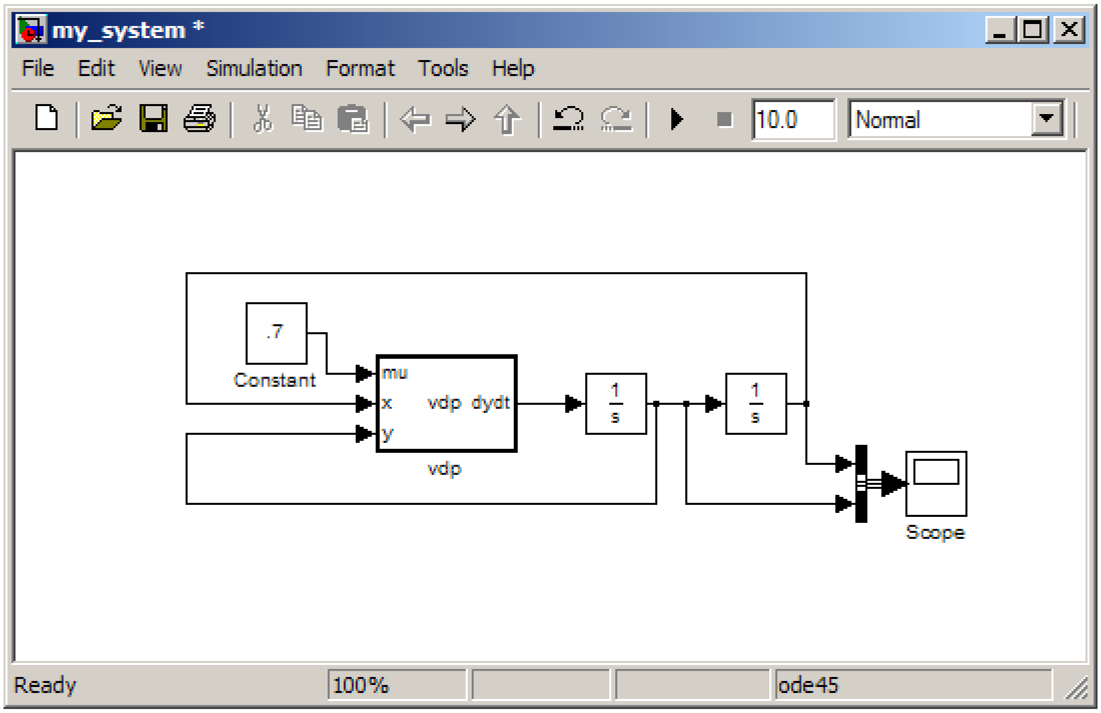
```
syms x y;
mu = sym('mu');
dydt = -x - mu*y*(x^2 - 1);
emlBlock('my_system/vdp', dydt);
```

If you use the name of an existing block, the `emlBlock` command replaces the definition of an existing block with the converted symbolic expression.

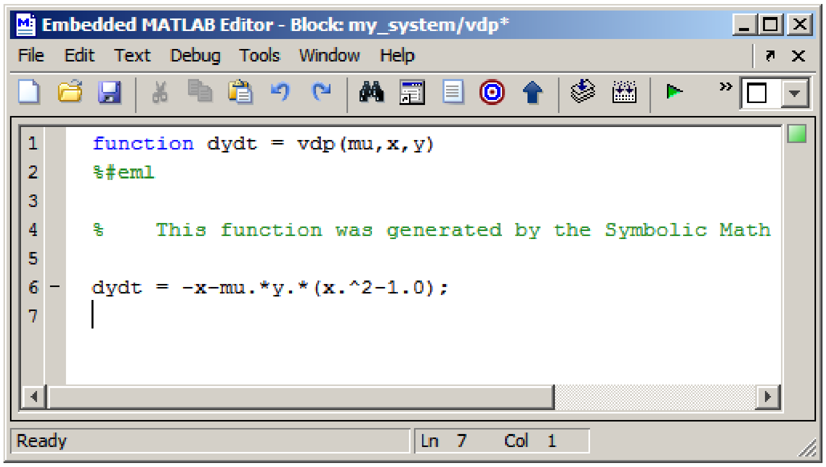
The model `my_system` contains the generated block.



Add other Simulink blocks and wiring to properly define the system.



You can open and edit the generated block. To open a block, select **Edit>Open Block** or use the context menu.


Embedded MATLAB Editor - Block: my_system/vdp*
File Edit Text Debug Tools Window Help
Ready
Ln 7 Col 1

```
function dydt = vdp(mu,x,y)
%#eml

%    This function was generated by the Symbolic Math

dydt = -x-mu.*y.*(x.^2-1.0);

```

Controlling the Order of Input Ports

`emlBlock` generates input variables and the corresponding input ports in alphabetical order from a symbolic expression. To change the order of input variables, use the `vars` option:

```
syms x y;
mu = sym('mu');
dydt = -x - mu*y*(x^2 - 1);
emlBlock('my_system/vdp', dydt,...
'vars', [y mu x]);
```

Naming the Output Ports

By default, `emlBlock` generates the names of the output ports as the word `out` followed by the output port number, for example, `out3`. The `output` option allows you to use the custom names of the output ports:

```
syms x y;
mu = sym('mu');
dydt = -x - mu*y*(x^2 - 1);
emlBlock('my_system/vdp', dydt,...
'outputs',{'name1'});
```

Converting MuPAD Expressions

You can convert a MuPAD expression or function to an Embedded MATLAB Function block:

```
syms x y;
f = evalin(symengine, 'arcsin(x) + arccos(y)');
emlBlock('my_system/my_block', f);
```

The resulting block contains the same expressions written in the MATLAB language:

```
function f = my_block(x,y)
%#eml

f = asin(x) + acos(y);
```

Note Some MuPAD expressions cannot be correctly converted to a block. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the results of conversion. To verify the results, you can:

- Run the simulation containing the resulting block
- Open the block and verify that all the functions are defined in the Embedded MATLAB Function Library

---

Generating Simscape Equations

Simscape™ software extends the Simulink product line with tools for modeling and simulating multidomain physical systems, such as those with mechanical, hydraulic, pneumatic, thermal, and electrical components. Unlike other Simulink blocks, which represent mathematical operations or operate on signals, Simscape blocks represent physical components or relationships directly. With Simscape blocks, you build a model of a system just as you would assemble a physical system. For more information about Simscape software see .

You can extend the Simscape modeling environment by creating custom components. When you define a component, use the equation section of the component file to establish the mathematical relationships among a component's variables, parameters, inputs, outputs, time, and the time derivatives of each of these entities. The Symbolic Math Toolbox and Simscape software let you perform symbolic computations and use the results of these computations in the equation section. The `simscapeEquation` function translates the results of symbolic computations to Simscape language equations.

Converting Algebraic and Differential Equations

Suppose, you want to generate a Simscape equation from the solution of the following ordinary differential equation. As a first step, use the `dsolve` function to solve the equation:

```
s = dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0');
s = simplify(s)
```

The solution is:

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

Then, use the `simscapeEquation` function to rewrite the solution in the Simscape language:

```
simscapeEquation(s)
```

`simscapeEquation` generates the following code :

```
ans =
s == cos(a*time)
```

The variable *time* replaces all instances of the variable *t* except for derivatives with respect to *t*. To use the generated equation, copy the equation and paste it to the equation section of the Simscape component file. Do not copy the automatically generated variable `ans` and the equal sign that follows it.

`simscapeEquation` converts any derivative with respect to the variable *t* to the Simscape notation, `X.der`, where `X` is the time-dependent variable. For example, convert the following differential equation to a Simscape equation. Also, here you explicitly specify the left and the right sides of the equation by using the syntax `simscapeEquation(LHS, RHS)`:

```
syms a;
x = sym('x(t)');
simscapeEquation(diff(x), -a^2*x)

ans =
x.der == -a^2*x
```

`simscapeEquation` also translates piecewise expressions to the Simscape language. For example, the result of the following Fourier transform is a piecewise function:

```
syms v u;
syms x real;
f = exp(-x^2*abs(v))*sin(v)/v;
s = fourier(f, v, u)

s =
piecewise([x <> 0, atan((u + 1)/x^2) - atan((u - 1)/x^2)])
```

From this symbolic piecewise equation, `simscapeEquation` generates valid code for the equation section of a Simscape component file:

```
simscapeEquation(s)

ans =
s == if x ~= 0, atan((u + 1)/x^2) - atan((u - 1)/x^2) end
```

Clear the assumption that *x* is real:

```
syms x clear
```

Converting MuPAD Equations

If you perform symbolic computations in the MuPAD Notebook Interface and want to convert the results to Simscape equations, use the `generate::Simscape` function in MuPAD.

Limitations

The equation section of a Simscape component file supports a limited number of functions. See the list of Supported Functions for more information. If a symbolic equation contains the functions that the equation section of a Simscape component file does not support. `simscapeEquation` cannot correctly convert these equations to Simscape equations. Such expressions do not trigger an error message. The following types of expressions are prone to invalid conversion:

- Expressions with infinities
- Expressions returned by `evalin` and `feval`.

MuPAD in Symbolic Math Toolbox

Understanding MuPAD

In this section...

Introduction to MuPAD

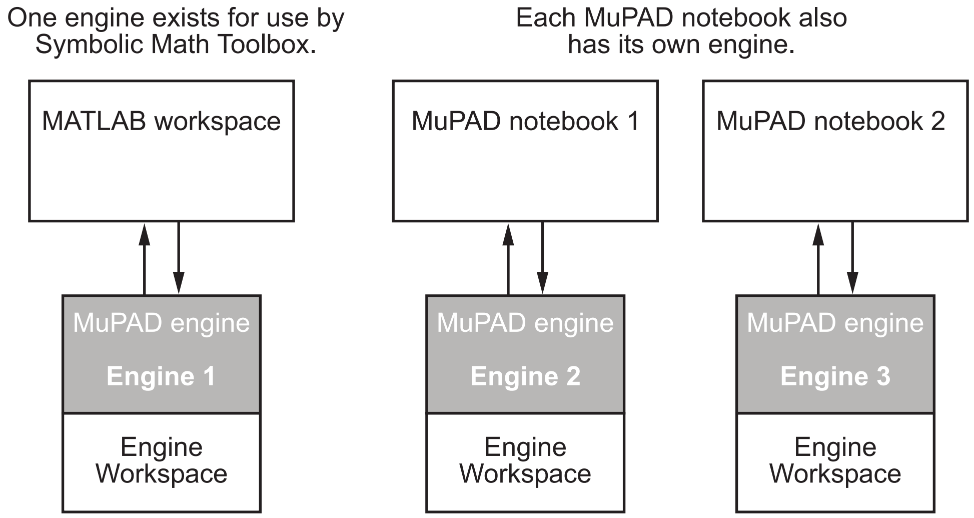
Version 5 of Symbolic Math Toolbox is powered by the MuPAD symbolic engine.

- 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 4-3 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 4-25.
- 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 4-28.

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.



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 4-33.

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 4-15.

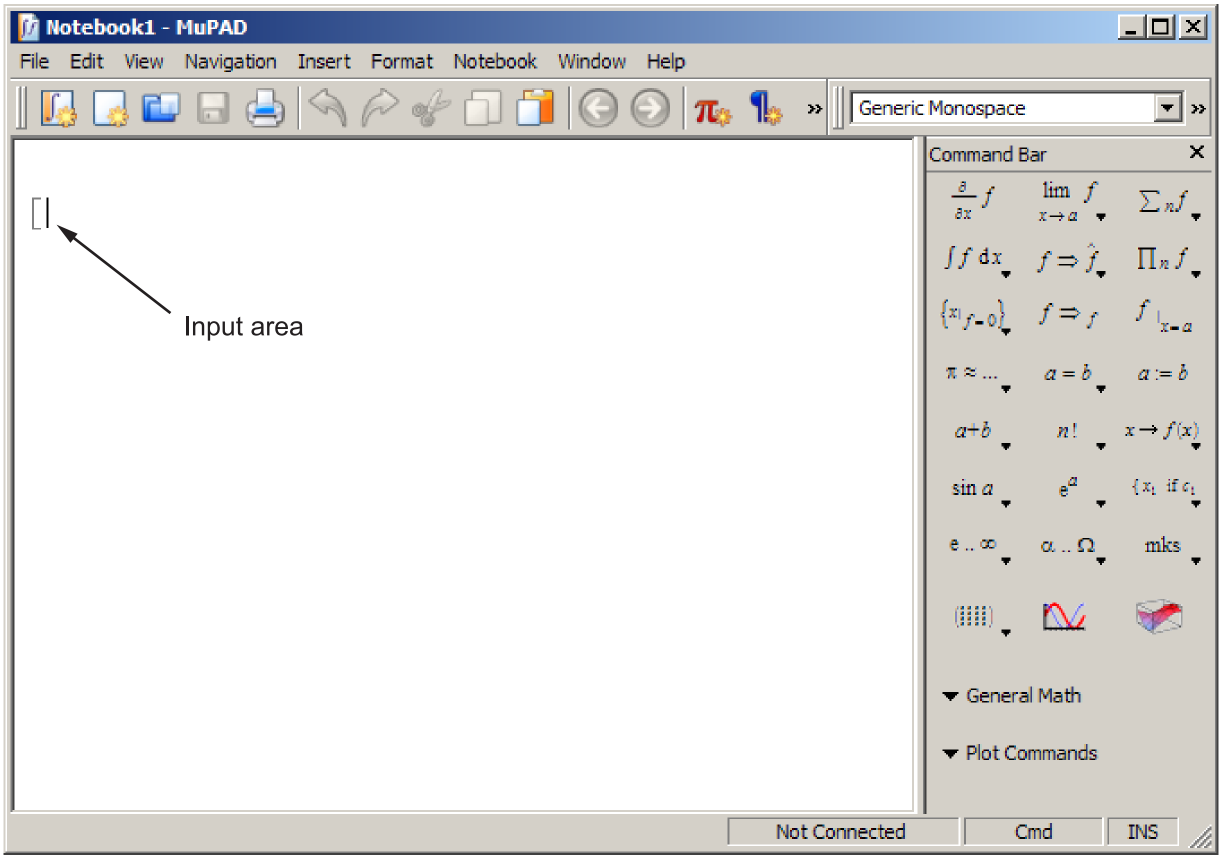
The density function of the normal and positive random variable is

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

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

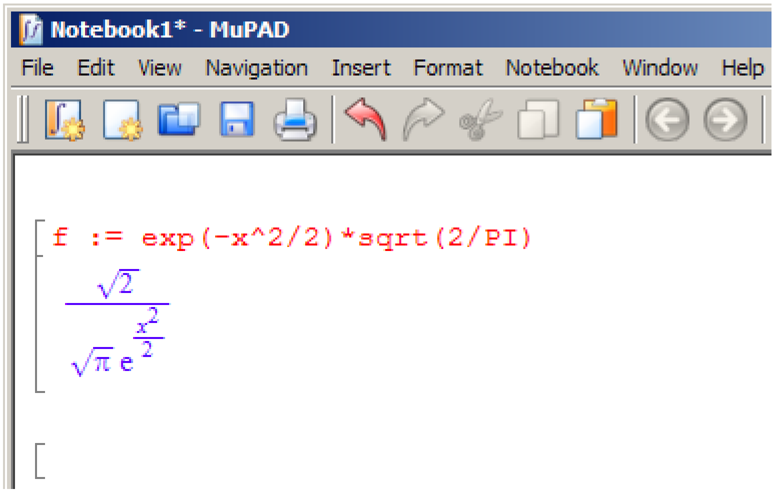
```
f := exp(-x^2/2)*sqrt(2/PI)
```

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 4-21.

The MuPAD notebook displays results in real math notation.

Your notebook appears as follows.


Notebook1* - MuPAD
File Edit View Navigation Insert Format Notebook Window Help

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

$$\frac{\sqrt{2}}{\sqrt{\pi}\, e^{\frac{x^2}{2}}}$$

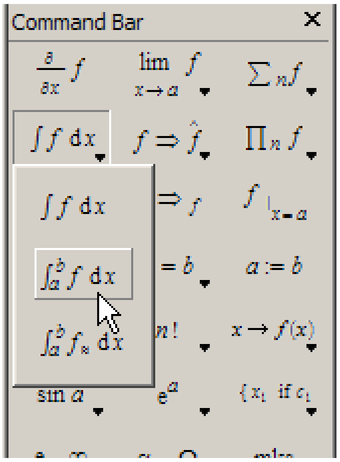
4 The mean of the random variable is

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

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 Use the **Tab** key to select the replaceable fields `#f`, `#x`, etc. Use **Ctrl+space** to autocomplete inputs. For example, type `infi` and press **Ctrl+space** to enter `infinity`.

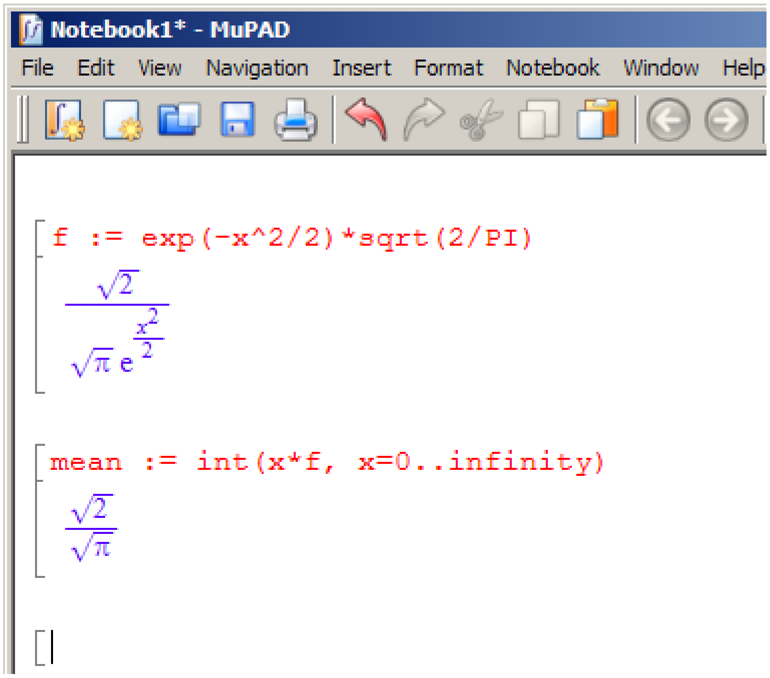
d Replace `#f` with `x*f`, `#x` with `x`, `#a` with `0`, and `#b` with `infinity`.

e 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.


Notebook1* - MuPAD
File Edit View Navigation Insert Format Notebook Window Help

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

$$\frac{\sqrt{2}}{\sqrt{\pi}\, e^{\frac{x^2}{2}}}$$

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

$$\frac{\sqrt{2}}{\sqrt{\pi}}$$

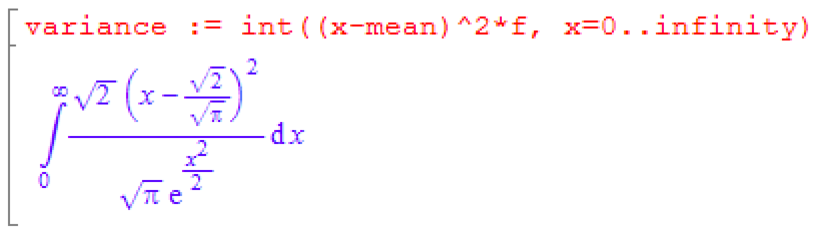
5 The variance of the random variable is

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

To calculate the variance of the random variable, type

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

and press **Enter**.



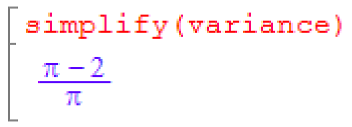
```
variance := int((x-mean)^2*f, x=0..infinity)
```

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

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)
```

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

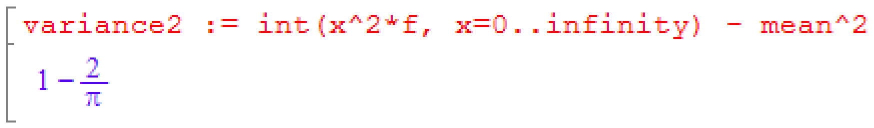
7 Another expression for the variance of the random variable is

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

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**.

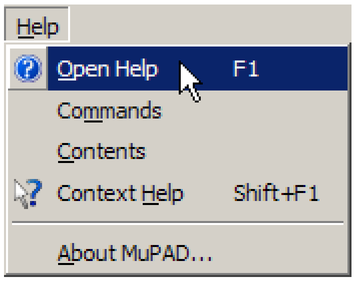


```
variance2 := int(x^2*f, x=0..infinity) - mean^2
```

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

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.

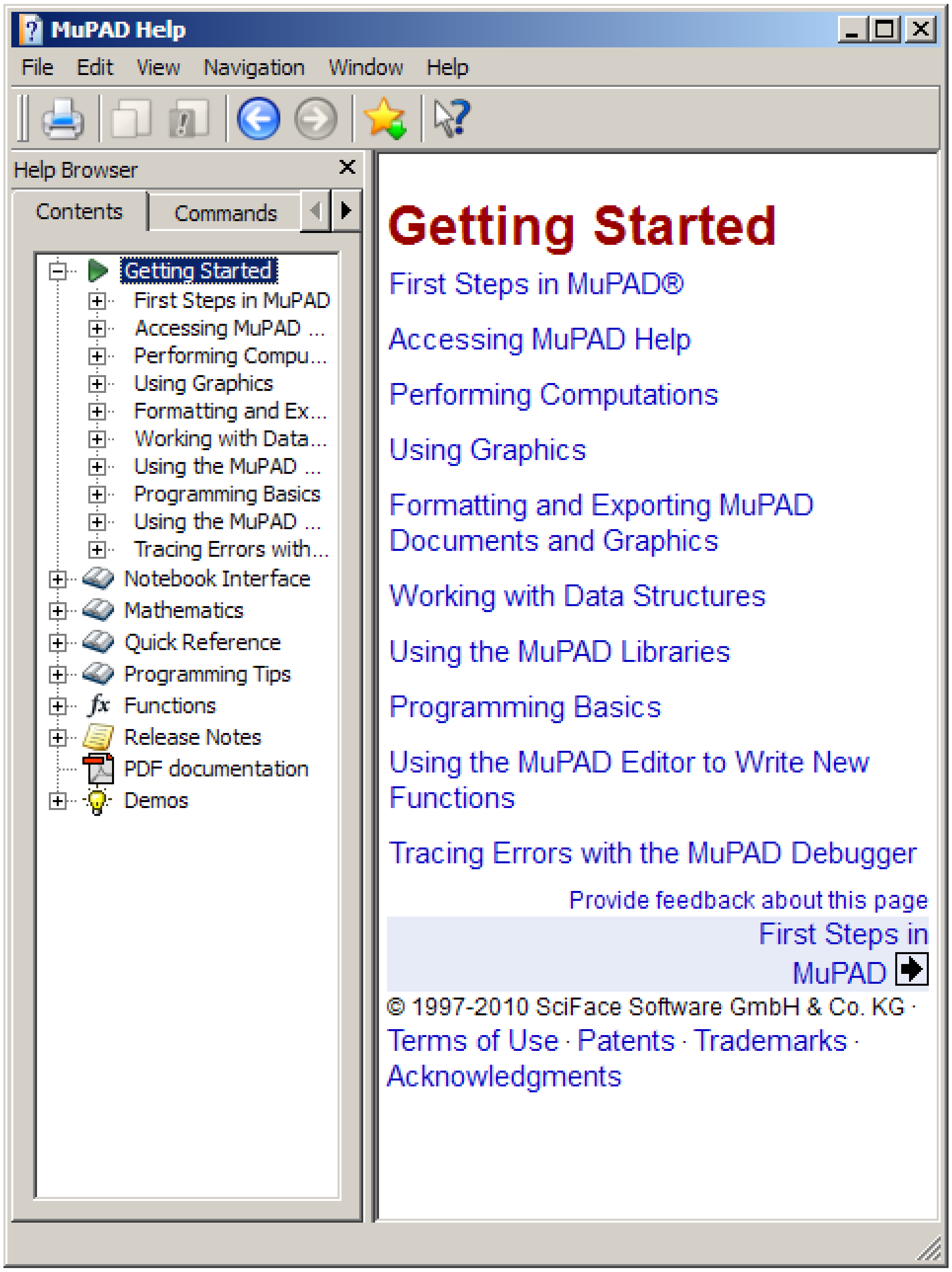
MuPAD for MATLAB Users

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.



Getting Started
First Steps in MuPAD
Accessing MuPAD ...
Performing Compu...
Using Graphics
Formatting and Ex...
Working with Data...
Using the MuPAD ...
Programming Basics
Using the MuPAD ...
Tracing Errors with...
Notebook Interface
Mathematics
Quick Reference
Programming Tips
Functions
Release Notes
PDF documentation
Demos

Getting Started

First Steps in MuPAD®

Accessing MuPAD Help

Performing Computations

Using Graphics

Formatting and Exporting MuPAD Documents and Graphics

Working with Data Structures

Using the MuPAD Libraries

Programming Basics

Using the MuPAD Editor to Write New Functions

Tracing Errors with the MuPAD Debugger

Provide feedback about this page

First Steps in MuPAD

© 1997-2010 SciFace Software GmbH & Co. KG · Terms of Use · Patents · Trademarks · Acknowledgments

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 syntax opens a blank MuPAD notebook.

The variable `nb` is a handle to the notebook. The toolbox uses this handle only for communication between the MATLAB workspace and the MuPAD notebook. Use handles as described in "Copying Variables and Expressions Between the MATLAB Workspace and MuPAD Notebooks" on page 4-25.

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

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

where *`file_name`* must be a full path unless the notebook is in the current folder. This command is useful in case you lose the handle to a notebook, in which case, you can save the notebook file and then 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, MuPAD does not save variables copied into a notebook using `setVar(nb,...)`.**

To open a notebook and automatically jump to a particular location, create a link target at that location inside a notebook and refer to it when opening a notebook. For information about creating link targets, see the "Formatting and Exporting MuPAD Documents and Graphics" section in the Getting Started chapter of the MuPAD documentation. To refer to a link target when opening a notebook, enter the command:

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

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.

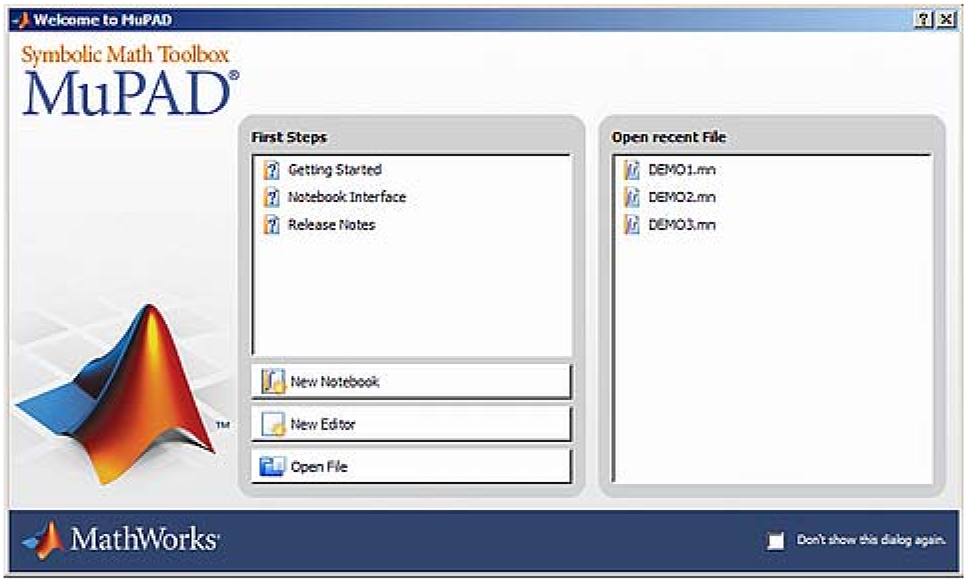
Note MuPAD notebook files open in an unevaluated state; in other words, the notebook is not synchronized with its engine when it opens. To synchronize a notebook with its engine, choose **Evaluate All** from the **Notebook** menu. For more information, see "Synchronizing a Notebook and its Engine" on page 4-19.

Opening Recent Files and Other MuPAD Interfaces

The command

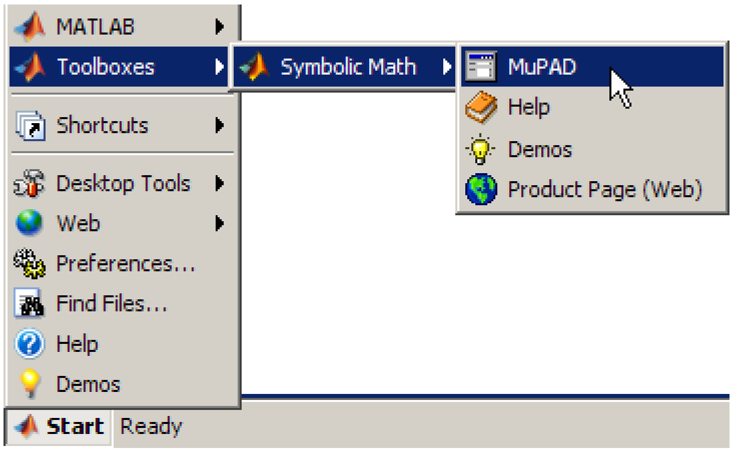
```
mupadwelcome
```

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.

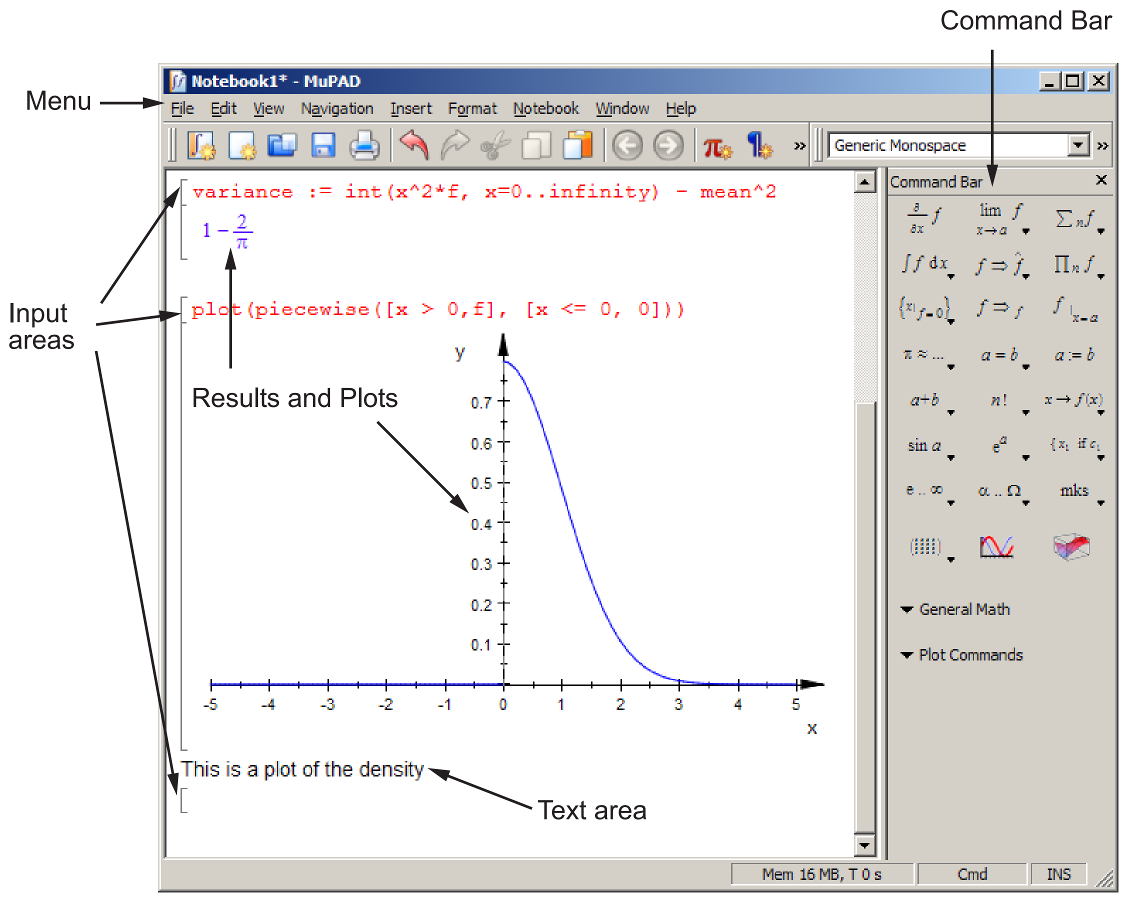
Alternatively, you can launch the `mupadwelcome` screen from the MATLAB **Start** menu as pictured.



Calculating in a MuPAD Notebook

The Visual Elements of a Notebook

A MuPAD notebook has the following main components.



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

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

```
plot(piecewise([x > 0,f], [x <= 0, 0]))
```

This is a plot of the density

- 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.

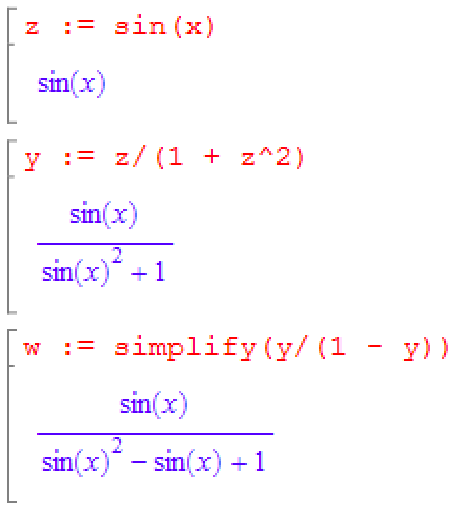
Working in a Notebook

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 4-17.
- 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)
```

$\sin(x)$

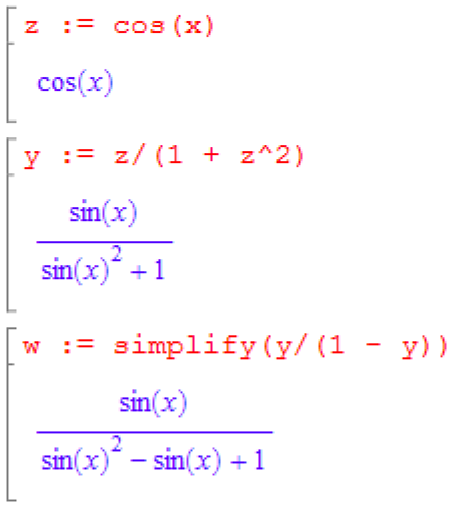
```
y := z/(1 + z^2)
```

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

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

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

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)
```

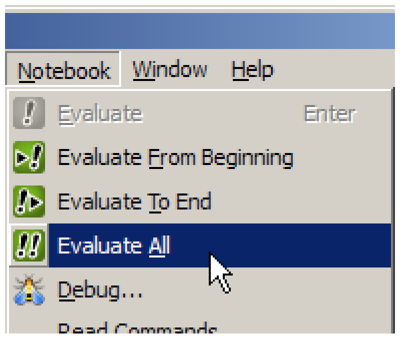
$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

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

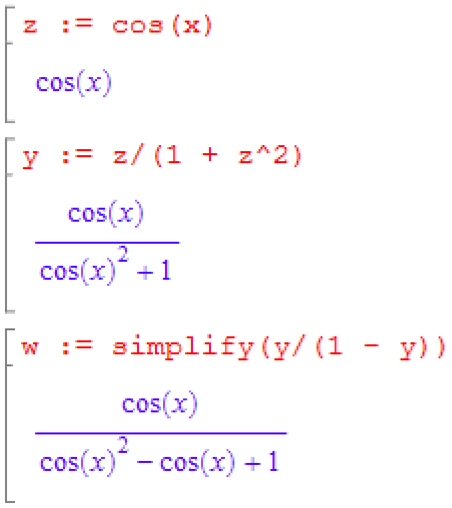
$$\frac{\sin(x)}{\sin(x)^2 - \sin(x) + 1}$$

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)
```

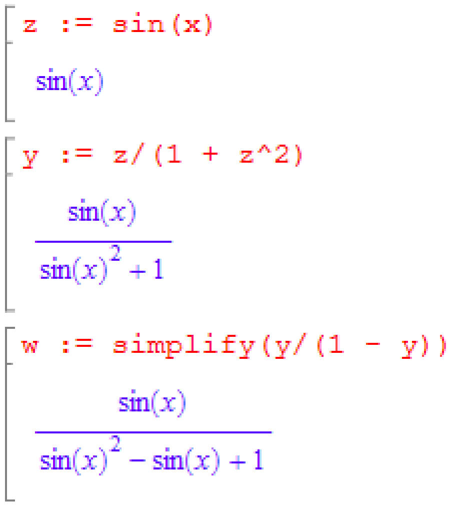
$$\frac{\cos(x)}{\cos(x)^2 + 1}$$

```
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 4-17:



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

$\sin(x)$

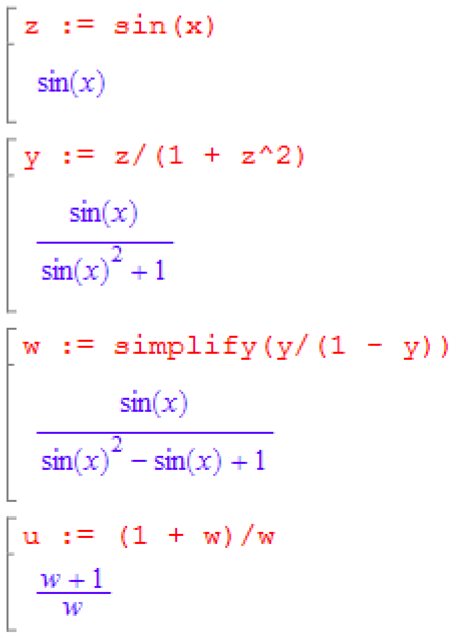
```
y := z/(1 + z^2)
```

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

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

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

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))
```

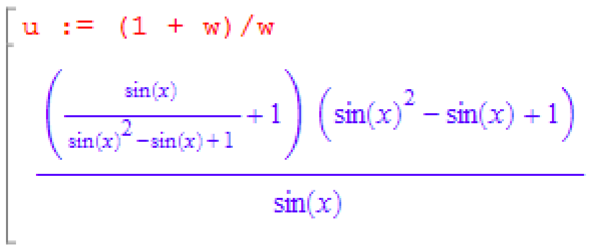
$$\frac{\sin(x)}{\sin(x)^2 - \sin(x) + 1}$$

```
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
```

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

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`

MATLAB vs. MuPAD Expressions (Continued)

MATLAB Expression	MuPAD Expression
`cosint`	`Ci`
`eulergamma`	`EULER`
`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	$F[f](w)=\int_{-\infty}^{\infty} f(x)e^{-iwx}dx$ `F = fourier(f)`	$F[f](w)=\int_{-\infty}^{\infty} f(x)e^{iwx}dx$ `F = transform::fourier(f,x,w)`
Inverse Fourier transform	$F^{-1}[f](x)=\frac{1}{2\pi}\int_{-\infty}^{\infty} f(w)e^{iwx}dw$ `Finv = ifourier(f)`	$F^{-1}[f](x)=\frac{1}{2\pi}\int_{-\infty}^{\infty} f(w)e^{-iwx}dw$ `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	expint(x) = –Ei(–x) = $\int_x^{\infty} \frac{\exp(-t)}{t} dt$ for $x > 0$ = Ei(1, x).	$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt$ for $x < 0$. $\mathrm{Ei}(n, x) = \int_1^{\infty} \frac{\exp(-xt)}{t^n} dt.$ 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);
```

Here *`file_name`* must be a full path unless the notebook is in the current folder. The *`nb`* 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

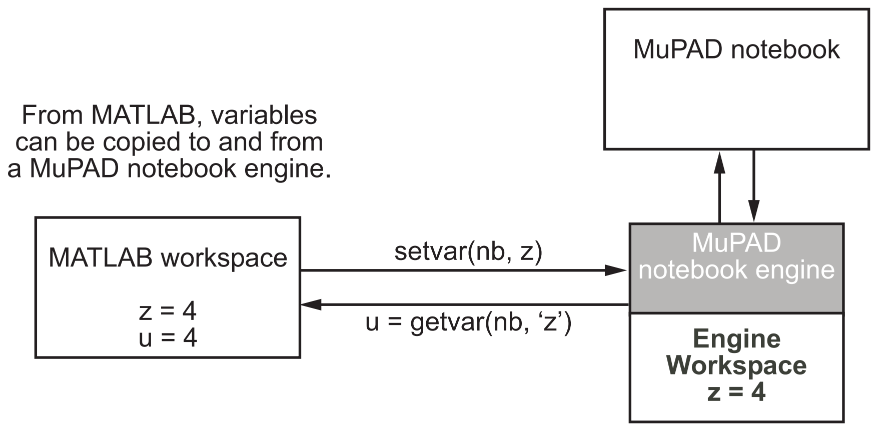
  ```
  MATLABvar = getVar(notebook_handle,'variable');
  ```

  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 4-19.

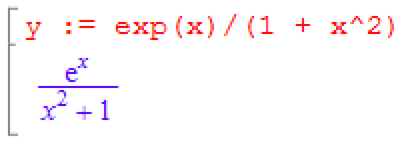
Note All such copying and assignment must be done from the MATLAB workspace, not from a MuPAD notebook.



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.

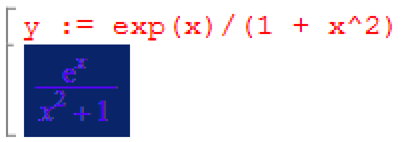
For example, consider the following MuPAD expression.



```
y := exp(x)/(1 + x^2)
```

$$\frac{e^x}{x^2+1}$$

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



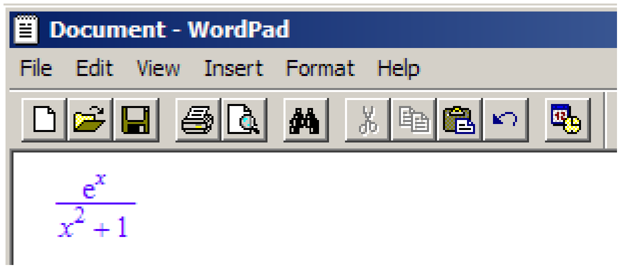
```
y := exp(x)/(1 + x^2)
```

$$\frac{e^x}{x^2+1}$$

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.



$$\frac{e^x}{x^2+1}$$

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`.

Note You cannot use `evalin` and `feval` to access the MuPAD function `log` that represents the logarithm to an arbitrary base. Instead, both commands evaluate the natural logarithm.

---

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.

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')
```

The result is

```
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)
```

The result is

```
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. The symbolic solver

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

returns

```
ans =
 0
 0
```

The numeric solver `fsolve`

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

returns

```
ans =
[x = 3.0699801238394654654386548746677946]
```

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)
```

Floating-Point Arguments of evalin and feval

By default, MuPAD performs all computations in an exact form. When you call the `evalin` or `feval` function with floating-point numbers as arguments, the toolbox converts these arguments to rational numbers before passing them to MuPAD. For example, when you calculate the incomplete gamma function, the result is the following symbolic expression:

```
y = feval(symengine,'igamma', 0.1, 2.5)

y =
igamma(1/10, 5/2)
```

To approximate the result numerically with double precision, use the `double` function:

```
format long;
double(y)

ans =
    0.028005841168289
```

Alternatively, use quotes to prevent the conversion of floating-point arguments to rational numbers. (The toolbox treats arguments enclosed in quotes as strings.) When MuPAD performs arithmetic operations on numbers involving at least one floating-point number, it automatically switches to numeric computations and returns a floating-point result:

```
feval(symengine,'igamma', '0.1', 2.5)

ans =
0.028005841168289177028337498391181
```

For further computations, set the format for displaying outputs back to `short`:

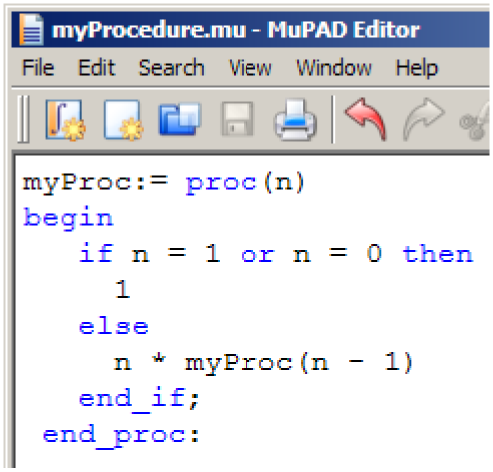
```
format short;
```

Calling Your Own MuPAD Procedures

You can extend the functionality available in the toolbox by writing your own procedures in the MuPAD language. This section explains how to call such procedures at the MATLAB command line. For information about creating

MuPAD procedures, see the "Programming Basics" section in the MuPAD Getting Started documentation.

Suppose, you wrote the `myProc` procedure that computes the factorial of a nonnegative integer.


myProcedure.mu - MuPAD Editor
File Edit Search View Window Help

```
myProc:= proc(n)
begin
   if n = 1 or n = 0 then
     1
   else
     n * myProc(n - 1)
   end_if;
 end_proc:
```

Save the procedure as a file with the extension `.mu`. For example, save the procedure as `myProcedure.mu` in the folder `C:/MuPAD`.

Now, switch to the MATLAB Command Window. Before calling the procedure at the MATLAB command line, enter:

```
eng=symengine;
eng.feval('read',' "C:/MuPAD/myProcedure.mu" ');
```

The `read` command reads and executes the `myProcedure.mu` file in MuPAD. After that, you can call the `myProc` procedure with any valid parameter. For example, compute the factorial of 15:

```
eng.feval('myProc', 15)

ans =
1307674368000
```

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 4-35.

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 4-34 and "Examples of the Effect of Assumptions" on page 4-35.

- 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.

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 x clear

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 4-28. 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, {}])
```

Function Reference

Calculus

`diff`	Differentiate symbolic expression
`int`	Integrate symbolic expression
`jacobian`	Compute Jacobian matrix
`limit`	Compute limit of symbolic expression
`symsum`	Evaluate symbolic sum of series
`taylor`	Taylor series expansion

Linear Algebra

`colspace`	Return basis for column space of matrix
`det`	Compute determinant of symbolic matrix
`diag`	Create or extract diagonals of symbolic matrices
`eig`	Compute symbolic eigenvalues and eigenvectors
`expm`	Compute symbolic matrix exponential
`inv`	Compute symbolic matrix inverse
`jordan`	Compute Jordan canonical form of matrix
`null`	Form basis for null space of matrix
`poly`	Compute characteristic polynomial of matrix
`rank`	Compute rank of symbolic matrix
`rref`	Compute reduced row echelon form of matrix

`svd`	Compute singular value decomposition of symbolic matrix
`tril`	Return lower triangular part of symbolic matrix
`triu`	Return upper triangular part of symbolic matrix

Simplification

`coeffs`	List 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
`gamma`	Gamma function
`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`	Compute Riemann zeta function

MuPAD

`clear all`	Remove items from MATLAB workspace and reset MuPAD engine
`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
`openmuphlp`	Open MuPAD help file
`openxvc`	Open MuPAD XVC graphics file
`openxvz`	Open MuPAD XVZ graphics file
`reset`	Close MuPAD engine
`setVar`	Assign variable in MuPAD notebook
`symengine`	Return symbolic engine
`trace`	Enable and disable tracing of MuPAD commands

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
`char`	Convert symbolic objects to strings
`double`	Convert symbolic matrix to MATLAB numeric form
`emlBlock`	Convert symbolic expression to Embedded MATLAB Function block
`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
`simscapeEquation`	Convert symbolic expressions to Simscape language equations
`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`	Perform symbolic equality test
`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 complex symbolic number
`round`	Symbolic matrix elementwise round
`size`	Symbolic matrix dimensions
`sort`	Sort symbolic vectors, matrices, or polynomials
`sym`	Define symbolic 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. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`.

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','ccodetest');
```

return a file named `ccodetest` containing the following:

```
  t2 = exp(-x);
  t3 = exp(-t2);
  t0 = t3*exp(x*(-2.0))*(-3.0)+t3*exp(x*(-3.0))+t2*t3;
```

See Also

`fortran`, `latex`, `matlabFunction`, `pretty`

"Generating Code from Symbolic Expressions" on page 3-145

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)]


ans =
[ -2, -3, -3, -2, -1/2]
```

See Also

`round`, `floor`, `fix`, `frac`

Purpose

Convert symbolic objects to strings

Syntax

```
char(A)
char(A, d)
```

Description

`char(A)` converts a symbolic scalar or a symbolic array to a string.

`char(A, d)` converts a symbolic scalar or array to a string. For symbolic arrays, the second parameter specifies the form of the resulting string. For symbolic scalars, this parameter does not affect the result.

Input Arguments

`A`

A symbolic scalar or a symbolic array

`d`

A number that specifies the format of the resulting string. For symbolic arrays:

`char (A, 1)` results in `matrix([...])`

`char(A, 2)` results in `matrix([[...],[...]])`

`char(A, d)` for all other values of the parameter `d` results in `array([1..m, 1..n, 1..p], [(1,1,1) = xxx,...,(m,n,p ) = xxx])`

Examples

Convert symbolic expressions to strings, and then concatenate the strings:

```
syms x;
y = char(x^3 + x^2 + 2*x - 1);
name = [y, ' presents a polynomial expression']
```

The result is:

```
name =
x^3 + x^2 + 2*x - 1 presents a polynomial expression
```

Convert a symbolic matrix to a string:

```
A = sym(hilb(3))
char(A)
```

The result is:

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

ans =
matrix([[1,1/2,1/3],[1/2,1/3,1/4],[1/3,1/4,1/5]])
```

See Also

`sym` | `double` | `pretty`

clear all

Purpose Remove items from MATLAB workspace and reset MuPAD engine

Syntax `clear all`

Description `clear all` clears all objects in the MATLAB workspace and closes the MuPAD engine associated with the MATLAB workspace resetting all its assumptions.

See Also `reset`

Purpose

List 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)` returns a list of the coefficients and a list of the terms of `p`. There is a one-to-one correspondence between the coefficients and the terms of `p`.

Examples

List the coefficients of the following single-variable polynomial:

```
syms x
t = 16*log(x)^2 + 19*log(x) + 11;
coeffs(t)
```

The result is:

```
ans =
[ 11, 19, 16]
```

List the coefficients of the following polynomial with respect to the indeterminate `sin(x)`:

```
syms a b c x
y = a + b*sin(x) + c*sin(2*x);
coeffs(y, sin(x))
```

The result is:

```
ans =
[ a + c*sin(2*x), b]
```

List the coefficients of the following multivariable polynomial with respect to all the indeterminates and with respect to the variable x only:

```
syms x y
z = 3*x^2*y^2 + 5*x*y^3;
coeffs(z)
coeffs(z,x)
```

The results are:

```
ans =
[ 5, 3]

ans =
[ 5*y^3, 3*y^2]
```

Display the list of the coefficients and the list of the terms of the following polynomial expression:

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

The results are:

```
c =
[ 5*x, 3*x^2]

t =
[ y^3, y^2]
```

See Also

`sym2poly`

Purpose

Collect coefficients

Syntax

```
R = collect(S)
R = collect(S,v)
```

Description

`R = collect(S)` returns an array of collected polynomials for each polynomial in the array `S` of polynomials.

`R = collect(S,v)` collects terms containing the variable `v`.

Examples

The following statements

```
syms x y;
R1 = collect((exp(x)+x)*(x+2))
R2 = collect((x+y)*(x^2+y^2+1), y)
R3 = collect([(x+1)*(y+1),x+y])
```

return

```
R1 =
x^2 + (exp(x) + 2)*x + 2*exp(x)

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

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

See Also

`expand`, `factor`, `simple`, `simplify`, `syms`

colspace

Purpose

Return basis for column space of matrix

Syntax

```
B = colspace(A)
```

Description

`B = colspace(A)` returns a matrix whose columns form a basis for the column space of `A`. The matrix `A` is symbolic or numeric.

Examples

Find the basis for the column space of the following matrix:

```
A = sym([2,0;3,4;0,5])
B = colspace(A)
```

The result is:

```
A =
[ 2, 0]
[ 3, 4]
[ 0, 5]

B =
[     1,   0]
[     0,   1]
[ -15/8, 5/4]
```

See Also

`null` | `size`

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 `symvar` and `y` is the symbolic variable of `g` as defined by `symvar`.

`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 `symvar`.

`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

```
a = compose(f,g)
b = compose(f,g,t)
c = compose(h,g,x,z)
d = compose(h,g,t,z)
e = compose(h,p,x,y,z)
f = compose(h,p,t,u,z)
```

returns:

```
a =
1/(sin(y)^2 + 1)

b =
1/(sin(t)^2 + 1)

c =
sin(z)^t

d =
x^sin(z)

e =
(1/exp(z/u))^t

f =
x^(1/exp(y/z))
```

See Also `finverse`, `subs`, `syms`

Purpose

Symbolic complex conjugate

Syntax

```
conj(X)
```

Description

`conj(X)` is the complex conjugate of `X`.

For a complex `X`, `conj(X) = real(X) - i*imag(X)`.

See Also

`real`, `imag`

Purpose

Cosine integral

Syntax

```
Y = cosint(X)
```

Description

`Y = cosint(X)` evaluates the cosine integral function at the elements of `X`, a numeric matrix, or a symbolic matrix. The cosine integral function is defined by

$$Ci(x) = \gamma + \ln(x) + \int_0^x \frac{\cos t - 1}{t} dt,$$

where γ is Euler's constant 0.577215664...

Examples

Compute cosine integral for a numerical value:

```
cosint(7.2)
```

The result is:

```
0.0960
```

Compute the cosine integral for `[0:0.1:1]` :

```
cosint([0:0.1:1])
```

The result is:

```
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`

Purpose

Compute determinant of symbolic matrix

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 for a symbolic `A` and a numeric value for a numeric `A`.

Examples

Compute the determinant of the following symbolic matrix:

```
syms a b c d;
det([a, b; c, d])
```

The result is:

```
ans =
a*d - b*c
```

Compute the determinant of the following matrix containing the symbolic numbers:

```
A = sym([2/3 1/3; 1 1])
r = det(A)
```

The result is:

```
A =
[ 2/3, 1/3]
[   1,   1]

r =
1/3
```

See Also

`rank` | `eig`

Purpose

Create or extract diagonals of symbolic matrices

Syntax

```
diag(A, k)
diag(A)
```

Description

`diag(A, k)` returns a square symbolic matrix of order `n + abs(k)`, with the elements of `A` on the *k*-th diagonal. `A` must present a row or column vector with `n` components. The value `k = 0` signifies the main diagonal. The value `k > 0` signifies the *k*-th diagonal above the main diagonal. The value `k < 0` signifies the *k*-th diagonal below the main diagonal. If `A` is a square symbolic matrix, `diag(A, k)` 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. If `A` is a square symbolic matrix, `diag(A)` returns the main diagonal of `A`.

Examples

Create a symbolic matrix with the main diagonal presented by the elements of the vector `v`:

```
syms a b c;
v = [a b c];
diag(v)
```

The result is:

```
ans =
[ a, 0, 0]
[ 0, b, 0]
[ 0, 0, c]
```

Create a symbolic matrix with the second diagonal below the main one presented by the elements of the vector `v`:

```
syms a b c;
```

```
v = [a b c];
diag(v, -2)
```

The result is:

```
ans =
[ 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]
```

Extract the main diagonal from a square matrix:

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

The result is

```
ans =
 a
 2
 z
```

Extract the first diagonal above the main one:

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

The result is:

```
ans =
 b
 3
```

See Also

`tril` | `triu`

diff

Purpose

Differentiate symbolic expression

Syntax

```
diff(expr)
diff(expr, v)
diff(expr, sym('v'))
diff(expr, n)
diff(expr, v, n)
diff(expr, n, v)
```

Description

`diff(expr)` differentiates a symbolic expression `expr` with respect to its free variable as determined by `symvar`.

`diff(expr, v)` and `diff(expr, sym('v'))` differentiate `expr` with respect to `v`.

`diff(expr, n)` differentiates `expr` *n* times. *n* is a positive integer.

`diff(expr, v, n)` and `diff(expr, n, v)` differentiate `expr` with respect to `v` `n` times.

Examples

Differentiate the following single-variable expression one time:

```
syms x;
diff(sin(x^2))
```

The result is

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

Differentiate the following single-variable expression six times:

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

The result is

```
ans =
720
```

Differentiate the following expression with respect to `t`:

```
syms x t;
diff(sin(x*t^2), t)
```

The result is

```
ans =
2*t*x*cos(t^2*x)
```

See Also

`int` | `jacobian` | `symvar`

digits

Purpose

Variable precision accuracy

Syntax

```
digits
digits(d)
d = digits
```

Description

`digits` specifies the minimum number of significant (nonzero) 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 at least `d` significant (nonzero) decimal digits. The value `d` must be a positive integer larger than 1 and smaller than $2^{29}+1$.

`d = digits` returns the current VPA accuracy.

Examples

The `digits` function specifies the number of significant (nonzero) digits. For example, use 4 significant digits to compute the ratio 1/3 and the ratio 1/3000:

```
old = digits;
digits(4);
vpa(1/3)
vpa(1/3000)
digits(old);

ans =
0.3333

ans =
0.0003333
```

To change the VPA accuracy for one operation without changing the current `digits` setting, use the `vpa` function. For example, compute the ratio 1/3 with the default 32 digits, 10 digits, and 40 digits:

```
vpa(1/3)
vpa(1/3, 10)
vpa(1/3, 40)

ans =
 0.33333333333333333333333333333333

ans =
0.3333333333

ans =
0.3333333333333333333333333333333333333333
```

The number of digits that you specify by the `vpa` function or the `digits` function is the minimal number of digits. Internally, the toolbox can use more digits than you specify. These additional digits are called guard digits. For example, set the number of digits to 4, and then display the floating-point approximation of 1/3 using 4 digits:

```
old = digits;
digits(4);
a = vpa(1/3)

a =
0.3333
```

Now, display `a` using 20 digits. The result shows that the toolbox internally used more than 4 digits when computing `a`. The last digits in the following result are incorrect because of the round-off error:

```
digits(20);
vpa(a)
digits(old);

ans =
0.33333333333303016843
```

Hidden round-off errors can cause unexpected results. For example, compute the number 1/10 with the default 32 digits accuracy and with the 10 digits accuracy:

```
a = vpa(1/10)
old = digits;
digits(10);
b = vpa(1/10)
digits(old);

a =
0.1

b =
0.1
```

Now, compute the difference `a - b`. The result is not zero:

```
a - b

ans =
0.0000000000000000000086736173798840354720600815844403
```

The difference `a - b` is not equal to zero because the toolbox approximates the number `b=0.1` with 32 digits. This approximation produces round-off errors because the floating-point number 0.1 is different from the rational number 1/10. When you compute the difference `a - b`, the toolbox actually computes the difference as follows:

```
b = vpa(b)
a - b

b =
0.09999999999999999999132638262011 6

ans =
0.0000000000000000000086736173798840354720600815844403
```

Suppose, you convert a number to a symbolic object, and then perform VPA operations on that object. The results can depend on the conversion technique that you used to convert a floating-point number to a symbolic object. The `sym` function lets you choose the conversion technique by specifying the optional second argument, which can be 'r', 'f', 'd' or 'e'. The default is 'r'. For example, convert the constant π=3.141592653589793... to a symbolic object:

```
r = sym(pi)
f = sym(pi, 'f')
d = sym(pi, 'd')
e = sym(pi, 'e')

r =
pi

f =
884279719003555/281474976710656

d =
3.1415926535897931159979634685442

e =
pi - (198*eps)/359
```

Set the number of digits to 4. Three of the four numeric approximations give the same result:

```
digits(4);
vpa(r)
vpa(f)
vpa(d)
vpa(e)

ans =
3.142
```

```
ans =
3.142

ans =
3.142

ans =
3.142 - 0.5515*eps
```

Now, set the number of digits to 40. The numeric approximation of 1/10 depends on the technique that you used to convert 1/10 to the symbolic object:

```
digits(40);
vpa(r)
vpa(f)
vpa(d)
vpa(e)

ans =
3.141592653589793238462643383279502884197

ans =
3.141592653589793115997963468544185161591

ans =
3.1415926535897931159979634685442

ans =
3.141592653589793238462643383279502884197 -...
0.5515320334261838440111420612813370473538*eps
```

See Also

`double`, `vpa`

Purpose

Dirac delta

Syntax

`dirac(x)`

Description

`dirac(x)` returns the Dirac delta function of `x`.

The Dirac delta function, `dirac`, has the value 0 for all `x` not equal to 0 and the value `Inf` for `x` = 0. Several Symbolic Math Toolbox functions return answers in terms of `dirac`.

Example

`dirac` has the property that

$$\int_{-\infty}^{\infty} dirac(x-a) * f(x) = f(a)$$

for any function `f` and real number `a`. For example:

```
syms x a
a = 5;
int(dirac(x-a)*sin(x),-inf, inf)

ans =
sin(5)
```

`dirac` also has the following relationship to the function `heaviside`:

```
syms x;
diff(heaviside(x),x)


ans =
dirac(x)
```

See Also

`heaviside`

doc

Purpose

Get help for MuPAD functions

Syntax

```
doc(symengine)
doc(symengine,'MuPAD_function_name')
```

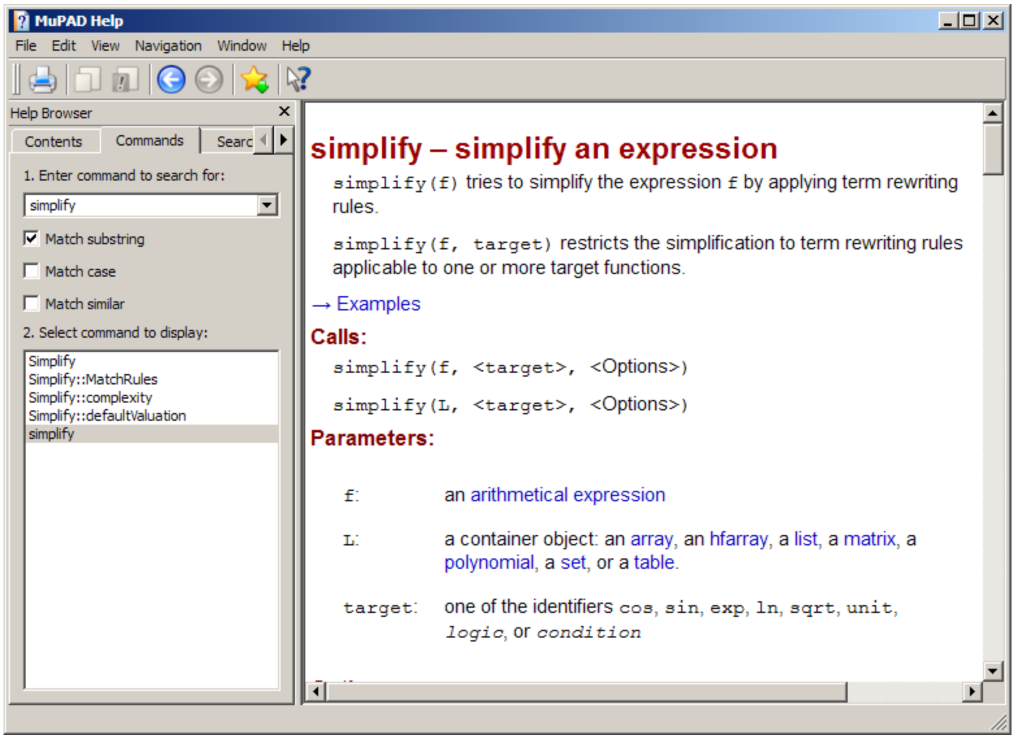
Description

`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.



simplify – simplify an expression

`simplify(f)` tries to simplify the expression `f` by applying term rewriting rules.

`simplify(f, target)` restricts the simplification to term rewriting rules applicable to one or more target functions.

→ Examples

Calls:

`simplify(f, <target>, <Options>)`

`simplify(L, <target>, <Options>)`

Parameters:

`f`:	an arithmetical expression
`L`:	a container object: an array, an hfarray, a list, a matrix, a polynomial, a set, or a table.
`target`:	one of the identifiers `cos`, `sin`, `exp`, `ln`, `sqrt`, `unit`, *`logic`*, or *`condition`*

See Also

“Getting Help for MuPAD” on page 4-10

double

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

Find the numeric value for the expression $\frac{1+\sqrt{5}}{2}$:

```
double(sym('(1+sqrt(5))/2'))
```

The result is:

```
1.6180
```

The following statements

```
a = sym(2*sqrt(2));
b = sym((1-sqrt(3))^2);
T = [a, b];
double(T)
```

return

```
ans =
    2.8284    0.5359
```

See Also

`sym`, `vpa`

Purpose

Symbolic solution of ordinary differential equations

Syntax

```
dsolve('eq1','eq2',...,'cond1','cond2',...,'v')
dsolve(...,'IgnoreAnalyticConstraints',value)
```

Description

`dsolve('eq1','eq2',...,'cond1','cond2',...,'v')` symbolically solves the ordinary differential equations `eq1, eq2,...` using `v` as the independent variable. Here `cond1,cond2,...` specify boundary or initial conditions or both. You also can use the following syntax: `dsolve('eq1, eq2',...,'cond1,cond2',...,'v')`. The default independent variable is `t`.

The letter `D` denotes differentiation with respect to the independent variable. The primary default is `d/dx`. The letter `D` followed by a digit denotes repeated differentiation. For example, `D2` is d^2/dx^2. Any character immediately following a differentiation operator is a dependent variable. For example, `D3y` denotes the third derivative of `y(x)` or `y(t)`.

You can specify initial and boundary conditions by 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 the specified initial conditions is less than the number of dependent variables, the resulting solutions contain the arbitrary constants `C1`, `C2,...`.

You can input each equation or a condition as a separate symbolic equation. The `dsolve` command accepts up to 12 input arguments.

`dsolve` can produce the following three types of outputs:

- 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 alphabetically 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 is an equivalent lower order differential equation or an integral.

`dsolve(...,'IgnoreAnalyticConstraints',value)` accepts the following values:

- `value = 'all'` applies the purely algebraic simplifications to the expressions on both sides of equations. These simplifications might not be generally valid. The default value of this option is `all`.
- `value = 'none'` solves ordinary differential equations without additional assumptions. The results obtained with this option are correct for all values of the arguments.

Note By default, the solver does not guarantee general correctness and completeness of the results. If you do not set the option `IgnoreAnalyticConstraints` to `none`, always verify results returned by the `dsolve` command.

If you do not set the value of the option `IgnoreAnalyticConstraints` to `none`, the solver applies the following rules to the expressions on both sides of an equation:

- The solutions of polynomial equations must be complete.
- $\ln(a)+\ln(b)=\ln(a\cdot b)$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$(a\cdot b)^c = a^c\cdot b^c$$

- $\ln\left(a^b\right) = b \cdot \ln a$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$\left(a^b\right)^c = a^{b \cdot c}$$

- The following equality is valid for all values of x:
  - $\ln\left(e^x\right) = x$
  - $\arcsin\left(\sin(x)\right) = x$, $\arccos\left(\cos(x)\right) = x$, $\arctan\left(\tan(x)\right) = x$
  - $\operatorname{arcsinh}\left(\sinh(x)\right)=x$, $\operatorname{arccosh}\left(\cosh(x)\right)=x$, $\operatorname{arctanh}\left(\tanh(x)\right)=x$
  - $W_k\left(x \cdot e^x\right) = x$ for all values of k
- The solver can multiply both sides of an equation by any expression except `0`.

Examples

Solving Ordinary Differential Equations Symbolically

```
dsolve('Dx = -a*x')

ans =
C2/exp(a*t)
```

Specifying the Dependent Variable

The following differential equation presents `f` as a dependent variable:

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

ans =
C4*exp(t) - sin(t)/2 - cos(t)/2
```

Specifying the Independent Variable

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

ans =
              1
             -1
 cosh(C7 + s*i)
 cosh(C11 - s*i)
```

Setting Initial and Boundary Conditions

```
dsolve('Dy = a*y', 'y(0) = b')

ans =
b*exp(a*t)

dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0')

ans =
(1/exp(a*t*i))/2 + exp(a*t*i)/2
```

Solving a System of Differential Equations

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

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

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

```
z.x

ans =
C20*cos(t) + C19*sin(t)


z.y
```

```
ans =
C19*cos(t) - C20*sin(t)
```

Using the IgnoreAnalyticConstraints Option

By default, the solver applies the set of purely algebraic simplifications that are not correct in general, but that can result in simple and practical solutions:

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

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

To obtain complete and generally correct solutions, set the value of the option `IgnoreAnalyticConstraints` to `none`:

```
y = dsolve('Dy=1+y^2','y(0)=1',...
'IgnoreAnalyticConstraints','none')

y =
piecewise([C29 in Z_, tan(pi/4 + t + pi*C29)])
```

The algebraic simplifications also allow you to obtain solutions for the equations that the solver cannot compute when it uses strict mathematical rules:

```
dsolve('Dv=19.6*v-0.00196*v^2','v(0)=1')

ans =
 10000/(exp(log(9999) - (98*t)/5) + 1)
```

versus

```
dsolve('Dv=19.6*v-0.00196*v^2','v(0)=1',...
'IgnoreAnalyticConstraints','none')

Warning: Possibly spurious solutions
[solvelib::checkSolutions]
ans =
piecewise([C38 in Z_, 1/(exp(log(9999) - (98*t)/5 +...
2*pi*C38*i)/10000 + 1/10000)])
```

Diagnostics

If `dsolve` cannot find an analytic solution for an equation, it prints the warning:

```
Warning: Explicit solution could not be found.
```

and returns an empty `sym` object.

See Also

`syms`

Purpose

Compute 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 matrices V and D. The columns of `V` present eigenvectors of `A`. The diagonal matrix `D` contains eigenvalues. If the resulting `V` has the same size as `A`, the matrix `A` has a full set of linearly independent eigenvectors that satisfy `A*V = V*D`.

`[V,D,P] = eig(A)` returns a vector of indices `P`. The length of `P` equals to the total number of linearly independent eigenvectors, so that `A*V = V*D(P,P)`.

`lambda = eig(vpa(A))` returns numeric eigenvalues using variable precision arithmetic.

`[V,D] = eig(vpa(A))` returns numeric eigenvectors using variable precision arithmetic. If `A` does not have a full set of eigenvectors, the columns of `V` are not linearly independent.

Examples

Compute the eigenvalues for the magic square of order 5:

```
M = sym(magic(5));
eig(M)
```

The result is:

```
ans =
                                    65
  (625/2 - (5*3145^(1/2))/2)^(1/2)
  ((5*3145^(1/2))/2 + 625/2)^(1/2)
```

```
   -(625/2 - (5*3145^(1/2))/2)^(1/2)
   -((5*3145^(1/2))/2 + 625/2)^(1/2)
```

Compute the eigenvalues for the magic square of order 5 using variable precision arithmetic:

```
M = sym(magic(5));
eig(vpa(M))
```

The result is:

```
ans =
                                     65.0
 21.27676547147379553062642669797423
 13.12628093070921880252564308594914
  -13.1262809307092188025256430859494
  -21.2767654714737955306264266979742
```

Compute the eigenvalues and eigenvectors for one of the MATLAB test matrices:

```
A = sym(gallery(5))
[v, lambda] = eig(A)
```

The results are:

```
A =
[    -9,     11,    -21,      63,    -252]
[    70,    -69,    141,    -421,    1684]
[  -575,    575,  -1149,    3451,  -13801]
[  3891,  -3891,   7782,  -23345,   93365]
[  1024,  -1024,   2048,   -6144,   24572]

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

Convert symbolic expression to Embedded MATLAB Function block

Syntax

```
emlBlock(block, f)
emlBlock(block, f1, f2, ...)
emlBlock(block, f1, f2, ..., param1, value1,...)
```

Description

`emlBlock(block, f)` converts the symbolic expression `f` to an Embedded MATLAB Function block that you can use in Simulink models. The parameter `block` specifies the name of the block you create or modify. The `block` should be a string.

`emlBlock(block, f1, f2, ...)` converts a list of the symbolic expressions `f1`, `f2`, `...` to an Embedded MATLAB Function block with multiple outputs.

`emlBlock(block, f1, f2, ..., param1, value1,...)` converts a list of the symbolic expressions `f1`, `f2`, `...` to an Embedded MATLAB Function block with multiple outputs, with the following options for parameter/value pairs:

- `Parameter` = `'functionName'` allows you to set the name of the function. `value` should be a string. By default, `value` coincides with the name of the block.
- `Parameter` = `'outputs'` allows you to set the names of the output ports. `value` should be a cell array of strings. The number of `value` entries should equal or exceed the number of free variables in the symbolic expression `f`. The default name of an output port consists of the word `out` followed by the output port number, for example, `out3`.
- `Parameter` = `'vars'` allows you to set the order of the variables and the corresponding input ports of a block. The default order is alphabetical. `value` should be either a cell array of strings or symbolic arrays, or a vector of symbolic variables. The number of `value` entries should equal or exceed the number of free variables in the symbolic expression `f`.

Note To convert a MuPAD expression or function to an Embedded MATLAB Function block, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine, 'MuPAD_Function',x1,...,xn)`. `emlBlock` cannot correctly convert some MuPAD expressions to a block. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the results of conversion. To verify the results, you can:

- Run the simulation containing the resulting block.
- Open the block and verify that all the functions are defined in the Embedded MATLAB Function Library.

Examples

Before you can convert a symbolic expression to an Embedded MATLAB Function block, create an empty model or open an existing one:

```
new_system('new_system');
open_system('new_system');
```

Use `emlBlock` to create the block *`new_block`* containing the symbolic expression:

```
syms x y z
f = x^2 + y^2 +z^2;
emlBlock('new_system/new_block',f);
```

If you use the name of an existing block, the `emlBlock` command replaces the definition of an existing block with the converted symbolic expression.

You can open and edit the resulting block. To open a block, select **Edit>Open Block** or use the context menu:

```
function f = new_block(x,y,z)
%#eml

f = x.^2 + y.^2 + z.^2;
```

The following example generates a block and sets the function name to *new_function*:

```
emlBlock('new_system/new_block', x, y, z,...
'functionName', 'new_function')
```

You can change the order of the input ports:

```
emlBlock('new_system/new_block', x, y, z,...
'vars', [y z x])
```

Also, you can rename the output variables and the corresponding ports:

```
emlBlock('new_system/new_block', x, y, z,...
'outputs',{'name1','name2','name3'})
```

`emlBlock` accepts several options simultaneously:

```
emlBlock('new_system/new_block', x, y, z,...
'functionName', 'new_function','vars', [y z x],...
'outputs',{'name1','name2','name3'})
```

You also can convert MuPAD expressions:

```
syms x y;
f = evalin(symengine, 'arcsin(x) + arccos(y)');
emlBlock('new_system/new_block', f);
```

The resulting block contains the same expressions written in the MATLAB language:

```
function f = new_block(x,y)
%#eml
```

```
f = asin(x) + acos(y);
```

See Also

`ccode`, `fortran`, `matlabFunction`, `subs`, `sym2poly`

"Generating Code from Symbolic Expressions" on page 3-145

Purpose

Perform symbolic equality test

Syntax

```
eq(A, B)
A == B
```

Description

`eq(A, B)` compares each element of `A` for equality with the corresponding element of `B`. If the elements are not equal or if either element is undefined, the test fails. `eq` does not expand or simplify expressions before making the comparison.

`A == B` is the alternate syntax for `eq(A, B)`.

Examples

Check equality of two symbolic matrices:

```
A = sym(hilb(10));
B = sym([1/11 1/12 1/13 1/14 1/15 1/16]);
eq(A(9, 3:8), B)
```

The result is:

```
ans =
     1     1     1     1     1     1
```

Check the trigonometric identity:

```
syms x;
sin(x)^2 + cos(x)^2 == 1
```

The symbolic equality test might fail to recognize mathematical equivalence of polynomial or trigonometric expressions because it does not simplify or expand them. The result is:

```
ans =
     0
```

When testing mathematical equivalence of such expressions, simplify the difference between the expressions, and then compare the result with 0:

```
syms x;
simplify(sin(x)^2 + cos(x)^2 - 1) == 0
```

The result is:

```
ans =
     1
```

See Also

simplify

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

Compute the discriminant of the following polynomial:

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

The result is:

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

Do not use `evalin` to access the MuPAD `log` function that represents the logarithm to an arbitrary base. The `evalin` command evaluates `log` as the natural logarithm (the appropriate MuPAD function is `ln`):

```
evalin(symengine,'log(E)')

ans =
1
```

Evaluating `log` with two parameters results in the following error:

```
evalin(symengine,'log(10, 10)')

??? Error using ==> mupadengine.mupadengine>mupadengine.evalin at 124
Error: wrong no of args [ln]
```

See Also

`doc`, `feval`

"Calling MuPAD Functions at the MATLAB Command Line" on page 4-28

expm

Purpose

Compute symbolic matrix exponential

Syntax

```
expm(A)
```

Description

`expm(A)` computes the matrix exponential of the symbolic matrix `A`.

Examples

Compute the matrix exponential for the following matrix and simplify the result:

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

The result is:

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

See Also

`eig`

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

Expand the expression:

```
syms x;
expand((x-2)*(x-4))
```

The result is:

```
ans =
x^2 - 6*x + 8
```

Expand the trigonometric expression:

```
syms x y;
expand(cos(x+y))
```

The result is:

```
ans =
cos(x)*cos(y) - sin(x)*sin(y)
```

Expand the exponent:

```
syms a b;
expand(exp((a+b)^2))
```

The result is:

```
ans =
exp(2*a*b)*exp(a^2)*exp(b^2)
```

Expand the expressions that form a vector:

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

The result is:

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

See Also

`collect`, `factor`, `horner`, `simple`, `simplify`, `syms`

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.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

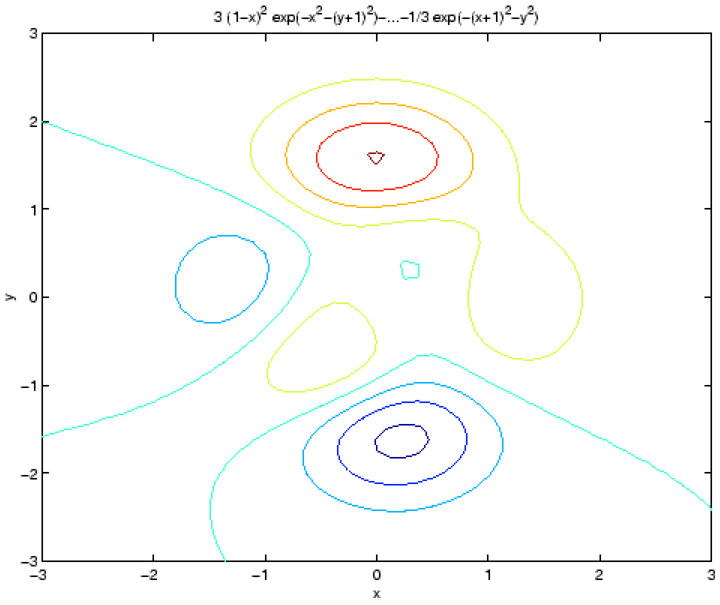
`ezcontour` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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`

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.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

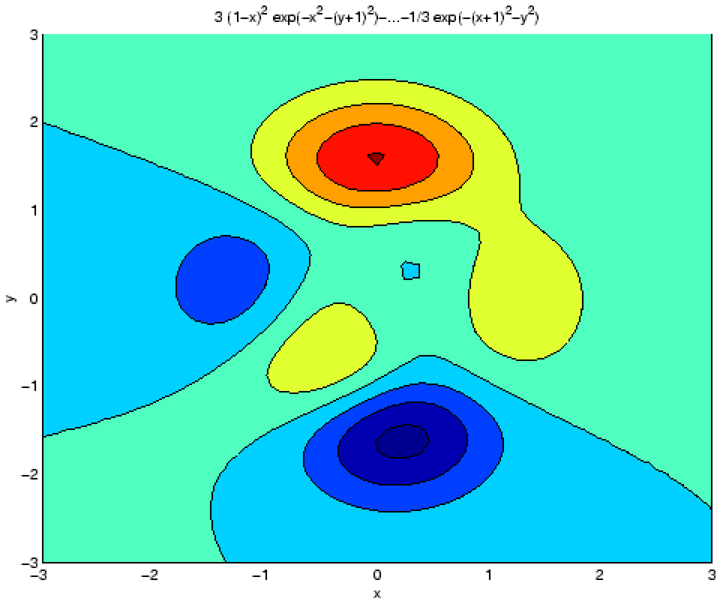
`ezcontourf` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB abbreviates the string.

See Also

`contourf`, `ezcontour`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`

Purpose

3-D mesh plotter

Syntax

```
ezmesh(f)
ezmesh(f, domain)
ezmesh(x,y,z)
ezmesh(x,y,z,[smin,smax,tmin,tmax])
ezmesh(x,y,z,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
```

Description

`ezmesh(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB 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.

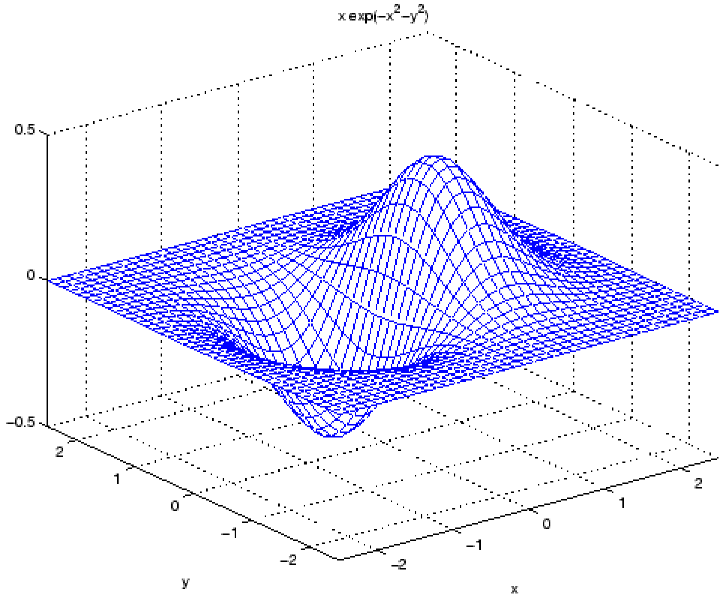
Examples

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`

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

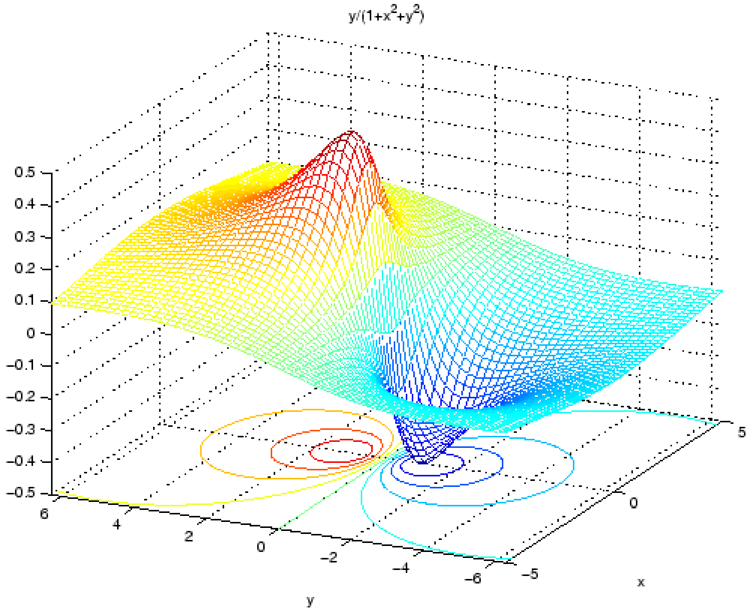
Create a mesh/contour graph of the expression,

$$f(x, y) = \frac{y}{1 + x^2 + y^2},$$

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`

Purpose

Function plotter

Syntax

```
ezplot(f)
ezplot(f,[xmin xmax])
ezplot(f,[xmin xmax], fign)
ezplot(f,[xmin, xmax, ymin, ymax])
ezplot(x,y)
ezplot(x,y,[tmin,tmax])
ezplot(...,figure)
```

Description

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

`ezplot(f,[xmin xmax])` plots $f = f(x)$ over the specified domain. It opens and displays the result in a window labeled **Figure No. 1**. If any plot windows are already open, `ezplot` displays the result in the highest numbered window.

`ezplot(f,[xmin xmax], fign)` opens (if necessary) and displays the plot in the window labeled `fign`.

For implicitly defined functions, $f = f(x,y)$.

`ezplot(f)` plots $f(x,y) = 0$ over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`ezplot(f,[xmin, xmax, ymin, ymax])` plots $f(x,y) = 0$ over `xmin` $< x <$ `xmax` and `ymin` $< y <$ `ymax`.

`ezplot(f,[min,max])`plots $f(x,y) = 0$ over `min` $< x <$ `max` and `min` $< y <$ `max`.

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezplot(u^2 - v^2 - 1,[-3,2,-2,3])` plots $u^2 - v^2 - 1 = 0$ over $-3 < u < 2$, $-2 < v < 3$.

`ezplot(x,y)` plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default domain $0 < t < 2\pi$.

`ezplot(x,y,[tmin,tmax])` plots $x = x(t)$ and $y = y(t)$ over `tmin` $< t$ $<$ `tmax`.

`ezplot(...,figure)` plots the given function over the specified domain in the figure window identified by the handle `figure`.

Algorithm

If you do not specify a plot range, `ezplot` samples the function between `-2*pi` and `2*pi` and selects a subinterval where the variation is significant as the plot domain. For the range, `ezplot` omits extreme values associated with singularities.

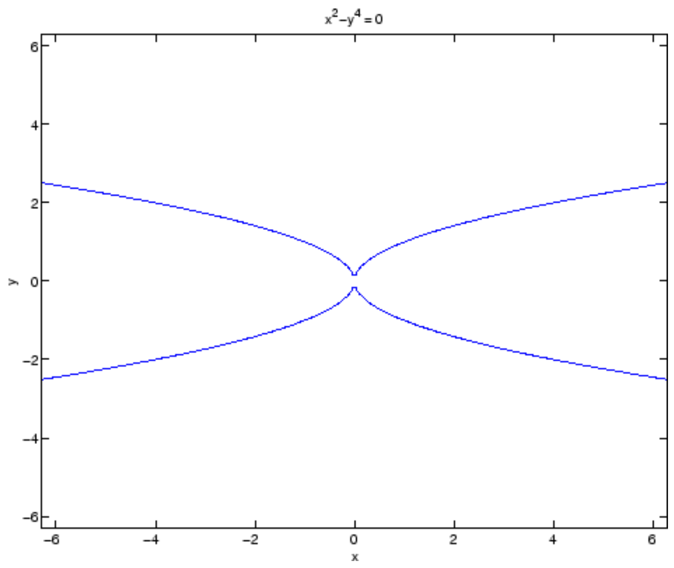
Examples

This example plots the implicitly defined function,

$$x^2 - y^4 = 0$$

over the domain $[-2\pi, 2\pi]$.

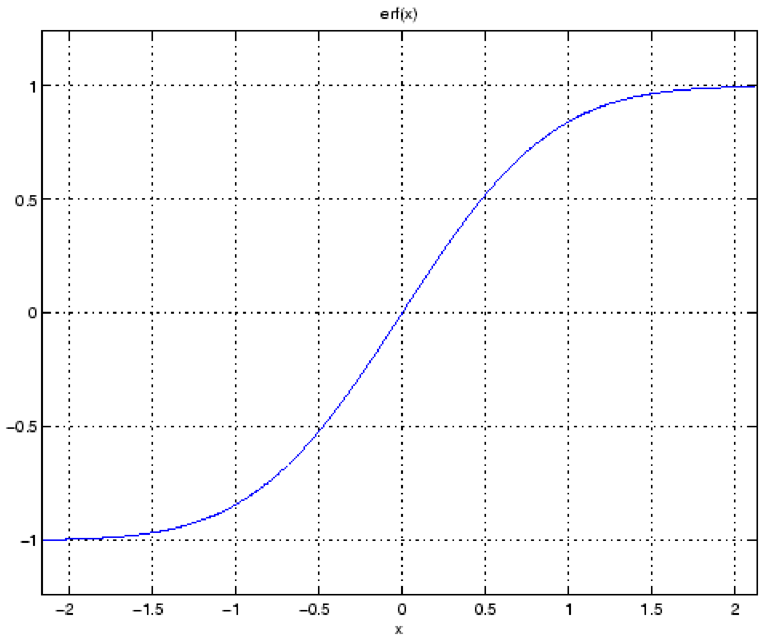
```
syms x y
ezplot(x^2-y^4)
```



The following statements

```
syms x
ezplot(erf(x))
grid
```

plot a graph of the error function.



See Also `ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`, `plot`

Purpose

3-D parametric curve plotter

Syntax

```
ezplot3(x,y,z)
ezplot3(x,y,z,[tmin,tmax])
ezplot3(...,'animate')
```

Description

`ezplot3(x,y,z)` plots the spatial curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the default domain $0 < t < 2\pi$.

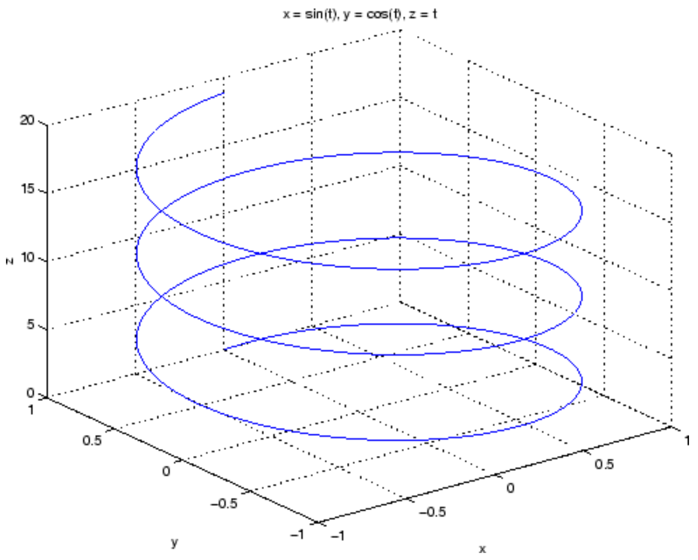
`ezplot3(x,y,z,[tmin,tmax])` plots the curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the domain $tmin < t < tmax$.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

Examples

This example plots the parametric curve, $x = sin(t)$, $y = cos(t)$, $z = t$ over the domain $[0, 6\pi]$.

```
syms t;
ezplot3(sin(t), cos(t), t,[0,6*pi])
```



See Also ezcontour, ezcontourf, ezmesh, ezmeshc, ezplot, ezpolar, ezsurf, ezsurfc, plot3

Purpose

Polar coordinate plotter

Syntax

```
ezpolar(f)
ezpolar(f, [a, b])
```

Description

`ezpolar(f)` plots the polar curve $r = f(\theta)$ over the default domain $0 < \theta < 2\pi$.

`ezpolar(f, [a, b])` plots f for `a` $< \theta <$ `b`.

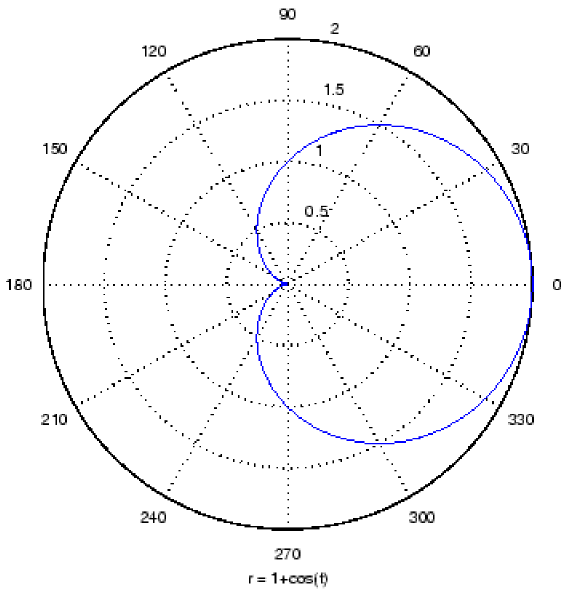
Example

This example creates a polar plot of the function,

```
1 + cos(t)
```

over the domain $[0, 2\pi]$.

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


90
2
120
60
1.5
150
30
1
0.5
180
0
210
330
240
300
270
r = 1+cos(t)

Purpose

3-D colored surface plotter

Syntax

```
ezsurf(f)
ezsurf(f,domain)
ezsurf(x,y,z)
ezsurf(x,y,z,[smin,smax,tmin,tmax])
ezsurf(x,y,z,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
```

`ezsurf(f)` plots over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB 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.

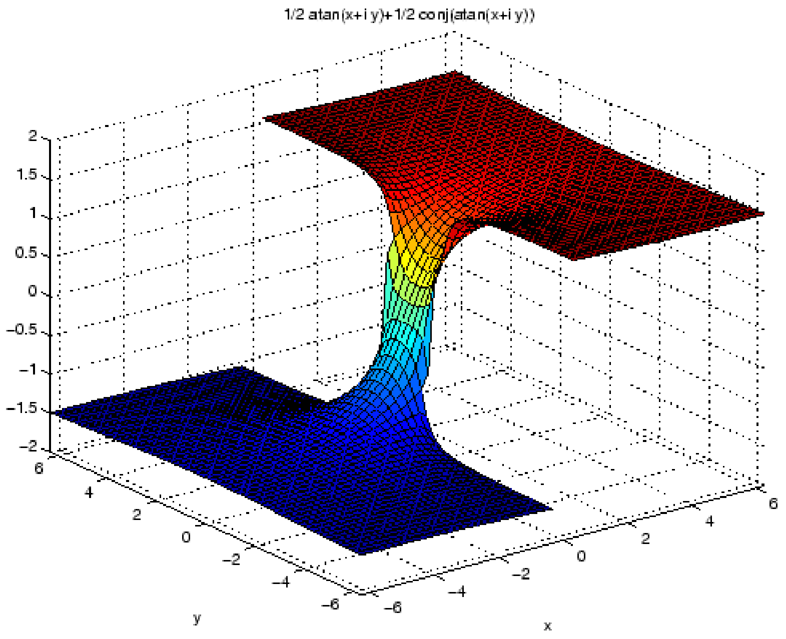
Examples

`ezsurf` does not graph points where the mathematical function is not defined (these data points are set to `NaN`s, which MATLAB does not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function,

$$f(x,y) = real(atan(x + iy))$$

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

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



Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

See Also

`ezcontour`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot`, `ezpolar`, `ezsurfc`, `surf`

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

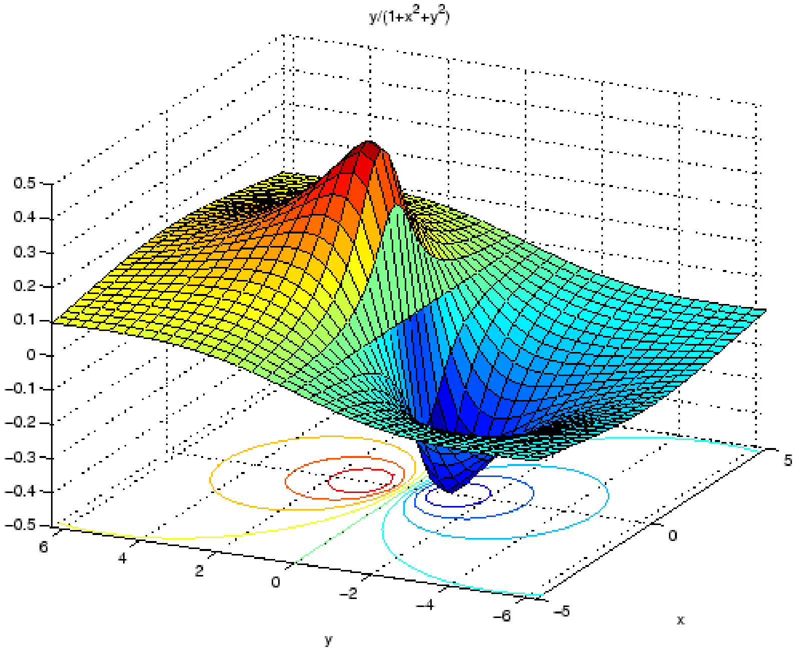
Create a surface/contour plot of the expression,

$$f(x, y) = \frac{y}{1 + x^2 + y^2},$$

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

Factorize the two-variable expression:

```
syms x y;
factor(x^3-y^3)
```

The result is:

```
ans =
(x - y)*(x^2 + x*y + y^2)
```

Factorize the expressions that form a vector:

```
syms a b;
factor([a^2 - b^2, a^3 + b^3])
```

The result is:

```
ans =
[ (a - b)*(a + b), (a + b)*(a^2 - a*b + b^2)]
```

Factorize the symbolic number:

```
factor(sym('12345678901234567890'))
```

The result is:

```
ans =
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 `result` if `status` is nonzero. If `status` is 0, `result` is a symbolic object. Otherwise, `result` 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
```

Do not use `feval` to access the MuPAD `log` function that represents the logarithm to an arbitrary base. The `feval` command evaluates `log` as the natural logarithm (the appropriate MuPAD function is `ln`):

```
feval(symengine,'log', 'E')

ans =
1
```

Evaluating `log` with two parameters results in the following error:

```
feval(symengine,'log', '10', '10')
```

```
??? Error using ==> mupadengine.mupadengine>mupadengine.feval at 163
Error: wrong no of args [ln]
```

See Also

`doc`, `evalin`

"Calling MuPAD Functions at the MATLAB Command Line" on page 4-28

findsym

Purpose

Determine variables in symbolic expression or matrix

Note `findsym` is not recommended. Use `symvar` instead.

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:

1. The variables are sorted by the first letters in their names. The ordering is x y w z v u ... a X Y W Z V U ... A. The name of a symbolic variable cannot begin with a number.
2. For all subsequent letters the ordering is alphabetical, with all uppercase letters having precedence over lowercase: 0 1 ... 9 A B ... Z a b ...z.

`findsym(S)` can return variables in different order than`findsym(S, n)`.

See Also

`symvar`

finverse

Purpose Functional inverse

Syntax

```
g = finverse(f)
g = finverse(f,v)
```

Description `g = finverse(f)` returns the functional inverse of `f`. `f` is a scalar `sym` representing a function of one symbolic variable, say `x`. Then `g` is a scalar `sym` that satisfies `g(f(x)) = x`. That is, `finverse(f)` returns f^{-1}, provided f^{-1} exists.

`g = finverse(f,v)` uses the symbolic variable `v`, where `v` is a `sym`, as the independent variable. Then `g` is a scalar `sym` that satisfies `g(f(v)) = v`. Use this form when `f` contains more than one symbolic variable.

Examples Compute functional inverse for the trigonometric function:

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

The result is:

```
ans =
atan(1/x)
```

Compute functional inverse for the exponent function:

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

The result is:

```
ans =
2*v + log(u)
```

See Also `compose`, `syms`

Purpose Round toward zero

Syntax `fix(X)`

Description `fix(X)` is the matrix of the integer parts of `X`.

`fix(X) = floor(X)` if `X` is positive and `ceil(X)` if `X` is negative.

See Also `round`, `ceil`, `floor`, `frac`

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)]

ans =
[ -2, -3, -3, -2, -1/2]
```

See Also `round`, `ceil`, `fix`, `frac`

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. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`.

Examples

The statements

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

return

```
ans =
      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

```
ans =
      H(1,1) = 1.0D0
      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','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 3-145

fourier

Purpose

Fourier integral transform

Syntax

```
F = fourier(f)
F = fourier(f,v)
F = fourier(f,u,v)
```

Description

`F = fourier(f)` is the Fourier transform of the symbolic scalar `f` with default independent variable `x`. The default return is a function of `w`. The Fourier transform is applied to a function of `x` and returns a function of `w`.

$$f = f(x) \Rightarrow F = F(w)$$

If `f = f(w)`, `fourier` returns a function of `t`.

$$F = F(t)$$

By definition,

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

where `x` is the symbolic variable in `f` as determined by `symvar`.

`F = fourier(f,v)` makes `F` a function of the symbol `v` instead of the default `w`.

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

`F = fourier(f,u,v)` makes `f` a function of `u` and `F` a function of `v` instead of the default variables `x and w`, respectively.

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

Examples

Fourier Transform	MATLAB Commands

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

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

$f(x) = xe^{-\|x\|}$
 	
$F[f](u) = \int_{-\infty}^{\infty} f(x) e^{-ixu} dx$
 	
$= -\frac{4iu}{(1+u^2)^2}$
	`syms x u;`
	`f = x*exp(-abs(x));`
	`fourier(f,u)`
	
	returns
	
	`ans =`
	`-(4*u*i)/(u^2 + 1)^2`

$f(x,v) = e^{-x^2 \frac{\|v\| \sin v}{v}}$, x real	`syms v u;`
	`syms x real;`

Fourier Transform	MATLAB Commands

$F[f(v)](u) = \int_{-\infty}^{\infty} f(x,v)e^{-ivu}dv$
 $= -\arctan\frac{u-1}{x^2} + \arctan\frac{u+1}{x^2}$
	`f = exp(-x^2*abs(v))*sin(v)/v;`
	 `fourier(f,v,u)`
	
	 returns
	
	 `ans =`
	 `piecewise([x <> 0,...`
	 `atan((u + 1)/x^2)...`
	 `- atan((u - 1)/x^2)])`

See Also

`ifourier`, `laplace`, `ztrans`

Purpose

Symbolic matrix elementwise fractional parts

Syntax

`frac(X)`

Description

`frac(X)` is the matrix of the fractional parts of the elements: `frac(X) = X - fix(X)`.

Example

```
x = sym(-5/2);
[fix(x) floor(x) round(x) ceil(x) frac(x)]

ans =
[ -2, -3, -3, -2, -1/2]
```

See Also

`round`, `ceil`, `floor`, `fix`

funtool

Purpose

Function calculator

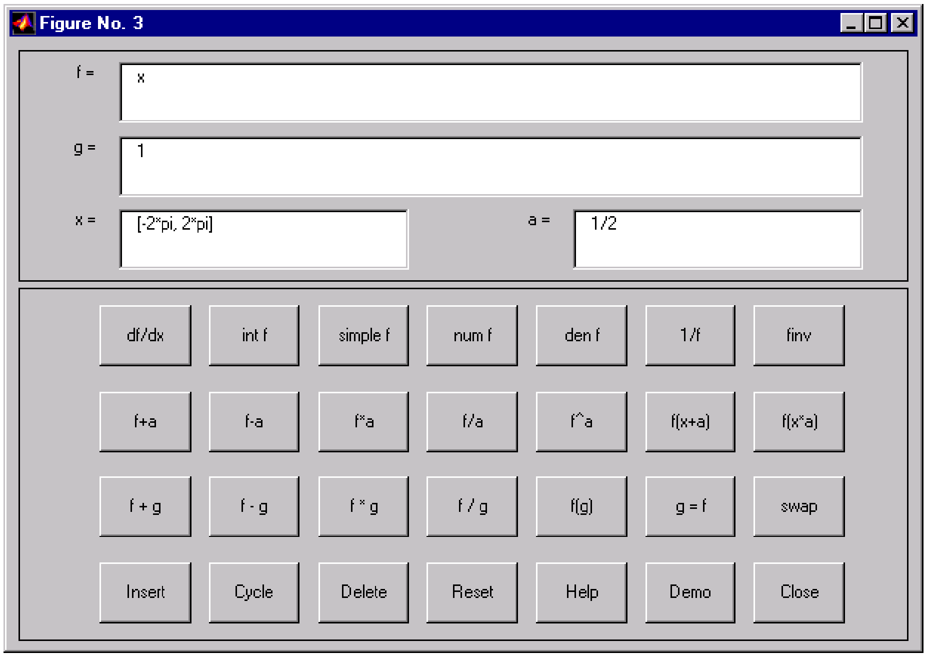
Syntax

```
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`

gamma

Purpose

Gamma function

Syntax

```
gamma(x)
gamma(A)
```

Description

`gamma(x)` returns the gamma function of a symbolic variable or symbolic expression x.

`gamma(A)` returns the gamma function of the elements of a symbolic vector or a symbolic matrix A.

Input Arguments

`x`

A symbolic variable or a symbolic expression

`A`

A vector or a matrix of symbolic variables or expressions

Definitions

gamma Function

The following integral defines the gamma function:

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

Examples

Differentiate the gamma function, and then substitute the variable *t* with the value 1:

```
syms t
u = diff(gamma(t^3 + 1))
subs(u, 1)

u =
3*t^2*gamma(t^3 + 1)*psi(t^3 + 1)

ans =
    1.2684
```

Compute the limit of the following expression that involves the gamma function:

```
syms x;
limit(x/gamma(x), x, inf)

ans =
0
```

Simplify the following expression:

```
syms x;
simplify(gamma(x)*gamma(1 - x))

ans =
pi/sin(pi*x)
```

References

See Also

`mfun` | `mfunlist`

getVar

Purpose

Get variable from MuPAD notebook

Syntax

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

Description

`y = getVar(nb,'z')` assigns the symbolic variable `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`.

Examples

For `x < 0` the function `heaviside(x)` returns `0`:

```
heaviside(sym(-3))

ans =
0
```

For `x > 0` the function, `heaviside(x)` returns `1`:

```
heaviside(sym(3))

ans =
1
```

For `x = 0` the function, `heaviside(x)` returns `1/2`:

```
heaviside(sym(0))

ans =
1/2
```

For numeric `x = 0` the function, `heaviside(x)` returns the numeric result:

```
heaviside(0)

ans =
    0.5000
```

See Also

`dirac`

Purpose

Horner nested polynomial representation

Syntax

`horner(P)`

Description

Suppose `P` is a matrix of symbolic polynomials. `horner(P)` transforms each element of `P` into its Horner, or nested, representation.

Examples

Find nested polynomial representation of the polynomial:

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

The result is

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

Find nested polynomial representation for the polynomials that form a vector:

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

The result is:

```
ans =
   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

Compute hypergeometric functions:

```
syms a z
q = hypergeom([],[],z)
r = hypergeom(1,[],z)
s = hypergeom(a,[],z)
```

The results are:

```
q =
exp(z)

r =
-1/(z - 1)

s =
1/(1 - z)^a
```

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

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

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

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

See Also

`fourier`, `ilaplace`, `iztrans`

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

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

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

$f(u) = \frac{1}{u^2 - a^2}$
$L^{-1}[f] = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} g(u) e^{xu} du$
$= \frac{\sinh(xa)}{a}$
	`syms x u;`
	`syms a real;`
	`f = 1/(u^2-a^2);`
	`simplify(ilaplace(f,x))`
	returns
	`ans =`
	`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 symbolic expression

Syntax

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

Description

`int(expr)` returns the indefinite integral of `expr` with respect to its symbolic variable as defined by `symvar`.

`int(expr, v)` returns the indefinite integral of `expr` with respect to the symbolic scalar variable `v`.

`int(expr, a, b)` returns the definite integral from *a* to *b* of `expr` with respect to the default symbolic variable. *a* and *b* are symbolic or double scalars.

`int(expr, v, a, b)` returns the definite integral of `expr` with respect to `v` from *a* to *b*.

Examples

Find indefinite integral of the following single-variable expression:

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

The result is:

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

Find indefinite integral of the following multivariable expression with respect to `z`:

```
syms x z;
int(x/(1 + z^2), z)
```

The result is:

```
ans =
x*atan(z)
```

Integral the following expression from `0` to `1`:

```
syms x;
int(x*log(1 + x), 0, 1)
```

The result is:

```
ans =
1/4
```

Integral the following expression from `sin(t)` to `1`:

```
syms x t;
int(2*x, sin(t), 1)
```

The result is:

```
ans =
cos(t)^2
```

Find indefinite integrals for the expressions listed as the elements of a matrix:

```
syms x t z;
alpha = sym('alpha');
int([exp(t), exp(alpha*t)])
```

The result is:

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

See Also

`diff` | `symsum` | `symvar`

int8, int16, int32, int64

Purpose

Convert symbolic matrix to signed integers

Syntax

```
int8(S)
int16(S)
int32(S)
int64(S)
```

Description

`int8(S)` converts a symbolic matrix `S` to a matrix of signed 8-bit integers.

`int16(S)` converts `S` to a matrix of signed 16-bit integers.

`int32(S)` converts `S` to a matrix of signed 32-bit integers.

`int64(S)` converts `S` to a matrix of signed 64-bit integers.

Note The output of `int8`, `int16`, `int32`, and `int64` does not have data type `symbolic`.

The following table summarizes the output of these four functions.

Function	Output Range	Output Type	Bytes per Element	Output Class
`int8`	-128 to 127	Signed 8-bit integer	1	`int8`
`int16`	-32,768 to 32,767	Signed 16-bit integer	2	`int16`
`int32`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	4	`int32`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

See Also

`sym`, `vpa`, `single`, `double`, `uint8`, `uint16`, `uint32`, `uint64`

Purpose

Compute symbolic matrix inverse

Syntax

`R = inv(A)`

Description

`R = inv(A)` returns inverse of the symbolic matrix `A`.

Examples

Compute the inverse of the following matrix of symbolic numbers:

```
A = sym([2,-1,0;-1,2,-1;0,-1,2]);
inv(A)
```

The result is:

```
ans =
[ 3/4, 1/2, 1/4]
[ 1/2,   1, 1/2]
[ 1/4, 1/2, 3/4]
```

Compute the inverse of the following symbolic matrix:

```
syms a b c d
A = [a b; c d];
inv(A)
```

The result is:

```
ans =
[  d/(a*d - b*c), -b/(a*d - b*c)]
[ -c/(a*d - b*c),  a/(a*d - b*c)]
```

Compute the inverse of the symbolic Hilbert matrix:

```
inv(sym(hilb(4)))
```

The result is:

```
ans =
[   16,  -120,   240,  -140]
[ -120,  1200, -2700,  1680]
[  240, -2700,  6480, -4200]
[ -140,  1680, -4200,  2800]
```

See Also

`eig` | `det` | `rank`

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 `symvar(F)` and returns a function of `k`.

$$F = F(w) \Rightarrow f = f(k)$$

Examples

Inverse Z-Transform	MATLAB Operation

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

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

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

See Also

`ifourier`, `ilaplace`, `ztrans`

Purpose

Compute 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 $\partial f(i) / \partial v(j)$. If `f` is scalar, the Jacobian of `f` is the gradient of `f`. If `v` is a scalar, the result equals to `diff(f, v)`.

Examples

Compute the Jacobians of the following vectors:

```
syms x y z
f = [x*y*z; y; x + z];
v = [x, y, z];
R = jacobian(f, v)
b = jacobian(x + z, v)
```

The result is

```
R =
[ y*z, x*z, x*y]
[   0,   1,   0]
[   1,   0,   1]

b =
[ 1, 0, 1]
```

See Also

`diff`

jordan

Purpose

Compute Jordan canonical form of matrix

Syntax

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

Description

`J = jordan(A)` computes the Jordan canonical (normal) form of a symbolic or numeric matrix `A`. The Jordan form of a numeric matrix is extremely sensitive to numerical errors. To compute Jordan canonical form of a matrix, represent the elements of the matrix by integers or ratios of small integers, if possible.

`[V, J] = jordan(A)` computes the Jordan canonical form `J` and the similarity transform `V`. The matrix `V` contains the generalized eigenvectors of `A` as columns, and `V\A*V = J`.

Examples

Compute the Jordan canonical form and the similarity transform for the following numeric matrix. Verify that the resulting matrix `V` satisfies the condition `V\A*V = J`:

```
A = [1 -3 -2; -1  1 -1; 2 4 5]
[V, J] = jordan(A)
V\A*V
```

The result is:

```
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

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

See Also `eig` | `inv` | `poly`

lambertw

Purpose

Lambert's *W* function

Syntax

```
W = lambertw(X)
W = lambertw(K,X)
```

Description

`W = lambertw(X)` evaluates Lambert's *W* function at the elements of `X`, a numeric matrix or a symbolic matrix. Lambert's *W* solves the equation

$$we^{w} = x$$

for `w` as a function of `x`.

`W = lambertw(K,X)` is the K-th branch of this multi-valued function.

Examples

Compute Lambert's *W* function:

```
lambertw([0 -exp(-1); pi 1])
```

The result is:

```
ans =
          0   -1.0000
     1.0737    0.5671
```

The statements

```
syms x y
lambertw([0 x;1 y])
```

return

```
ans =
[              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

$$L(s) = \int_0^\infty F(t)e^{-st}dt$$

where `t` is the symbolic variable in `F` as determined by `symvar`.

`L = laplace(F,t)` makes `L` a function of `t` instead of the default `s`.

$$L(t) = \int_0^\infty F(x)e^{-tx}dx$$

Here `L` is returned as a scalar symbol.

`L = laplace(F,w,z)` makes `L` a function of `z` and `F` a function of `w` instead of the default variables `s` and `t`, respectively.

$$L(z) = \int_0^\infty F(w)e^{-zw}dw$$

Examples

Laplace Transform	MATLAB Command

$f(t) = t^4$
 	
$L[f] = \int_0^\infty f(t)e^{-ts}dt$
 	
$= \frac{24}{s^5}$
	`syms t;`
	`f = t^4;`
	`laplace(f)`
	
	returns
	
	`ans =`
	`24/s^5`

$g(s) = \frac{1}{\sqrt{s}}$
 	
$L[g](t) = \int_0^\infty g(s)e^{-st}ds$
 	
$= \sqrt{\frac{\pi}{t}}$
	`syms s;`
	`g = 1/sqrt(s);`
	`laplace(g)`
	
	returns
	
	`ans =`
	`pi^(1/2)/t^(1/2)`

$f(t) = e^{-at}$
 	
$L[f](x) = \int_0^\infty f(t)e^{-tx}dt$
 	
$= \frac{1}{x+a}$
	`syms t a x;`
	`f = exp(-a*t);`
	`laplace(f,x)`
	
	returns
	
	`ans =`
	`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

```
ans =
\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

```
ans =
\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 t;
alpha = sym('alpha');
A = [alpha t alpha*t];
latex(A)
```

return

```
ans =
\left(\begin{array}{ccc} \mathrm{alpha} & t & \mathrm{alpha}\, t...
\end{array}\right)
```

You can use the `latex` command to annotate graphs:

```
syms x
f = taylor(log(1+x));
ezplot(f)
hold on
title(['$' latex(f) '$'],'interpreter','latex')
hold off
```

See Also

`pretty`, `ccode`, `fortran`

Purpose

Compute limit of symbolic expression

Syntax

```
limit(expr, x, a)
limit(expr, a)
limit(expr)
limit(expr, x, a, 'left')
limit(expr, x, a, 'right')
```

Description

`limit(expr, x, a)` computes bidirectional limit of the symbolic expression `expr` when `x` approaches *a*.

`limit(expr, a)` computes bidirectional limit of the symbolic expression `expr` when the default variable approaches *a*.

`limit(expr)` computes bidirectional limit of the symbolic expression `expr` when the default variable approaches `0`.

`limit(expr, x, a, 'left')` computes the limit of the symbolic expression `expr` when `x` approaches *a* from the left.

`limit(expr, x, a, 'right')` computes the limit of the symbolic expression `expr` when `x` approaches *a* from the right.

Examples

Compute bidirectional limits for the following expressions:

```
syms x h;
limit(sin(x)/x)
limit((sin(x + h) - sin(x))/h, h, 0)
```

The results are

```
ans =
1

ans =
cos(x)
```

Compute the limits from the left and right for the following expressions:

```
syms x;
limit(1/x, x, 0, 'right')
limit(1/x, x, 0, 'left')
```

The results are

```
ans =
Inf

ans =
-Inf
```

Compute the limit for the functions presented as elements of a vector:

```
syms x a;
v = [(1 + a/x)^x, exp(-x)];
limit(v, x, inf)
```

The result is

```
ans =
[ exp(a), 0]
```

See Also

diff | taylor

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`

Purpose

Convert symbolic expression to function handle or file

Syntax

```
g = matlabFunction(f)
g = matlabFunction(f1,f2,...)
g = matlabFunction(f,param1,value1,...)
```

Description

`g = matlabFunction(f)` converts the symbolic expression `f` to a MATLAB function with the handle `g`.

`g = matlabFunction(f1,f2,...)` converts a list of the symbolic expressions `f1`, `f2`, `...` to a MATLAB function with multiple outputs. The function handle is `g`.

`g = matlabFunction(f,param1,value1,...)` converts the symbolic expression `f` to a MATLAB function with the handle `g`. The command accepts the following options for parameter/value pairs:

- `Parameter` = `'file'` allows you to generate a file with *optimized* code. The generated file can accept double or matrix arguments and evaluate the symbolic expression applied to the arguments. Optimized means intermediate variables are automatically generated to simplify or speed the code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. The value of this parameter must be a string representing the path to the file. If the string is empty, `matlabFunction` generates an anonymous function. If the string does not end in `.m`, the function appends `.m`.
- `Parameter` = `'outputs'` allows you to set the names of the output variables. `value` should be a cell array of strings. The default names of output variables coincide with the names you use calling `matlabFunction`. If you call `matlabFunction` using an expression instead of individual variables, the default names of output variables consist of the word `out` followed by the number, for example, `out3`.
- `Parameter` = `'vars'` allows you to set the order of the input variables or symbolic vectors in the resulting function handle or the file. The default order is alphabetical. The value of this parameter must be either a cell array of strings or symbolic arrays, or a vector of

symbolic variables. The number of value entries should equal or exceed the number of free variables in the symbolic expression `f`.

Note To convert a MuPAD expression or function to a MATLAB function, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine, 'MuPAD_Function',x1,...,xn)`. `matlabFunction` cannot correctly convert some MuPAD expressions to MATLAB functions. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the results of conversion. To verify the results, execute the resulting function.

Examples

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

ht =
   @(x,y)sin(sqrt(x.^2+y.^2)).*1.0./sqrt(x.^2+y.^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','myfile');
```

If the file `myfile.m` already exists in the current folder, `matlabFunction` replaces the existing function with the converted symbolic expression. You can open and edit the resulting file:

```
function out1 = myfile(x,y,z)
%MYFILE
%    OUT1 = MYFILE(X,Y,Z)

t2 = x.^2;
t3 = y.^2;
```

```
t4 = z.^2;
t5 = t2 + t3 + t4;
out1 = log(t5) + 1.0./sqrt(t5);
```

You can change the order of the input variables:

```
syms x y z
r = x^2 + y^2 + z^2;
matlabFunction(r, 'file', 'new_function',...
'vars', [y z x]);
```

The created `new_function` accepts variables in the required order:

```
function r = new_function(y,z,x)
%NEW_FUNCTION
%    R = NEW_FUNCTION(Y,Z,X)

r = x.^2 + y.^2 + z.^2;
```

You can specify that the input arguments are vectors:

```
syms x y z t
r = (x^2 + y^2 + z^2)*exp(-t);
matlabFunction(r, 'file', 'new_function',...
'vars', {t, [x y z]});
```

The resulting function operates on vectors:

```
function r = new_function(t,in2)
%NEW_FUNCTION
%    R = NEW_FUNCTION(T,IN2)

x = in2(:,1);
y = in2(:,2);
z = in2(:,3);
r = exp(-t).*(x.^2+y.^2+z.^2);
```

You can specify the names of the output variables:

```
syms x y z
r = x^2 + y^2 + z^2;
q = x^2 - y^2 - z^2;
f = matlabFunction(r, q, 'file', 'new_function',...
'outputs', {'name1','name2'});
```

The generated function returns *name1* and *name2*:

```
function [name1,name2] = new_function(x,y,z)
%NEW_FUNCTION
%    [NAME1,NAME2] = NEW_FUNCTION(X,Y,Z)

t9 = x.^2;
t10 = y.^2;
t11 = z.^2;
name1 = t10 + t11 + t9;
if nargout > 1
    name2 = -t10 - t11 + t9;
end
```

Also, you can convert MuPAD expressions:

```
syms x y;
f = evalin(symengine, 'arcsin(x) + arccos(y)');
matlabFunction(f, 'file', 'new_function');
```

The created file contains the same expressions written in the MATLAB language:

```
function f = new_function(x,y)
%NEW_FUNCTION
%    F = NEW_FUNCTION(X,Y)

f = asin(x) + acos(y);
```

See Also

`ccode`, `fortran`, `subs`, `sym2poly`, `emlBlock`

"Generating Code from Symbolic Expressions" on page 3-145

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 listed in “Syntax and Definitions of mfun Special Functions” on page 3-121. 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 6-135.

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 Evaluate Fresnel cosine integral:

```
mfun('FresnelC',0:5)
```

The result is:

```
ans =
0    0.7799    0.4883    0.6057    0.4984    0.5636
```

Evaluate hyperbolic cosine integral:

```
mfun('Chi',[3*i 0])

ans =
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)`	$n \geq 0$ $0<\lvert t\rvert<2\pi$
Bessel functions	`BesselI, BesselJ`—Bessel functions of the first kind. `BesselK, BesselY`—Bessel functions of the second kind.	`BesselJ(v,x)` `BesselY(v,x)` `BesselI(v,x)` `BesselK(v,x)`	`v` is real.
Beta function	$B(x,y)=\frac{\Gamma(x)\cdot\Gamma(y)}{\Gamma(x+y)}$	`Beta(x,y)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Binomial coefficients	$\binom{m}{n} = \frac{m!}{n!(m-n)!}$ $= \frac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$	`binomial(m,n)`	
Complete elliptic integrals	Legendre's complete elliptic integrals of the first, second, and third kind. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.	`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. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.	`EllipticCK(k)` `EllipticCE(k)` `EllipticCPi(a,k)`	`a` is real, $-\infty < a < \infty$. `k` is real, $0 < k < 1$.

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments

Complementary error function and its iterated integrals	$erfc(z) = \frac{2}{\sqrt{\pi}} \cdot \int_z^\infty e^{-t^2} dt = 1 - erf(z)$
	$erfc(-1, z) = \frac{2}{\sqrt{\pi}} \cdot e^{-z^2}$
	$erfc(n, z) = \int_z^\infty erfc(n-1, t) dt$
		`erfc(z)`
		`erfc(n,z)`
			$n > 0$

Dawson's integral	$F(x) = e^{-x^2} \cdot \int_0^x e^{t^2} dt$	`dawson(x)`	
Digamma function	$\Psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	
Dilogarithm integral	$f(x) = \int_1^x \frac{\ln(t)}{1-t} dt$	`dilog(x)`	$x > 1$
Error function	$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	`erf(z)`	

Euler numbers and polynomials	Generating function for Euler numbers:
	$\frac{1}{\cosh(t)} = \sum_{n=0}^\infty E_n \frac{t^n}{n!}$
		`euler(n)`
		`euler(n,z)`
			$n \geq 0$
			$\|t\| < \frac{\pi}{2}$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments

Exponential integrals	$Ei(n,z) = \int_1^{\infty} \frac{e^{-zt}}{t^n} dt$
	 $Ei(x) = PV\left(-\int_{-\infty}^{x} \frac{e^t}{t}\right)$
		`Ei(n,z)`
		 `Ei(x)`
			$n \geq 0$
			 $\mathrm{Real}(z) > 0$

Fresnel sine and cosine integrals	$C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt$
	 $S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt$
		`FresnelC(x)`
		 `FresnelS(x)`	

Gamma function	$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$	`GAMMA(z)`	
Harmonic function	$h(n) = \sum_{k=1}^{n} \frac{1}{k} = \Psi(n+1) + \gamma$	`harmonic(n)`	$n > 0$

Hyperbolic sine and cosine integrals	$Shi(z) = \int_0^z \frac{\sinh(t)}{t} dt$
	 $Chi(z) = \gamma + \ln(z) + \int_0^z \frac{\cosh(t) - 1}{t} dt$
		`Shi(z)`
		 `Chi(z)`	

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
(Generalized) hypergeometric function	$F(n,d,z)=\sum_{k=0}^{\infty}\frac{\prod_{i=1}^{j}\frac{\Gamma(n_i+k)}{\Gamma(n_i)}\cdot z^k}{\prod_{i=1}^{m}\frac{\Gamma(d_i+k)}{\Gamma(d_i)}\cdot k!}$ 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. This definition uses modulus k. The numerical `ellipke` function and the MuPAD functions for computing elliptic integrals use the parameter $m=k^2=\sin^2\alpha$.	`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	$\mathrm{lnGAMMA}(z)=\ln(\Gamma(z))$	`lnGAMMA(z)`	
Logarithmic integral	$Li(x)=PV\left\{\int_0^x\frac{dt}{\ln t}\right\}=Ei(\ln x)$	`Li(x)`	$x>1$

MFUN Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Polygamma function	$\Psi^{(n)}(z) = \frac{d^n}{dz}\Psi(z)$ where $\Psi(z)$ is the Digamma function.	`Psi(n,z)`	$n \geq 0$
Shifted sine integral	$Ssi(z) = Si(z) - \frac{\pi}{2}$	`Ssi(z)`	

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)`	

Orthogonal Polynomials (Continued)

Polynomial	mfun Name	Arguments
Generalized Laguerre	`L(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	

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 elementwise 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)

ans =
24

syms x
mod(x^3 - 2*x + 999, 10)

ans =
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`. Also, you can use the argument `file#linktargetname` to refer to the particular link target inside a notebook. In this case, the `mupad` function opens the MuPAD notebook or program file (`file`) and jumps to the beginning of the link target `linktargetname`. If there are multiple link targets with the name `linktargetname`, the `mupad` function uses the last `linktargetname` occurrence.

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 current folder, and define a handle `mphandle` to the notebook, enter:

```
mphandle = mupad('notebook1.mn');
```

To open a notebook and jump to a particular location, create a link target at that location inside a notebook and refer to it when opening a notebook. For example, if you have the Conclusions section in `notebook1.mn`, create a link target named `conclusions` and refer to it when opening the notebook. The `mupad` function opens `notebook1.mn` and scroll it to display the Conclusions section:

```
mphandle = mupad('notebook1.mn#conclusions');
```

For information about creating link targets, see the Formatting and Exporting MuPAD Documents and Graphics section in the MuPAD Getting Started documentation.

See Also

`getVar`, `mupadwelcome`, `openmn`, `openmu`, `setVar`

Purpose

Launch MuPAD interfaces

Syntax

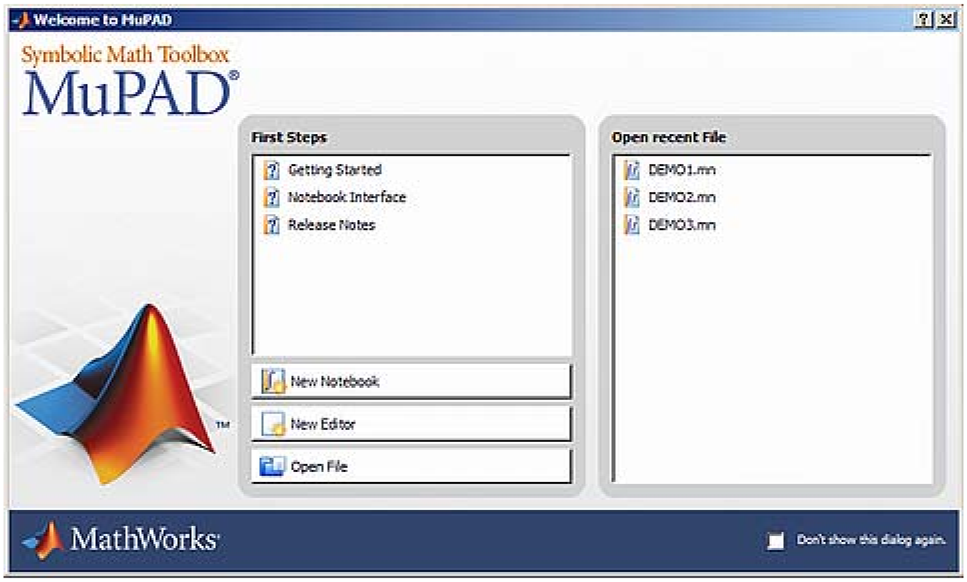
`mupadwelcome`

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.



See Also

`mupad`

"Opening Recent Files and Other MuPAD Interfaces" on page 4-13

Purpose

Form basis for null space of matrix

Syntax

```
Z = null(A)
```

Description

`Z = null(A)` returns a list of vectors that form the basis for the null space of a matrix `A`. The product `A*Z` is zero. `size(Z, 2)` is the nullity of `A`. If `A` has full rank, `Z` is empty.

Examples

Find the basis for the null space and the nullity of the magic square of symbolic numbers. Verify that `A*Z` is zero:

```
A = sym(magic(4));
Z = null(A)
nulllityOfA = size(Z, 2)
A*Z
```

The results are:

```
Z =
 -1
 -3
  3
  1

nulllityOfA =
     1

ans =
 0
 0
 0
 0
```

Find the basis for the null space of the matrix `B` that has full rank:

```
B = sym(hilb(3))
Z = null(B)
```

The result is:

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

Z =
[ empty sym ]
```

See Also

`rank` | `rref` | `size` | `svd`

numden

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

Find the numerator and denominator of the symbolic number:

```
[n, d] = numden(sym(4/5))
```

The result is:

```
n =
4

d =
5
```

Find the numerator and denominator of the symbolic expression:

```
syms x y;
[n,d] = numden(x/y + y/x)
```

The result is:

```
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 `h = openmn(file)`

Description `h = openmn(file)` opens the MuPAD notebook file named `file`, and returns a handle to the file in `h`. The command `h = mupad(file)` accomplishes the same task.

Example To open a notebook named `e-e-x.mn` in the folder `\Documents\Notes` of drive `H:`, enter

```
h = openmn('H:\Documents\Notes\e-e-x.mn');
```

See Also `mupad`, `open`, `openmu`, `openmuphlp`, `openxvc`, `openxvz`

Purpose

Open MuPAD program file

Syntax

`h = openmu(file)`

Description

`h = openmu(file)` opens the MuPAD program file named `file`, and returns a handle to the file in `h`. The command `h = mupad(file)` accomplishes the same task.

Example

To open a program file named `yyx.mu` in the folder `\Documents\Notes` of drive `H:`, enter

```
h = openmu('H:\Documents\Notes\yyx.mu');
```

See Also

`mupad`, `open`, `openmn`, `openmuphlp`, `openxvc`, `openxvz`

openmuphlp

Purpose

Open MuPAD help file

Syntax

`h = openmuphlp(file)`

Description

`h = openmuphlp(file)` opens the MuPAD help file named file, and returns a handle to the file in h. The command `h = mupad(file)` accomplishes the same task.

Input Arguments

`file`

A MuPAD help file

Output Arguments

`h`

A handle to the file

Examples

To open a help file named `helpPage.muphlp` in the folder `\Documents\Notes` of drive `H:`, enter

```
h = openmuphlp('H:\Documents\Notes\helpPage.muphlp');
```

See Also

`mupad` | `open` | `openmn` | `openmu` | `openxvc` | `openxvz`

Purpose

Open MuPAD XVC graphics file

Syntax

`h = openxvc(file)`

Description

`h = openxvc(file)` opens the MuPAD XVC graphics file named file, and returns a handle to the file in h. The command `h = mupad(file)` accomplishes the same task.

Input Arguments

`file`

An MuPAD XVC graphics file

Output Arguments

`h`

A handle to the file

Examples

To open a graphics file named `image1.xvc` in the folder `\Documents\Notes` of drive `H:`, enter

```
h = openxvc('H:\Documents\Notes\image1.xvc');
```

See Also

`mupad` | `open` | `openmn` | `openmu` | `openmuphlp` | `openxvz`

openxvz

Purpose

Open MuPAD XVZ graphics file

Syntax

`h = openxvz(file)`

Description

`h = openxvz(file)` opens the MuPAD XVZ graphics file named file, and returns a handle to the file in h. The command `h = mupad(file)` accomplishes the same task.

Input Arguments

`file`

An MuPAD XVZ graphics file

Output Arguments

`h`

A handle to the file

Examples

To open a graphics file named `image1.xvz` in the folder `\Documents\Notes` of drive `H:`, enter

```
h = openxvz('H:\Documents\Notes\image1.xvz');
```

See Also

`mupad` | `open` | `openmn` | `openmu` | `openmuphlp` | `openxvc`

Purpose

Compute characteristic polynomial of matrix

Syntax

```
p = poly(A)
p = poly(A, v)
poly(sym(A))
```

Description

`p = poly(A)` returns the coefficients of the characteristic polynomial of a numeric matrix `A`. For symbolic `A`, `poly(A)` returns the characteristic polynomial of `A` in terms of the default variable `x`. If the elements of `A` already contain the variable `x`, the default variable is `t`. If the elements of `A` contain both `x` and `t`, the default variable is still `t`.

`p = poly(A, v)`, for both numeric and symbolic matrices, returns the characteristic polynomial of `A` in terms of the variable `v`.

`poly(sym(A))`, for numeric `A`, approximately equals `poly2sym(poly(A))`. The approximation is due to round-off error.

Examples

Compute characteristic polynomials of one of the MATLAB test matrices:

```
syms z
A = gallery(3)
p = poly(A)
q = poly(sym(A))
s = poly(A, z)
```

The results are:

```
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
```

Compute the characteristic polynomials of the following symbolic matrix in terms of the default variable. Also compute the characteristic polynomials in terms of the specified variable `y`:

```
syms x y;
B = x*hilb(3)
a = poly(B)
b = poly(B, y)
```

The results are:

```
B =
[   x, x/2, x/3]
[ x/2, x/3, x/4]
[ x/3, x/4, x/5]

a =
t^3 - (23*t^2*x)/15 + (127*t*x^2)/720 - x^3/2160

b =
- x^3/2160 + (127*x^2*y)/720 - (23*x*y^2)/15 + y^3
```

See Also

`eig` | `jordan` | `poly2sym` | `solve`

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

$$c_1 x^{n-1} + c_2 x^{n-2} + \ldots + c_n$$

`poly2sym` uses `sym`'s default (rational) conversion mode to convert the numeric coefficients to symbolic constants. This mode expresses the symbolic coefficient approximately as a ratio of integers, if `sym` can find a simple ratio that approximates the numeric value, otherwise as an integer multiplied by a power of 2.

`r = poly2sym(c, v)` is 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

The command

```
poly2sym([1 3 2])
```

returns

```
ans =
x^2 + 3*x + 2
```

The command

```
poly2sym([.694228, .333, 6.2832])
```

returns

```
ans =
(6253049924220329*x^2)/9007199254740992 + (333*x)/1000 + 3927/625
```

The command

```
poly2sym([1 0 1 -1 2], y)
```

returns

```
ans =
y^4 + y^2 - y + 2
```

See Also `sym`, `sym2poly`, `polyval` in the online MATLAB Function Reference

Purpose

Pretty-print symbolic expressions

Syntax

Description

The `pretty` function prints symbolic output in a format that resembles typeset mathematics.

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  |
  |  3   5      |
  |  - - ----   |
  |  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 `symvar(A,1)` or `symvar(B,1)`.

Example

```
syms x
p = x^3 - 2*x + 5;
[q, r] = quorem(x^5, p)

q =
x^2 + 2

r =
- 5*x^2 + 4*x - 10

[q, r] = quorem(10^5, subs(p,'10'))

q = 101
r = 515
```

See Also

`mod`

Purpose

Compute rank of symbolic matrix

Syntax

`rank(A)`

Description

`rank(A)` computes the rank of the symbolic matrix `A`.

Examples

Compute the rank of the following numeric matrix:

```
B = magic(4);
rank(B)
```

The result is:

```
ans =
     3
```

Compute the rank of the following symbolic matrix:

```
syms a b c d
A = [a b;c d];
rank(A)
```

The result is:

```
ans =
2
```

See Also

`eig | null | rref | size`

Purpose Real part of complex symbolic 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)]

ans =
[ -2, -3, -3, -2, -1/2]
```

See Also

`floor`, `ceil`, `fix`, `frac`

Purpose

Compute reduced row echelon form of matrix

Syntax

`rref(A)`

Description

`rref(A)` computes the reduced row echelon form of the symbolic matrix `A`. If the elements of a matrix contain free symbolic variables, `rref` regards the matrix as nonzero.

Examples

Compute the reduced row echelon form of the magic square matrix:

```
rref(sym(magic(4)))
```

The result is:

```
ans =
[ 1, 0, 0,  1]
[ 0, 1, 0,  3]
[ 0, 0, 1, -3]
[ 0, 0, 0,  0]
```

Compute the reduced row echelon form of the following symbolic matrix:

```
syms a b c;
A = [a b c; b c a; a + b, b + c, c + a];
rref(A)
```

The result is:

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

See Also

`eig` | `jordan` | `rank` | `size`

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 for *x* from 0 to 1. `rsums(f)` displays a graph of *f*(*x*) using 10 terms (rectangles). You can 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. The height of each rectangle is determined by the value of the function in the middle of each interval.

`rsums(f, a, b)` and `rsums(f, [a, b])` approximates the integral for *x* from `a` to `b`.

Examples Both `rsums('exp(-5*x^2)')` and `rsums exp(-5*x^2)` create the following plot.

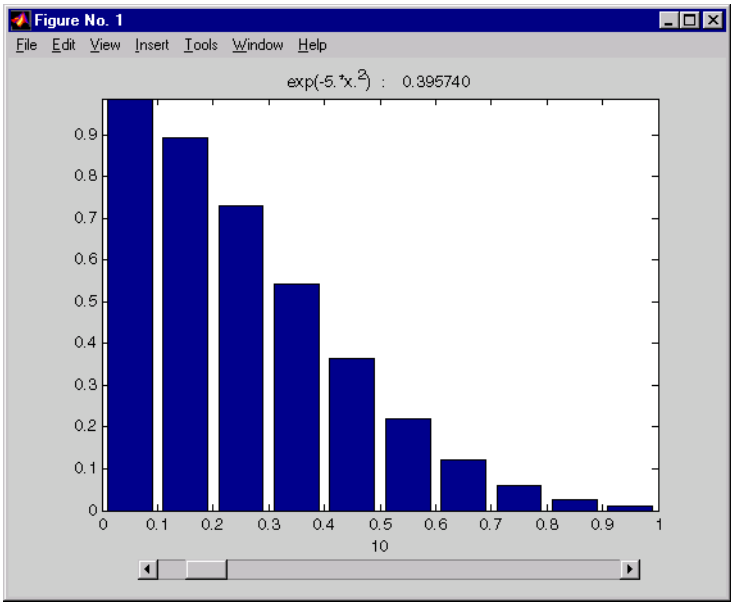


Figure No. 1
File Edit View Insert Tools Window Help
exp(-5.*x.^2) : 0.395740
10

setVar

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`

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

Simplify the expressions:

```
syms x;
f = cos(x)^2 + sin(x)^2;
f = simple(f)
g = cos(3*acos(x));
g = simple(g)
```

The results are:

```
f =
1

g =
4*x^3 - 3*x
```

Simplify the expressions displaying all possible simplifications:

```
syms x;
f = cos(x) + i*sin(x);
```

```
simple(f)
```

The result is:

```
simplify:
cos(x) + sin(x)*i

radsimp:
cos(x) + sin(x)*i

simplify(100):
cos(x) + sin(x)*i

combine(sincos):
cos(x) + sin(x)*i

combine(sinhcosh):
cos(x) + sin(x)*i

combine(ln):
cos(x) + sin(x)*i

factor:
cos(x) + sin(x)*i

expand:
cos(x) + sin(x)*i

combine:
cos(x) + sin(x)*i

rewrite(exp):
exp(x*i)

rewrite(sincos):
cos(x) + sin(x)*i
```

```
rewrite(sinhcosh):
cosh(-x*i) - sinh(-i*x)

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

mwcos2sin:
- 2*sin(x/2)^2 + sin(x)*i + 1

collect(x):
cos(x) + sin(x)*i

ans =
exp(x*i)
```

Simplify the expression and display the simplification method:

```
syms x;
f = (x + 1)*x*(x - 1);
[f, how] = simple(f)

f =
x^3 - x

how =
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` = 100.

Examples

Simplify the trigonometric expression:

```
syms x;
simplify(sin(x)^2 + cos(x)^2)
```

The result is:

```
ans =
1
```

Simplify the expression:

```
syms a b c;
simplify(exp(c*log(sqrt(a+b))))
```

The result is:

```
ans =
(a + b)^(c/2)
```

Simplify the expressions from the list:

```
S = [(x^2 + 5*x + 6)/(x + 2), sqrt(16)];
R = simplify(S)
```

The result is:

```
R =
[ x + 3, 4]
```

See Also

`collect`, `expand`, `factor`, `horner`, `simple`

simscapeEquation

Purpose

Convert symbolic expressions to Simscape language equations

Syntax

```
simscapeEquation(f)
simscapeEquation(LHS, RHS)
```

Description

`simscapeEquation(f)` converts the symbolic expression *f* to a Simscape language equation. This function call converts any derivative with respect to the variable *t* to the Simscape notation `X.der`. Here `X` is the time-dependent variable. In the resulting Simscape equation, the variable *time* replaces all instances of the variable *t* except for derivatives with respect to *t*.

`simscapeEquation(LHS, RHS)` returns a Simscape equation `LHS == RHS`.

Tips

The equation section of a Simscape component file supports a limited number of functions. See the list of Supported Functions for more information. If a symbolic equation contains the functions that are not available in the equation section of a Simscape component file, `simscapeEquation` cannot correctly convert these equations to Simscape equations. Such expressions do not trigger an error message. The following types of expressions are prone to invalid conversion:

- Expressions with infinities
- Expressions returned by `evalin` and `feval`.

If you perform symbolic computations in the MuPAD Notebook Interface and want to convert the results to Simscape equations, use the `generate::Simscape` function in MuPAD.

Examples

Convert the following expressions to Simscape language equations:

```
syms t
x = sym('x(t)');
y = sym('y(t)');
phi = diff(x)+5*y + sin(t);
```

```
simscapeEquation(phi)
simscapeEquation(diff(y),phi)
```

The result is:

```
ans =
phi == x.der + sin(time) + 5*y

ans =
y.der == x.der + sin(time) + 5*y
```

See Also

emlBlock | matlabFunction | ccode | fortran

How To

- "Generating Simscape Equations" on page 3-155

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

$$Si(x) = \int_0^x \frac{\sin t}{t} dt$$

Examples

Evaluate sine integral for the elements of the matrix:

```
sinint([pi 0;-2.2 exp(3)])

ans =
    1.8519         0
   -1.6876    1.5522
```

The statement

```
sinint(1.2)
```

returns

```
ans =
1.1080
```

The statement

```
syms x;
diff(sinint(x))
```

returns

```
ans =
sin(x)/x
```

See Also `cosint`

Purpose

Symbolic matrix dimensions

Syntax

```
d = size(A)
[m, n] = size(A)
d = size(A, n)
```

Description

Suppose `A` is an `m`-by-`n` symbolic or numeric matrix. The statement `d = size(A)` returns a numeric vector with two integer components, `d = [m,n]`.

The multiple assignment statement `[m, n] = size(A)` returns the two integers in two separate variables.

The statement `d = size(A, n)` returns the length of the dimension specified by the scalar n. For example, `size(A,1)` is the number of rows of `A` and `size(A,2)` is the number of columns of `A`.

Examples

The statements

```
   syms a b c d
   A = [a b c ; a b d; d c b; c b a];
   d = size(A)
   r = size(A, 2)
```

return

```
   d =
        4     3

   r =
        3
```

See Also

`length`, `ndims` in the online MATLAB Function Reference

solve

Purpose

Symbolic solution of algebraic equations

Syntax

```
solve(eq)
solve(eq, var)
solve(eq1, eq2, ..., eqn)
g = solve(eq1, eq2, ..., eqn, var1, var2, ..., varn)
```

Description

Single Equation/Expression

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 `symvar`).

`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 `symvar` 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 the quadratic equation:

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

The result is:

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

Solve the quadratic equation for the variable `b`:

```
syms a b c x;
solve('a*x^2 + b*x + c','b')
```

The result is:

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

Solve the system of equations:

```
syms x;
S = solve('x + y  = 1','x - 11*y = 5');
S = [S.x S.y]
```

The result is:

```
S =
[ 4/3, -1/3]
```

The statement

```
syms a u v;
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]
```

To see the elements of A, enter:

```
Aa = A.a
Au = A.u
Av = A.v

Aa =
 2
 3
 3
 2
Au =
 1/3 + (2^(1/2)*i)/3
 1/4 + (3^(1/2)*i)/4
 1/4 - (3^(1/2)*i)/4
 1/3 - (2^(1/2)*i)/3

Av =
 - 2/3 + (2^(1/2)*i)/3
 - 3/4 + (3^(1/2)*i)/4
 - 3/4 - (3^(1/2)*i)/4
 - 2/3 - (2^(1/2)*i)/3
```

See Also

`Arithmetic Operations`, `dsolve`, `symvar`

Purpose

Sort symbolic vectors, matrices, or polynomials

Syntax

```
Y = sort(X)
Y = sort(X, dim)
Y = sort(X, mode)
[Y, I] = sort(X)
```

Description

`Y = sort(X)` sorts the elements of a symbolic vector or matrix in ascending order. If `X` is a vector, `sort(X)` sorts the elements of `X` in numerical or lexicographic order. If `X` is a matrix, `sort(X)` sorts each column of `X`.

`Y = sort(X, dim)` sorts the elements of a symbolic vector or matrix along the dimension of `X` specified by the integer *`dim`*. For two-dimensional matrices, use `1` to sort element of each column and `2` to sort element of each row.

`Y = sort(X, mode)` sorts the elements of a symbolic vector or matrix in the specified direction, depending on the value of *`mode`*. Use `ascend` to sort in ascending order, and `descend` to sort in descending order.

`[Y, I] = sort(X)` sorts a symbolic vector or a matrix `X`. This call also returns the array `I` that shows the indices that each element of a new vector or matrix `Y` had in the original vector or matrix `X`. If `X` is an m-by-n matrix, then each column of `I` is a permutation vector of the corresponding column of `X`, such that

```
for j = 1:n
    Y(:,j) = X(I(:,j),j);
end
```

If `X` is a two-dimensional matrix and you sort the elements of each column, the array `I` shows the row indices that the elements of `Y` had in the original matrix `X`. If you sort the elements of each row, `I` shows the original column indices.

Examples

Sort the elements of the following symbolic vector in ascending and descending order:

```
syms a b c d e;
sort([7 e 1 c 5 d a b])
sort([7 e 1 c 5 d a b], 'descend')
```

The results are:

```
ans =
[ 1, 5, 7, a, b, c, d, e]

ans =
[ e, d, c, b, a, 7, 5, 1]
```

Sort the elements of the following symbolic matrix:

```
X = sym(magic(3))

X =
[ 8, 1, 6]
[ 3, 5, 7]
[ 4, 9, 2]
```

By default, the `sort` command sorts elements of each column:

```
sort(X)

ans =
[ 3, 1, 2]
[ 4, 5, 6]
[ 8, 9, 7]
```

To sort the elements of each row, use set the value of the *`dim`* option to 2:

```
sort(X, 2)

ans =
[ 1, 6, 8]
[ 3, 5, 7]
[ 2, 4, 9]
```

Sort the elements of each row of `X` in descending order:

```
sort(X, 2, 'descend')

ans =
[ 8, 6, 1]
[ 7, 5, 3]
[ 9, 4, 2]
```

Sort the matrix `X` returning the array with indices that each element of the resulting matrix had in `X`:

```
[Y, I] = sort(X)

Y =
[ 3, 1, 2]
[ 4, 5, 6]
[ 8, 9, 7]

I =
     2     1     3
     3     2     1
     1     3     2
```

See Also

`sym2poly`, `coeffs`

subexpr

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

```
h = solve('a*x^3+b*x^2+c*x+d = 0');
[r,s] = subexpr(h,'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)*(s^(1/3)...
                    + (c/(3*a) - b^2/(9*a^2))/s^(1/3))*i)/2
 (c/(3*a) - b^2/(9*a^2))/(2*s^(1/3)) - s^(1/3)/2 - b/(3*a) + (3^(1/2)*(s^(1/3)...
              + (c/(3*a) - b^2/(9*a^2))/s^(1/3))*i)/2

s =
((d/(2*a) + b^3/(27*a^3) - (b*c)/(6*a^2))^2 + (c/(3*a) - b^2/(9*a^2))^3)^(1/2)...
                                          - b^3/(27*a^3) - d/(2*a) + (b*c)/(6*a^2)
```

See Also

`pretty`, `simple`, `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 vectors or cell arrays of the same size and type, 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

```
y =
C2/exp(a*t)
```

Then the statements

```
a = 980; C2 = 3; subs(y)
```

produce

```
ans =
3/exp(980*t)
```

Single Substitution

```
syms a b;
subs(a + b, a, 4)
```

returns

```
ans =
b + 4
```

Multiple Substitutions

```
syms a b;
subs(cos(a) + sin(b), {a, b}, {sym('alpha'), 2})
```

returns

```
ans =
sin(2) + cos(alpha)
```

Scalar Expansion Case

```
syms t;
subs(exp(a*t), 'a', -magic(2))
```

returns

```
ans =
[   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

```
ans =
     0    -1
     2     0
```

See Also

`simplify`, `subexpr`

Purpose

Compute singular value decomposition of symbolic matrix

Syntax

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

Description

`sigma = svd(A)` returns a symbolic vector containing the singular values of a symbolic matrix `A`. With symbolic inputs, `svd` does not accept complex values as inputs.

`sigma = svd(vpa(A))` returns a vector with the 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` with the columns containing the singular vectors and a diagonal matrix `S` containing the singular values. The matrices satisfy `A = U*S*V'`. The `svd` command does not compute symbolic singular vectors. With multiple outputs, `svd` does not accept complex values as inputs.

Examples

Compute the symbolic and numeric singular values and the numeric singular vectors of the following magic square:

```
digits(5)
A = sym(magic(4));
svd(A)
svd(vpa(A))
[U, S, V] = svd(A)
```

The results are:

```
ans =
        0
 2*5^(1/2)
 8*5^(1/2)
       34
```

```
ans =
           34.0
         17.889
         4.4721
 2.8024*10^(-7)

U =
[ 0.5,  0.67082,  0.5,  0.22361]
[ 0.5, -0.22361, -0.5,  0.67082]
[ 0.5,  0.22361, -0.5, -0.67082]
[ 0.5, -0.67082,  0.5, -0.22361]

S =
[ 34.0,      0,      0, 0]
[    0, 17.889,      0, 0]
[    0,      0, 4.4721, 0]
[    0,      0,      0, 0]

V =
[ 0.5,  0.5,  0.67082,  0.22361]
[ 0.5, -0.5, -0.22361,  0.67082]
[ 0.5, -0.5,  0.22361, -0.67082]
[ 0.5,  0.5, -0.67082, -0.22361]
```

See Also

digits | eig | inv | vpa

Purpose

Define symbolic objects

Syntax

```
S = sym(A)
x = sym('x')
x = sym('x', 'real')
k = sym('x', 'positive')
x = sym('x', 'clear')
A = sym('A', [m n])
A = sym('A', n)
A = sym(A, 'real')
A = sym(A, 'positive')
A = sym(A, 'clear')
S = sym(A, flag)
```

Description

`S = sym(A)` constructs an object `S`, of the `sym` class, 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')` creates the symbolic variable `x` and 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('x', 'positive')` creates the symbolic variable `x` and assumes that `x` is real and positive.

`x = sym('x', 'clear')` clears all previously set assumptions on the variable `x`. 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.

`A = sym('A', [m n])` creates a `m`-by-`n` matrix of symbolic variables. The dimensions `m` and `n` of a matrix must be integers. You can use this syntax to create a `1`-by-`n` or an `n`-by-`1` vector. The `sym` function does not allow you to use symbolic variables without assigned numeric values as dimensions. By default, the generated names of elements of a vector use the form `Ak`, and the generated names of elements of a matrix use the form `Ai_j`. The base, `A`, must be a valid variable name. (To verify whether the name is a valid variable name, use `isvarname`). The values of `k`, `i`, and `j` range from 1 to `m` or 1 to `n`. To specify other form for generated names of matrix elements, use `'%d'` in the first input. For example, `A = sym('A%d%d', [3 3])` generates the 3-by-3 symbolic matrix `A` with the elements `A11, A12, ..., A33`.

`A = sym('A', n)` creates an `n`-by-`n` matrix.

`A = sym(A, 'real')` sets an assumption that all elements of a symbolic matrix (or a symbolic vector) `A` are real. To create a symbolic vector or a symbolic matrix `A`, use `A = sym('A', [m n])` or `A = sym('A', n)`.

`A = sym(A, 'positive')` sets an assumption that all elements of a symbolic matrix (or a symbolic vector) `A` are real and positive. To create a symbolic vector or a symbolic matrix `A`, use `A = sym('A', [m n])` or `A = sym('A', n)`.

`A = sym(A, 'clear')` clears assumptions previously set on the symbolic matrix (or the symbolic vector) `A`.

`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 `N*2^e` or `-N*2^e` , where `N` and `e` are integers, `N`   `0`. For example, `sym(1/10,'f')` is `3602879701896397/36028797018963968` .

`'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 round-off 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 `3`s, but is an accurate decimal representation of the floating-point number nearest to `4/3`.

Examples

Create the 3-by-4 symbolic matrix `A` with the auto-generated elements `A1_1, ..., A3_4` :

```
A = sym('A', [3 4])

A =
[ A1_1, A1_2, A1_3, A1_4]
[ A2_1, A2_2, A2_3, A2_4]
[ A3_1, A3_2, A3_3, A3_4]
```

Now, create the 4-by-4 matrix `B` with the elements `x_1_1, ..., x_4_4`:

```
B = sym('x_%d_%d', [4 4])

B =
[ x_1_1, x_1_2, x_1_3, x_1_4]
[ x_2_1, x_2_2, x_2_3, x_2_4]
[ x_3_1, x_3_2, x_3_3, x_3_4]
[ x_4_1, x_4_2, x_4_3, x_4_4]
```

This syntax does not define elements of a symbolic matrix as separate symbolic objects. To access an element of a matrix, use parentheses:

```
A(2, 3)
B (4, 2)

ans =
A2_3

ans =
x_4_2
```

You can use symbolic matrices and vectors generated by the `sym` function to define other matrices:

```
A = diag(sym('A',[1 4]))

A =
[ A1,  0,  0,  0]
[  0, A2,  0,  0]
[  0,  0, A3,  0]
[  0,  0,  0, A4]
```

Perform operations on symbolic matrices by using the operators that you use for numeric matrices. For example, find the determinant and the trace of the matrix `A`:

```
det(A)

ans =
A1*A2*A3*A4

trace(A)

ans =
A1 + A2 + A3 + A4
```

Also, use the `sym` function to set assumptions on all elements of a symbolic matrix. You cannot create a symbolic matrix and set an assumption on all its elements in one sym function call. Use two separate sym function calls: the first call creates a matrix, and the second call specifies an assumption:

```
A = sym('A%d%d', [2 2]);
A = sym(A, 'positive')

A =
[ A11, A12]
[ A21, A22]
```

Now, MATLAB assumes that all elements of A are positive:

```
solve(A(1, 1)^2 - 1, A(1, 1))

ans =
1
```

To clear all previously set assumptions on elements of a symbolic matrix, also use the sym function:

```
A = sym(A, 'clear');
solve(A(1, 1)^2 - 1, A(1, 1))

ans =
 -1
  1
```

The sym function lets you choose the conversion technique by specifying the optional second argument, which can be 'r', 'f', 'd' or 'e'. The default is 'r'. For example, convert the number 1/3 to a symbolic object:

```
r = sym(1/3)
f = sym(1/3, 'f')
d = sym(1/3, 'd')
e = sym(1/3, 'e')

r =
1/3
```

```
f =
6004799503160661/18014398509481984

d =
0.3333333333333333148296162562473909929395

e =
1/3 - eps/12
```

See Also `digits`, `double`, `findsym`, `reset`, `syms`, `symvar`

`eps` in the online MATLAB Function Reference

symengine

Purpose

Return symbolic engine

Syntax

`s = symengine`

Description

`s = symengine` returns the currently active symbolic engine.

Examples

To see which symbolic computation engine is currently active, enter:

```
s = symengine
```

The result is:

```
s =
MuPAD symbolic engine
```

Now you can use the variable *s* in function calls that require symbolic engine:

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

The result is:

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

See Also

`feval`

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 a shortcut for

```
arg1 = sym('arg1');
arg2 = sym('arg2'); ...
```

`syms arg1 arg2 ...  real` is a shortcut for

```
arg1 = sym('arg1','real');
arg2 = sym('arg2','real'); ...
```

`syms arg1 arg2 ...  clear` is a shortcut for

```
arg1 = sym('arg1','clear');
arg2 = sym('arg2','clear'); ...
```

`syms arg1 arg2 ...  positive` is a shortcut 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`.

In functions and scripts, do not use the `syms` command to create symbolic variables with the same names as MATLAB functions. For these names MATLAB does not create symbolic variables, but keeps the names assigned to the functions. If you want to create a symbolic variable with the same name as some MATLAB function inside a function or a script, use the `sym` command. For example:

```
alpha = sym('alpha')
```

Examples

`syms x y real` is equivalent to

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

To clear the symbolic objects `x` and `y` of `'real'` status, type

```
syms x y 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)` resets the MuPAD engine.
- `clear all` clears all objects in the MATLAB workspace and resets the MuPAD engine.

See Also

`findsym`, `reset`, `sym`, `symvar`

"Clearing Assumptions and Resetting the Symbolic Engine" on page 4-33

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 command

```
syms x u v
sym2poly(x^3 - 2*x - 5)
```

returns

```
ans =
 1     0    -2    -5
```

The command

```
sym2poly(u^4 - 3 + 5*u^2)
```

returns

```
ans =
 1     0     5     0    -3
```

and the command

```
sym2poly(sin(pi/6)*v + exp(1)*v^2)
```

returns

```
ans =
2.7183    0.5000         0
```

See Also `poly2sym`, `subs`, `sym`, `polyval` in the online MATLAB Function Reference

Purpose

Evaluate symbolic sum of series

Syntax

```
r = symsum(expr)
r = symsum(expr, v)
r = symsum(expr, a, b)
r = symsum(expr, v, a, b)
```

Description

`r = symsum(expr)` evaluates the sum of the symbolic expression `expr` with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `0` to `defaultVar - 1`.

`r = symsum(expr, v)` evaluates the sum of the symbolic expression `expr` with respect to the symbolic variable `v`. The value of the variable `v` changes from `0` to `v - 1`.

`r = symsum(expr, a, b)` evaluates the sum of the symbolic expression `expr` with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `a` to `b`.

`r = symsum(expr, v, a, b)` evaluates the sum of the symbolic expression `expr` with respect to the symbolic variable `v`. The value of the default variable changes from `a` to `b`.

Examples

Evaluate the sum of the following symbolic expressions `k` and `k^2`:

```
syms k
symsum(k)
symsum(k^2)
```

The results are

```
ans =
k^2/2 - k/2

ans =
k^3/3 - k^2/2 + k/6
```

Evaluate the sum of the following expression specifying the limits:

```
symsum(k^2, 0, 10)
```

The result is

```
ans =
385
```

Evaluate the sum of the following multivariable expression with respect to k:

```
syms x;
symsum(x^k/sym('k!'), k, 0, inf)
```

The result is

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

See Also

int | syms | symvar

How To

- "Symbolic Summation" on page 3-19

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 the alphabetical order with uppercase letters preceding lowercase letters. 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 alphabetically closest to `'x'`:

1. The variables are sorted by the first letter in their names. The ordering is x y w z v u ... a X Y W Z V U ... A. The name of a symbolic variable cannot begin with a number.
2. For all subsequent letters, the ordering is alphabetical, with all uppercase letters having precedence over lowercase: 0 1 ... 9 A B ... Z a b ...z.

Note `symvar(s)` can return variables in different order than `symvar(s,n)`.

Examples

```
syms wa wb wx yx ya yb
f = wa + wb + wx + ya + yb + yx;
symvar(f)
```

The result is:

```
ans =
[ wa, wb, wx, ya, yb, yx]

syms x y z a b
```

```
w = x^2/(sin(3*y - b));
symvar(w)
```

The result is:

```
ans =
[ b, x, y]

symvar(w, 3)
```

The result is:

```
ans =
[ x, y, b]
```

`symvar(s,1)` returns the variable closest to `x`. When performing differentiation, integration, substitution or solving equations, MATLAB uses this variable as a default variable.

```
syms v z
g = v + z;
symvar(g, 1)
```

The result is:

```
ans =
z

syms aaa aab
g = aaa + aab;
symvar(g, 1)
```

The result is:

```
ans =
aaa

syms X1 x2 xa xb
g = X1 + x2 + xa + xb;
```

```
symvar(g, 1)
```

The result is:

```
ans =
x2
```

See Also `findsym`, `sym`, `syms`

taylor

Purpose

Taylor series expansion

Syntax

```
taylor(f)
taylor(f, n)
taylor(f, a)
taylor(f, n, v)
taylor(f, n, v, a)
```

Description

`taylor(f)` returns the fifth order Maclaurin polynomial approximation to `f`.

`taylor(f, n)` returns the (`n-1`)-order Maclaurin polynomial approximation to `f`. Here `n` is a positive integer.

`taylor(f, a)` returns the fifth order Taylor series approximation to `f` about point `a`. Here `a` is a real number. If `a` is a positive integer or if you want to change the expansion order, use `taylor(f,n,a)` to specify the base point and the expansion order.

`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. If `a` is a symbol or a string, do not omit `v`.

If a is neither an integer nor a symbol or a string, 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 also can omit any of the arguments `n`, `v`, and `a`. If you do not specify `v`, `taylor` uses`symvar` to determine the function's independent variable. `n` defaults to `6`, and `a` defaults to `0`.

The following expression present the Taylor series for an analytic function $f(x)$ about the base point $x=a$:

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

Examples

The following table describes the various uses of the `taylor` command and its relation to Taylor and Maclaurin series. Before using the `taylor` command, define the function you want to expand. For example:

```
syms x
f = exp(x^2);
```

Mathematical Operation	MATLAB Operation
$\sum_{m=0}^{5} x^m \cdot \frac{f^{(m)}(0)}{m!}$	`taylor(f)`
$\sum_{m=0}^{n-1} x^m \cdot \frac{f^{(m)}(0)}{m!}$ n is a positive integer	`taylor(f,n)` n is a positive integer.
$\sum_{m=0}^{5} (x-a)^m \cdot \frac{f^{(m)}(a)}{m!}$ a is a real number	`taylor(f,a)` a is a real number.
$\sum_{m=0}^{n-1} (x-a)^m \cdot \frac{f^{(m)}(a)}{m!}$ n is a positive integer and `a` is real. Also, `a` can be a positive integer.	`taylor(f,n,a)` a is real and n is a positive integer.

In the case where `f` is a function of two or more variables (`f=f(x,y,...)`), there is an additional 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
$\sum_{m=0}^{5} \frac{y^m}{m!} \cdot \frac{\partial^n}{\partial y^m} f(x,y) \Big\vert_{y=0}$	`taylor(f,y)`
$\sum_{m=0}^{n-1} \frac{y^m}{m!} \cdot \frac{\partial^m}{\partial y^m} f(x,y) \Big\vert_{y=0}$ n is a positive integer	`taylor(f,y,n)` or `taylor(f,n,y)` n is a positive integer.
$\sum_{m=0}^{5} \frac{(y-a)^m}{m!} \cdot \frac{\partial^m}{\partial y^m} f(x,y) \Big\vert_{y=a}$ a is real	`taylor(f,y,a)` a is real.
$\sum_{m=0}^{n-1} \frac{(y-a)^m}{m!} \cdot \frac{\partial^m}{\partial y^m} f(x,y) \Big\vert_{y=a}$ a is real and n is a positive integer	`taylor(f,n,y,a)` a is real and n is a positive integer.

See Also `symvar`, `taylortool`

Purpose

Taylor series calculator

Syntax

```
taylortool
taylortool('f')
```

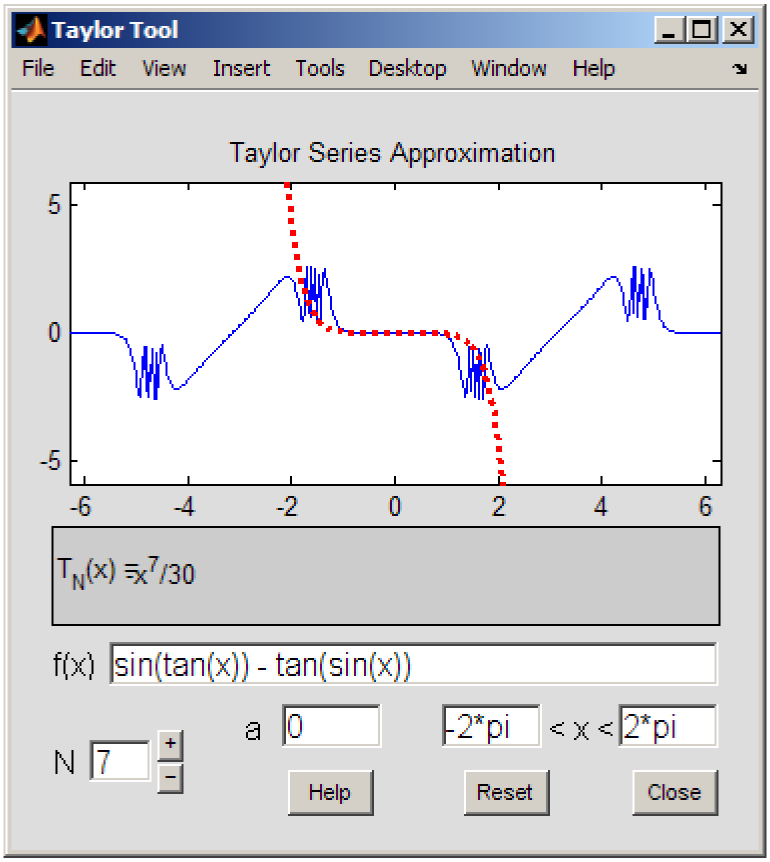
Description

`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('sin(tan(x)) - tan(sin(x))')
```


Taylor Tool
File Edit View Insert Tools Desktop Window Help
Taylor Series Approximation
5
0
-5
-6
-4
-2
0
2
4
6
Help
Reset
Close

$T_N(x) = x^7/30$

f(x) sin(tan(x)) - tan(sin(x))

N 7

a 0

-2*pi < x < 2*pi

See Also `funtool`, `rsums`

Purpose

Enable and disable tracing of MuPAD commands

Syntax

```
trace(symengine, 'on')
trace(symengine, 'off')
```

Description

`trace(symengine, 'on')` enables tracing of all subsequent MuPAD commands. Tracing means that for each command Symbolic Math Toolbox shows all internal calls to MuPAD functions and the results of these calls.

`trace(symengine, 'off')` disables MuPAD commands tracing.

See Also

`evalin` | `feval`

Purpose

Return lower triangular part of symbolic matrix

Syntax

```
tril(A)
tril(A, k)
```

Description

`tril(A)` returns a triangular matrix that retains the lower part of the matrix `A`. The upper triangle of the resulting matrix is padded with zeros.

`tril(A, k)` returns a matrix that retains the elements of `A` on and below the *k*-th diagonal. The elements above the *k*-th diagonal equal to zero. The values $k = 0$, $k > 0$, and $k < 0$ correspond to the main, superdiagonals, and subdiagonals, respectively.

Examples

Display the matrix retaining only the lower triangle of the original symbolic matrix:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
tril(A)
```

The result is:

```
ans =
[     a,     0,     0]
[     1,     2,     0]
[ a + 1, b + 2, c + 3]
```

Display the matrix that retains the elements of the original symbolic matrix on and below the first superdiagonal:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
tril(A, 1)
```

The result is:

```
ans =
[     a,     b,     0]
[     1,     2,     3]
[ a + 1, b + 2, c + 3]
```

Display the matrix that retains the elements of the original symbolic matrix on and below the first subdiagonal:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
tril(A, -1)
```

The result is:

```
ans =
[     0,     0, 0]
[     1,     0, 0]
[ a + 1, b + 2, 0]
```

See Also

`diag` | `triu`

Purpose

Return upper triangular part of symbolic matrix

Syntax

```
triu(A)
triu(A, k)
```

Description

`triu(A)` returns a triangular matrix that retains the upper part of the matrix `A`. The lower triangle of the resulting matrix is padded with zeros.

`triu(A, k)` returns a matrix that retains the elements of `A` on and above the *k*-th diagonal. The elements below the *k*-th diagonal equal to zero. The values `k = 0`, `k > 0`, and `k < 0` correspond to the main, superdiagonals, and subdiagonals, respectively.

Examples

Display the matrix retaining only the upper triangle of the original symbolic matrix:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
triu(A)
```

The result is:

```
ans =
[ a, b,     c]
[ 0, 2,     3]
[ 0, 0, c + 3]
```

Display the matrix that retains the elements of the original symbolic matrix on and above the first superdiagonal:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
triu(A, 1)
```

The result is:

```
ans =
[ 0, b, c]
[ 0, 0, 3]
[ 0, 0, 0]
```

Display the matrix that retains the elements of the original symbolic matrix on and above the first subdiagonal:

```
syms a b c
A = [a b c; 1 2 3; a + 1 b + 2 c + 3];
triu(A, -1)
```

The result is:

```
ans =
[ 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 at least `d` decimal digits of accuracy, where `d` is the current setting of `digits`. The toolbox also increases the internal precision of calculations by several digits (guard digits).

`R = vpa(A, d)` uses at least `d` significant (nonzero) digits, instead of the current setting of `digits`. The value `d` must be a positive integer larger than 1 and smaller than $2^{29}+1$.

Examples

Approximate the following expressions with the 25 digits precision:

```
old = digits(25);
q = vpa('1/2')
p = vpa(pi)
w = vpa('(1+sqrt(5))/2')
digits(old)

q =
0.5

p =
3.141592653589793238462643

w =
1.618033988749894848204587
```

Solve the following equation:

```
y = solve('x^2 - 2')
```

```
y =
  2^(1/2)
 -2^(1/2)
```

Approximate the solutions with floating-point numbers:

```
vpa(y(1))
vpa(y(2))

ans =
1.4142135623730950488016887242097

ans =
-1.4142135623730950488016887242097
```

Use the `vpa` function to approximate elements of the following matrices:

```
A = vpa(hilb(2), 25)
B = vpa(hilb(2), 5)

A =
[ 1.0,                         0.5]
[ 0.5, 0.3333333333333333333333333]

B =
[ 1.0,     0.5]
[ 0.5, 0.33333]
```

The `vpa` function lets you specify the number of significant (nonzero) digits that is different from the current `digits` setting. For example, compute the ratio 1/3 and the ratio 1/3000 with 4 significant digits:

```
vpa(1/3, 4)
vpa(1/3000, 4)

ans =
0.3333

ans =
0.0003333
```

The number of digits that you specify by the `vpa` function or the `digits` function is the minimal number of digits. Internally, the toolbox can use more digits than you specify. These additional digits are called guard digits. For example, set the number of digits to 4, and then display the floating-point approximation of 1/3 using 4 digits:

```
old = digits;
digits(4);
a = vpa(1/3, 4)

a =
0.3333
```

Now, display `a` using 20 digits. The result shows that the toolbox internally used more than 4 digits when computing `a`. The last digits in the following result are incorrect because of the round-off error:

```
vpa(a, 20)
digits(old);

ans =
0.33333333333303016843
```

Hidden round-off errors can cause unexpected results. For example, compute the number 1/10 with the default 32 digits accuracy and with the 10 digits accuracy:

```
a = vpa(1/10, 32)
b = vpa(1/10, 10)

a =
0.1

b =
0.1
```

Now, compute the difference `a - b`. The result is not zero:

```
a - b

ans =
0.000000000000000000086736173798840354720600815844403
```

The difference is not equal to zero because the toolbox approximates the number `b=0.1` with 32 digits. This approximation produces round-off errors because the floating point number 0.1 is different from the rational number 1/10. When you compute the difference `a - b`, the toolbox actually computes the difference as follows:

```
a - vpa(b, 32)

ans =
0.000000000000000000086736173798840354720600815844403
```

Suppose, you convert a number to a symbolic object, and then perform VPA operations on that object. The results can depend on the conversion technique that you used to convert a floating-point number to a symbolic object. The `sym` function lets you choose the conversion technique by specifying the optional second argument, which can be '`r`', '`f`', '`d`' or '`e`'. The default is '`r`'. For example, convert the constant π=3.141592653589793... to a symbolic object:

```
r = sym(pi)
f = sym(pi, 'f')
d = sym(pi, 'd')
e = sym(pi, 'e')

r =
pi

f =
884279719003555/281474976710656

d =
3.1415926535897931159979634685442

e =
pi - (198*eps)/359
```

Compute these numbers with the 4 digits VPA precision. Three of the four numeric approximations give the same result:

```
vpa(r, 4)
vpa(f, 4)
vpa(d, 4)
vpa(e, 4)

ans =
3.142

ans =
3.142

ans =
3.142

ans =
3.142 - 0.5515*eps
```

Now, increase the VPA precision to 40 digits. The numeric approximation of 1/10 depends on the technique that you used to convert 1/10 to the symbolic object:

```
vpa(r, 40)
vpa(f, 40)
vpa(d, 40)
vpa(e, 40)

ans =
3.141592653589793238462643383279502884197

ans =
3.141592653589793115997963468544185161591

ans =
3.1415926535897931159979634685442

ans =
3.141592653589793238462643383279502884197 -...
0.5515320334261838440111420612813370473538*eps
```

See Also `digits`, `double`

Purpose

Compute Riemann zeta function

Syntax

```
Y = zeta(X)
Y = zeta(n, X)
```

Description

`Y = zeta(X)` evaluates the zeta function at the elements of `X`, a numeric matrix, or a symbolic matrix. The zeta function is defined by

$$\zeta(w) = \sum_{k=1}^{\infty} \frac{1}{k^w}$$

`Y = zeta(n, X)` returns the `n`-th derivative of `zeta(X)`.

Examples

Compute the Riemann zeta function for the number:

```
zeta(1.5)
```

The result is:

```
ans =
    2.6124
```

Compute the Riemann zeta function for the matrix:

```
zeta(1.2:0.1:2.1)
```

The result is:

```
ans =
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
```

Compute the Riemann zeta function for the matrix of the symbolic expressions:

```
syms x y;
zeta([x 2; 4 x + y])
```

The result is:

```
ans =
[ zeta(x),      pi^2/6]
[ pi^4/90, zeta(x + y)]
```

Differentiate the Riemann zeta function:

```
diff(zeta(x), x, 3)
```

The result is:

```
ans =
zeta(3, x)
```

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 `symvar`. If `f = f(z)`, then `ztrans(f)` returns a function of `w`.

$$F = F(w)$$

`F = ztrans(f, w)` makes `F` a function of the symbol `w` instead of the default `z`.

$$F(w) = \sum_{0}^{\infty} \frac{f(n)}{w^n}$$

`F = ztrans(f, k, w)` takes `f` to be a function of the symbolic variable `k`.

$$F(w) = \sum_{0}^{\infty} \frac{f(k)}{w^k}$$

Examples

Z-Transform	MATLAB Operation

$f(n) = n^4$
$Z[f] = \sum_{n=0}^{\infty} f(n) z^{-n}$
$= \frac{z(z^3 + 11z^2 + 11z + 1)}{(z-1)^5}$
	`syms n;`
	`f = n^4;`
	`ztrans(f)`
	
	`ans =`
	`(z^4 + 11*z^3 + 11*z^2`
	`+ z)/(z - 1)^5`

$g(z) = a^z$
$Z[g] = \sum_{z=0}^{\infty} g(z) w^{-z}$
$= \frac{w}{w-a}$
	`syms a z;`
	`g = a^z;`
	`ztrans(g)`
	
	`ans =`
	`-w/(a - w)`

$f(n) = sin(an)$
$Z[f] = \sum_{n=0}^{\infty} f(n) w^{-n}$
$= \frac{w \sin a}{1 - 2w \cos a + w^2}$
	`syms a n w;`
	`f = sin(a*n);`
	`ztrans(f, w)`
	
	`ans =`
	`(w*sin(a))/(w^2 -`
	`2*cos(a)*w + 1)`

See Also

`fourier`, `iztrans`, `laplace`

Index