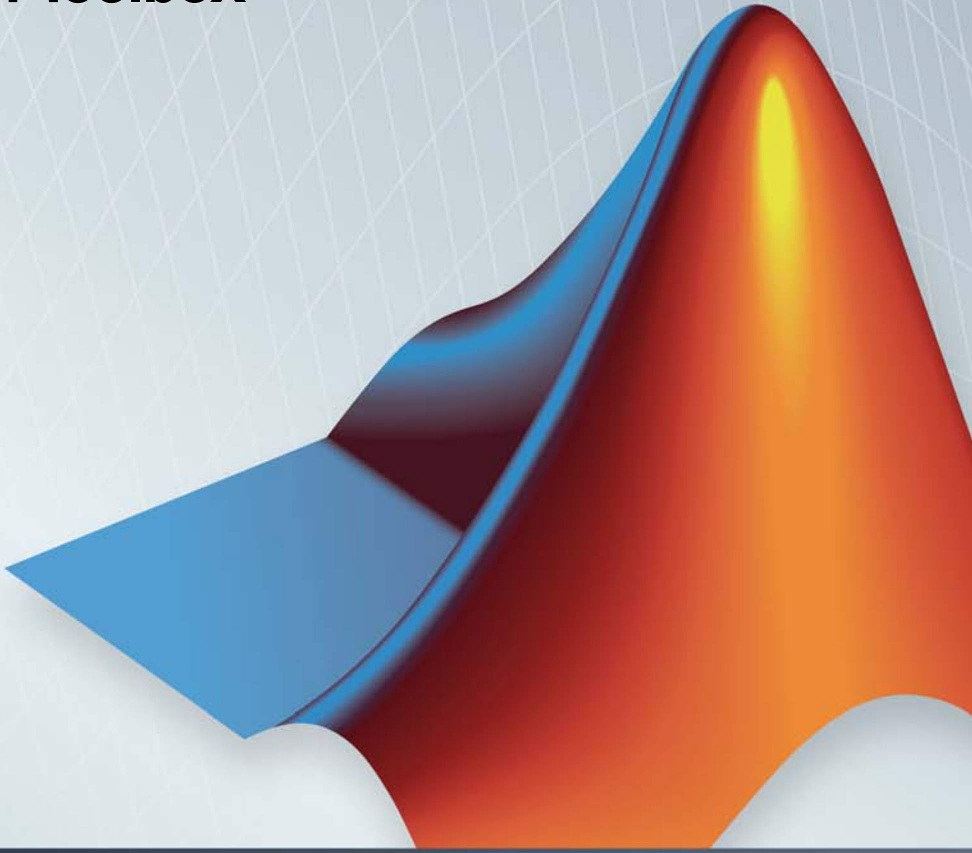
Symbolic Math Toolbox™

User's Guide

R2013a



MATLAB®

MathWorks®

How to Contact MathWorks

Symbolic Math Toolbox™ User's Guide

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)
April 2011	Online only	Revised for Version 5.6 (Release 2011a)
September 2011	Online only	Revised for Version 5.7 (Release 2011b)
March 2012	Online only	Revised for Version 5.8 (Release 2012a)
September 2012	Online only	Revised for Version 5.9 (Release 2012b)
March 2013	Online only	Revised for Version 5.10 (Release 2013a)

Acknowledgments

Contents

1 Getting Started

2 Using Symbolic Math Toolbox Software

Getting Started

Product Description

Perform symbolic math computations

Symbolic Math Toolbox™ provides functions for solving and manipulating symbolic math expressions and performing variable-precision arithmetic. You can analytically perform differentiation, integration, simplification, transforms, and equation solving. You can also generate code for MATLAB®, Simulink®, and Simscape™ from symbolic math expressions.

Symbolic Math Toolbox includes the MuPAD language, which is optimized for handling and operating on symbolic math expressions. It provides libraries of MuPAD functions in common mathematical areas such as calculus and linear algebra and in specialized areas such as number theory and combinatorics. You can also write custom symbolic functions and libraries in the MuPAD language. The MuPAD Notebook app lets you document symbolic math derivations with embedded text, graphics, and typeset math. You can share the annotated derivations as HTML or as a PDF.

Key Features

- Functions for symbolic equation solving, differentiation, integration, and simplification, as well as for computing transforms and special functions
- Variable-precision arithmetic
- MuPAD symbolic math language
- MuPAD Notebook app with embedded text, graphics, and typeset math for documenting and managing computations performed in the MuPAD language
- MuPAD function libraries for common mathematical areas such as calculus and linear algebra and for specialized areas such as number theory and combinatorics
- Functions for generating code for MATLAB, Simulink, Simscape, C, Fortran, MathML, and TeX from symbolic expressions

Access Symbolic Math Toolbox Functionality

Work 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.

Work from MuPAD

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.

A 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 "Call Built-In MuPAD Functions from MATLAB Command Window" on page 3-32

If you are a new user of the MuPAD Notebook Interface, see Getting Started with MuPAD.

Symbolic Objects

In this section...

Overview of Symbolic Objects

Symbolic objects are a special MATLAB data type introduced by the Symbolic Math Toolbox software. They enable 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 are symbolic variables, symbolic numbers, symbolic expressions, symbolic matrices, and symbolic functions.

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 "Create Symbolic Variables and Expressions" on page 1-8 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 15 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 “Estimate Precision of Numeric to Symbolic Conversions” on page 1-22.

Create Symbolic Variables and Expressions

In this section...

Create 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 15 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.

Create Symbolic Expressions

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

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

The command

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

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

```
f = phi^2 - phi - 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;
```

Tip To create a symbolic expression that is a constant, you must use the `sym` command. Do not use the `syms` function 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.

Create Symbolic Functions

You also can use `sym` and `syms` to create symbolic functions. For example, you can create an arbitrary function `f(x, y)` where `x` and `y` are function variables. The simplest way to create an arbitrary symbolic function is to use `syms`:

```
syms f(x, y)
```

This syntax creates the symbolic function `f` and symbolic variables `x` and `y`.

Alternatively, you can use `sym` to create a symbolic function. Note that `sym` only creates the function. It does not create symbolic variables that represent its arguments. You must create these variables before creating a function:

```
syms x y;
f(x, y) = sym('f(x, y)');
```

If instead of an arbitrary symbolic function you want to create a function defined by a particular mathematical expression, use this two-step approach. First create symbolic variables representing the arguments of the function:

```
syms x y
```

Then assign a mathematical expression to the function. In this case, the assignment operation also creates the new symbolic function:

```
f(x, y) = x^3*y^3

f(x, y) =
x^3*y^3
```

After creating a symbolic function, you can differentiate, integrate, or simplify it, substitute its arguments with values, and perform other mathematical operations. For example, find the second derivative on `f(x, y)` with respect to variable `y`. The result `d2fy` is also a symbolic function.

```
d2fy = diff(f, y, 2)

d2fy(x, y) =
6*x^3*y
```

Now evaluate `f(x, y)` for `x = y + 1`:

```
f(y + 1, y)

ans =
y^3*(y + 1)^3
```

Create 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 “Delete Symbolic Objects and Their Assumptions” on page 1-36 for more information.

Create a Matrix of Symbolic Variables

Use 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
```

To check if the sum of the elements of the first row equals the sum of the elements of the second column, use the `logical` function:

```
logical(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.

Generate 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 that 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]
```

Create 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 "Estimate Precision of Numeric to Symbolic Conversions" on page 1-22.

Find Symbolic Variables in Expressions, Functions, Matrices

To find symbolic variables in an expression, function, or matrix, use `symvar`. For example, find all symbolic variables in symbolic expressions `f` and `g`:

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

ans =
[ n, x]
```

Here, `symvar` sorts all returned variables alphabetically. Similarly, you can find the symbolic variables in `g` by entering:

```
symvar(g)

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

`symvar` also can return the first `n` symbolic variables found in a symbolic expression, matrix, or function. To specify the number of symbolic variables that you want `symvar` to return, use the second parameter of `symvar`. For example, return the first two variables found in symbolic expression `g`:

```
symvar(g, 2)

ans =
[ t, b]
```

Notice that the first two variables in this case are not `a` and `b`. When you call `symvar` with two arguments, it sorts symbolic variables by their proximity to `x`.

You also can find symbolic variables in a function:

```
syms x y w z
f(w, z) = x*w + y*z;
symvar(f)

ans =
[ w, x, y, z]
```

When you call `symvar` with two arguments, it returns the function inputs in front of other variables:

```
symvar(f, 2)

ans =
[ w, z]
```

Find a Default Symbolic Variable

If you do not specify an independent variable when performing substitution, differentiation, or integration, MATLAB uses a *default* variable. The default variable is typically the one closest alphabetically to `x` or, for symbolic functions, the first input argument of a function. To find which variable is chosen as a default variable, use the `symvar(f, 1)` command. For example:

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

ans =
t

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

ans =
tx
```

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

Perform Symbolic Computations

In this section...

Simplify Symbolic Expressions

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

```
phi = sym('(1 + sqrt(5))/2');
f = phi^2 - phi - 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 parentheses 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 2-33.

Substitutions in Symbolic Expressions

Substitute 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` = 1/3:

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

ans =
2/9
```

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

```
f

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

Substitute 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)
```

Substitute 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)
```

Substitute 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:

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

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

Substitute 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, see "Substitution".

Estimate 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', which stands for rational approximation "Conversion to Rational Symbolic Form" on page 1-23.

Conversion to Floating-Point Symbolic Form

The 'f' option to sym converts double-precision floating-point numbers to exact numeric values 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
```

Conversion 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
```

Conversion 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
```

Conversion 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
```

Differentiate 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 2-3.

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 its 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 "Find a Default Symbolic Variable" on page 1-15.

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
```

Integrate 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 2-13.

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 "Find a Default Symbolic Variable" on page 1-15.

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 = sin(x)^(1/sqrt(n));
int(f, n, 1, 10)
```

```
Warning: Explicit integral could not be found.

ans =
int(sin(x)^(1/n^(1/2)), n == 1..10)
```

Solve 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 "Equation Solving".

Solve Algebraic Equations with One Symbolic Variable

Use the double equal sign (==) to define an equation. Then you can `solve` the equation by calling the solve function. For example, solve this equation:

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

ans =
 1
 2
 3
```

If you do not specify the right side of the equation, `solve` assumes that it is zero:

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

ans =
 1
 2
 3
```

Solve Algebraic Equations with Several Symbolic Variables

If an equation contains several symbolic variables, you can specify a variable for which this equation should be solved. For example, solve this multivariable equation with respect to `y`:

```
syms x y
solve(6*x^2 - 6*x^2*y + x*y^2 - x*y + y^3 - y^2 == 0, 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 "Find a Default Symbolic Variable" on page 1-15.

Solve 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
```

Create 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

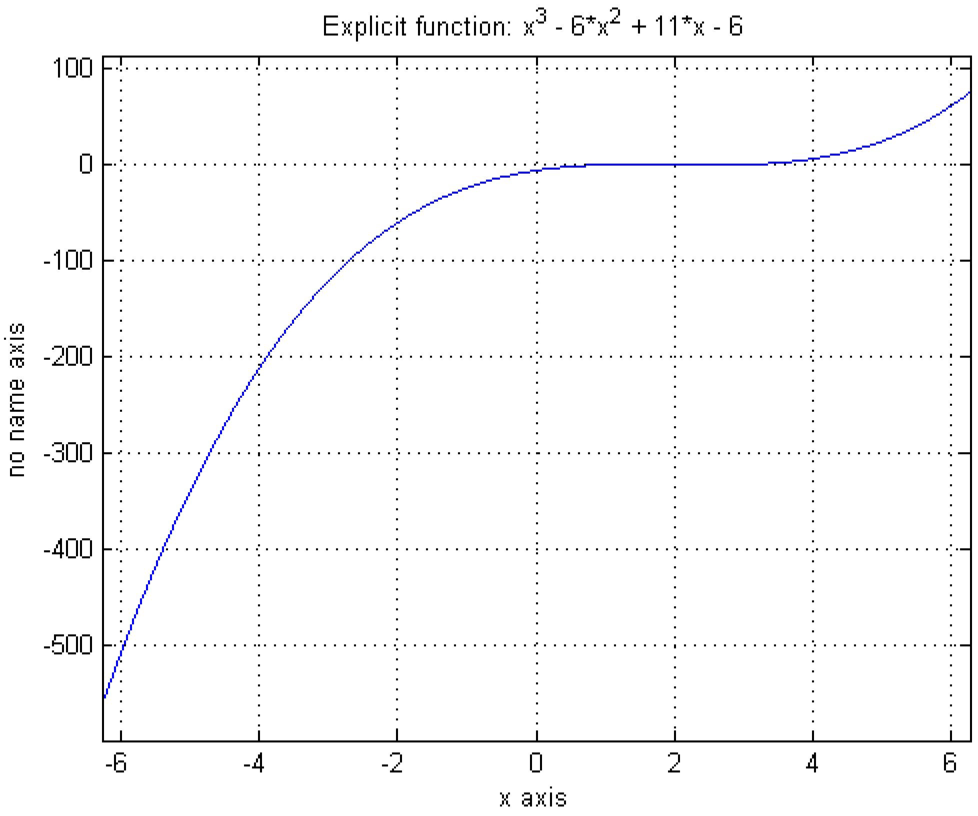
Explicit Function Plot

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:

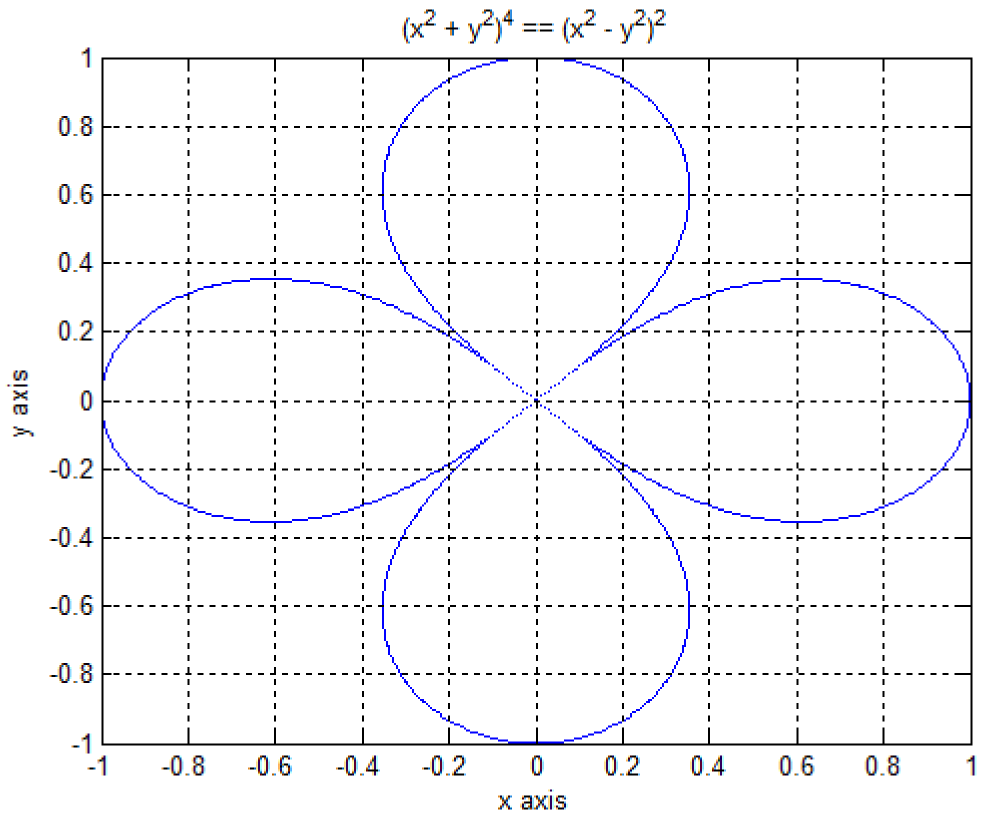
```
xlabel('x axis')
ylabel('no name axis')
title('Explicit function: x^3 - 6*x^2 + 11*x - 6')
grid on
hold off
```



Implicit Function Plot

Using `ezplot`, you can also plot equations. For example, plot the following equation over $-1 < x < 1$:

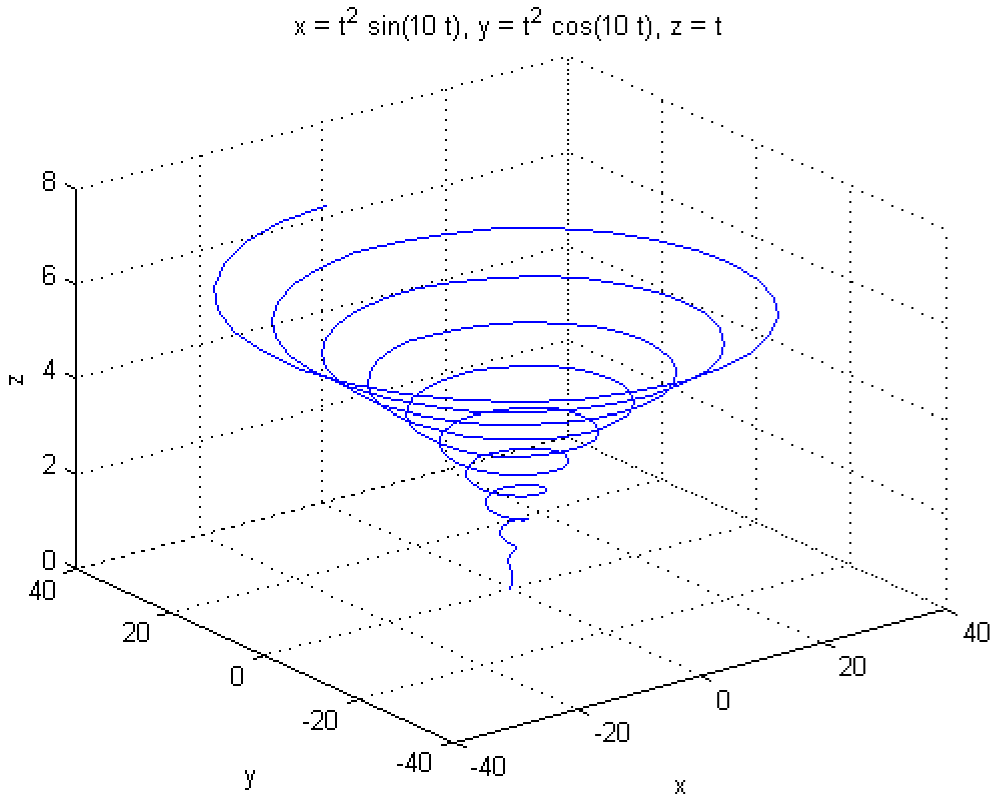
```
syms x y
ezplot((x^2 + y^2)^4 == (x^2 - y^2)^2, [-1 1])
hold on
xlabel('x axis')
ylabel('y axis')
grid on
hold off
```



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:

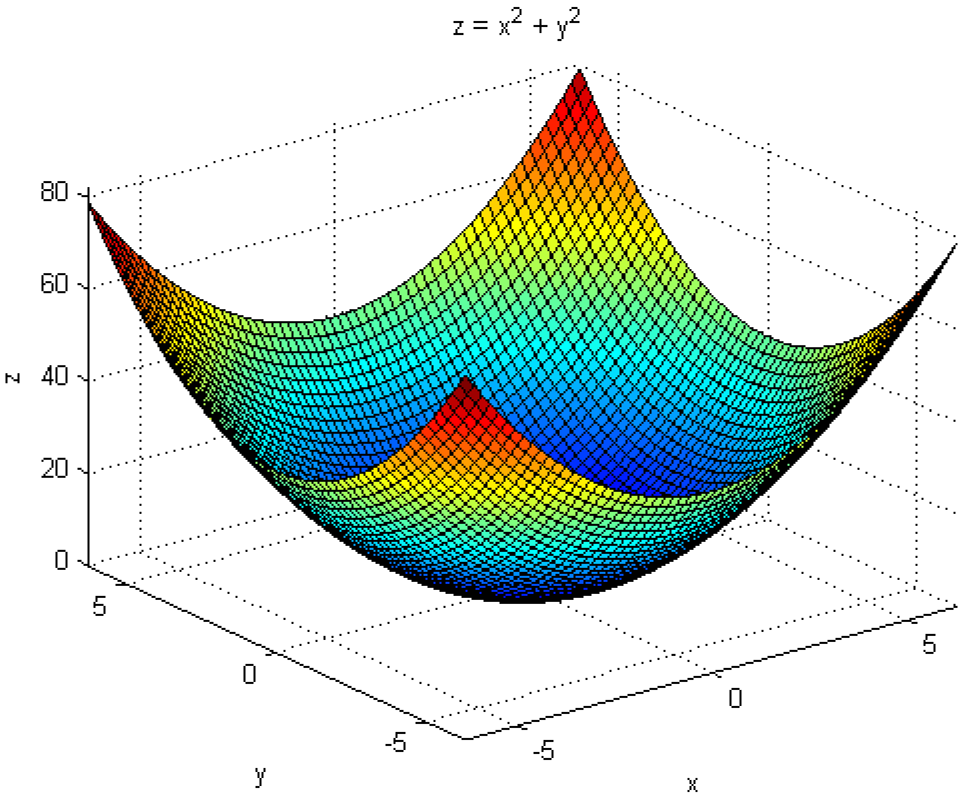
```
syms t
ezplot3(t^2*sin(10*t), t^2*cos(10*t), t)
```



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 on Symbolic Objects

In this section...

Default Assumption

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

```
syms z
```

then MATLAB assumes that `z` is a complex variable. You can always check if a symbolic variable is assumed to be complex or real by using `assumptions`. If `z` is complex, `assumptions(z)` returns an empty symbolic object:

```
assumptions(z)

ans =
[ empty sym ]
```

Set Assumptions

To set an assumption on a symbolic variable, use the `assume` function. For example, assume that the variable `x` is nonnegative:

```
syms x
assume(x >= 0)
```

`assume` replaces all previous assumptions on the variable with the new assumption. If you want to add a new assumption to the existing assumptions, use `assumeAlso`. For example, add the assumption that `x` is also an integer. Now the variable `x` is a nonnegative integer:

```
assumeAlso(x,'integer')
```

assume and assumeAlso let you state that a variable or an expression belongs to one of these sets: integers, rational numbers, and real numbers.

Alternatively, you can set an assumption while declaring a symbolic variable using sym or syms. For example, create the real symbolic variables a and b, and the positive symbolic variable c:

```
a = sym('a', 'real');
b = sym('b', 'real');
c = sym('c', 'positive');
```

or more efficiently:

```
syms a b real
syms c positive
```

There are two assumptions that you can assign to a symbolic object within the sym or syms command: real and positive.

Check Existing Assumptions

To see all assumptions set on a symbolic variable, use the assumptions function with the name of the variable as an input argument. For example, this command returns the assumptions currently used for the variable x:

```
assumptions(x)
```

To see all assumptions used for all symbolic variables in the MATLAB workspace, use assumptions without input arguments:

```
assumptions
```

For details, see "Check Assumptions Set On Variables" on page 3-44.

Delete Symbolic Objects and Their Assumptions

Symbolic objects and their assumptions are stored separately. When you set an assumption that x is real using

```
syms x
assume(x,'real')
```

you actually create a symbolic object x and the assumption that the object is real. The object is stored in the MATLAB workspace, and the assumption is stored in the symbolic engine. When you delete a symbolic object from the MATLAB workspace using

```
clear x
```

the assumption that x is real stays in the 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 == 0, x)

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 details on clearing symbolic variables, see “Clear Assumptions and Reset the Symbolic Engine” on page 3-43.

Using Symbolic Math Toolbox Software

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 2-5.

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

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

returns

```
ans =
0
```

If you just enter

```
diff(5)
```

MATLAB returns

```
ans =
     []
```

because `5` is not a symbolic expression.

Derivatives of Expressions with Several Variables

To differentiate an expression that contains more than one symbolic variable, 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 "Find a Default Symbolic Variable" on page 1-15. 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 `symvar`:

```
symvar(f, 1)

ans =
t
```

Calculate the second derivative of `f` with respect to `t`:

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

This command 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
```

This table 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)]
```

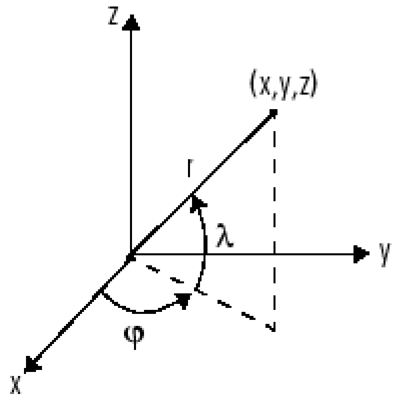
The command

```
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 = simplify(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 `simplify` to make trigonometric substitutions and reductions (simplifications).

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)`

Mathematical Operator	MATLAB Command
$\frac{d^2f}{db^2}$	`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)
```

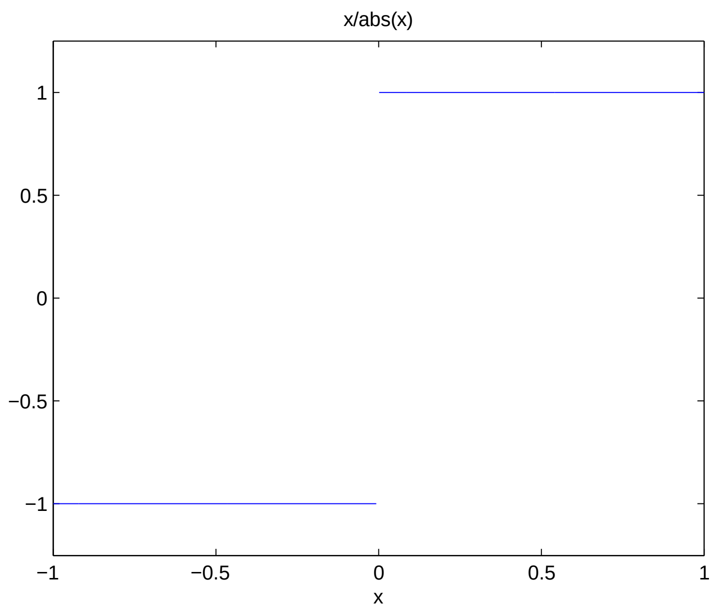
which returns

```
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')

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')

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

Integration

If `f` is a symbolic expression, then

```
int(f)
```

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

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

uses the symbolic object `v` as the variable of integration, rather than the variable determined by `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
```

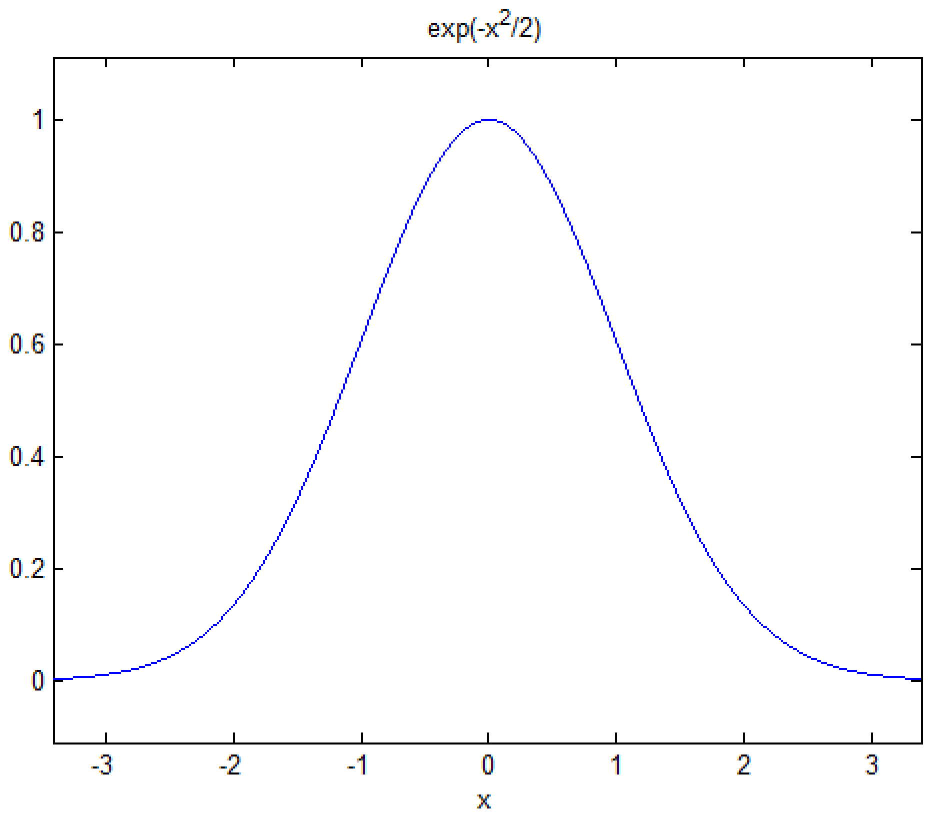
Integration with Real Parameters

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, use `assume` to set an assumption on `a`:

```
syms a
assume(a > 0);
```

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)
```

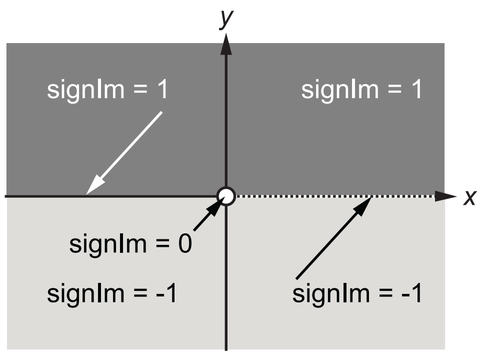
`syms` is used with the `clear` option to clear the all assumptions on `a`. For more information about symbolic variables and assumptions on them, see "Delete Symbolic Objects and Their Assumptions" on page 1-36.

The preceding commands produce the complex output

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

The function `signIm` is defined as:

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



To evaluate `F` at `a = 1 + i`, enter

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

```
g =
pi*(1/2 - i/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, 'Order', 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 expansion point is `a = 0`.

The command

```
pretty(T)
```

prints `T` in a format resembling typeset mathematics:

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

These commands

```
syms x
g = exp(x*sin(x));
t = taylor(g, 'ExpansionPoint', 2, 'Order', 12);
```

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
```

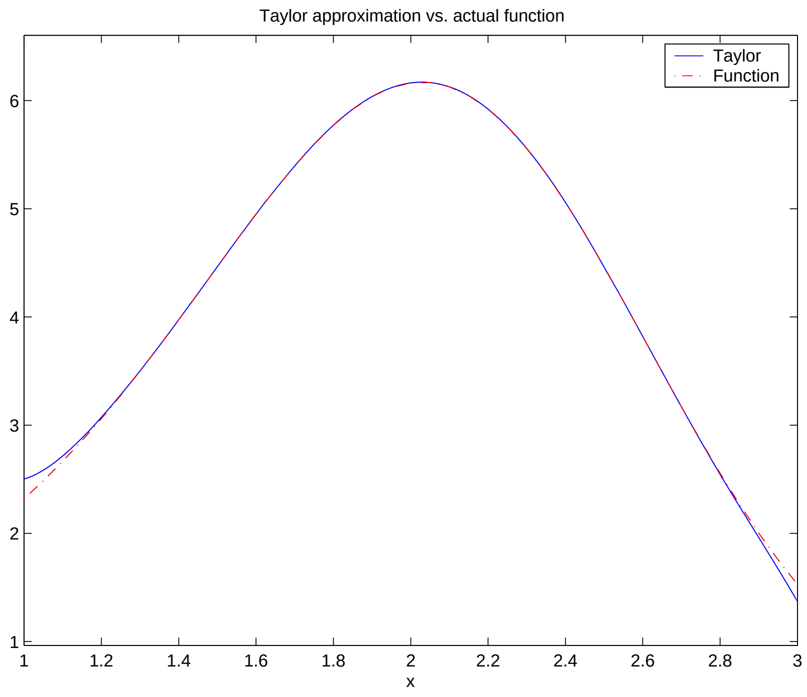
to find that `t` has about 100,000 characters in its printed form. In order to proceed with using `t`, first simplify its presentation:

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

```
ans =
           1        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 is given to Professor Gunnar Bäckstrøm of UMEA in Sweden for this example.

Find Asymptotes, Critical and Inflection Points

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

In this section…

Define a 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
```

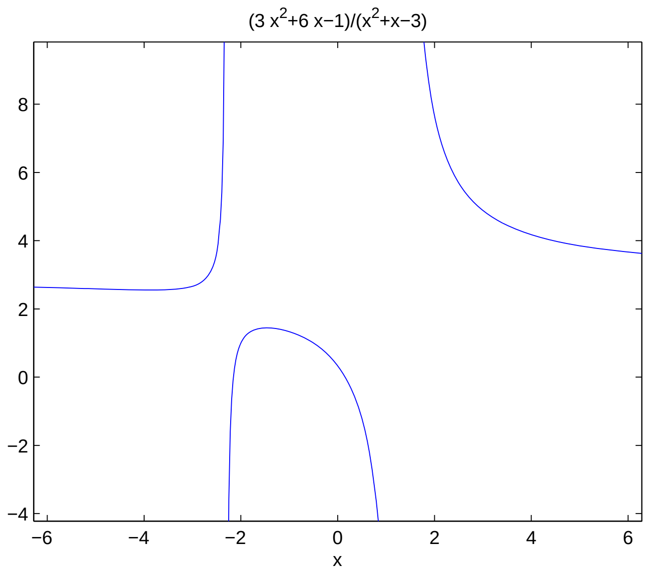
This returns

```
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.



Find 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

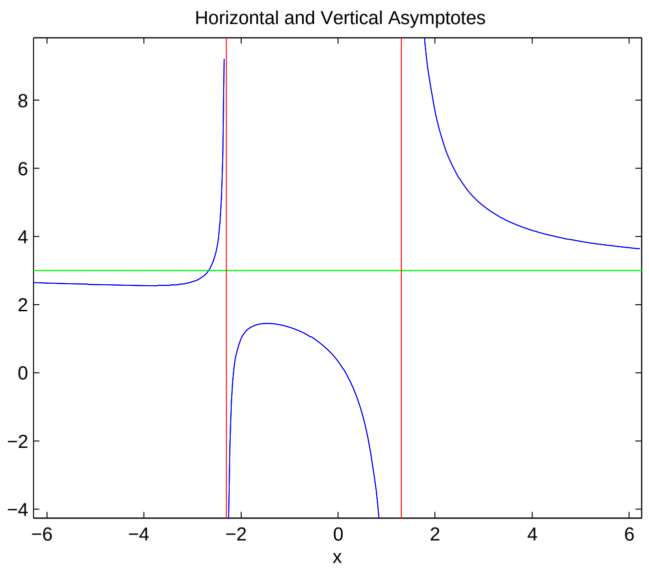
$$x = \frac{-1-\sqrt{13}}{2}.$$

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)
```

Find 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)

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)

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)

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

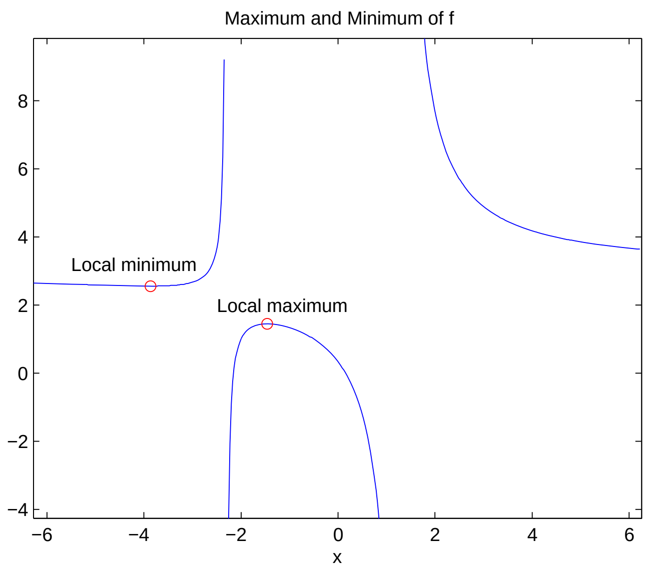
$$x_2 = \frac{-8+\sqrt{13}}{3}.$$

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.



Find 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);
```

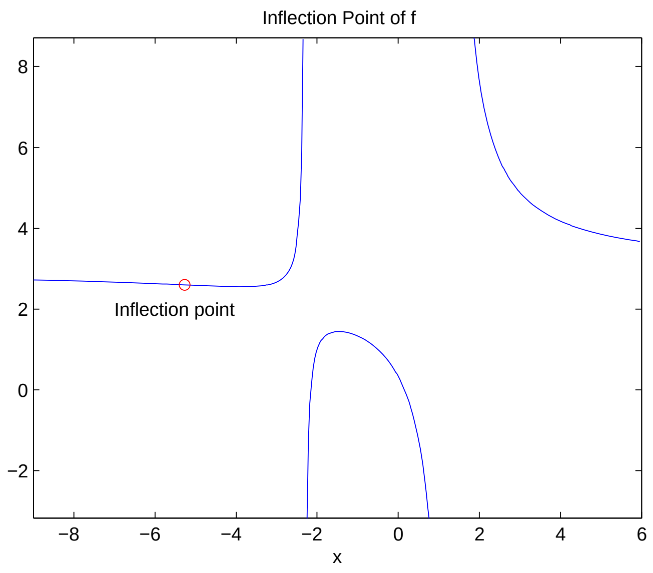
To see the symbolic expression for the inflection point, enter

```
pretty(simplify(inflec_pt))


     2/3   1/3            1/2 1/3    2/3   1/3      1/2      1/3
    2    13    (13 - 3 13   )       2    13    (3 13    + 13)
  - -------------------------- - -------------------------- - 8/3
                 6                              6
```

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.


Inflection Point of f
Inflection point
x
8
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

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`, and `simplify`.

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)`

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

`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 =`

f	factor(f)
	`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]
```

simplifyFraction

The statement `simplifyFraction(f)` represents the expression `f` as a fraction where both the numerator and denominator are polynomials whose greatest common divisor is 1. The `Expand` option lets you expand the numerator and denominator in the resulting expression.

`simplifyFraction` is significantly more efficient for simplifying fractions than the general simplification function `simplify`.

f	simplifyFraction(f)

`syms x`
`f =(x^3 - 1)/(x`
`- 1);`
	`simplifyFraction(f)`
	
	`ans =`
	`x^2 + x + 1`

`syms x`
`f = (x^3 - x^2*y -`
`x*y^2 + y^3)/(x^3`
`+ y^3);`
	`simplifyFraction(f)`
	
	`ans =`
	`(x^2 - 2*x*y + y^2)/(x^2 - x*y + y^2)`

`syms x`	`simplifyFraction(f)`

f	simplifyFraction(f)

`f = (1 - exp(x)^4)/(1 + exp(x))^4;`	`ans =`
	`(exp(2*x) - exp(3*x) - exp(x) + 1)/(exp(x) + 1)^3`
	
	`simplifyFraction(f, 'Expand', true)`
	
	`ans =`
	`(exp(2*x) - exp(3*x) - exp(x) + 1)/(3*exp(2*x) + exp(3*x) + 3*exp(x) + 1)`

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 = (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)`

f	simplify(f)

`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, 'Steps', n)` where `n` is a positive integer that controls how many steps `simplify` takes. By default, `n = 1`. For example,

```
syms x
z = (cos(x)^2 - sin(x)^2)*sin(2*x)*(exp(2*x) - 2*exp(x) + 1)/(exp(2*x) - 1);

simplify(z)

ans =
(sin(4*x)*(exp(x) - 1))/(2*(exp(x) + 1))

simplify(z, 'Steps', 30)

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

Substitute with subexpr

These commands solve the equation `x^3 + a*x + 1 = 0` for the variable `x`:

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

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))

(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

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

This long expression has many repeated pieces, or subexpressions. The `subexpr` function allows you to save these common subexpressions as well as the symbolic object rewritten in terms of the subexpressions. The subexpressions are saved in a column vector called `sigma`.

Continuing with the example

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

returns

```
sigma =
(a^3/27 + 1/4)^(1/2) - 1/2


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

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

```
 sigma
```

which returns

```
sigma =
(a^3/27 + 1/4)^(1/2) - 1/2
```

You can use other variable names instead of `sigma`. For example, replace the common subexpression in `s` by `u`:

```
r1 = subexpr(s,'u')
```

```
u =
(a^3/27 + 1/4)^(1/2) - 1/2

r1 =
                                              u^(1/3) - a/(3*u^(1/3))
 (3^(1/2)*(a/(3*u^(1/3)) + u^(1/3))*i)/2 + a/(6*u^(1/3)) - u^(1/3)/2
 a/(6*u^(1/3)) - (3^(1/2)*(a/(3*u^(1/3)) + u^(1/3))*i)/2 - u^(1/3)/2
```

`subexpr` does not let you control which subexpressions need to be replaced.

Substitute with subs

Use this code to find 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, also appear in a different order.

Replace the rather lengthy expression (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2) throughout v and E:

```
syms S
v = subs(v,(a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2), S)
E = subs(E,(a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2), S)

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

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

Simplify v:

```
v = simplify(v)

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]
```

Note that subs does not assign (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2) to S:

```
S

S =
S
```

Assign this expression to S:

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

Substitute variables a, b, and c in S with the values 1, 2, and 3:

```
subs(S, {a, b, c}, {1, 2, 3})

ans =
3^(1/2)
```

Substitute a, b, and c in v with the same values. Note that you must call subs twice. The first call, subs(v), replaces S in v with the expression (a^2 - a*b - a*c + b^2 - b*c + c^2)^(1/2). The second call replaces the variables a, b, and c in v with the values 1, 2, and 3:

```
subs(subs(v), {a, b, c}, {1, 2, 3})

ans =
[   3^(1/2)/2 - 1/2, - 3^(1/2)/2 - 1/2, 1]
[ - 3^(1/2)/2 - 1/2,   3^(1/2)/2 - 1/2, 1]
[                    1,                     1, 1]
```

These substitutions do not modify a, b, c, S, and v:

```
[a, b, c]

ans =
[ a, b, c]

S

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

v

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]
```

To modify the original values S and v, assign the results returned by subs to S and v. This approach does not modify a, b, and c.

```
S = subs(S, {a, b, c}, {1, 2, 3})

S =
3^(1/2)

v = subs(subs(v), {a, b, c}, {1, 2, 3})

v =
[   3^(1/2)/2 - 1/2, - 3^(1/2)/2 - 1/2, 1]
[ - 3^(1/2)/2 - 1/2,   3^(1/2)/2 - 1/2, 1]
[                 1,                 1, 1]
```

Alternatively, you can assign values to the variables a, b, and c:

```
a = 1; b = 2; c = 3;
```

The new values of a, b, and c now exist in the MATLAB workspace:

```
[a, b, c]
```

```
ans =
     1     2     3
```

Use `subs` with one input argument to evaluate `S` and `v` for these values:

```
S = subs(S)

S =
3^(1/2)

v = subs(v)

v =
[   3^(1/2)/2 - 1/2, - 3^(1/2)/2 - 1/2, 1]
[ - 3^(1/2)/2 - 1/2,   3^(1/2)/2 - 1/2, 1]
[                 1,                 1, 1]
```

Combine subs and double for Numeric Evaluations

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

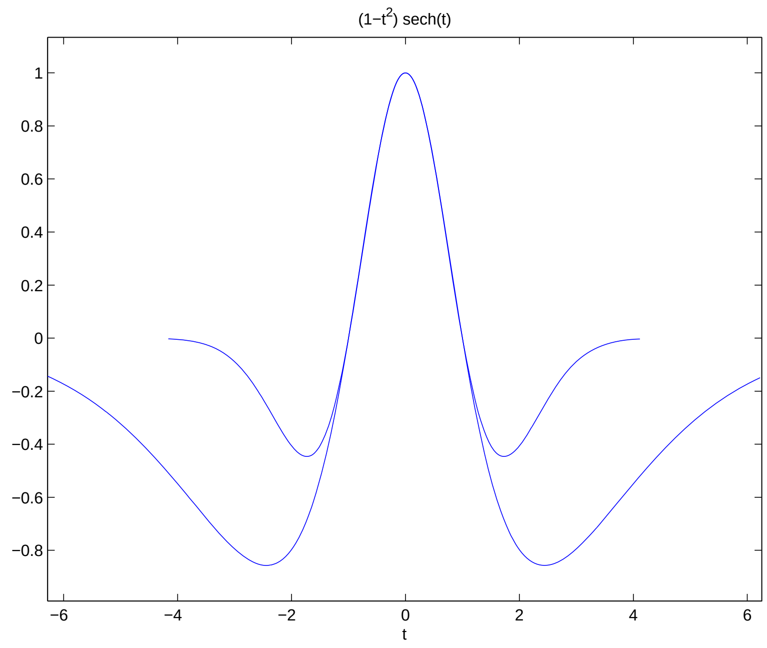
```
syms t
M = (1 - t^2)*exp(-1/2*t^2);
P = (1 - t^2)*sech(t);
```

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

One approach is to type

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

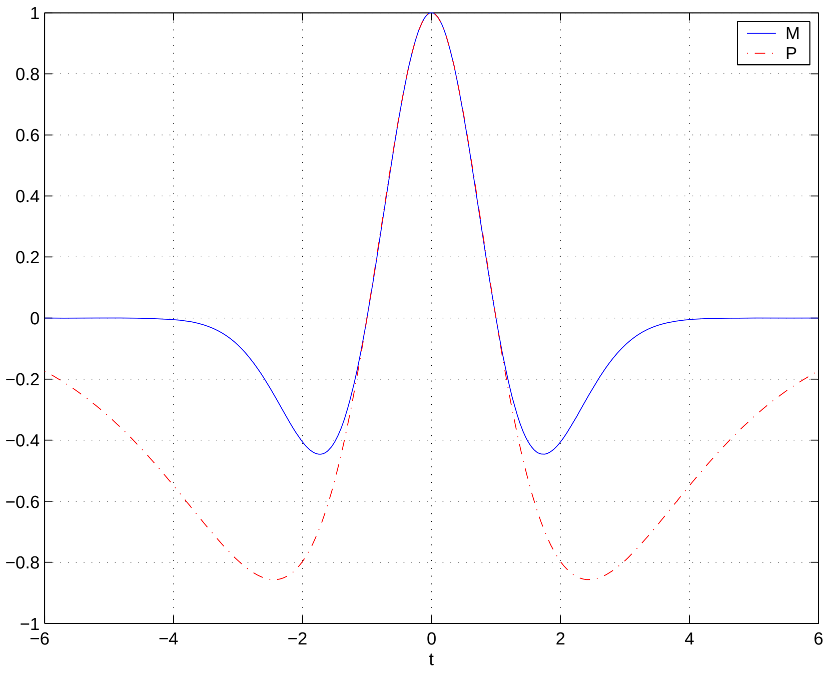
but this plot does not readily help you identify the curves.



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`.


M
P
t
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−6
−4
−2
0
2
4
6

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 exact symbolic arithmetic
VPA	MuPAD 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 =
0.83333333333333333333333333333
```

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.

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);
```

Conversion 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`.

Accuracy of Numeric Computations

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 =
262537412640768743.99999999999925007259 72
```

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

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 `simplify` function

```
A = simplify(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 = simplify(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,                0.2]
```

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

```
inv(V)
```

which returns

```
ans =
[    9.0,   -36.0,    30.0]
[  -36.0,   192.0,  -180.0]
[   30.0,  -180.0,   180.0]
```

shows the loss of two digits. So does

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

which gives

```
ans =
 2160.000000000018
```

and

```
V\b
```

which is

```
ans =
   3.0
 -24.0
  30.0
```

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


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

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

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

return

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

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

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

$x^2 - \frac{64}{45}x + \frac{253}{2160}$ which is the quadratic factor in `factor(charpoly(H, x))`:

```
syms x
g = factor(charpoly(H, x))/x;
solve(g)

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

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 =charpoly(R, x);
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 = charpoly(S, x)
```

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 =
 -1020.053214255892
  -0.17053529728769
 0.2180398054830161
  999.9469178604428
  1000.120698293384
  1019.524355263202
  1019.993550129163
  1020.420188201505
```

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

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

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

Symbolic Math Toolbox commands

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

return

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

You can simplify this expression using `simplify`:

```
G = simplify(G)

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

Next, the command

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

produces

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

You can rewrite `g` in terms of exponents:

```
g = rewrite(g, 'exp')

g =
 exp(-t*i)
  exp(t*i)
```

Jordan Canonical Form

The Jordan canonical form results from attempts to convert a matrix to its diagonal form 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 converted to diagonal forms. 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

Singular value decomposition expresses an m-by-n matrix A as A = U*S*V'. Here, S is an m-by-n diagonal matrix with singular values of A on its diagonal. The columns of the m-by-m matrix U are the left singular vectors for corresponding singular values. The columns of the n-by-n matrix V are the right singular vectors for corresponding singular values. V' is the Hermitian transpose (the complex conjugate of the transpose) of V.

To compute the singular value decomposition of a matrix, use svd. This function lets you compute singular values of a matrix separately or both singular values and singular vectors in one function call. To compute singular values only, use svd without output arguments

```
svd(A)
```

or with one output argument

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

To compute singular values and singular vectors of a matrix, use three output arguments:

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

svd returns two unitary matrices, U and V, the columns of which are singular vectors. It also returns a diagonal matrix, S, containing singular values on its diagonal. The elements of all three matrices are floating-point numbers. The accuracy of computations is determined by the current setting of digits.

Create 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 = 3;
for i=1:n
    for j=1:n
      A(i,j) = sym(1/(i-j+1/2));
   end
end
```

For n = 3, the matrix is

```
A

A =
[   2,  -2, -2/3]
[ 2/3,   2,   -2]
[ 2/5, 2/3,    2]
```

Compute the singular values of this matrix. If you use `svd` directly, it will return exact symbolic result. For this matrix, the result is very long. If you prefer a shorter numeric result, convert the elements of `A` to floating-point numbers using `vpa`. Then use `svd` to compute singular values of this matrix using variable-precision arithmetic:

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

S =
 3.1387302525015353960741348953506
 3.0107425975027462353291981598225
 1.6053456783345441725883965978052
```

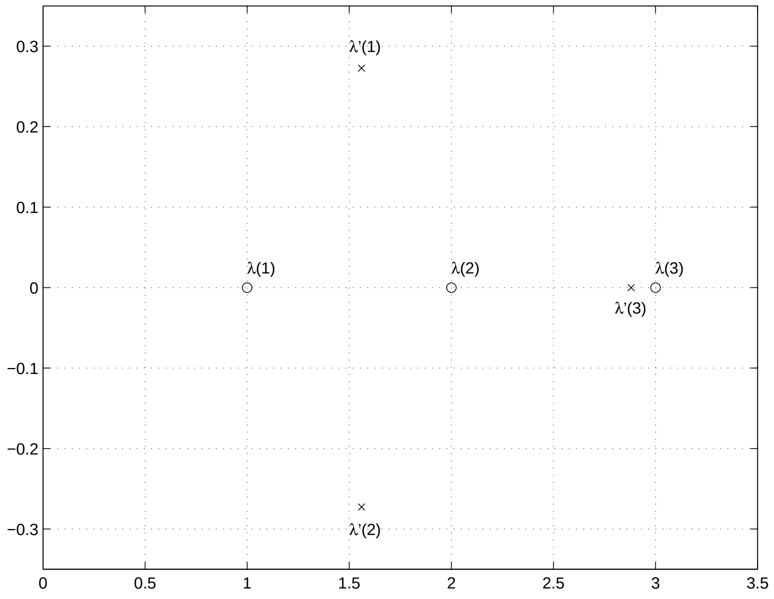
Now, compute the singular values and singular vectors of `A`:

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

U =
[  0.53254331027335338470683368360204,  0.76576895948802052989304092179952, 0.360548919520962147911898877283
[ -0.82525689650849463222502853672224,  0.37514965283965451993171338605042, 0.4221537548565148952248803191 73
[  0.18801243961043281839917114171742, -0.52236064041897439447429784257224, 0.8317395529207519217842187433 14

S =
[ 3.1387302525015353960741348953506,                                 0,                                 0]
[                                 0, 3.0107425975027462353291981598225,                                 0]
[                                 0,                                 0, 1.6053456783345441725883965978052]

V =
[  0.18801243961043281839917114171742,  0.52236064041897439447429784257224, 0.8317395529207519217842187433 14
[ -0.82525689650849463222502853672224, -0.37514965283965451993171338605042, 0.4221537548565148952248803191 73
[  0.53254331027335338470683368360204, -0.76576895948802052989304092179952,
0.36054891952096214791189887728353]
```

Eigenvalue Trajectories

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

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



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

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

```
A = gallery(3)

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

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

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

produce

```
e =
   1.000000000010722
   1.999999999991790
   2.999999999997399
```

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

```
B = sym(A);
e = eig(B)'
p = charpoly(B, x)
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 = charpoly(A, x)
```

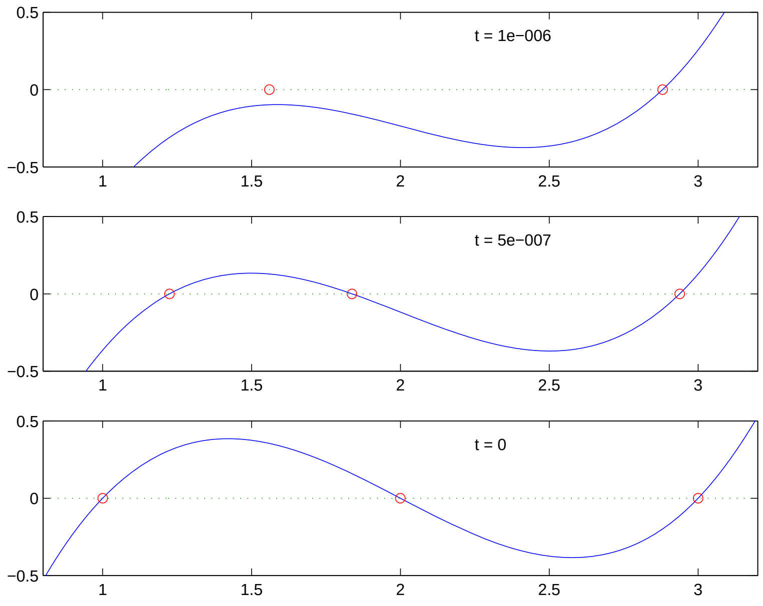
gives

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

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

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

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



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

The next statements compute and display the actual eigenvalues

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

sigma =
(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:

```
+-                              -+
|     t        1/3               |
|     - + sigma    - #3 + 2      |
|     3                          |
|                                |
|           1/3                  |
|  t   sigma                     |
|  - - -------- + #1 + 2 - #2    |
|  3      2                      |
|                                |
|           1/3                  |
|  t   sigma                     |
|  - - -------- + #1 + 2 + #2    |
|  3      2                      |
+-                              -+
```

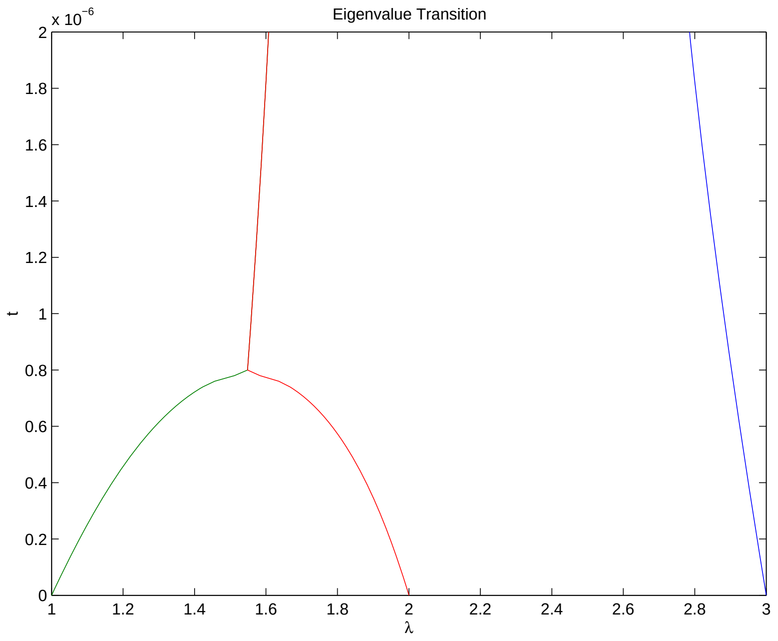
where

```
                             2
         492512 t   (t + 6)
         -------- - -------- + 11/3
            3          9
   #1 == -------------------------
                          1/3
                 2 sigma

          1/2       1/3
         3    (sigma    + #3) i
   #2 == ---------------------
                   2

                             2
         492512 t   (t + 6)
         -------- - -------- + 11/3
            3          9
   #3 == -------------------------
                         1/3
                  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 the `polylib::discrim` function in the MuPAD language.

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
                                                           0.00000078379249059679401048587946985451882055609055 3664
                        0.0000010769248160492151380753716016059778420823631 1263 - 0.0000030854463650228906549274 7*i
0.0000030854463650228906549274653827563618021771075729 5*i + 0.0000010769248160492151380753716016059778424916787370 7
```

Of the four solutions, you know that

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

is the transition point

```
tau =
0.00000078379249059679401048084
```

because it is closest to the previous estimate.

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

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

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

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

Find τ now by

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

Solve an Algebraic Equation

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)

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

Note that these examples assume equations of the form $f(x) = 0$. To solve equations of the form $f(x) = q(x)$, use the operator ==. For example, this command

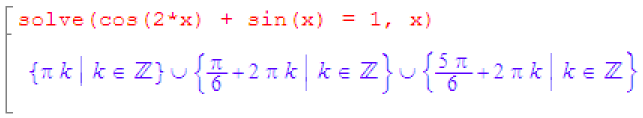
```
syms x
s = solve(cos(2*x) + sin(x) == 1)
```

returns a vector with three solutions.

```
s =
        0
     pi/6
```

```
(5*pi)/6
```

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



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

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

Solve a System of Algebraic Equations

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

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

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

```
syms x y alpha
```

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

```
[sol_x,sol_y] = solve(x^2*y^2 == 0, x-y/2 == alpha)
```

which returns

```
sol_x =
 alpha
     0

sol_y =
        0
 -2*alpha
```

Modify the first equation to $x^2y^2 = 1$. The new system has more solutions.

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

produces four distinct solutions:

```
sol_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

sol_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. For example, solve the system of equations `u^2 - v^2 = a^2`, `u + v = 1`, `a^2 - 2*a = 3`:

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

The solver returns its results enclosed in this structure:

```
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 equations can also be solved using matrix division. For example, solve this system:

```
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

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

Solve a Single Differential Equation

Use `dsolve` to compute symbolic solutions to ordinary differential equations. You can specify the equations as symbolic expressions containing `diff` or as strings with the letter `D` to indicate differentiation.

Note Because `D` indicates differentiation, the names of symbolic variables must not contain `D`.

Before using `dsolve`, create the symbolic function for which you want to solve an ordinary differential equation. Use `sym` or `syms` to create a symbolic function. For example, create a function `y(x)`:

```
syms y(x)
```

For details, see "Create Symbolic Functions" on page 1-10.

To specify initial or boundary conditions, use additional equations. If you do not specify initial or boundary conditions, the solutions will contain integration constants, such as `C1`, `C2`, and so on.

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.
- Place the output in a structure whose fields contain the solutions of the differential equations.

First-Order Linear ODE

Suppose you want to solve the equation `y'(t) = t*y`. First, create the symbolic function `y(t)`:

```
syms y(t)
```

Now use `dsolve` to solve the equation:

```
y(t) = dsolve(diff(y) == t*y)
```

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

`y(t) = C2*exp(t^2/2)` is a solution to the equation for any constant `C2`.

Solve the same ordinary differential equation, but now specify the initial condition `y(0) = 2`:

```
syms y(t)
y(t) = dsolve(diff(y) == t*y, y(0) == 2)

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

Nonlinear ODE

Nonlinear equations can have multiple solutions, even if you specify initial conditions. For example, solve this equation:

```
syms x(t)
x(t) = dsolve((diff(x) + x)^2 == 1, x(0) == 0)
```

results in

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

Second-Order ODE with Initial Conditions

Solve this second-order differential equation with two initial conditions. One initial condition is a derivative `y'(x)` at `x = 0`. To be able to specify this initial condition, create an additional symbolic function `Dy = diff(y)`. (You also can use any valid function name instead of `Dy`.) Then `Dy(0) = 0` specifies that `Dy = 0` at `x = 0`.

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

y(x) =
1 - (8*sin(x/2)^4)/3
```

Third-Order ODE

Solve this third-order ordinary differential equation:

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

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

Because the initial conditions contain the first- and the second-order derivatives, create two additional symbolic functions, `Dy` and `D2y` to specify these initial conditions:

```
syms u(x)
Du = diff(u);
D2u = diff(u, 2);
u(x) = dsolve(diff(u, 3) == u, u(0) == 1, Du(0)
== -1, D2u(0) == pi)

u(x) =

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

More ODE Examples

This table shows examples of differential equations and their Symbolic Math Toolbox syntax. The last example is the Airy differential equation, whose solution is called the Airy function.

Differential Equation	MATLAB Command

$\frac{dy}{dt} + 4y(t) = e^{-t}$
 	
$y(0) = 1$
	`syms y(t)`
	`dsolve(diff(y) + 4*y == exp(-t),`
	`y(0) == 1)`

$2x^2y'' + 3xy' - y = 0$
$(' = d/dx)$
	`syms y(x)`
	`dsolve(2*x^2*diff(y, 2) +`
	`3*x*diff(y) - y == 0)`

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

Solve a System of Differential Equations

`dsolve` can handle several ordinary differential equations in several variables, with or without initial conditions. For example, solve these linear first-order equations. First, create the symbolic functions `f(t)` and `g(t)`:

```
syms f(t) g(t)
```

Now use `dsolve` to solve the system. The toolbox returns the computed solutions as elements of the structure `S`:

```
S = dsolve(diff(f) == 3*f + 4*g, diff(g) == -4*f + 3*g)

S =
    g: [1x1 sym]
    f: [1x1 sym]
```

To return the values of `f(t)` and `g(t)`, enter these commands:

```
f(t) = S.f
g(t) = S.g

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

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

If you prefer to recover `f(t)` and `g(t)` directly, as well as include initial conditions, enter these commands:

```
syms f(t) g(t)
[f(t), g(t)] = dsolve(diff(f) == 3*f + 4*g,...
diff(g) == -4*f + 3*g, f(0) == 0, g(0) == 1)

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

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

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

```
syms x(t) y(t)
A = [1 2; -1 1];
B = [1; t];
Y = [x; y];
```

Solve the system using `dsolve`:

```
S = dsolve(diff(Y) == A*Y + B);
x = S.x
y = S.y

x =
2^(1/2)*exp(t)*cos(2^(1/2)*t)*(C2 + (exp(-t)*(4*sin(2^(1/2)*t) +...
2^(1/2)*cos(2^(1/2)*t) + 6*t*sin(2^(1/2)*t) +...
6*2^(1/2)*t*cos(2^(1/2)*t)))/18) +...
2^(1/2)*exp(t)*sin(2^(1/2)*t)*(C1 - (exp(-t)*(4*cos(2^(1/2)*t) -...
2^(1/2)*sin(2^(1/2)*t) +...
6*t*cos(2^(1/2)*t) - 6*2^(1/2)*t*sin(2^(1/2)*t)))/18)

y =
exp(t)*cos(2^(1/2)*t)*(C1 - (exp(-t)*(4*cos(2^(1/2)*t) -...
2^(1/2)*sin(2^(1/2)*t) + 6*t*cos(2^(1/2)*t) -...
6*2^(1/2)*t*sin(2^(1/2)*t)))/18) - exp(t)*sin(2^(1/2)*t)*(C2 +...
(exp(-t)*(4*sin(2^(1/2)*t) + 2^(1/2)*cos(2^(1/2)*t) +...
6*t*sin(2^(1/2)*t) + 6*2^(1/2)*t*cos(2^(1/2)*t)))/18)
```

Compute 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

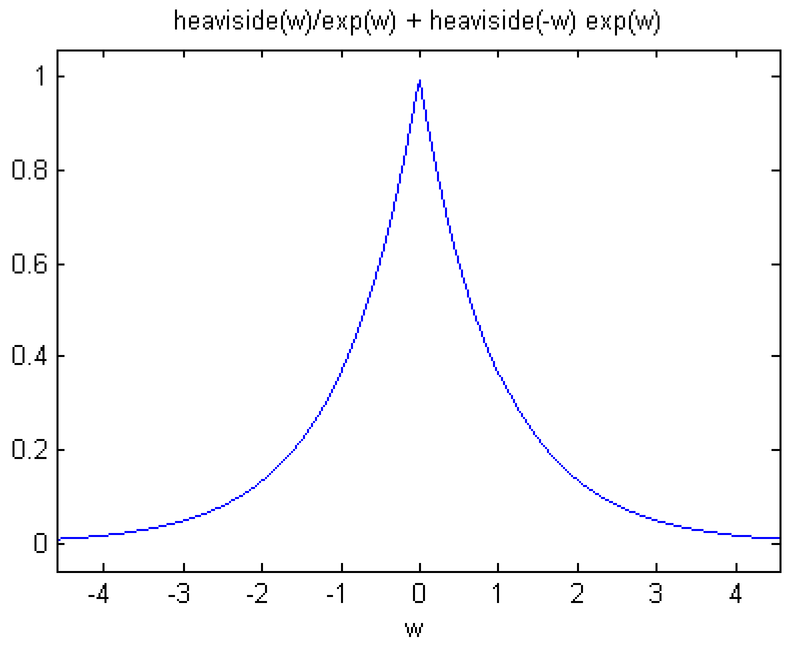
$$F^{-1}[f](x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(w)e^{iwx}dw.$$

This documentation refers 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 indicate 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. This documentation refers 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*exp(-w)*heaviside(w) + pi*heaviside(-w)*exp(w))/pi

fcauchy = expand(fcauchy)

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

```
ezplot(fcauchy)
```

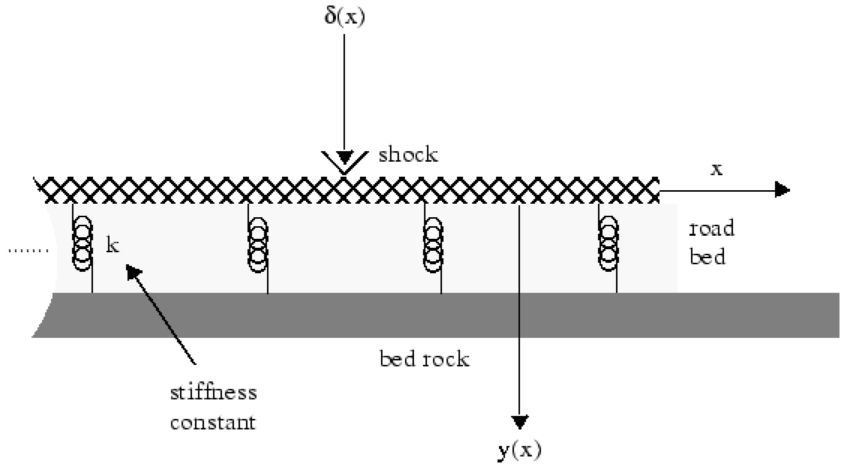


The Fourier transform is symmetric, since the original Cauchy density function is symmetric.

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

```
finvfcauchy = ifourier(fcauchy)

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

simplify(finvfcauchy)

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

An application of the Fourier transform is the solution of ordinary and partial differential equations over the real line. Consider the deformation of an infinitely long beam resting on an elastic foundation with a shock applied to it at a point. A "real world" analogy to this phenomenon is a set of railroad tracks atop a road bed.



The shock could be induced by a pneumatic hammer blow.

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 (railroad track), I is the "beam constant," and *k* is the spring (road bed) stiffness. The shock force on the right side of the differential equation is modeled by the Dirac Delta function δ(*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

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

Let $Y(w) = F[y(x)](w)$ and $\Delta(w) = F[\delta(x)](w)$. Indeed, try the command `fourier(dirac(x), 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 y(x)
fourier(diff(y(x), x, 4), x, w)
```

which returns

```
ans =
w^4*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.

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

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

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 the differential equation, observe that the ratio

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

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

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

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

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

and see

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

```
     / pi       \                             \ \
  cos| -- + 4 x | exp(8 x) (heaviside(x) - 1) | | / 512
     \ 4        /                             / /
```

Notice that g contains the Heaviside distribution

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

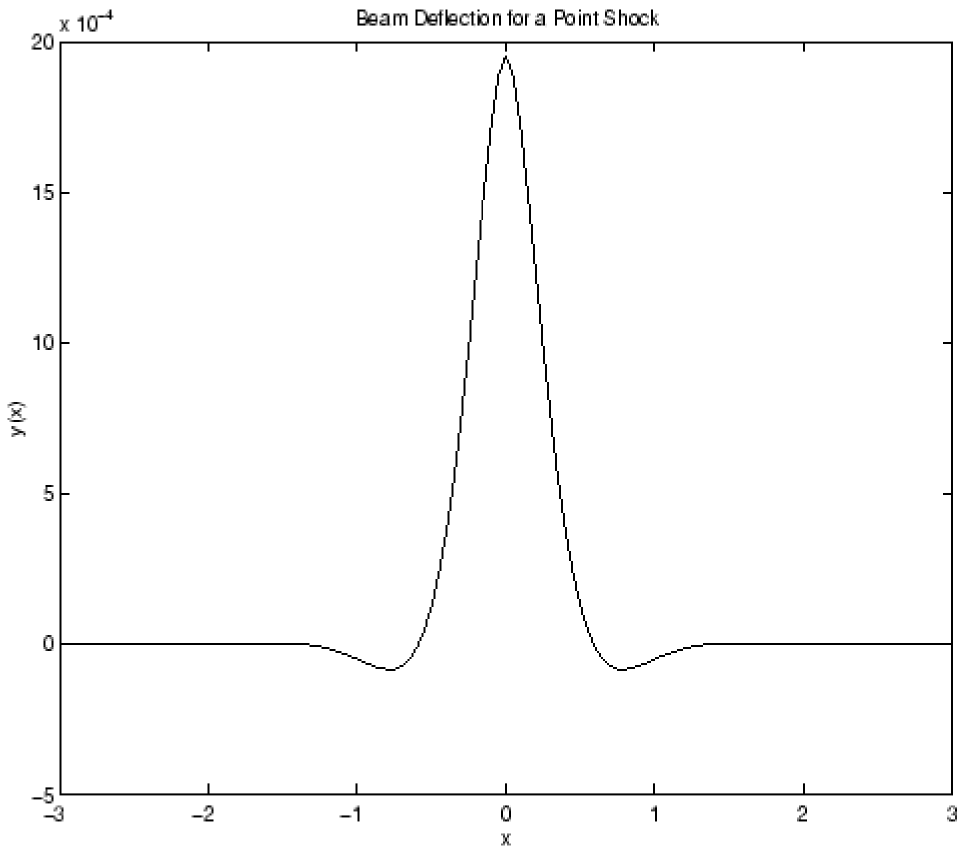
Since Y is the product of Fourier transforms, y is the convolution of the transformed functions. That is, $F[y] = Y(w) = \Delta(w)\ G(w) = F[\delta]\ F[g]$ implies

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

by the special property of the Dirac Delta function. To plot this function, 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.

Compute Laplace and Inverse Laplace Transforms

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

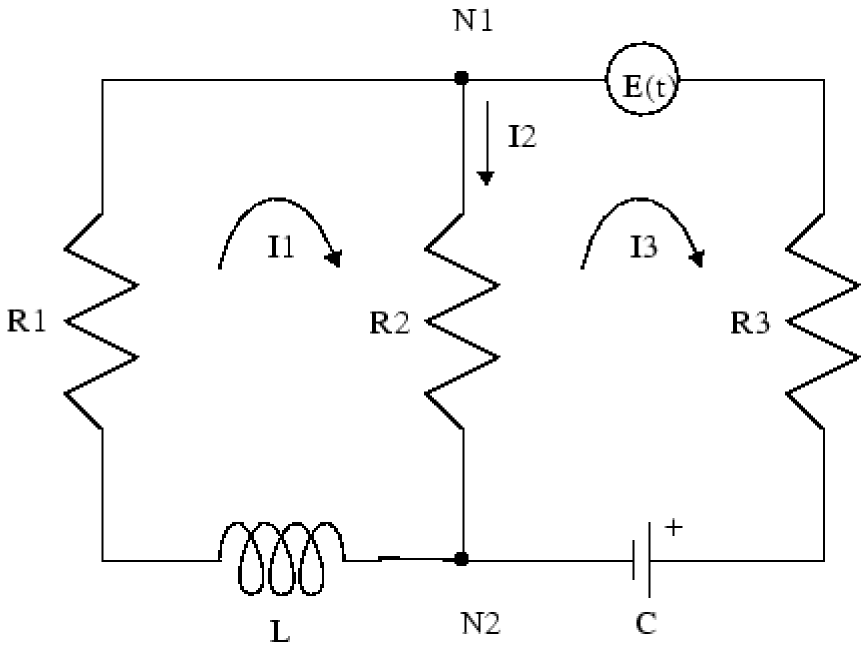
$$L[f](s) = \int_0^\infty f(t)e^{-ts}dt,$$

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*] indicates 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.

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

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

Solve this system of differential equations using `laplace`. 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;
syms I1(t) Q(t) s;
dI1(t) = diff(I1(t), t);
dQ(t) = diff(Q(t),t);
E(t) = sin(t);  % Voltage
eq1(t) = dI1(t) + R2*dQ(t)/L - (R2 - R1)*I1(t)/L;
eq2(t) = dQ(t) - (E(t) - Q/C)/(R2 + R3) - R2*I1(t)/(R2 + R3);
```

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

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

returns

```
L1(t) =
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(t) =
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 you 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, 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(t),{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(t) =
s*laplace(Q(t), t, s) - 1/(5*(s^2 + 1)) -...
(2*laplace(I1(t), t, s))/5 + (4*laplace(Q(t), t, s))/5 - 2
```

To solve for `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)`, make a final pair of substitutions. First, replace the strings `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)` by the `sym` objects `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(t) =
(4*LQ)/5 - (2*LI1)/5 + LQ*s - 1/(5*(s^2 + 1)) - 2
```

which, after collecting terms,

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

gives

```
NQ(t) =
(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)
```

you 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, compute the inverse Laplace transform of LI1 and LQ. Inverting LI1

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

produces

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

Inverting LQ

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

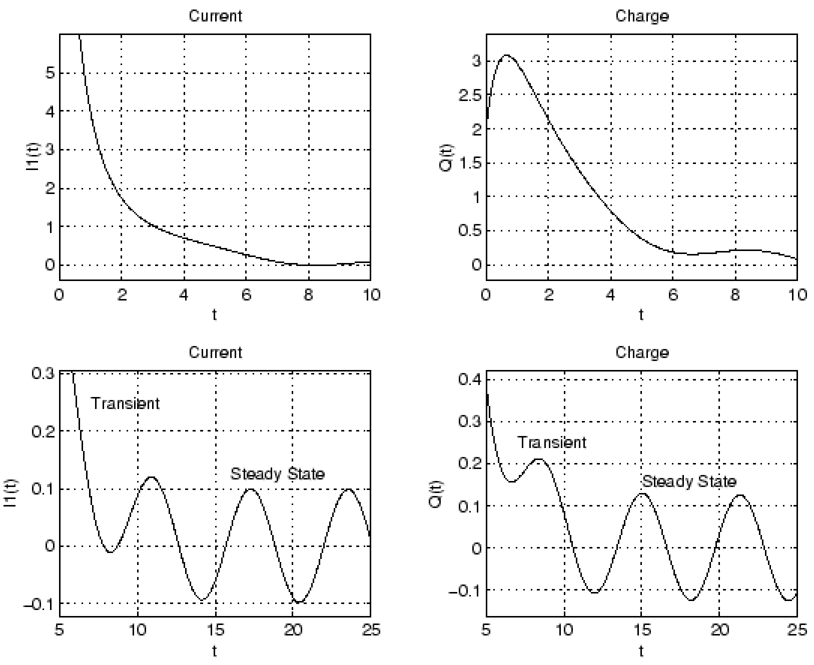
yields

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

Now plot the current I1(t) and charge Q(t) in two different time domains, 0 ≤ t ≤ 10 and 5 ≤ t ≤ 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).

Compute Z-Transforms 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

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

You can use `ztrans` to find the population each year *p*(*n*). First, apply `ztrans` to the equations

```
syms p(n) z
eq = p(n + 2) - p(n + 1) - p(n);
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 $p(n)$, take the inverse z-transform of P.

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

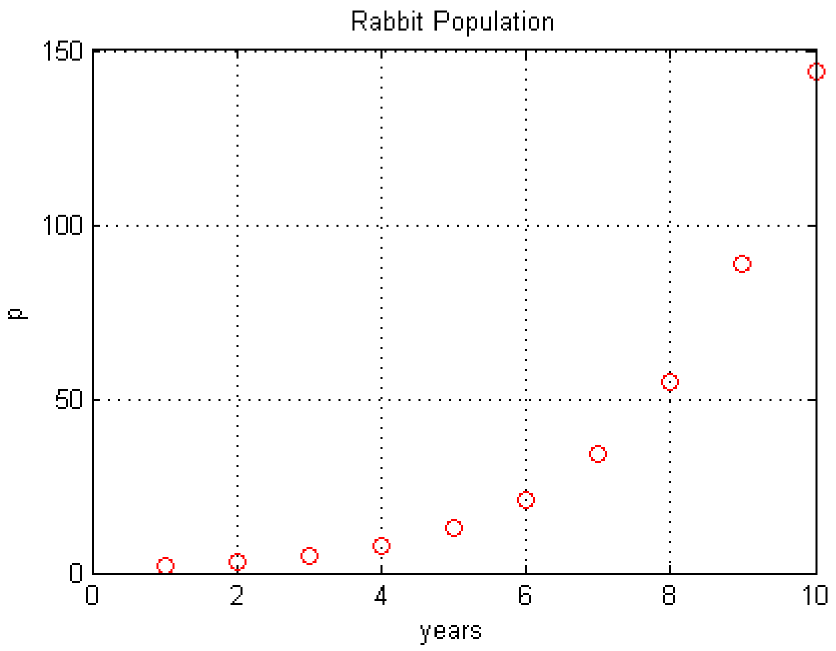
The result is a bit complicated, but explicit:

```
p =
4*(-1)^(n/2)*cos(n*(pi/2 + asinh(1/2)*i)) +...
1/2^n*((3*5^(1/2))/10 - 3/2)*(5^(1/2) + 1)^n -...
1/2^n*((3*5^(1/2))/10 + 3/2)*(1 - 5^(1/2))^n
```

Finally, plot p:

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

to show the growth in rabbit population over time.



References

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

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

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

Create Plots

In this section...

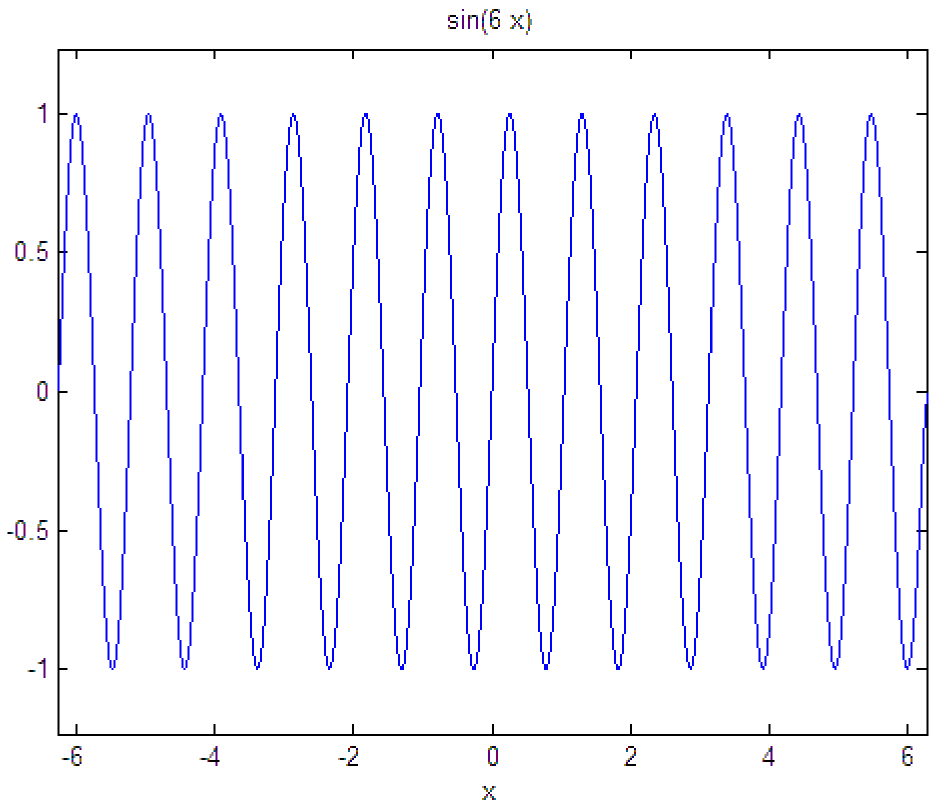
Plot with Symbolic Plotting Functions

MATLAB provides many 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. Symbolic Math Toolbox expands these graphical capabilities and lets you plot symbolic functions using:

- `ezplot` to create 2-D plots of symbolic expressions, equations, or functions in Cartesian coordinates.
- `ezplot3` to create 3-D parametric plots. To create animated plots, use the `animate` option.
- `ezpolar` that creates plots in polar coordinates.
- `ezsurf` to create surface plots. The `ezsurfc` plotting function creates combined surface and contour plots.
- `ezcontour` to create contour plots. The `ezcontourf` function creates filled contour plots.
- `ezmesh` to create mesh plots. The `ezmeshc` function creates combined mesh and contour plots.

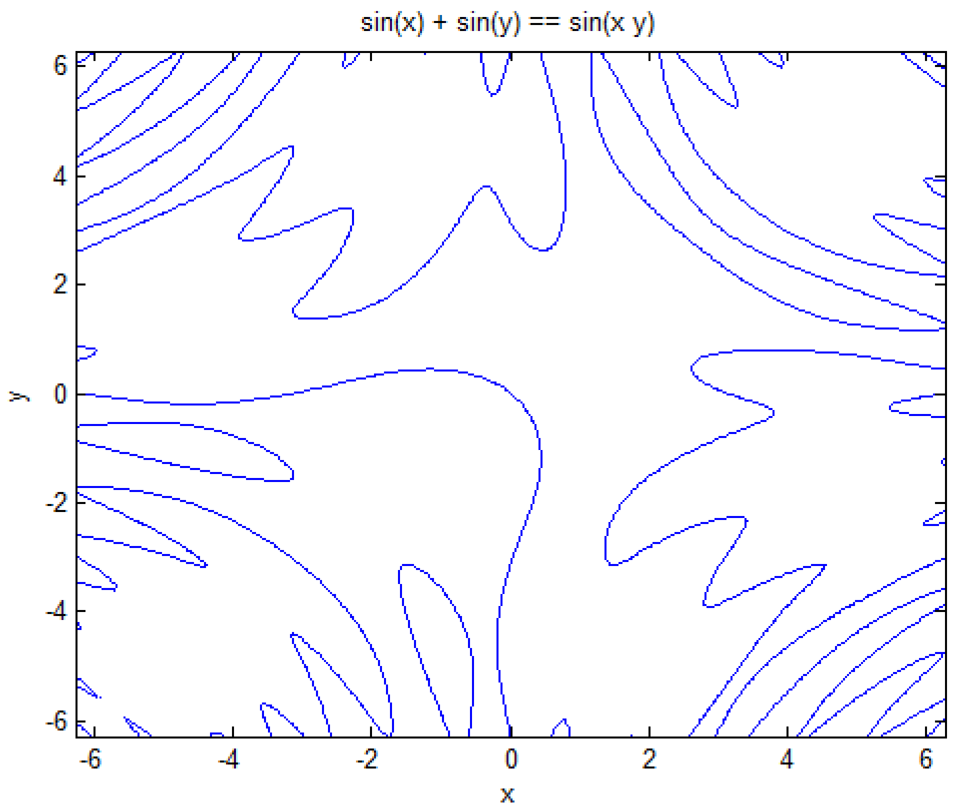
For example, plot the symbolic expression `sin(6x)` in Cartesian coordinates. By default, `ezplot` uses the range $-2\pi < x < 2\pi$:

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



`ezplot` also can plot symbolic equations that contain two variables. To define an equation, use ==. For example, plot this trigonometric equation:

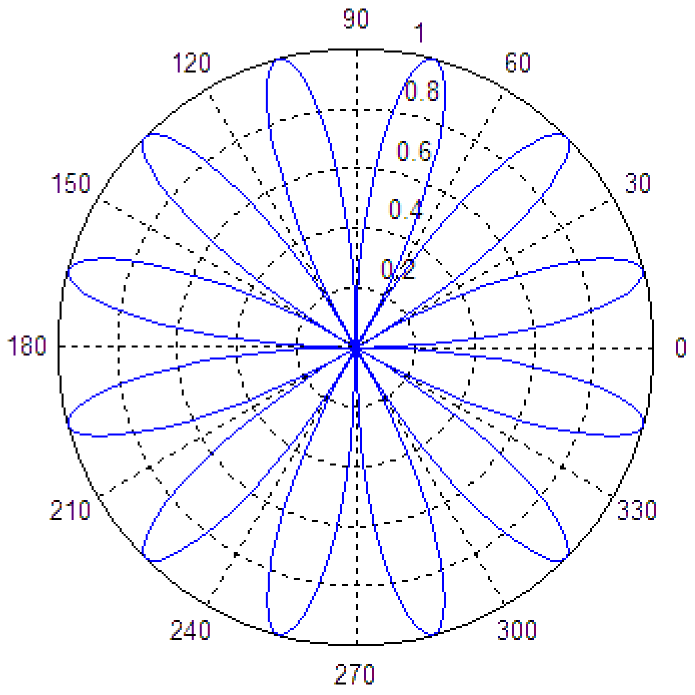
```
syms x y
ezplot(sin(x) + sin(y) == sin(x*y))
```



When plotting a symbolic expression, equation, or 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 let you change the mesh setting.

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

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



r = sin(6 t)

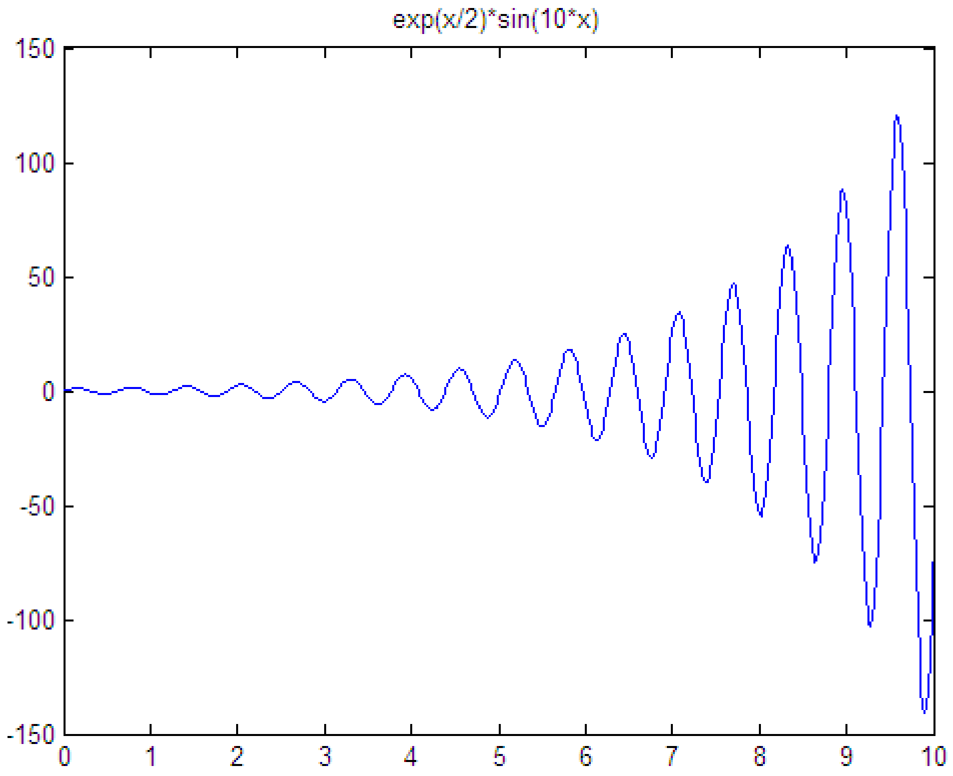
Plot with MATLAB Plotting Functions

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
```

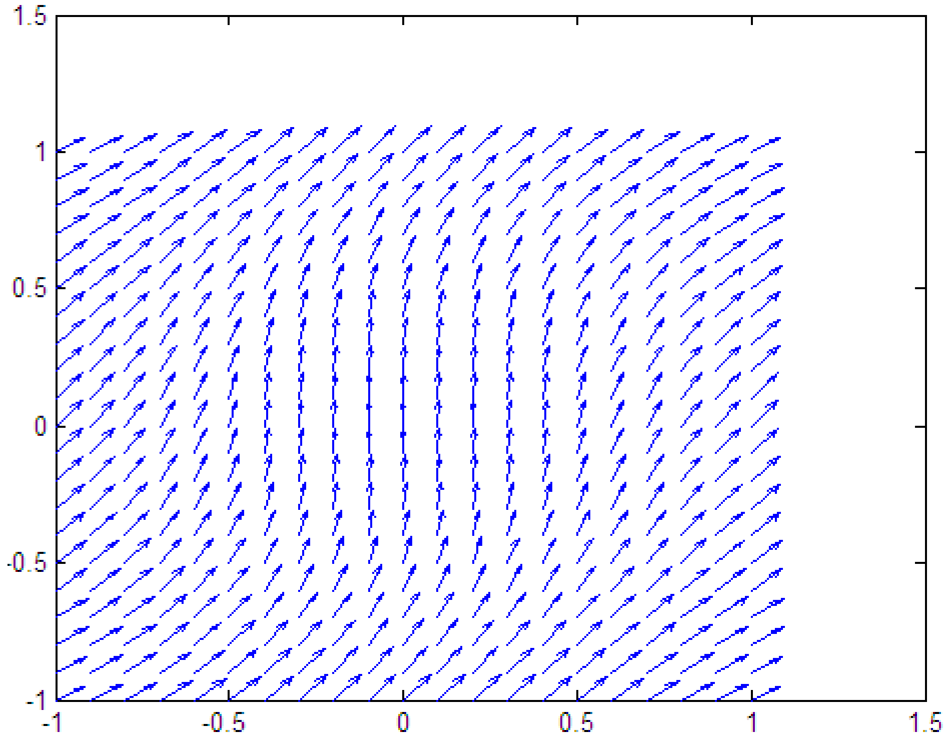


An alternative approach is to replace symbolic variables in an expression with numeric values by using the `subs` function. For example, in the following expressions u and v, substitute the symbolic variables x and y with the numeric values defined by `meshgrid`:

```
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*):

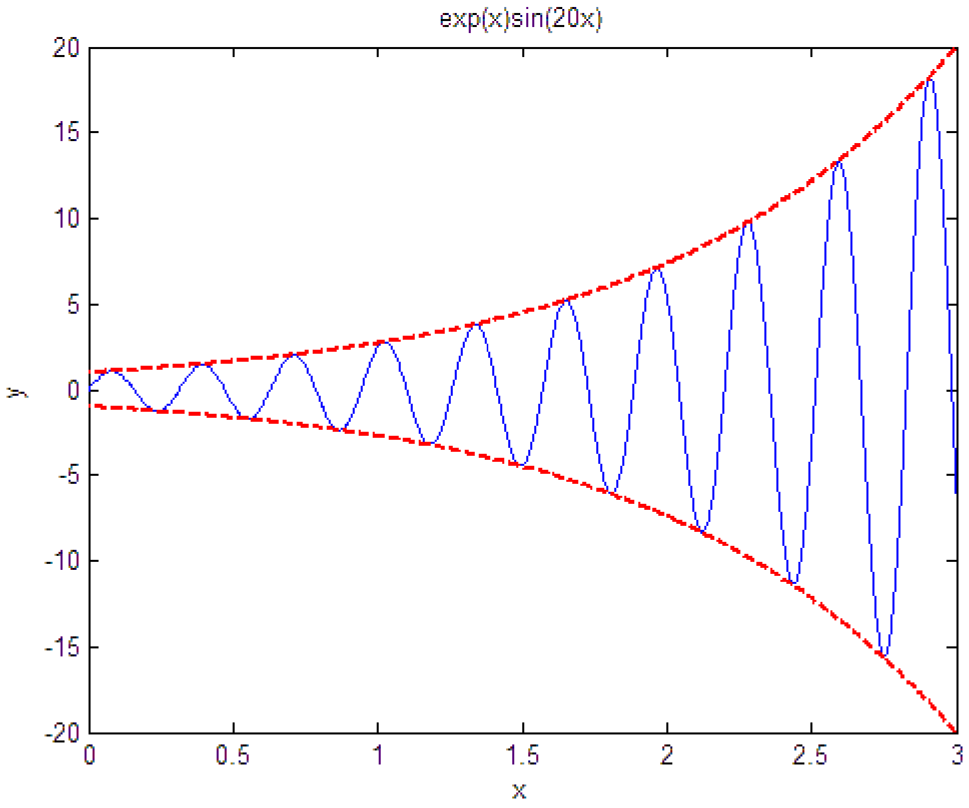
```
quiver(X, Y, U, V)
```



Plot 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
```



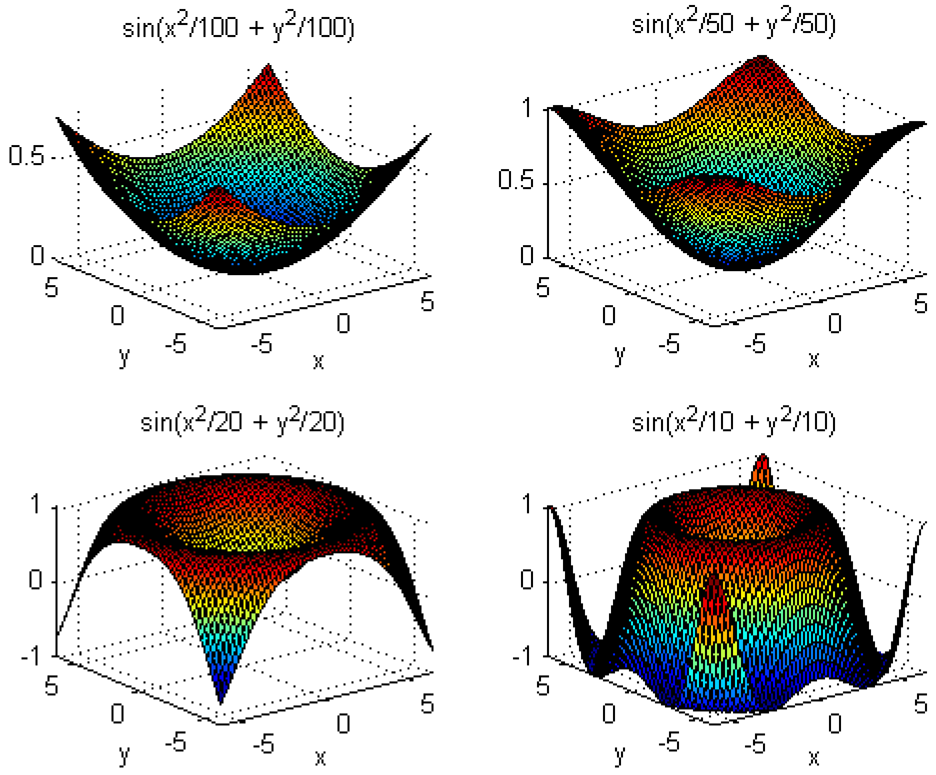
Plot Multiple Symbolic Functions in One Figure

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`:

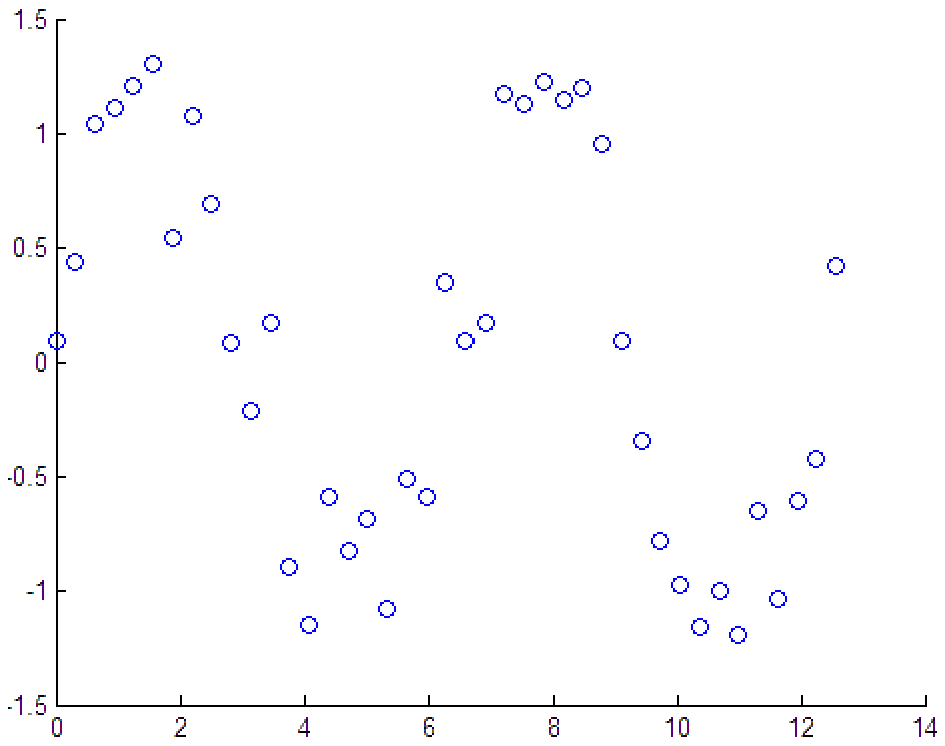
```
syms x y
z = x^2 + y^2;
subplot(2, 2, 1); ezsurf(sin(z/100))
subplot(2, 2, 2); ezsurf(sin(z/50))
subplot(2, 2, 3); ezsurf(sin(z/20))
subplot(2, 2, 4); ezsurf(sin(z/10))
```



Combine Symbolic Function Plots and Numeric Data Plots

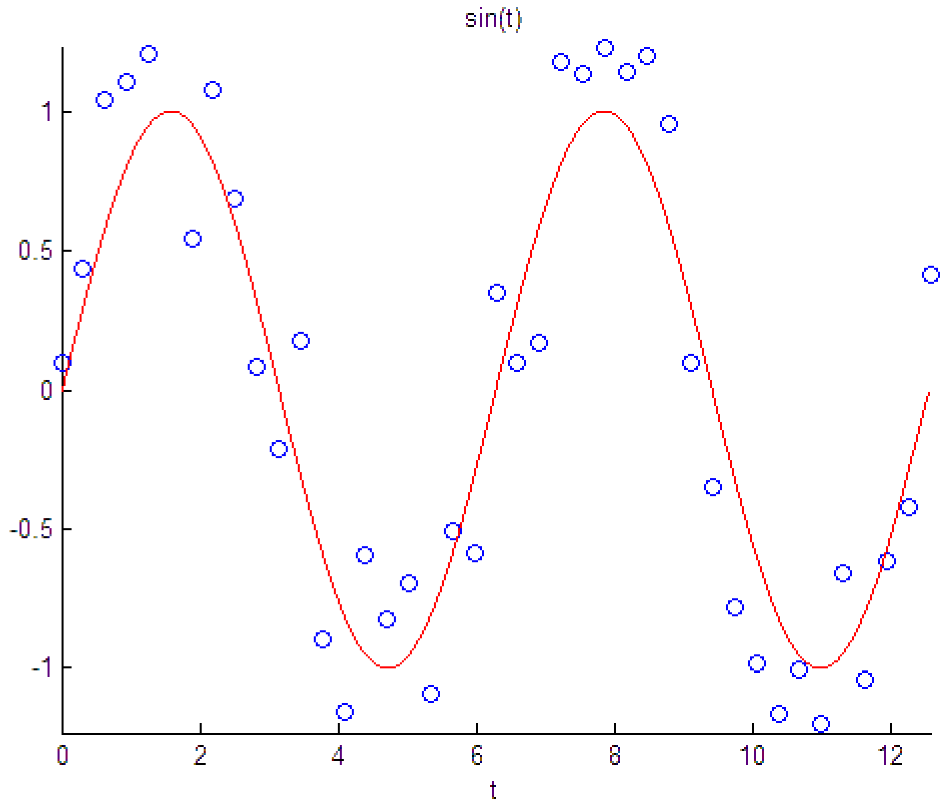
The combined graphical capabilities of MATLAB and the Symbolic Math Toolbox software let you plot numeric data and symbolic functions in one graph. Suppose, you have two discrete data sets, *x* and *y*. Use the `scatter` plotting function to plot these data sets as a collection of points with coordinates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



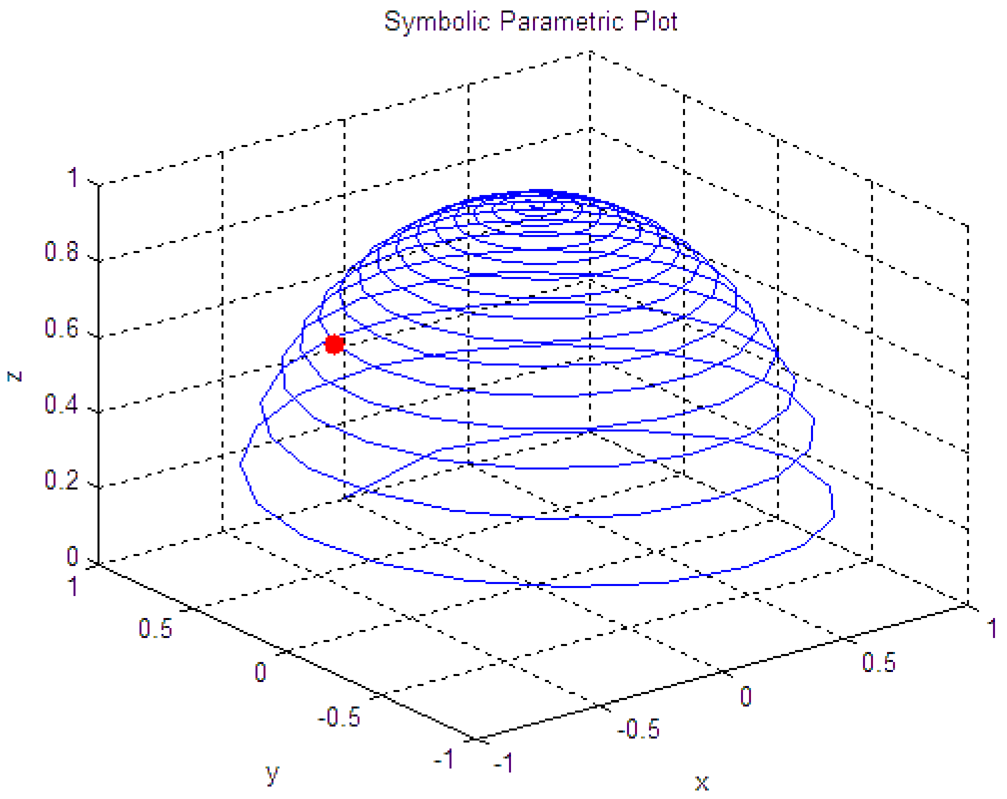
Now, suppose you want to plot the sine function on top of the scatter plot in the same graph. First, use the `hold on` command to retain the current plot in the figure. (Without this command, the symbolic plot that you are about to create replaces the numeric data plot.) Then, use `ezplot` to plot the 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')
```



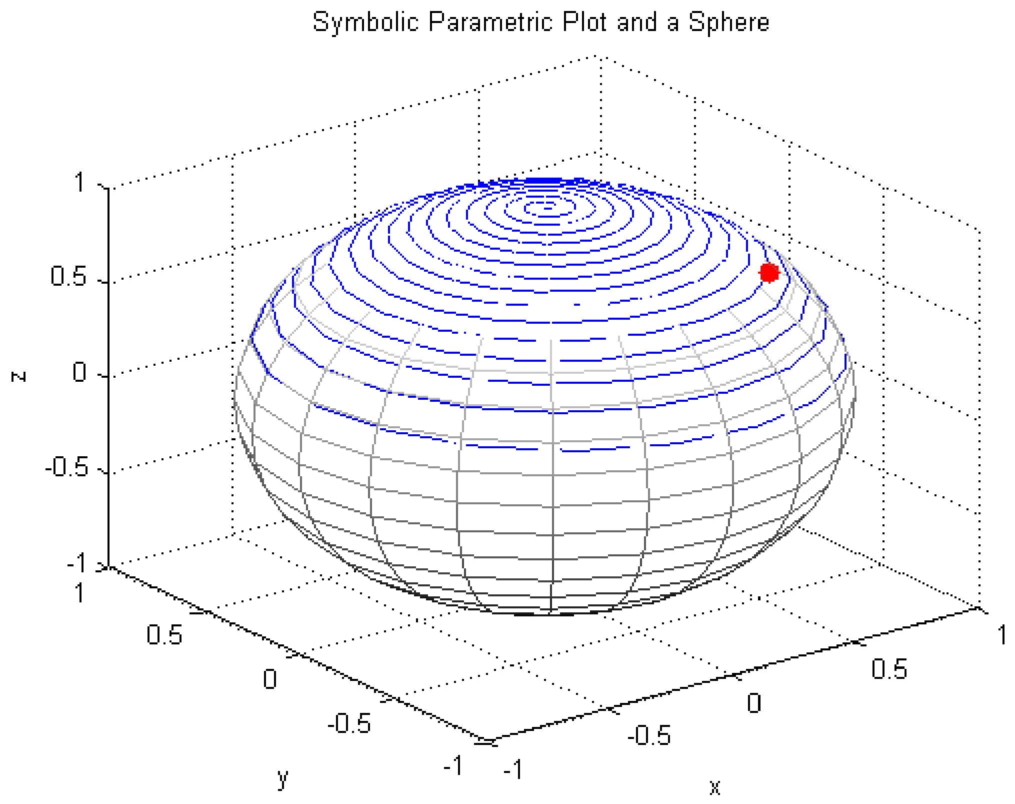
MATLAB provides the plotting functions that simplify the process of generating spheres, cylinders, ellipsoids, and so on. The Symbolic Math Toolbox software lets you create a symbolic function plot in the same graph with these volumes. For example, use the following commands to generate the spiral function 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')
```



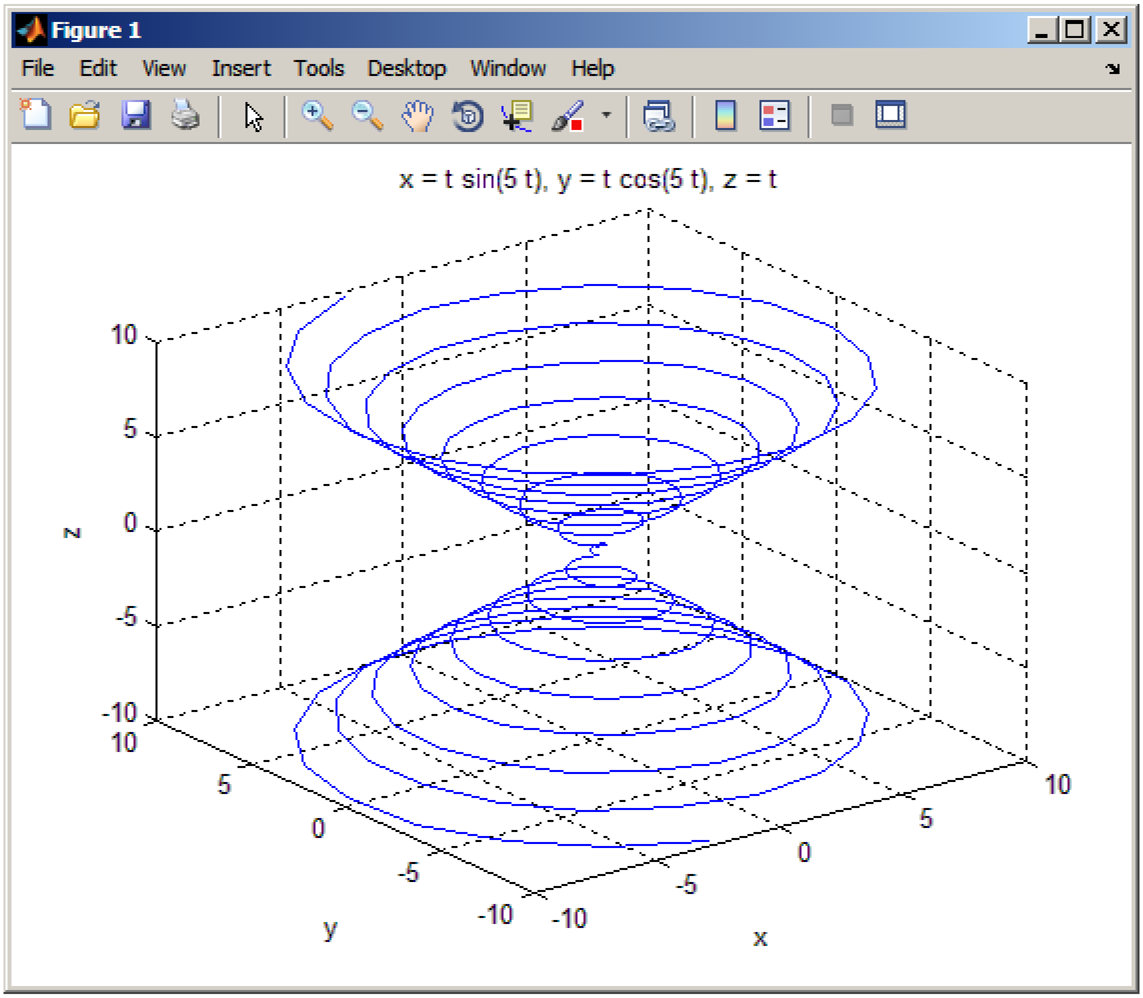
Add the sphere with radius 1 and the center at (0, 0, 0) to this graph. The `sphere` function generates the required sphere, and the `mesh` function creates a mesh plot for that sphere. Combining the plots clearly shows that the symbolic parametric function plot is wrapped around the top hemisphere:

```
hold on
[X,Y,Z] = sphere;
mesh(X, Y, Z)
colormap(gray)
title('Symbolic Parametric Plot and a Sphere')
```


Symbolic Parametric Plot and a Sphere
z
1
0.5
0
-0.5
-1
y
1
0.5
0
-0.5
-1
x
-1
-0.5
0
0.5
1

Explore 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.



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.

- 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.

- Rotate 3-D graphs (). Rotating 3-D graphs allows you to see more features of the surface and mesh function plots.
- Display particular data values on a graph and export them to MATLAB workspace variables ().

Edit 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.

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.

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

Save 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.
- 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.
- 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.
- 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.

Generate 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.

If you work in the MuPAD notebook interface, see `generate::C` and `generate::fortran`.

Generate 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.

If you work in the MuPAD notebook interface, see "Create MATLAB Functions from MuPAD Expressions" on page 3-48.

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
```

Control the Order of Variables

matlabFunction generates input variables in alphabetical order from a symbolic expression. That is why the function handle in "Generating a Function Handle" on page 2-131 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))
```

Generate 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)
...
```

Name 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)
...
```

Convert 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);
```

Tip `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.

Generate MATLAB Function Blocks

Using `matlabFunctionBlock`, you can generate a 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.

If you work in the MuPAD notebook interface, see "Create MATLAB Function Blocks from MuPAD Expressions" on page 3-52.

Generate and Edit a Block

Suppose, you want to create a model involving the symbolic expression `r = sqrt(x^2 + y^2)`. Before you can convert a symbolic expression to a 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 `matlabFunctionBlock` command. Also specify the block name:

```
syms x y
r = sqrt(x^2 + y^2);
matlabFunctionBlock('my_system/my_block', r)
```

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

You can open and edit the generated block. To open a block, double-click it.

```
function r = my_block(x,y)
%#codegen

r = sqrt(x.^2+y.^2);
```

Control the Order of Input Ports

`matlabFunctionBlock` 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);
matlabFunctionBlock('my_system/vdp', dydt,...
'vars', [y mu x])
```

Name the Output Ports

By default, `matlabFunctionBlock` 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);
matlabFunctionBlock('my_system/vdp', dydt,...
'outputs',{'name1'})
```

Convert MuPAD Expressions

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

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

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

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

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

Tip 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 Functions Supported for Code Generation.

Generate 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 www.mathworks.com/products/simscape/.

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.

If you work in the MuPAD notebook interface, see "Create Simscape Equations from MuPAD Expressions" on page 3-54.

Convert 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:

```
syms a y(t)
Dy = diff(y);
s = dsolve(diff(y, 2) == -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(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 x
assume(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 ~= 0.0),
    -atan(1.0/x^2*(u-1.0))+atan(1.0/x^2*(u+1.0));
  else
    NaN;
  end;
```

Clear the assumption that *x* is real:

```
syms x clear
```

Convert 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`

Special Functions of Applied Mathematics

In this section...

Evaluate Special Functions Numerically Using mfun

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

For example, suppose you want to evaluate the hyperbolic cosine integral at the points 2 + `i`, 0, and 4.5. Look in the tables in "Syntax and Definitions of mfun Special Functions" on page 2-143 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)`	$n \geq 0$ $\|t\| < \frac{\pi}{2}$
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)`	$n \geq 0$ $\mathrm{Real}(z) > 0$
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)`	

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)`
			$0 < x \le \infty$.
			 `a` is real, $-\infty < a < \infty$.
			 `k` is real, $0 < k < 1$.

Incomplete gamma function	$\Gamma(a,z) = \int_z^\infty e^{-t} \cdot t^{a-1} dt$	`GAMMA(z1,z2)` `z1` $= a$ `z2` $= z$	

mfun Special Functions (Continued)

Function Name	Definition	mfun Name	Arguments
Logarithm of the gamma function	$\ln\mathrm{GAMMA}(z) = \ln(\Gamma(z))$	`lnGAMMA(z)`	
Logarithmic integral	$Li(x) = PV\left\{\int_0^x \frac{dt}{\ln t}\right\} = Ei(\ln x)$	`Li(x)`	$x > 1$
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)`	

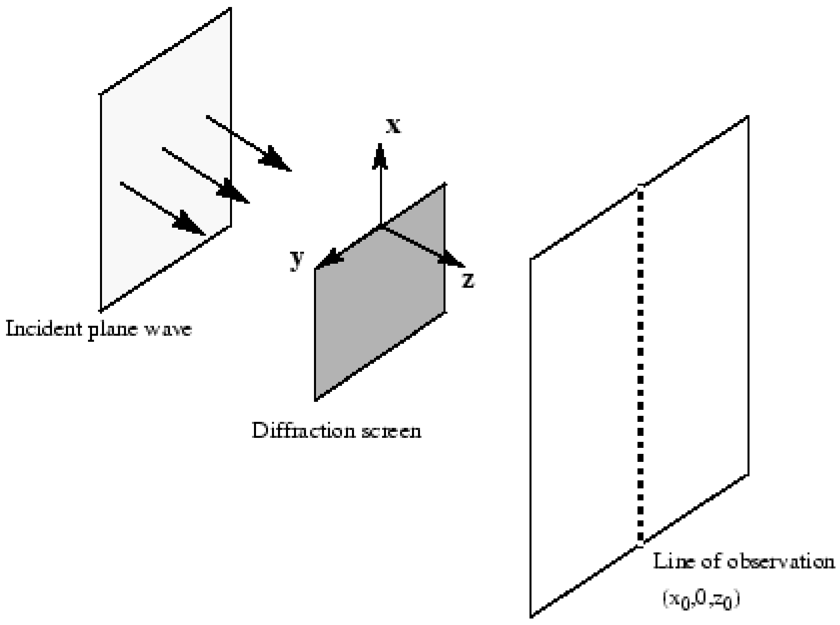
Orthogonal Polynomials (Continued)

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:

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

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

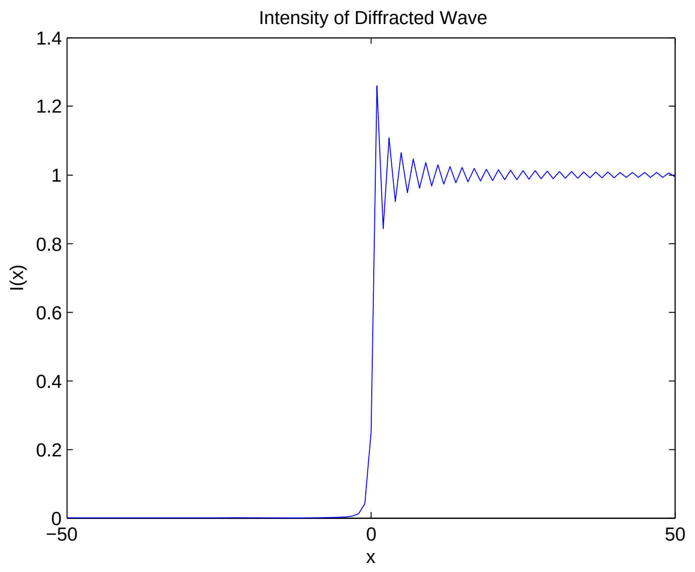
How does the intensity of the diffracted wave behave along the line of observation? Since k and z_0 are constants independent of x, you set

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

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]
```

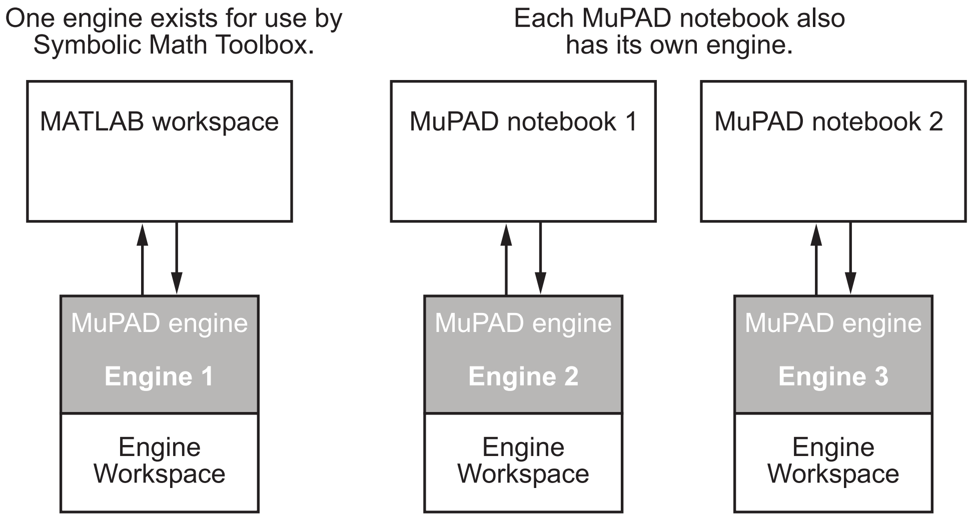
in the code to calculate I.

MuPAD in Symbolic Math Toolbox

MuPAD Engines and MATLAB Workspace

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 an 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 details, see “Clear Assumptions and Reset the Symbolic Engine” on page 3-43.

Create, Open, and Save MuPAD Notebooks

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

```
nb = mupad
```

You can use any variable name 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 "Copy Variables and Expressions Between MATLAB and MuPAD" on page 3-22.

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

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

where ***`file_name`*** must be a full path unless the notebook is in the current folder. This command is useful if you lose the handle to a notebook, in which case, you can save the notebook file and then reopen it with a fresh handle.

Caution 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 "Work with Links". To refer to a link target when opening a notebook, enter:

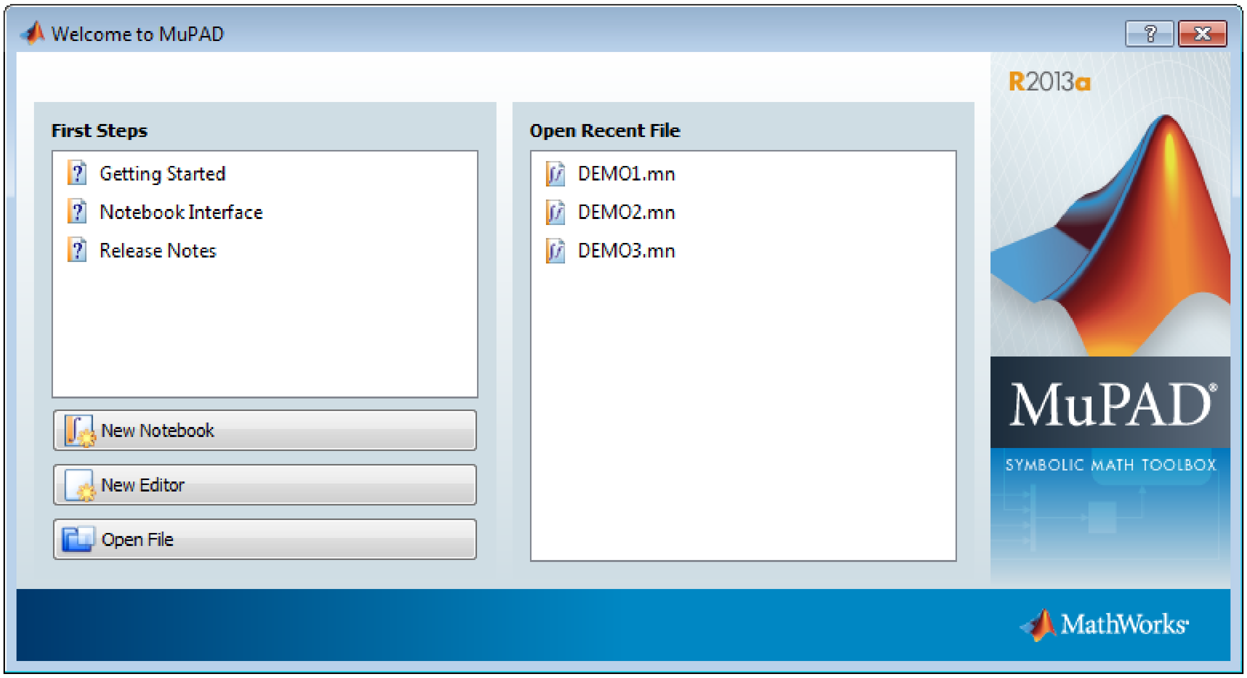
```
nb2 = mupad('file_name#linktarget_name')
```

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

Tip 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, select **Notebook > Evaluate All**. For details, see "Synchronize Notebook and its Engine" on page 3-10.

You also can use the Welcome to MuPAD dialog box to access various MuPAD interfaces. To open this dialog box, enter:

```
mupadwelcome
```



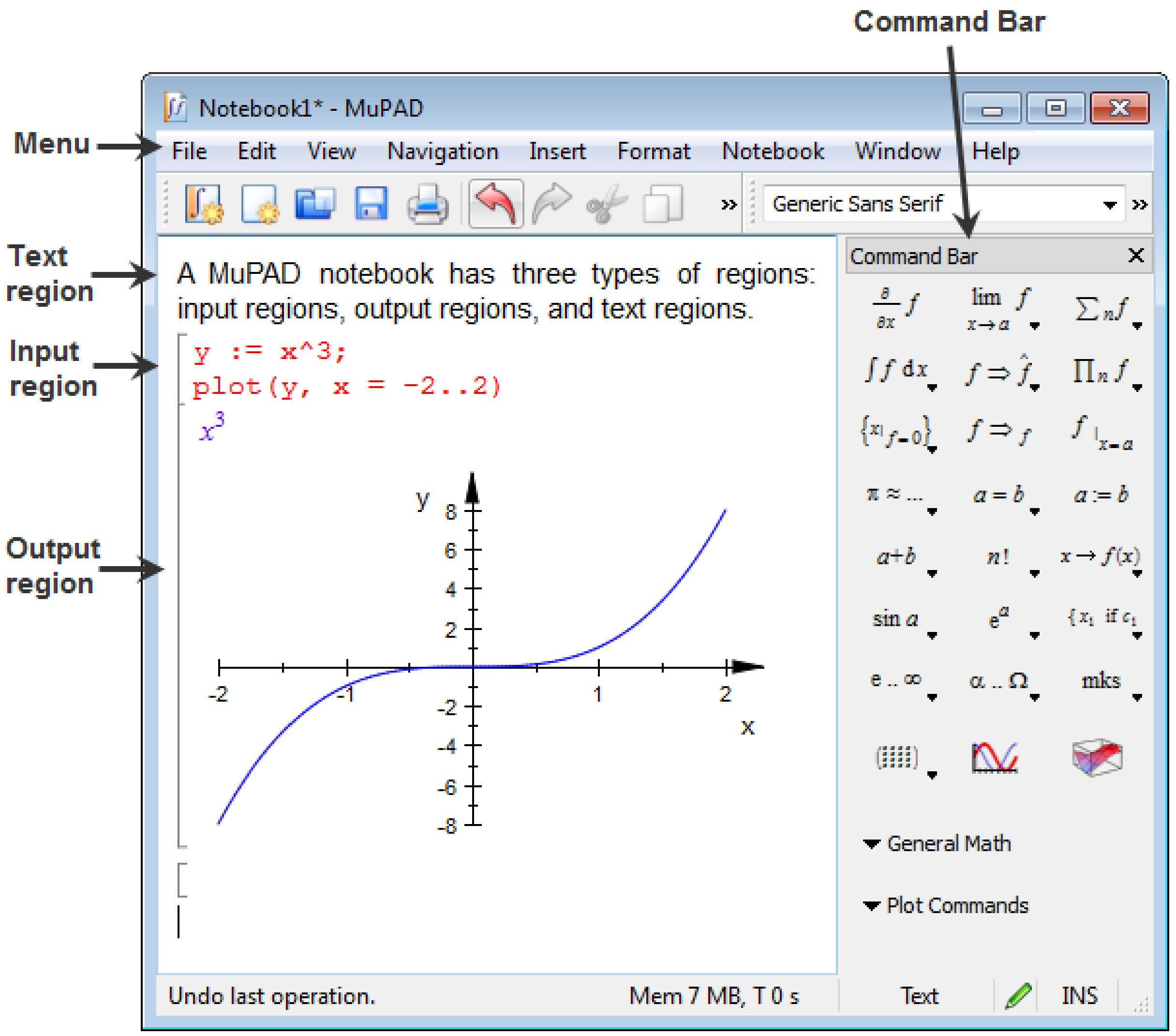
- To access documentation, click one of the three options in the **First Steps** pane.
- To open an existing file, click its name in the **Open Recent File** pane.
- To open a new notebook, click the **New Notebook** button.
- To open a new MATLAB Editor window, click the **New Editor** button.

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

Calculate in a MuPAD Notebook

Visual Elements of a Notebook

A MuPAD notebook has the following main components.



A MuPAD notebook has three types of regions: input regions, output regions, and text regions.

```
y := x^3;
plot(y, x = -2..2)
```

x^3

- Enter commands for execution, evaluation, or plotting in input regions.
- Enter comments in text regions. You can type and format text in a notebook similar to working in any word processing application.
- Use the **Command Bar** to help you enter commands into input regions with the proper syntax.
- Use the **Insert** menu to add a text area (called **Text Paragraph**) or input regions (called **Calculation**).
- Use the **Notebook** menu to evaluate expressions in input regions.

Work in a Notebook

The MuPAD notebook interface differs from the MATLAB interface. Things to remember when working in a MuPAD notebook are:

- 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 want to change and typing over it. Press **Enter** to reevaluate the result.
- Results do not automatically cascade or propagate through a notebook, as described in "Cascade Calculations" on page 3-7.
- The MATLAB method of recalling a previous command by pressing 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.

Cascade Calculations

If you change a variable in a notebook, the change does 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)
```

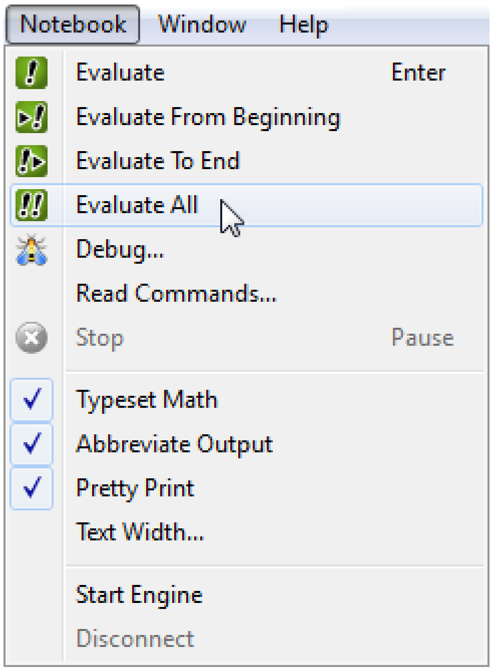
$$\frac{\sin(x)}{\sin(x)^2 + 1}$$

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

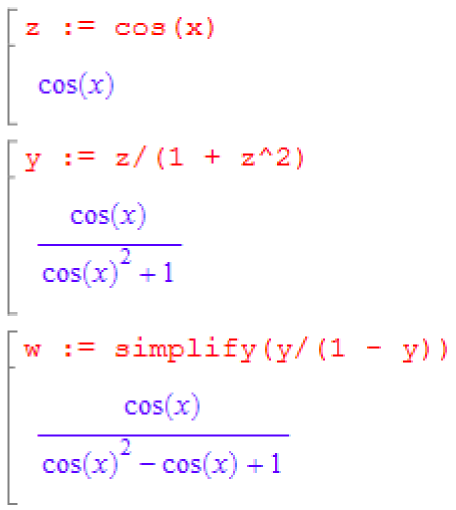
$$\frac{\sin(x)}{\sin(x)^2 - \sin(x) + 1}$$

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)
```

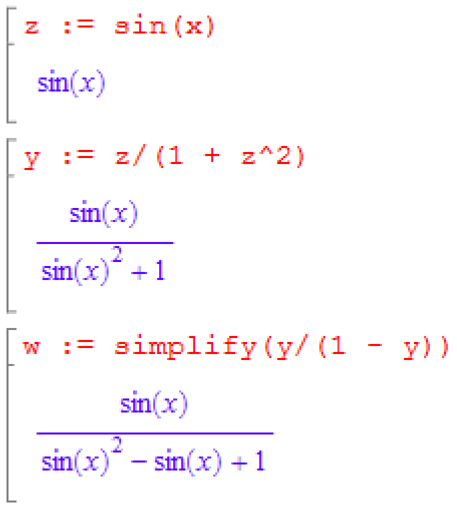
$$\frac{\cos(x)}{\cos(x)^2 + 1}$$

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

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

Synchronize 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 "Cascade Calculations" on page 3-7:



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

$$\sin(x)$$

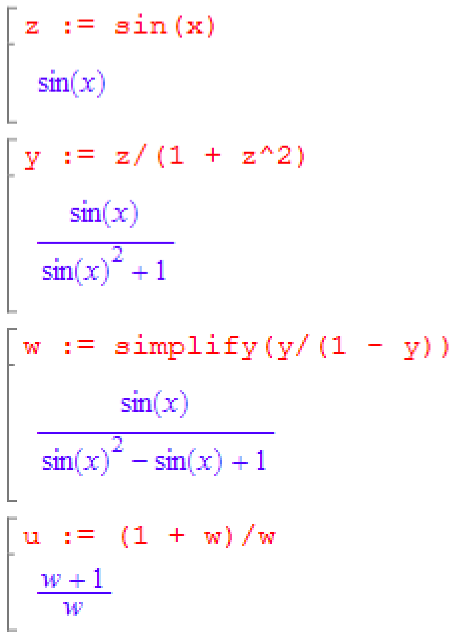
```
y := z/(1 + z^2)
```

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

```
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))
```

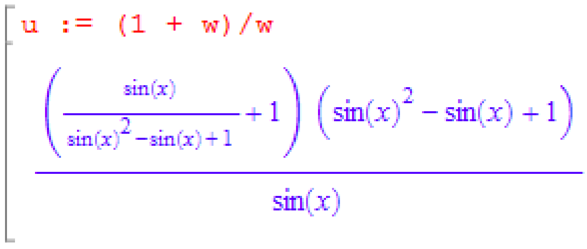
$$\frac{\sin(x)}{\sin(x)^2 - \sin(x) + 1}$$

```
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
```

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

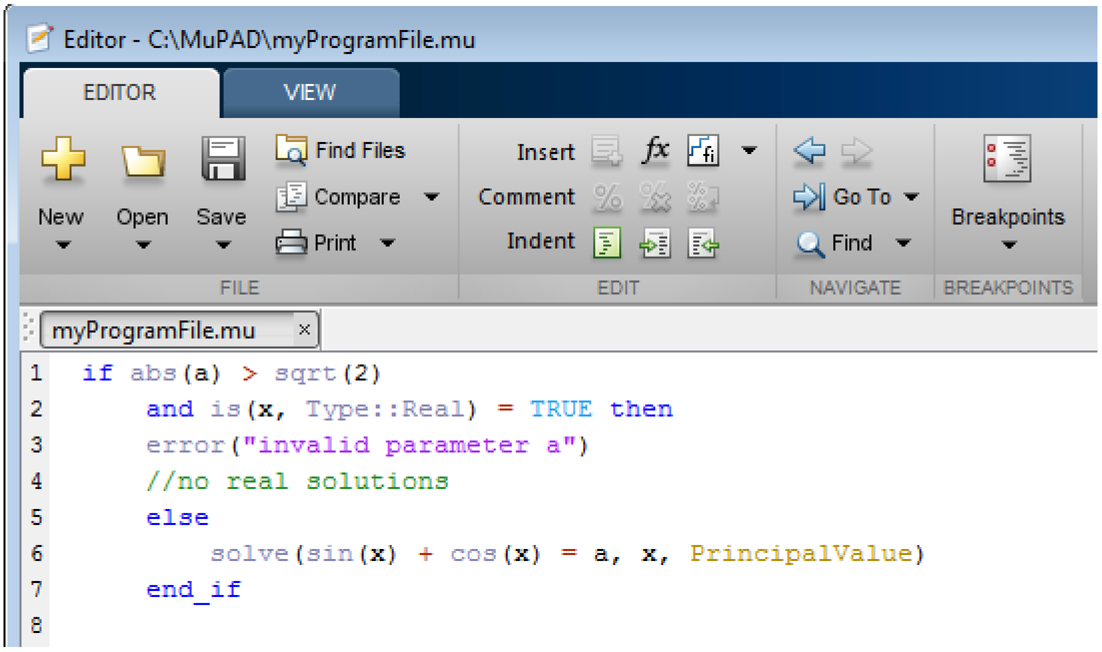
Edit and Debug MuPAD Code

Edit the Code in the MATLAB Editor

The default interface for editing MuPAD code is the MATLAB Editor. Alternatively, you can create and edit your code in any text editor. The MATLAB Editor automatically formats the code and, therefore, helps you avoid errors, or at least reduce their number.

Note The MATLAB Editor cannot evaluate or debug MuPAD code.

To open an existing MuPAD file with the extension `.mu` in the MATLAB Editor, double-click the file name or select **Open** and navigate to the file.



```
if abs(a) > sqrt(2)
    and is(x, Type::Real) = TRUE then
    error("invalid parameter a")
    //no real solutions
    else
        solve(sin(x) + cos(x) = a, x, PrincipalValue)
    end_if
```

After editing the code, save the file. Note that the extension `.mu` allows the Editor to recognize and open MuPAD program files. Thus, if you intend to open the files in the MATLAB Editor, save them with the extension `.mu`. Otherwise, you can specify other extensions suitable for text files, for example, `.txt` or `.tst`.

Debug the Code in the MuPAD Debugger

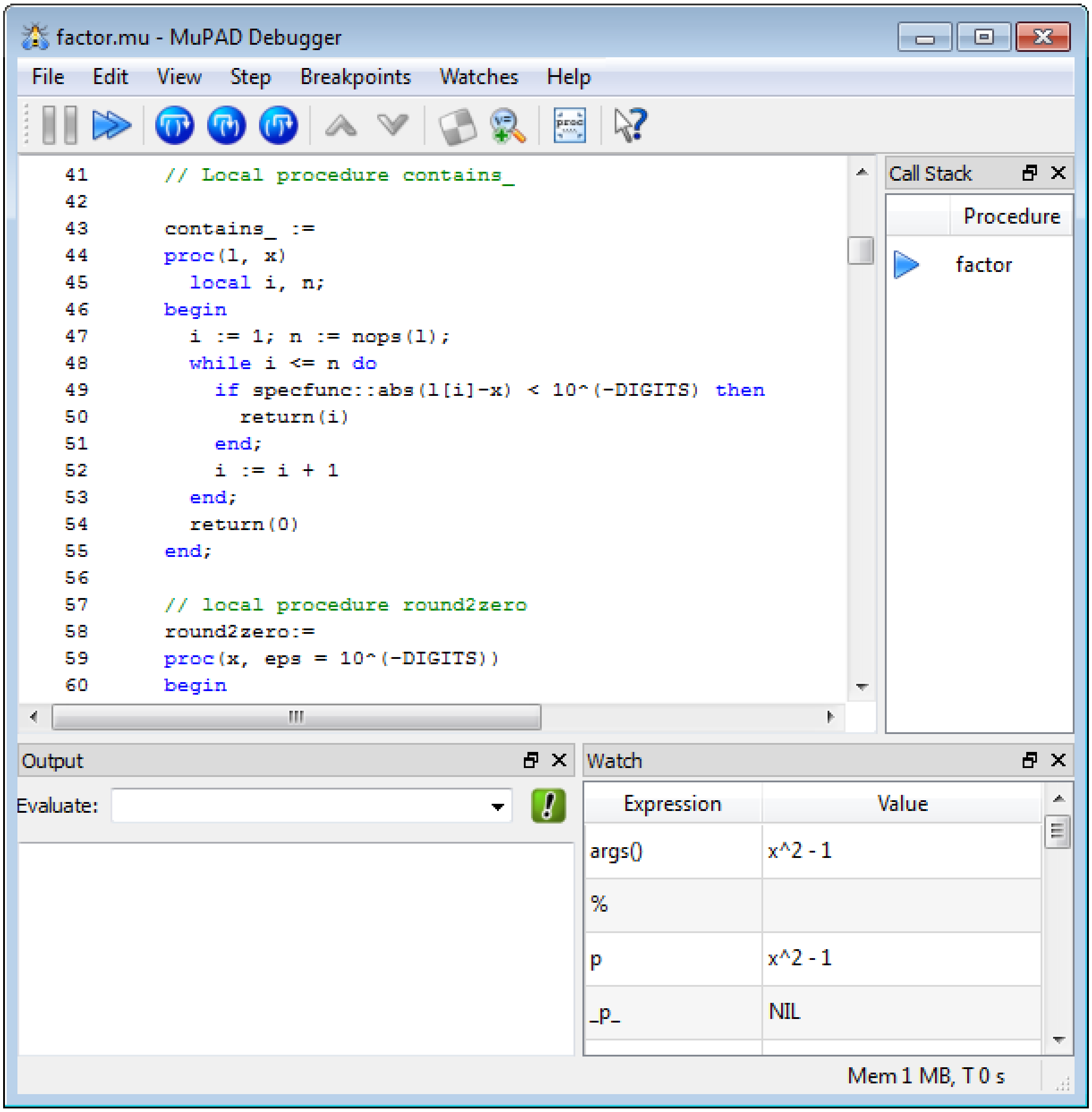
The MuPAD Debugger helps you find runtime errors in your code. This interface lets you:

- Execute your code step by step.
- Set breakpoints, including conditional breakpoints.
- Observe the values of the variables and expressions in each step.

To open the Debugger:

1. Open a new or existing MuPAD notebook. For instructions, see "Create, Open, and Save MuPAD Notebooks" on page 3-3.
2. In the main menu of a notebook, select **Notebook > Debug**.
3. In the resulting dialog box, enter the procedure call that you want to debug.

Alternatively, use the `debug` function in the MuPAD notebook.


factor.mu - MuPAD Debugger
File Edit View Step Breakpoints Watches Help
Call Stack
Procedure
factor
Output
Evaluate:
Watch
Mem 1 MB, T 0 s

```
// Local procedure contains_

contains_ :=
proc(l, x)
  local i, n;
begin
  i := 1; n := nops(l);
  while i <= n do
    if specfunc::abs(l[i]-x) < 10^(-DIGITS) then
      return(i)
    end;
    i := i + 1
  end;
  return(0)
end;

// local procedure round2zero
round2zero:=
proc(x, eps = 10^(-DIGITS))
begin
```

Expression	Value
args()	x^2 - 1
%	
p	x^2 - 1
_p_	NIL

The default layout of the Debugger window displays four panes:

- The main pane (top-left by default) displays the code that you debug. The Debugger only shows the code, but does not allow you to update it.
- The **Output** pane lets you type an expression and evaluate it anytime during the debugging process.
- The **Watch** pane shows values of the variables at each step during the debugging process.
- The **Call Stack** pane shows the names of the procedures that you debug.

You can close any pane, except for the main pane. If you close a pane, you can restore it again by selecting **View** and the name of the required pane. Using the **View** menu, you can also open the **Breakpoints** pane that shows the list of breakpoints in the code.

You cannot fix bugs directly in the Debugger window. If you work in the Debugger window and want to edit the code:

1. Open the file with the code in the MATLAB Editor.

   Tip If you did not yet save this code to a program file, display the code in a new Editor window by selecting **File > New Editor with Source**.

2. Close the Debugger if it is open.
3. Update the code in the MATLAB Editor and save it.
4. Open a notebook.
5. In the notebook, select **Notebook > Read Commands** from the main menu and navigate to your updated file.
6. Open the Debugger from the notebook.

For details about the MuPAD Debugger, see “Trace Errors with the MuPAD Debugger”.

Notebook Files and Program Files

The two main types of files in MuPAD are:

- Notebook files, or notebooks
- Program files

A *notebook file* has the extension `.mn` and lets you store the result of the work performed in the notebook interface. A notebook file can contain text, graphics, and any MuPAD commands and their outputs. A notebook file can also contain procedures and functions.

By default, a notebook file opens in the notebook interface. Creating a new notebook or opening an existing one does not automatically start the MuPAD engine. This means that although you can see the results of computations as they were saved, MuPAD does not remember evaluating them. (The "MuPAD Workspace" is empty.) You can evaluate any or all commands after opening a notebook.

A *program file* is a text file that contains any code snippet that you want to store separately from other computations. Saving a code snippet as a program file can be very helpful when you want to use the code in several notebooks. Typically, a program file contains a single procedure, but it also can contain one or more procedures or functions, assignments, statements, tests, or any other valid MuPAD code.

Tip If you use a program file to store a procedure, MuPAD does not require the name of that program file to match the name of a procedure.

The most common approach is to write a procedure and save it as a program file with the extension `.mu`. This extension allows the MATLAB Editor to recognize and open the file later. Nevertheless, a program file is just a text file. You can save a program file with any extension that you use for regular text files.

To evaluate the commands from a program file, you must execute a program file in a notebook. For details about executing program files, see "Read MuPAD Procedures" on page 3-40.

Source Code of the MuPAD Library Functions

You can display the source code of the MuPAD built-in library functions. If you work in the MuPAD notebook interface, enter `expose(name)`, where `name` is the library function name. The notebook interface displays the code as plain text with the original line breaks and indentations.

You can also display the code of a MuPAD library function in the MATLAB Command Window. To do this, use the `evalin` or `feval` function to call the MuPAD `expose` function:

```
sprintf(char(feval(symengine, 'expose', 'numlib::tau')))


ans =

proc(a)
  name numlib::tau;
begin
  if args(0) <> 1 then
    error(message("symbolic:numlib:IncorrectNumberOfArguments"))
  else
    if not testtype(a, Type::Numeric) then
      return(procname(args()))
    else
      if domtype(a) <> DOM_INT then
        error(message("symbolic:numlib:ArgumentInteger"))
      end_if
    end_if
  end_if;
  numlib::numdivisors(a)
end_proc
```

MuPAD also includes kernel functions written in C++. You cannot access the source code of these functions.

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 must be especially careful when working in MuPAD notebooks, since fewer of your typos 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
Auto-completion	**Tab**	**Ctrl+space bar**
Equality, inequality comparison	`==`, `~=`	`=`, `<>`

The next table lists 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`
`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`

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

	Symbolic Math Toolbox Definition	MuPAD Definition
Exponential integral	$\mathrm{expint}(x) = -\mathrm{Ei}(-x) =$ $\int_x^\infty \frac{\exp(-t)}{t}dt$ for $x > 0 =$ $\mathrm{Ei}(1,\ x)$.	$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}dt$ 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.

Copy Variables and Expressions Between MATLAB and MuPAD

You can copy a variable from 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. 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 open the notebook using the following syntax:

```
nb = mupad;
```

You can use any variable name for the handle `nb`. To open an existing notebook file, use the following syntax:

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

Here *`file_name`* must be a full path unless the notebook is in the current folder. The handle `nb` 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 this command in the MATLAB Command Window:

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

  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 variable `z` exists. To check that it exists, enter 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 variable `z` exists. Check that it exists 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 in the MATLAB Command Window:

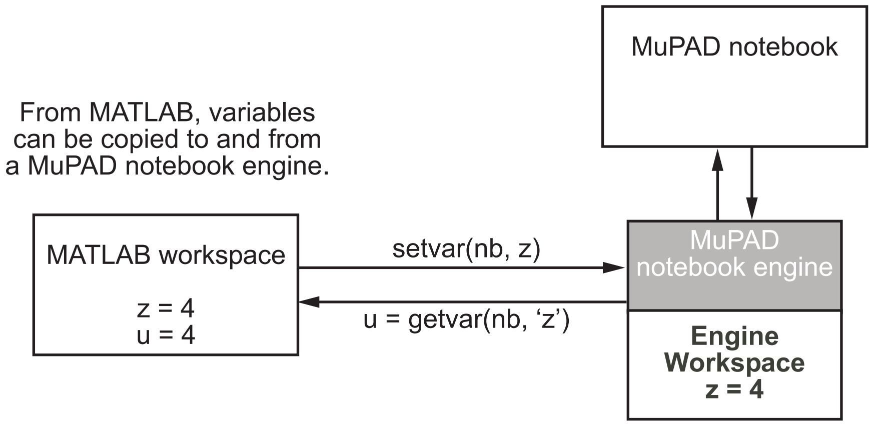
  ```
  MATLABvar = getVar(notebook_handle,'variable');
  ```

  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')
  ```

  Communication between the MATLAB workspace and the MuPAD notebook occurs in the notebook's engine. Therefore, 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 an undefined variable `z` in the MuPAD engine, the resulting MATLAB variable `u` is empty. For details, see "Synchronize Notebook and its Engine" on page 3-10.

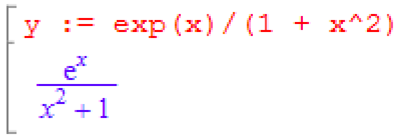
Tip Do all copying and assignments from the MATLAB workspace, not from a MuPAD notebook.



Copy and Paste 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 might get the text associated with the expression, or a picture, depending on your operating system and application support.

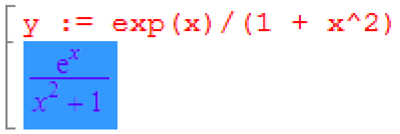
For example, consider this MuPAD expression:



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

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

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



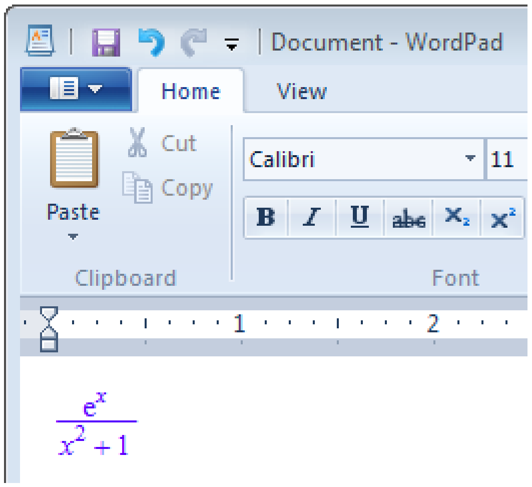
```
y := exp(x)/(1 + x^2)
```

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

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

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

If you paste it into Microsoft® WordPad on a Windows® system, the result is a picture.



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

Reserved Variable and Function Names

Both MATLAB and MuPAD have their own reserved keywords, such as function names, special values, and names of mathematical constants. Using reserved keywords as variable or function names can result in errors. If a variable name or a function name is a reserved keyword in one or both interfaces, you can get errors or incorrect results. If you work in one interface and a name is a reserved keyword in another interface, the error and warning messages are produced by the interface you work in. These messages can specify the cause of the problem incorrectly.

Tip The best approach is to avoid using reserved keywords as variable or function names, especially if you use both interfaces.

Conflicts Caused by MuPAD Function Names

In MuPAD, function names are protected. Normally, the system does not let you redefine a standard function or use its name as a variable. (To be able to modify a standard MuPAD function you must first remove its protection.) Even when you work in the MATLAB Command Window, the MuPAD engine handles symbolic computations. Therefore, MuPAD function names are reserved keywords in this case. Using a MuPAD function name while performing symbolic computations in the MATLAB Command Window can lead to incorrect results:

```
solve('D - 10')
```

The warning message does not indicate the real cause of the problem:

```
Warning: 1 equations in 0 variables.
Warning: Explicit solution could not be found.
> In solve at 81

ans =
[ empty sym ]
```

To fix this issue, use a variable name that is not a reserved keyword:

```
solve('x - 10')
```

```
ans =
10
```

Alternatively, use the `syms` function to declare `D` as a symbolic variable. Then call the symbolic solver without using quotes:

```
syms D;
solve(D - 10)
```

In this case, the toolbox replaces `D` with some other variable name before passing the expression to the MuPAD engine:

```
ans =
10
```

To list all MuPAD function names, enter this command in the MATLAB Command Window:

```
evalin(symengine, 'anames()')
```

If you work in a MuPAD notebook, enter:

```
anames()
```

Conflicts Caused by Syntax Conversions

Many mathematical functions, constants, and special values use different syntaxes in MATLAB and MuPAD. See the table MATLAB® vs. MuPAD® Expressions on page 3-20 for these expressions. When you use such functions, constants, or special values in the MATLAB Command Window, the toolbox internally converts the original MATLAB expression to the corresponding MuPAD expression and passes the converted expression to the MuPAD engine. When the toolbox gets the results of computations, it converts the MuPAD expressions in these results to the MATLAB expressions.

Suppose you write MuPAD code that introduces a new alias. For example, this code defines that `pow2` computes 2 to the power of `x`:

```
alias(pow2(x)=2^(x)):
```

Save this code in the `myProcPow.mu` program file in the `C:/MuPAD` folder. Before you can use this code, you must read the program file into the symbolic engine. Typically, you can read a program file into the symbolic engine by using `read`. This approach does not work for code defining aliases because `read` ignores them. If your code defines aliases, use `feval` to call the MuPAD `read` function. For example, enter these commands in the MATLAB Command Window:

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

Now you can use `pow2` to compute 2^x. For example, compute 2^2:

```
feval(eng, 'pow2', '2')

ans =
4
```

Now suppose you want to introduce the same alias and the following procedure in one program file:

```
alias(pow2(x)=2^(x)):

mySum := proc(n)
local i, s;
begin
    s := 0:
    for i from 1 to n do
        s := s + s/i + i
    end_for:
    return(s);
end_proc:
```

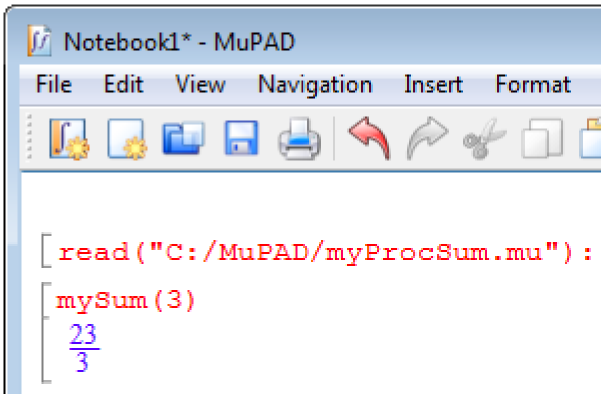
Save this code in the `myProcSum.mu` program file in the `C:/MuPAD` folder. Again, you must read the program file into the symbolic engine, and you cannot use `read` because the code defines an alias. Enter these commands in the MATLAB Command Window:

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

Error using mupadengine/feval (line 157)
MuPAD error: Error: Identifier expected (check aliases). [proc]

  Evaluating: read
```

```
Reading File: C:/MuPAD/myProcSum.mu
```

In this example, using the variable `i` causes the problem. The toolbox treats `i` as the imaginary unit, and therefore, converts it to `I` before passing the procedure to the MuPAD engine. Then the toolbox passes the converted code, with all instances of `i` replaced by `I`, to the MuPAD engine. This causes an error because `I` is protected, and the code tries to overwrite its value.

Reading the `myProcSum` procedure in a MuPAD notebook does not cause an error.



```
read("C:/MuPAD/myProcSum.mu"):
mySum(3)
```

$$\frac{23}{3}$$

Open MuPAD Interfaces from MATLAB

You can open an existing MuPAD notebook, a program file, or a graphic file (`.xvc` or `.xvz`) by double-clicking the file name. The system opens the file in the appropriate interface. Alternatively, use the `mupad` function or the MATLAB `open` function in the MATLAB Command Window and specify the path to the file:

```
mupad('H:\Documents\Notes\myNotebook.mn')
```

```
open('H:\Documents\Notes\myNotebook.mn')
```

If you perform computations in both interfaces, do not forget to use handles to notebooks. The toolbox uses this handle for communication between the MATLAB workspace and the MuPAD notebook. If you use the MATLAB Command Window only to open a notebook, and then perform all your computations in that notebook, you can skip using a handle. Also, you can skip using a handle when opening program files and graphic files.

Symbolic Math Toolbox also provides these functions for opening MuPAD files in the interfaces with which these files are associated:

- `openmn` opens a notebook in the notebook interface.
- `openmu` opens a program file with the extension `.mu` in the MATLAB Editor.
- `openxvc` opens an `XVC` graphic file in the MuPAD Graphics window.
- `openxvz` opens an `XVZ` graphic file in the MuPAD Graphics window.

These functions accomplish the same task as the `mupad` function. The system calls these functions when you double-click the file name.

You also can use the Welcome to MuPAD dialog box to open existing files as well as create new empty notebooks and program files. To open this dialog box, type `mupadwelcome` in the MATLAB Command Window. For details, see "Create, Open, and Save MuPAD Notebooks" on page 3-3.

After opening any MuPAD interface, you can use the main menu or the toolbar in that interface to open other interfaces or additional files.

Note You cannot access the MuPAD Debugger from the MATLAB Command Window.

For information about the Debugger, see "Edit and Debug MuPAD Code" on page 3-12.

Call Built-In MuPAD Functions from MATLAB Command Window

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

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]')

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

feval

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

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

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

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

z =
55
```

The next example compares the use of a symbolic solution of an equation to the solution returned by the MuPAD numeric `fsolve` function near the point `x = 3`. The symbolic solver returns these results:

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

ans =
 0
 0
```

The numeric solver `fsolve` returns this result:

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

ans =
x == 3.0699801238394654654386548746678
```

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

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)
```

Variable `y` is not expressed in terms of `x` because `y` is unknown to the MuPAD engine.

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;
```

Computations in MATLAB Command Window vs. MuPAD Notebook Interface

When computing with Symbolic Math Toolbox, you can choose to work in the MATLAB Command Window or in the MuPAD notebook interface. The MuPAD engine that performs all symbolic computations is the same for both interfaces. The choice of the interface mostly depends on your preferences.

Working in the MATLAB Command Window lets you perform all symbolic computations using the familiar MATLAB language. The toolbox contains hundreds of MATLAB symbolic functions for common tasks, such as differentiation, integration, simplification, transforms, and equation solving. If your task requires a few specialized symbolic functions not available directly from this interface, you can use `evalin` or `feval` to call MuPAD functions. See "Call Built-In MuPAD Functions from MATLAB Command Window" on page 3-32.

Working in the MATLAB Command Window is recommended if you use other toolboxes or MATLAB as a primary tool for your current task and only want to embed a few symbolic computations in your code.

Working in the MuPAD notebook interface requires you to use the MuPAD language, which is optimized for symbolic computations. In addition to solving common mathematical problems, MuPAD functions cover specialized areas, such as number theory and combinatorics. Also, for some computations the performance is better in the MuPAD notebook interface than in the MATLAB Command Window. The reason is that the engine returns the results in the MuPAD language. To display them in the MATLAB Command Window, the toolbox translates the results to the MATLAB language.

Working in the MuPAD notebook interface is recommended when your task mainly consists of symbolic computations. It is also recommended if you want to document your work and results, for example, embed graphics, animations, and descriptive text with your calculations. Symbolic results computed in the MuPAD notebook interface can be accessed from the MATLAB Command Window, which helps you integrate symbolic results into larger MATLAB applications.

Learning the MuPAD language and using the MuPAD notebook interface for your symbolic computations provides the following benefits.

Results Displayed in Typeset Math

By default, the MuPAD notebook interface displays results in typeset math making them look very similar to what you see in mathematical books. In addition, the notebook interface

- Uses standard mathematical notations in output expressions.
- Uses abbreviations to make a long output expression with common subexpressions shorter and easier to read. You can disable abbreviations.
- Wraps long output expressions, including long numbers, fractions and matrices, to make them fit the page. If you resize the notebook window, MuPAD automatically adjusts outputs. You can disable wrapping of output expressions.

Alternatively, you can display pretty-printed outputs similar to those that you get in the MATLAB Command Window when you use `pretty`. You can also display outputs as plain text. For details, see "Use Different Output Modes".

In a MuPAD notebook, you can copy or move output expressions, including expressions in typeset math, to any input or text region within the notebook, or to another notebook. If you copy or move an output expression to an input region, the expression appears as valid MuPAD input.

Graphics and Animations

The MuPAD notebook interface provides very extensive graphic capabilities to help you visualize your problem and display results. Here you can create a wide variety of plots, including:

- 2-D and 3-D plots in Cartesian, polar, and spherical coordinates
- Plots of continuous and piecewise functions and functions with singularities
- Plots of discrete data sets
- Surfaces and volumes by using predefined functions
- Turtle graphics and Lindenmayer systems

- Animated 2-D and 3-D plots

Graphics in the MuPAD notebook interface is interactive. You can explore and edit plots, for example:

- Change colors, fonts, legends, axes appearance, grid lines, tick marks, line, and marker styles.
- Zoom and rotate plots without reevaluating them.
- Display coordinates of any point on the plot.

After you create and customize a plot, you can export it to various vector and bitmap file formats, including EPS, SVG, PDF, PNG, GIF, BMP, TIFF, and JPEG. The set of the file formats available for exporting graphics from a MuPAD notebook can be limited by your operating system.

You can export animations as AVI files or as sequences of static images.

More Functionality in Specialized Mathematical Areas

While both MATLAB and MuPAD interfaces provide functions for performing common mathematical tasks, the notebook interface also provides functions that cover many specialized areas. For example, MuPAD libraries support computations in the following areas:

- Combinatorics
- Graph theory
- Gröbner bases
- Linear optimization
- Polynomial algebra
- Number theory
- Statistics

MuPAD libraries also provide large collections of functions for working with ordinary differential equations, integral and discrete transforms, linear algebra, and more.

More Options for Common Symbolic Functions

Most functions for performing common mathematical computations are available in both MATLAB and MuPAD interfaces. For example, you can solve equations and systems of equations using `solve`, simplify expressions using `simplify`, compute integrals using `int`, and compute limits using `limit`. Note that although the function names are the same, the syntax of the function calls depends on the interface that you use.

Results of symbolic computations can be very long and complicated, especially because the toolbox assumes all values to be complex by default. For many symbolic functions you can use additional parameters and options to help you limit the number and complexity and also to control the form of returned results. For example, `solve` accepts the `Real` option that lets you restrict all symbolic parameters of an equation to real numbers. It also accepts the `VectorFormat` option that you can use to get solutions of a system as a set of vectors.

Typically, the functions available in the notebook interface accept more options than the analogous functions in the MATLAB Command Window. For example, in the notebook interface you can use the `VectorFormat` option. This option is not directly available for the `solve` function called in the MATLAB Command Window.

Possibility to Expand Existing Functionality

The MuPAD programming language supports multiple programming styles, including imperative, functional, and object-oriented programming. The system includes a few basic functions written in C++, but the majority of the MuPAD built-in functionality is implemented as library functions written in the MuPAD language. You can extend the built-in functionality by writing custom symbolic functions and libraries, defining new function environments, data types, and operations on them in the MuPAD language. MuPAD implements data types as domains (classes). Domains with similar mathematical structure typically belong to a category. Domains and categories allow you to use the concepts of inheritance, overloading methods and operators. The language also uses axioms to state properties of domains and categories.

"Object-Oriented Programming" contains information to get you started with object-oriented programming in MuPAD.

Use Your Own MuPAD Procedures

Write MuPAD Procedures

A MuPAD procedure is a text file that you can write in any text editor . The recommended practice is to use the MATLAB Editor.

To define a procedure, use the `proc` function. Enclose the code in the `begin` and `end_proc` functions:

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

By default, a MuPAD procedure returns the result of the last executed command. You can force a procedure to return another result by using `return`. In both cases, a procedure returns only one result. To get multiple results from a procedure, use the `print` function or data structures inside the procedure.

- If you just want to display the results, and do not need to use them in further computations, use the `print` function. With `print`, your procedure still returns one result, but prints intermediate results on screen. For example, this procedure prints the value of its argument in each call:

  ```
  myProcPrint:= proc(n)
  begin
     print(n);
     if n = 0 or n = 1 then
        return(1);
     end_if;
     n * myProcPrint(n - 1);
  end_proc:
  ```

- If you want to use multiple results of a procedure, use ordered data structures, such as lists or matrices as return values. In this case, the result of the last executed command is technically one object, but it can contain more than one value. For example, this procedure returns the list of two entries:

```
myProcSort:= proc(a, b)
begin
  if a < b then
    [a, b]
  else
    [b, a]
  end_if;
end_proc:
```

Avoid using unordered data structures, such as sequences and sets, to return multiple results of a procedure. The order of the entries in these structures can change unpredictably.

When you save the procedure, it is recommended to use the extension `.mu`. For details, see "Notebook Files and Program Files" on page 3-16. The name of the file can differ from the name of the procedure. Also, you can save multiple procedures in one file.

Steps to Take Before Calling a Procedure

To be able to call a procedure, you must first execute it in a notebook. If you write a procedure in the same notebook, simply evaluate the input region that contains the procedure. If you write a procedure in a separate file, you must *read* the procedure into a notebook. *Reading* a procedure means finding and executing the procedure.

Read MuPAD Procedures

If you work in the MuPAD notebook interface and create a separate program file that contains a procedure, use one of the following methods to execute the procedure in a notebook. The first approach is to select **Notebook > Read Commands** from the main menu.

Alternatively, you can use the `read` function. The function call `read(filename)` searches for the program file in this order:

1 Folders specified by the environment variable `READPATH`

2 `filename` regarded as an absolute path

3 Current folder (depends on the operating system)

4 Folders specified by the environment variable `LIBPATH`

If you want to call the procedure from the MATLAB Command Window, you still need to execute that procedure before calling it. See "Call Your Own MuPAD Procedures" on page 3-41.

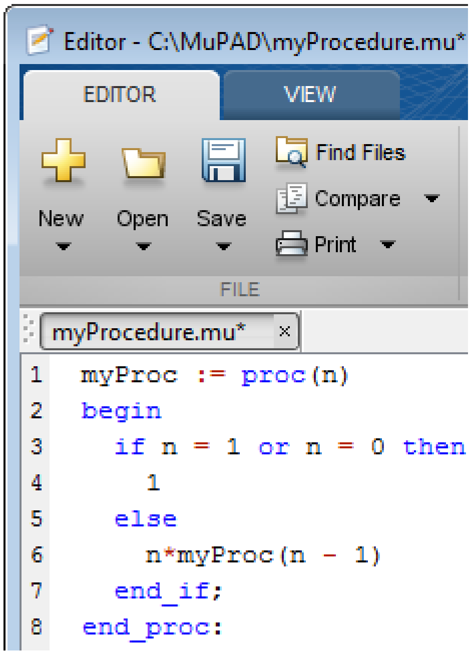
Use Startup Commands and Scripts

Alternatively, you can add a MuPAD procedure to startup commands of a particular notebook. This method lets you execute the procedure every time you start a notebook engine. Startup commands are executed silently, without any visible outputs in the notebook. You can copy the procedure to the dialog box that specifies startup commands or attach the procedure as a startup script. For information, see "Hide Code Lines".

Call 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 Window.

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



```
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`.

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

```
read(symengine, '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:

```
feval(symengine, 'myProc', 15)
```

```
ans =
1307674368000
```

Clear Assumptions and Reset the Symbolic Engine

The symbolic engine workspace associated with the MATLAB workspace is usually empty. The MATLAB workspace tracks 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, positive, integer, greater or less than some value, and so on. These assumptions can affect solutions to equations, simplifications, and transformations, as explained in "Effects of Assumptions on Computations" on page 3-45.

Note These 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, for example, using the commands

```
syms x
assume(x,'real')
```

or

```
syms x
assume(x > 0)
```

then 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 details, see "Check Assumptions Set On Variables" on page 3-44 and "Effects of Assumptions on Computations" on page 3-45.

If you reset the symbolic engine by entering the command

```
reset(symengine)
```

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` command.

This example shows 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` or `syms x;` `assume(x > 0)`	`x`	`x > 0`
2	`clear x`	empty	`x > 0`
3	`syms x`	`x`	`x > 0`
4	`syms x clear`	`x`	empty

Check Assumptions Set On Variables

To check whether a variable, say `x`, has any assumptions in the symbolic engine workspace associated with the MATLAB workspace, use the `assumptions` function in the MATLAB Command Window:

```
assumptions(x)
```

If the function returns an empty symbolic object, there are no additional assumptions on the variable. (The default assumption is that `x` can be any complex number.) Otherwise, there are additional assumptions on the value of that variable.

For example, while declaring the symbolic variable `x` make an assumption that the value of this variable is a real number:

```
syms x real;
assumptions(x)
```

```
ans =
x in R_
```

Another way to set an assumption is to use the `assume` function:

```
syms z;
assume(z ~= 0);
assumptions(z)

ans =
z ~= 0
```

To see assumptions set on all variables in the MATLAB workspace, use `assumptions` without input arguments:

```
assumptions

ans =
[ x in R_, z ~= 0]
```

Clear assumptions set on `x` and `z`:

```
syms x z clear;

assumptions

ans =
[ empty sym ]
```

Effects of Assumptions on Computations

Assumptions can affect many computations, including results returned by the `solve` function. They also can affect the results of simplifications. For example, solve this equation without any additional assumptions on its variable:

```
syms x
solve(x^4 == 1, x)

ans =
  1
 -1
  i
 -i
```

Now solve the same equation assuming that x is real:

```
syms x real
solve(x^4 == 1, x)

ans =
  1
 -1
```

Use the assumeAlso function to add the assumption that x is also positive:

```
assumeAlso(x > 0);
solve(x^4 == 1, x)

ans =
  1
```

Clearing x does not change the underlying assumptions that x is real and positive:

```
clear x
syms x
assumptions(x)
solve(x^4 == 1, x)

ans =
[ 0 < x, x in R_]

ans =
1
```

Clearing x with syms x clear clears the assumption:

```
syms x clear
assumptions(x)

ans =
[ empty sym ]
```

Tip `syms x clear` clears the assumptions and the value of `x`. To clear the assumptions only, use `sym('x','clear')`.

Create MATLAB Functions from MuPAD Expressions

Symbolic Math Toolbox lets you create a MATLAB function from a symbolic expression. A MATLAB function created from a symbolic expression accepts numeric arguments and evaluates the expression applied to the arguments. You can generate a function handle or a file that contains a MATLAB function. The generated file is available for use in any MATLAB calculation, independent of a license for Symbolic Math Toolbox functions.

If you work in the MATLAB Command Window, see "Generate MATLAB Functions" on page 2-131.

When you use the MuPAD notebook interface, all your symbolic expressions are written in the MuPAD language. To be able to create a MATLAB function from such expressions, you must convert it to the MATLAB language. There are two approaches for converting a MuPAD expression to the MATLAB language:

- Assign the MuPAD expression to a variable, and copy that variable from a notebook to the MATLAB workspace. This approach lets you create a function handle or a file that contains a MATLAB function. It also requires using a handle to the notebook.
- Generate MATLAB code from the MuPAD expression in a notebook. This approach limits your options to creating a file. You can skip creating a handle to the notebook.

The generated MATLAB function can depend on the approach that you chose. For example, code can be optimized differently or not optimized at all.

Suppose you want to create a MATLAB function from a symbolic matrix that converts spherical coordinates of any point to its Cartesian coordinates. First, open a MuPAD notebook with the handle `notebook_handle`:

```
notebook_handle = mupad;
```

In this notebook, create the symbolic matrix `S` that converts spherical coordinates to Cartesian coordinates:

```
x := r*sin(a)*cos(b):
y := r*sin(a)*sin(b):
```

```
z := r*cos(b):
S := matrix([x, y, z]):
```

Now convert matrix `S` to the MATLAB language. Choose the best approach for your task.

Copy MuPAD Variables to the MATLAB Workspace

If your notebook has a handle, like `notebook_handle` in this example, you can copy variables from that notebook to the MATLAB workspace with the `getVar` function, and then create a MATLAB function. For example, to convert the symbolic matrix `S` to a MATLAB function:

1 Copy variable `S` to the MATLAB workspace:

```
S = getVar(notebook_handle,'S')
```

Variable `S` and its value (the symbolic matrix) appear in the MATLAB workspace and in the MATLAB Command Window:

```
S =
 r*cos(b)*sin(a)
 r*sin(a)*sin(b)
        r*cos(b)
```

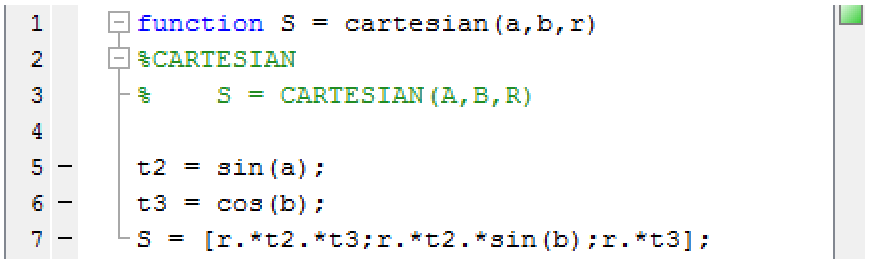
2 Use `matlabFunction` to create a MATLAB function from the symbolic matrix. To generate a MATLAB function handle, use `matlabFunction` without additional parameters:

```
h = matlabFunction(S)

h =
    @(a,b,r)[r.*cos(b).*sin(a);r.*sin(a).*sin(b);r.*cos(b)]
```

To generate a file containing the MATLAB function, use the parameter `file` and specify the path to the file and its name. For example, save the MATLAB function to the file `cartesian.m` in the current folder:

```
S = matlabFunction(S,'file', 'cartesian.m');
```

You can open and edit `cartesian.m` in the MATLAB Editor.



```
function S = cartesian(a,b,r)
%CARTESIAN
%     S = CARTESIAN(A,B,R)

t2 = sin(a);
t3 = cos(b);
S = [r.*t2.*t3;r.*t2.*sin(b);r.*t3];
```

Generate MATLAB Code in a MuPAD Notebook

To generate the MATLAB code from a MuPAD expression within the MuPAD notebook, use the `generate::MATLAB` function. Then, you can create a new file that contains an empty MATLAB function, copy the code, and paste it there. Alternatively, you can create a file with a MATLAB formatted string representing a MuPAD expression, and then add appropriate syntax to create a valid MATLAB function.

1 In the MuPAD notebook interface, use the `generate::MATLAB` function to generate MATLAB code from the MuPAD expression. Instead of printing the result on screen, use the `fprint` function to create a file and write the generated code to that file:

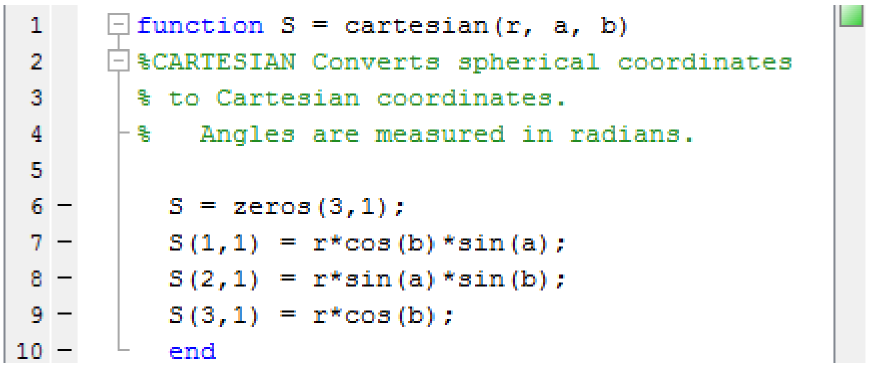
```
fprint(Unquoted, Text, "cartesian.m", generate::MATLAB(S)):
```

Note If the file with this name already exists, `fprint` replaces the contents of this file with the converted expression.

2 Open `cartesian.m`. It contains a MATLAB formatted string representing matrix `S`:

```
S = zeros(3,1);
S(1,1) = r*cos(b)*sin(a);
S(2,1) = r*sin(a)*sin(b);
S(3,1) = r*cos(b);
```

3 To convert this file to a valid MATLAB function, add the keywords `function` and `end`, the function name (must match the file name), input and output arguments, and comments:



```
function S = cartesian(r, a, b)
%CARTESIAN Converts spherical coordinates
% to Cartesian coordinates.
%   Angles are measured in radians.

  S = zeros(3,1);
  S(1,1) = r*cos(b)*sin(a);
  S(2,1) = r*sin(a)*sin(b);
  S(3,1) = r*cos(b);
  end
```

Create MATLAB Function Blocks from MuPAD Expressions

Symbolic Math Toolbox lets you create a MATLAB function block from a symbolic expression. The generated block is available for use in Simulink models, whether or not the computer that runs the simulations has a license for Symbolic Math Toolbox.

If you work in the MATLAB Command Window, see "Generate MATLAB Function Blocks" on page 2-136.

The MuPAD notebook interface does not provide a function for generating a block. Therefore, to be able to create a block from the MuPAD expression:

1 In a MuPAD notebook, assign that expression to a variable.

2 Use the `getVar` function to copy that variable from a notebook to the MATLAB workspace.

For details about these steps, see "Copy MuPAD Variables to the MATLAB Workspace" on page 3-49.

When the expression that you want to use for creating a MATLAB function block appears in the MATLAB workspace, use the `matlabFunctionBlock` function to create a block from that expression.

For example, open a MuPAD notebook with the handle `notebook_handle`:

```
notebook_handle = mupad;
```

In this notebook, create the following symbolic expression:

```
r := sqrt(x^2 + y^2)
```

Use `getVar` to copy variable `r` to the MATLAB workspace:

```
r = getVar(notebook_handle,'r')
```

Variable `r` and its value appear in the MATLAB workspace and in the MATLAB Command Window:

```
r =
(x^2 + y^2)^(1/2)
```

Before generating a MATLAB Function block from the expression, create an empty model or open an existing one. For example, create and open the new model `my_system`:

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

Since the variable and its value are in the MATLAB workspace, you can use `matlabFunctionBlock` to generate the block `my_block`:

```
matlabFunctionBlock('my_system/my_block', r)
```

You can open and edit the block in the MATLAB Editor. To open the block, double-click it:

```
function r = my_block(x,y)
%#codegen

r = sqrt(x.^2+y.^2);
```

Create Simscape Equations from MuPAD Expressions

Symbolic Math Toolbox lets you integrate symbolic computations into the Simscape modeling workflow by using the results of these computations in the Simscape equation section.

If you work in the MATLAB Command Window, see “Generate Simscape Equations” on page 2-139.

If you work in the MuPAD notebook interface, you can:

- Assign the MuPAD expression to a variable, copy that variable from a notebook to the MATLAB workspace, and use `simscapeEquation` to generate the Simscape equation in the MATLAB Command Window.
- Generate the Simscape equation from the MuPAD expression in a notebook.

In both cases, to use the generated equation, you must manually copy the equation and paste it to the equation section of the Simscape component file.

For example, follow these steps to generate a Simscape equation from the solution of the ordinary differential equation computed in the notebook interface:

1. Open a MuPAD notebook with the handle `notebook_handle`:

   ```
   notebook_handle = mupad;
   ```

2. In this notebook, define the following equation:

   ```
   s:= ode(y'(t) = y(t)^2, y(t)):
   ```

3. Decide whether you want to generate the Simscape equation in the MuPAD notebook interface or in the MATLAB Command Window.

GenerateSimscape Equations in the MuPAD Notebook Interface

To generate the Simscape equation in the same notebook, use `generate::Simscape`. To display generated Simscape code on screen, use the `print` function. To remove quotes and expand special characters like line breaks and tabs, use the printing option `Unquoted`:

```
print(Unquoted, generate::Simscape(s))
```

This command returns the Simscape equation that you can copy and paste to the Simscape equation section:

```
  -y^2+y.der == 0.0;
```

Generate Simscape Equations in the MATLAB Command Window

To generate the Simscape equation in the MATLAB Command Window, follow these steps:

1. Use `getVar` to copy variable `s` to the MATLAB workspace:

   ```
   s = getVar(notebook_handle, 's')
   ```

   Variable `s` and its value appear in the MATLAB workspace and in the MATLAB Command Window:

   ```
   s =
   ode(D(y)(t) - y(t)^2, y(t))
   ```

2. Use `simscapeEquation` to generate the Simscape equation from `s`:

   ```
   simscapeEquation(s)
   ```

You can copy and paste the generated equation to the Simscape equation section. Do not copy the automatically generated variable `ans` and the equal sign that follows it.

```
ans =
s == (-y^2+y.der == 0.0);
```

Functions — Alphabetical List

abs

Purpose

Absolute value of real or complex value

Syntax

```
abs(z)
abs(A)
```

Description

`abs(z)` returns the absolute value of `z`. If `z` is complex, `abs(z)` returns the complex modulus (magnitude) of `z`.

`abs(A)` returns the absolute value of each element of `A`. If `A` is complex, `abs(A)` returns the complex modulus (magnitude) of each element of `A`.

Tips

- Calling `abs` for a number that is not a symbolic object invokes the MATLAB `abs` function.

Input Arguments

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Complex Modulus

The absolute value of a complex number $z = x + y{*}i$ is the value $|z| = \sqrt{x^2 + y^2}$. Here, x and y are real numbers. The absolute value of a complex number is also called a complex modulus.

Examples

Compute absolute values of these symbolic real numbers:

```
[abs(sym(1/2)), abs(sym(0)), abs(sym(pi) - 4)]

ans =
[ 1/2, 0, 4 - pi]
```

Compute the absolute values of each element of matrix `A`:

```
A = sym([(1/2 + i), -25; i*(i + 1), pi/6 - i*pi/2]);
abs(A)

ans =
[ 5^(1/2)/2,                          25]
[   2^(1/2), (pi*5^(1/2)*18^(1/2))/18]
```

Compute the absolute value of this expression assuming that the value `x` is negative:

```
syms x
assume(x < 0)
abs(5*x^3)

ans =
-5*x^3
```

For further computations, clear the assumption:

```
syms x clear
```

See Also

`absangle` | `imag` | `real` | `sign`

Purpose

Adjoint of symbolic square matrix

Syntax

```
X = adjoint(A)
```

Description

`X = adjoint(A)` returns the adjoint matrix `X` of `A`. The adjoint of a matrix `A` is the matrix `X`, such that `A*X = det(A)*eye(n) = X*A`, where `n` is the number of rows in `A` and `eye(n)` is the `n`-by-`n` identity matrix.

Input Arguments

A

Symbolic square matrix.

Output Arguments

X

Symbolic square matrix of the same size as `A`.

Definitions

Adjoint of a Square Matrix

The adjoint of a square matrix A is the square matrix X, such that the (i,j)-th entry of X is the (j,i)-th cofactor of A.

Cofactor of a Matrix

The (j,i)-th cofactor of A is defined as

$$a_{ji}' = (-1)^{i+j} \det\left(A_{ij}\right)$$

A_{ij} is the submatrix of A obtained from A by removing the i-th row and j-th column.

Examples

Compute the adjoint of this symbolic matrix:

```
syms x y z
A = sym([x y z; 2 1 0; 1 0 2]);
X = adjoint(A)

X =
[  2,    -2*y,      -z]
[ -4, 2*x - z,     2*z]
[ -1,       y, x - 2*y]
```

Verify that `A*X = det(A)*eye(3)`, where `eye(3)` is the 3-by-3 identity matrix:

```
isAlways(A*X == det(A)*eye(3))

ans =
     1     1     1
     1     1     1
     1     1     1
```

Also verify that `det(A)*eye(3) = X*A`:

```
isAlways(det(A)*eye(3) == X*A)

ans =
     1     1     1
     1     1     1
     1     1     1
```

Compute the inverse of this matrix by computing its adjoint and determinant:

```
syms a b c d
A = [a b; c d];
invA = adjoint(A)/det(A)

invA =
[  d/(a*d - b*c), -b/(a*d - b*c)]
[ -c/(a*d - b*c),  a/(a*d - b*c)]
```

Verify that `invA` is the inverse of `A`:

```
isAlways(invA == inv(A))

ans =
            1      1
            1      1
```

See Also

det | invlinalg::adjoint | rank

Purpose

Airy function

Syntax

```
airy(x)
airy(0,x)
airy(1,x)
airy(2,x)
airy(3,x)
```

Description

`airy(x)` returns the Airy function of the first kind, Ai(x).

`airy(0,x)` is equivalent to `airy(x)`.

`airy(1,x)` returns the derivative of the Airy function of the first kind, Ai′(x).

`airy(2,x)` returns the Airy function of the second kind, Bi(x).

`airy(3,x)` returns the derivative of the Airy function of the second kind, Bi′(x).

Tips

- Calling `airy` for a number that is not a symbolic object invokes the MATLAB `airy` function.

Input Arguments

x

Symbolic number, variable, expression, or vector or matrix of symbolic numbers, variables, expressions.

Definitions

Airy Functions

The Airy functions Ai(x) and Bi(x) are linearly independent solutions of this differential equation:

$$\frac{\partial^2 y}{\partial x^2} - xy = 0$$

Examples

Solve this second-order differential equation. The solutions are the Airy functions of the first and the second kind.

```
syms y(x)
dsolve(diff(y, 2) - x*y == 0)

ans =
C2*airy(0, x) + C3*airy(2, x)
```

Verify that the Airy function of the first kind is a valid solution of the Airy differential equation:

```
syms x
simplify(diff(airy(0, x), x, 2) - x*airy(0, x)) == 0

ans =
     1
```

Verify that the Airy function of the second kind is a valid solution of the Airy differential equation:

```
simplify(diff(airy(2, x), x, 2) - x*airy(2, x)) == 0

ans =
     1
```

Compute the Airy functions for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[airy(1), airy(1, 3/2 + 2*i), airy(2, 2), airy(3, 1/101)]

ans =
   0.1353             0.1641 + 0.1523i
3.2981             0.4483
```

Compute the Airy functions for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `airy` returns unresolved symbolic calls.

```
[airy(sym(1)), airy(1, sym(3/2 + 2*i)), airy(2,
sym(2)), airy(3, sym(1/101))]

ans =
[ airy(0, 1), airy(1, 3/2 + 2*i), airy(2,
2), airy(3, 1/101)]
```

For symbolic variables and expressions, `airy` also returns unresolved symbolic calls:

```
syms x y
[airy(x), airy(1, x^2), airy(2, x - y), airy(3, x*y)]

ans =
[ airy(0, x), airy(1, x^2), airy(2, x - y), airy(3, x*y)]
```

Compute the Airy functions for $x = 0$. The Airy functions have special values for this parameter.

```
airy(sym(0))

ans =
3^(1/3)/(3*gamma(2/3))

airy(1, sym(0))

ans =
-(3^(1/6)*gamma(2/3))/(2*pi)

airy(2, sym(0))

ans =
3^(5/6)/(3*gamma(2/3))
```

```
airy(3, sym(0))

ans =
(3^(2/3)*gamma(2/3))/(2*pi)
```

If you do not use `sym`, you call the MATLAB `airy` function that returns numeric approximations of these values:

```
[airy(0), airy(1, 0), airy(2, 0), airy(3, 0)]

ans =
    0.3550   -0.2588    0.6149    0.4483
```

Differentiate the expressions involving the Airy functions:

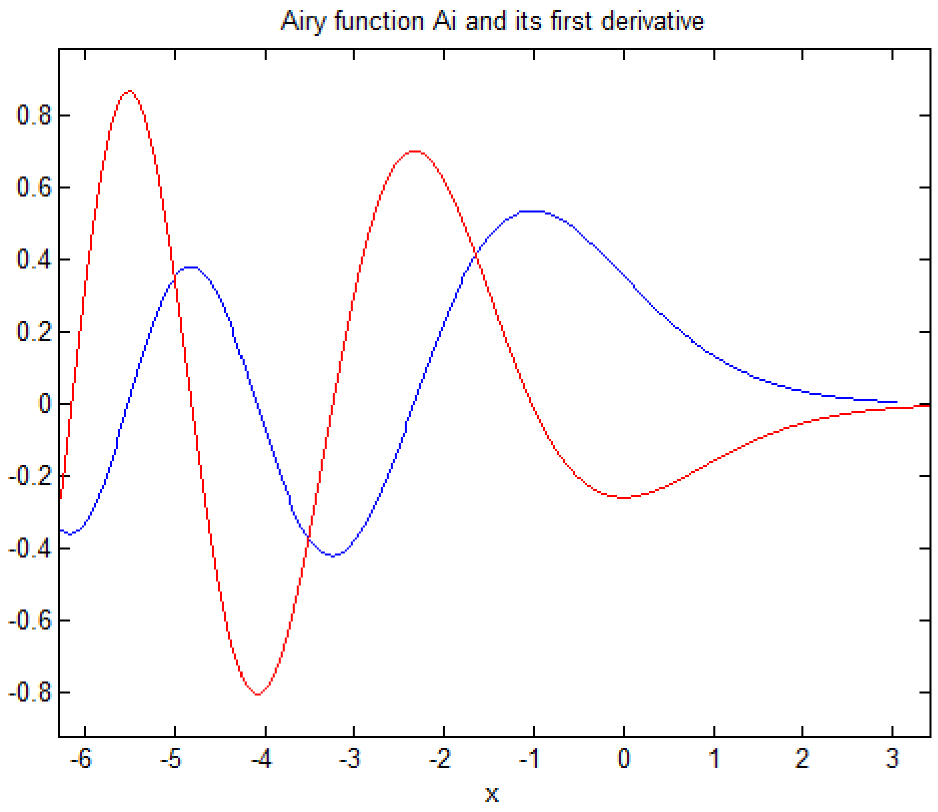
```
syms x y
diff(airy(x^2))
diff(diff(airy(3, x^2 + x*y -y^2), x), y)

ans =
2*x*airy(1, x^2)

ans =
airy(2, x^2 + x*y - y^2)*(x^2 + x*y - y^2) +...
airy(2, x^2 + x*y - y^2)*(x - 2*y)*(2*x + y) +...
airy(3, x^2 + x*y - y^2)*(x - 2*y)*(2*x + y)*(x^2 + x*y - y^2)
```

Compute the Airy function of the first kind for the elements of matrix `A`:

```
syms x
A = [-1, 0; 0, x];
airy(A)

ans =
[             airy(0, -1), 3^(1/3)/(3*gamma(2/3))]
[ 3^(1/3)/(3*gamma(2/3)),             airy(0, x)]
```

Plot the Airy function Ai(*x*) and its derivative Ai'(*x*):

```
syms x
ezplot(airy(x));
hold on

p = ezplot(airy(1,x));
set(p,'Color','red')

title('Airy function Ai and its first derivative')
hold off
```



References

Antosiewicz, H. A. “Bessel Functions of Fractional Order.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`besseli` | `besselj` | `besselk` | `bessely` | `mfun` | `mfunlist`

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Test whether all equations and inequalities represented as elements of symbolic array are valid

Syntax

```
all(A)
all(A,dim)
```

Description

`all(A)` tests whether all elements of `A` return logical `1` (`true`). If `A` is a matrix, `all` tests all elements of each column. If `A` is a multidimensional array, `all` tests all elements along one dimension.

`all(A,dim)` tests along the dimension of `A` specified by `dim`.

Tips

- If `A` is an empty symbolic array, `all(A)` returns logical `1`.
- If some elements of `A` are just numeric values (not equations or inequalities), `all` converts these values as follows. All numeric values except 0 become logical `1`. The value 0 becomes logical `0`.
- If `A` is a vector and all its elements return logical `1`, `all(A)` returns logical `1`. If one or more elements are zero, `all(A)` returns logical `0`.
- If `A` is a multidimensional array, `all(A)` treats the values along the first dimension that is not equal to 1 (nonsingleton dimension) as vectors, returning logical `1` or `0` for each vector.

Input Arguments

A

Symbolic vector, matrix, or multidimensional symbolic array. For example, it can be an array of symbolic equations, inequalities, or logical expressions with symbolic subexpressions.

dim

Integer. For example, if `A` is a matrix, `all(A,1)` tests elements of each column and returns a row vector of logical `1`s and `0`s. `all(A,2)` tests elements of each row and returns a column vector of logical `1`s and `0`s.

Default: The first dimension that is not equal to 1 (non-singleton dimension). For example, if A is a matrix, all(A) treats the columns of A as vectors.

Examples

Create vector V that contains the symbolic equation and inequalities as its elements:

```
syms x
V = [x ~= x + 1, abs(x) >= 0, x == x];
```

Use all to test whether all of them are valid for all values of x:

```
all(V)

ans =
     1
```

Create this matrix of symbolic equations and inequalities:

```
syms x
M = [x == x, x == abs(x); abs(x) >= 0, x ~= 2*x]

M =
[      x == x, x == abs(x)]
[ 0 <= abs(x),    x ~= 2*x]
```

Use all to test equations and inequalities of this matrix. By default, all tests whether all elements of each column are valid for all possible values of variables. If all equations and inequalities in the column are valid (return logical 1), then all returns logical 1 for that column. Otherwise, it returns logical 0 for the column. Thus, it returns 1 for the first column and 0 for the second column:

```
all(M)

ans =
     1     0
```

Create this matrix of symbolic equations and inequalities:

```
syms x
M = [x == x, x == abs(x); abs(x) >= 0, x ~= 2*x]

M =
[      x == x, x == abs(x)]
[ 0 <= abs(x),    x ~= 2*x]
```

For matrices and multidimensional arrays, `all` can test all elements along the specified dimension. To specify the dimension, use the second argument of `all`. For example, to test all elements of each column of a matrix, use the value 1 as the second argument:

```
all(M, 1)

ans =
     1     0
```

To test all elements of each row, use the value 2 as the second argument:

```
all(M, 2)

ans =
     0
     1
```

Test whether all elements of this vector return logical `1`s. Note that `all` also converts all numeric values outside equations and inequalities to logical `1`s and `0`s. The numeric value 0 becomes logical `0`:

```
syms x
all([0, x == x])

ans =
     0
```

All nonzero numeric values, including negative and complex values, become logical 1s:

```
all([1, 2, -3, 4 + i, x == x])
```

```
ans =
     1
```

See Also

and | any | isAlways | logical | not | or | xor

Purpose

Logical AND for symbolic expressions

Syntax

```
A & B
and(A,B)
```

Description

`A & B` represents the logical conjunction. `A & B` is true only when both `A` and `B` are true.

`and(A,B)` is equivalent to `A & B`.

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.

Input Arguments

A

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

B

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

Examples

Combine these symbolic inequalities into the logical expression using `&`:

```
syms x y
xy = x >= 0 & y >= 0;
```

Set the corresponding assumptions on variables `x` and `y` using `assume`:

```
assume(xy)
```

Verify that the assumptions are set:

```
assumptions

ans =
[ 0 <= x, 0 <= y]
```

Combine two symbolic inequalities into the logical expression using &:

```
syms x
range = 0 < x & x < 1;
```

Replace variable x with these numeric values. If you replace x with 1/2, then both inequalities are valid. If you replace x with 10, both inequalities are invalid. Note that subs does not evaluate these inequalities to logical 1 or 0.

```
x1 = subs(range, x, 1/2)
x2 = subs(range, x, 10)

x1 =
0 < 1/2 and 1/2 < 1

x2 =
0 < 10 and 10 < 1
```

To evaluate these inequalities to logical 1 or 0, use logical or isAlways:

```
logical(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

Note that simplify does not simplify these logical expressions to logical 1 or 0. Instead, they return *symbolic* values TRUE or FALSE.

```
s1 = simplify(x1)
s2 = simplify(x2)
```

```
s1 =
TRUE

s2 =
FALSE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)

ans =
     1

ans =
     0
```

The recommended approach to define a range of values is using `&`. Nevertheless, you can define a range of values of a variable as follows:

```
syms x
range = 0 < x < 1;
```

Now if you want to replace variable `x` with numeric values, use symbolic numbers instead of MATLAB double-precision numbers. To create a symbolic number, use `sym`

```
x1 = subs(range, x, sym(1/2))
x2 = subs(range, x, sym(10))

x1 =
(0 < 1/2) < 1

x2 =
(0 < 10) < 1
```

To evaluate these inequalities to logical `1` or `0`, use `isAlways`. Note that `logical` cannot resolve such inequalities.

```
isAlways(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

See Also

`all` | `any` | `isAlways` | `logical` | `not` | `or` | `xor`

Purpose

Symbolic polar angle

Syntax

`angle(Z)`

Description

`angle(Z)` computes the polar angle of the complex value `Z`.

Tips

- Calling `angle` for numbers (or vectors or matrices of numbers) that are not symbolic objects invokes the MATLAB `angle` function.
- If `Z = 0`, then `angle(Z)` returns `0`.

Input Arguments

Z

Symbolic number, variable, expression, function. The function also accepts a vector or matrix of symbolic numbers, variables, expressions, functions.

Examples

Compute the polar angles of these complex numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[angle(1 + i), angle(4 + pi*i), angle(inf + inf*i)]

ans =
    0.7854    0.6658    0.7854
```

Compute the polar angles of these complex numbers which are converted to symbolic objects:

```
[angle(sym(1) + i), angle(sym(4) + sym(pi)*i),
angle(inf + sym(inf)*i)]

ans =
[ pi/4, atan(pi/4), angle(Inf + Inf*i)]
```

Compute the limits of these symbolic expressions:

```
syms x;
limit(angle(x + x^2*i/(1 + x)), x, -inf)
limit(angle(x + x^2*i/(1 + x)), x, inf)
```

```
ans =
-(3*pi)/4

ans =
pi/4
```

Compute the polar angles of the elements of matrix `Z`:

```
Z = sym([sqrt(3) + 3*i, 3 + sqrt(3)*i; 1 + i, i]);
angle(Z)
```

```
ans =
[ pi/3, pi/6]
[ pi/4, pi/2]
```

Alternatives

For real `X` and `Y` such that `Z = X + Y*i`, the call `angle(Z)` is equivalent to `atan2(Y,X)`.

See Also

`atan2` | `conj` | `imag` | `real`

Purpose

Test whether at least one of equations and inequalities represented as elements of symbolic array is valid

Syntax

```
any(A)
any(A,dim)
```

Description

`any(A)` tests whether at least one element of `A` returns logical `1` (`true`). If `A` is a matrix, `any` tests elements of each column. If `A` is a multidimensional array, `any` tests elements along one dimension.

`any(A,dim)` tests along the dimension of `A` specified by `dim`.

Tips

- If `A` is an empty symbolic array, `any(A)` returns logical `0`.
- If some elements of `A` are just numeric values (not equations or inequalities), `any` converts these values as follows. All nonzero numeric values become logical `1`. The value 0 becomes logical `0`.
- If `A` is a vector and any of its elements returns logical `1`, `any(A)` returns logical `1`. If all elements are zero, `any(A)` returns logical `0`.
- If `A` is a multidimensional array, `any(A)` treats the values along the first dimension that is not equal to 1 (non-singleton dimension) as vectors, returning logical `1` or `0` for each vector.

Input Arguments

A

Symbolic vector, matrix, or multidimensional symbolic array. For example, it can be an array of symbolic equations, inequalities, or logical expressions with symbolic subexpressions.

dim

Integer. For example, if `A` is a matrix, `any(A,1)` tests elements of each column and returns a row vector of logical `1`s and `0`s. `any(A,2)` tests elements of each row and returns a column vector of logical `1`s and `0`s.

Default: The first dimension that is not equal to 1 (non-singleton dimension). For example, if A is a matrix, any(A) treats the columns of A as vectors.

Examples

Create vector V that contains the symbolic equation and inequalities as its elements:

```
syms x real
V = [x ~= x + 1, abs(x) >= 0, x == x];
```

Use any to test whether at least one of them is valid for all values of x:

```
any(V)

ans =
     1
```

Create this matrix of symbolic equations and inequalities:

```
syms x real
M = [x == 2*x, x == abs(x); abs(x) >= 0, x == 2*x]

M =
[    x == 2*x, x == abs(x)]
[ 0 <= abs(x),    x == 2*x]
```

Use any to test equations and inequalities of this matrix. By default, any tests whether any element of each column is valid for all possible values of variables. If at least one equation or inequality in the column is valid (returns logical 1), then any returns logical 1 for that column. Otherwise, it returns logical 0 for the column. Thus, it returns 1 for the first column and 0 for the second column:

```
any(M)

ans =
     1     0
```

Create this matrix of symbolic equations and inequalities:

```
syms x real
M = [x == 2*x, x == abs(x); abs(x) >= 0, x == 2*x]

M =
[    x == 2*x, x == abs(x)]
[ 0 <= abs(x),    x == 2*x]
```

For matrices and multidimensional arrays, `any` can test elements along the specified dimension. To specify the dimension, use the second argument of `any`. For example, to test elements of each column of a matrix, use the value 1 as the second argument:

```
any(M, 1)

ans =
     1     0
```

To test elements of each row, use the value 2 as the second argument:

```
any(M, 2)

ans =
     0
     1
```

Test whether any element of this vector returns logical `1`. Note that `any` also converts all numeric values outside equations and inequalities to logical `1`s and `0`s. The numeric value 0 becomes logical `0`:

```
syms x
any([0, x == x + 1])

ans =
     0
```

All nonzero numeric values, including negative and complex values, become logical 1s:

```
any([-4 + i, x == x + 1])
```

```
ans =
     1
```

See Also

`all` | `and` | `isAlways` | `logical` | `not` | `or` | `xor`

Purpose

Input variables of symbolic function

Syntax

```
argnames(f)
```

Description

`argnames(f)` returns input variables of `f`.

Input Arguments

f

Symbolic function.

Examples

Create this symbolic function:

```
syms f(x, y)
f(x, y) = x + y;
```

Use `argnames` to find input variables of `f`:

```
argnames(f)

ans =
[ x, y]
```

Create this symbolic function:

```
syms f(a, b, x, y)
f(x, b, y, a) = a*x + b*y;
```

Use `argnames` to find input variables of `f`. When returning variables, `argnames` uses the same order as you used when you defined the function:

```
argnames(f)

ans =
[ x, b, y, a]
```

See Also

`formula` | `sym` | `syms` | `symvar`

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 Hermitian 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 b1 b2
A = sym('a%d%d', [2 2]);
B = [b1 b2];
X = B/A;
x1 = X(1)
x2 = X(2)
```

return

```
x1 =
-(a21*b2 - a22*b1)/(a11*a22 - a12*a21)

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

See Also

`null` | `solve`

Purpose

Set assumption on symbolic object

Syntax

```
assume(assumption)
assume(expr,set)
```

Description

`assume(assumption)` states that `assumption` is valid for all symbolic variables in `assumption`. It also removes any assumptions previously made on these symbolic variables.

`assume(expr,set)` states that `expr` belongs to `set`. This new assumption replaces previously set assumptions on all variables in `expr`.

Tips

- `assume` removes any assumptions previously set on the symbolic variables. To retain previous assumptions while adding a new one, use `assumeAlso`.
- When you delete a symbolic variable from the MATLAB workspace using `clear`, all assumptions that you set on that variable remain in the symbolic engine. If you later declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable and the value of the variable, use this command:

  ```
   syms x clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('x','clear')
  ```

- To delete all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace clearing all assumptions, use this command:

  ```
  clear all
  ```

- If `assumption` is an inequality, then both sides of the inequality must represent real values. Inequalities with complex numbers are invalid because the field of complex numbers is not an ordered field. (It is impossible to tell whether `5 + i` is greater or less than `2 + 3*i`.) MATLAB projects complex numbers in inequalities to real axis. For example, `x > i` becomes `x > 0`, and `x <= 3 + 2*i` becomes `x <= 3`.
- The toolbox does not support assumptions on symbolic functions. Make assumptions on symbolic variables and expressions instead.

Input Arguments

assumption

Symbolic expression, equation, relation, or vector or matrix of symbolic expressions, equations, or relations. You also can combine several assumptions by using the logical operators `and`, `or`, `xor`, `not`, or their shortcuts.

expr

Symbolic variable, expression, vector, or matrix.

set

One of these strings: `integer`, `rational`, or `real`.

Examples

Compute this integral. If you do not make any assumptions, `int` returns this piecewise result:

```
syms x a
int(x^a, x)

ans =
piecewise([a == -1, log(x)], [a ~= -1, x^(a + 1)/(a + 1)])
```

Use `assume` to set an assumption that `x` does not equal `-1`:

```
assume(a ~= -1)
```

Compute the same integral again. Now `int` returns this result:

```
int(x^a, x)

ans =
x^(a + 1)/(a + 1)
```

For further computations, clear the assumption:

```
syms a clear
```

Calculate the time during which the object falls from a certain height by solving the kinematic equation for the free fall motion. If you do not consider the special case where no gravitational forces exist, you can assume that the gravitational acceleration `g` is positive:

```
syms g h t
assume(g > 0)
solve(h == g*t^2/2, t)

ans =
  (2^(1/2)*h^(1/2))/g^(1/2)
 -(2^(1/2)*h^(1/2))/g^(1/2)
```

You can also set assumptions on variables for which you solve an equation. When you set assumptions on such variables, the solver compares obtained solutions with the specified assumptions. This additional task can slow down the solver.

```
assume(t > 0)
solve(h == g*t^2/2, t)

ans =
(2^(1/2)*h^(1/2))/g^(1/2)
```

For further computations, clear the assumptions:

```
syms g t clear
```

Simplify this sine function:

```
syms n
simplify(sin(2*n*pi))

ans =
sin(2*pi*n)
```

Suppose `n` in this expression is an integer. Then you can simplify the expression further using the appropriate assumption:

```
assume(n,'integer')
simplify(sin(2*n*pi))

ans =
0
```

For further computations, clear the assumption:

```
syms n clear
```

You can set assumptions not only on variables, but also on expressions. For example, compute this integral:

```
syms x
int(1/abs(x^2 - 1), x)

ans =
-atanh(x)/sign(x^2 - 1)
```

If you know that $x^2 - 1 > 0$, set the appropriate assumption:

```
assume(x^2 - 1 > 0)
int(1/abs(x^2 - 1), x)

ans =
-atanh(x)
```

For further computations, clear the assumption:

```
syms x clear
```

Solve this equation:

```
syms x
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6
+ (365*x)/2 + 250/3, x)

ans =
   -5
   -1
  1/2
  100
 -1/3
```

Use `assume` to restrict the solutions to the interval $-1 <= x <= 1$:

```
assume(-1 <= x <= 1)
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6
+ (365*x)/2 + 250/3, x)

ans =
   -1
  1/2
 -1/3
```

To set several assumptions simultaneously, use the logical operators `and`, `or`, `xor`, `not`, or their shortcuts. For example, all negative solutions less than `-1` and all positive solutions greater than `1`:

```
assume(x < -1 | x > 1)
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6
+ (365*x)/2 + 250/3, x)

ans =
  -5
 100
```

For further computations, clear the assumptions:

```
syms x clear
```

Alternatives

When you create a new symbolic variable using `sym` and `syms`, you also can set an assumption that the variable is real or positive:

```
a = sym('a','real');
b = sym('b','real');
c = sym('c','positive');
```

or more efficiently

```
syms a b real
syms c positive
```

See Also

`and` | `assumeAlso` | `assumptions` | `clear all` | `isAlways` | `logical` | `not` | `or` | `sym` | `syms`

Concepts

- "Assumptions on Symbolic Objects" on page 1-35

Purpose

Add assumption on symbolic object

Syntax

```
assumeAlso(assumption)
assumeAlso(expr,set)
```

Description

`assumeAlso(assumption)` states that `assumption` is valid for all symbolic variables in `assumption`. It retains all assumptions previously set on these symbolic variables.

`assumeAlso(expr,set)` states that `expr` belongs to `set` in addition to all previously made assumptions.

Tips

- `assumeAlso` keeps all assumptions previously set on the symbolic variables. To replace previous assumptions with the new one, use `assume`.
- When adding assumptions, always check that a new assumption does not contradict the existing assumptions. To see existing assumptions, use `assumptions`. Symbolic Math Toolbox does not guarantee to detect conflicting assumptions. Conflicting assumptions can lead to unpredictable and inconsistent results.
- When you delete a symbolic variable from the MATLAB workspace using `clear`, all assumptions that you set on that variable remain in the symbolic engine. If later you declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable and the value of the variable, use this command:

  ```
   syms x clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('x','clear')
  ```

- To clear all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace resetting all its assumptions, use this command:

  ```
  clear all
  ```

- If `assumption` is an inequality, then both sides of the inequality must represent real values. Inequalities with complex numbers are invalid because the field of complex numbers is not an ordered field. (It is impossible to tell whether `5 + i` is greater or less than `2 + 3*i`.) MATLAB projects complex numbers in inequalities to real axis. For example, `x > i` becomes `x > 0`, and `x <= 3 + 2*i` becomes `x <= 3`.
- The toolbox does not support assumptions on symbolic functions. Make assumptions on symbolic variables and expressions instead.

Input Arguments

assumption

Symbolic expression, equation, relation, or vector or matrix of symbolic expressions, equations, or relations. You also can combine several assumptions by using the logical operators `and`, `or`, `xor`, `not`, or their shortcuts.

expr

Symbolic variable, expression, vector, or matrix.

set

One of these strings: `integer`, `rational`, or `real`.

Examples

Solve this equation assuming that both `x` and `y` are nonnegative:

```
syms x y
assume(x >= 0 & y >= 0)
s = solve(x^2 + y^2 == 1, y)
```

```
s =
```

```
{[(- x + 1)^(1/2)*(x + 1)^(1/2), 1],...
[-(- x + 1)^(1/2)*(x + 1)^(1/2), 1]} intersect...
Dom::Interval([0], Inf)
```

Now add the assumption that `x < 1`. To add a new assumption without removing the previous one, use `assumeAlso`:

```
assumeAlso(x < 1)
```

Solve the same equation under the expanded set of assumptions:

```
s = solve(x^2 + y^2 == 1, y)

s =
(1 - x)^(1/2)*(x + 1)^(1/2)
```

For further computations, clear the assumptions:

```
syms x y clear
```

When declaring the symbolic variable `n`, set an assumption that `n` is positive:

```
syms n positive
```

Using `assumeAlso`, you can add more assumptions on the same variable `n`. For example, assume also that n is and integer:

```
assumeAlso(n,'integer')
```

To see all assumptions currently valid for the variable `n`, use `assumptions`. In this case, `n` is a positive integer.

```
assumptions(n)

ans =
[ n in Z_, 0 < n]
```

For further computations, clear the assumptions:

```
syms n clear
```

When you add assumptions, ensure that the new assumptions do not contradict the previous assumptions. Contradicting assumptions can lead to inconsistent and unpredictable results. In some cases, `assumeAlso` detects conflicting assumptions and issues the following error:

```
syms y
assume(y,'real')
assumeAlso(y == i)

Error using mupadmex
Error in MuPAD command: Inconsistent assumptions
detected. [property::_setgroup]
```

`assumeAlso` does not guarantee to detect contradicting assumptions. For example, you can assume that `y` is nonzero, and both `y` and `y*i` are real values:

```
syms y
assume(y ~= 0)
assumeAlso(y,'real')
assumeAlso(y*i,'real')
```

To see all assumptions currently valid for the variable `y`, use `assumptions`:

```
assumptions(y)

ans =
[ y in R_, y ~= 0, y*i in R_]
```

For further computations, clear the assumptions:

```
syms y clear
```

Alternatives

Instead of adding assumptions one by one, you can set several assumptions in one function call. To set several assumptions, use `assume` and combine these assumptions by using the logical operators `and`, `or`, `xor`, `not`, `all`, `any`, or their shortcuts.

See Also

`and` | `assume` | `assumptions` | `clear all` | `isAlways` | `logical` | `not` | `or` | `sym` | `syms`

Concepts

- "Assumptions on Symbolic Objects" on page 1-35

assumptions

Purpose

Show assumptions set on symbolic variable

Syntax

```
assumptions(var)
assumptions
```

Description

`assumptions(var)` returns all assumptions set on variable `var`.

`assumptions` returns all assumptions set on all variables in MATLAB Workspace.

Tips

- When you delete a symbolic object from the MATLAB workspace by using `clear`, all assumptions that you set on that object remain in the symbolic engine. If later you declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable `var` and the value of the variable, use this command:

  ```
  syms var clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('var','clear')
  ```

- To clear all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace resetting all its assumptions, use this command:

  ```
  clear all
  ```

Input Arguments

var

Symbolic variable or array of symbolic variables.

Examples

Assume that the variable `n` is integer and the variable `x` is rational. In addition to that , assume that the product `n*x` belongs to the interval from `-100` to `100`:

```
syms n x
assume(n,'integer')
assume(x,'rational')
assumeAlso(-100 <= n*x <= 100)
```

To see the assumptions set on the variable `n`, enter:

```
assumptions(n)

ans =
[ -100 <= n*x, n*x <= 100, n in Z_]
```

To see the assumptions set on the variable `x`, enter:

```
assumptions(x)

ans =
[ -100 <= n*x, n*x <= 100, x in Q_]
```

To see the assumptions set on all variables, use `assumptions` without any arguments:

```
assumptions

ans =
[ -100 <= n*x, n*x <= 100, n in Z_, x in Q_]
```

For further computations, clear the assumptions:

```
syms n x clear
```

Use `assumptions` to return all assumptions, including those set by the `syms` command:

```
syms x real
assumeAlso(x < 0)
assumptions(x)
```

```
ans =
[ x < 0, x in R_]
```

See Also

`and | assume | assumeAlso | clear all | isAlways | logical | not | or | sym | syms`

Concepts

- "Assumptions on Symbolic Objects" on page 1-35

Purpose

Symbolic four-quadrant inverse tangent

Syntax

`atan2(Y,X)`

Description

`atan2(Y,X)` computes the four-quadrant inverse tangent (arctangent) of `Y` and `X`. If `Y` and `X` are vectors or matrices, `atan2` computes arctangents element by element.

Tips

- Calling `atan2` for numbers (or vectors or matrices of numbers) that are not symbolic objects invokes the MATLAB `atan2` function.
- If one of the arguments `X` and `Y` is a vector or a matrix, and another one is a scalar, then `atan2` expands the scalar into a vector or a matrix of the same length with all elements equal to that scalar.
- Symbolic arguments `X` and `Y` are assumed to be real.
- If `X = 0` and `Y > 0`, then `atan2(Y,X)` returns `pi/2`.

  If `X = 0` and `Y < 0`, then `atan2(Y,X)` returns `-pi/2`.

  If `X = Y = 0`, then `atan2(Y,X)` returns `0`.

Input Arguments

Y

Symbolic number, variable, expression, function. The function also accepts a vector or matrix of symbolic numbers, variables, expressions, functions. If `Y` is a number, it must be real. If `Y` is a vector or matrix, it must either be a scalar or have the same dimensions as `X`. All numerical elements of `Y` must be real.

X

Symbolic number, variable, expression, function. The function also accepts a vector or matrix of symbolic numbers, variables, expressions, functions. If `X` is a number, it must be real. If `X` is a vector or matrix, it must either be a scalar or have the same dimensions as `Y`. All numerical elements of `X` must be real.

Definitions

atan2 vs. atan

If $X \neq 0$ and $Y \neq 0$, then

$$\text{atan2}(Y, X) = \text{atan}(\frac{Y}{X}) + \frac{\pi}{2}\text{sign}(Y)(1 - \text{sign}(X))$$

Results returned by `atan2` belong to the closed interval `[-pi,pi]`. Results returned by `atan` belong to the closed interval `[-pi/2,pi/2]`.

Examples

Compute the arctangents of these parameters. Because these numbers are not symbolic objects, you get floating-point results.

```
[atan2(1, 1), atan2(pi, 4), atan2(inf, inf)]

ans =
    0.7854    0.6658    0.7854
```

Compute the arctangents of these parameters which are converted to symbolic objects:

```
[atan2(sym(1), 1), atan2(sym(pi), sym(4)),
atan2(inf, sym(inf))]

ans =
[ pi/4, atan(pi/4), angle(Inf + Inf*i)]
```

Compute the limits of this symbolic expression:

```
syms x;
limit(atan2(x^2/(1 + x), x), x, -inf)
limit(atan2(x^2/(1 + x), x), x, inf)

ans =
-(3*pi)/4
```

```
ans =
pi/4
```

Compute the arctangents of the elements of matrices `Y` and `X`:

```
Y = sym([3 sqrt(3); 1 1]);
X = sym([sqrt(3) 3; 1 0]);
atan2(Y, X)
```

```
ans =
[ pi/3, pi/6]
[ pi/4, pi/2]
```

Alternatives

For complex `Z = X + Y*i`, the call `atan2(Y,X)` is equivalent to `angle(Z)`.

See Also

`angle | conj | imag | real`

besseli

Purpose Modified Bessel function of the first kind

Syntax

```
besseli(nu,z)
besseli(nu,A)
```

Description `besseli(nu,z)` returns the modified Bessel function of the first kind, $I_\nu(z)$.

`besseli(nu,A)` returns the modified Bessel function of the first kind for each element of `A`.

Tips

- Calling `besseli` for a number that is not a symbolic object invokes the MATLAB `besseli` function.

Input Arguments

nu

Symbolic number, variable, or expression.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Modified Bessel Functions of the First Kind

The modified Bessel differential equation

$$z^2 \frac{d^2w}{dz^2} + z\frac{dw}{dz} - \left(z^2 + \nu^2\right)w = 0$$

has two linearly independent solutions. These solutions are represented by the modified Bessel functions of the first kind, $I_\nu(z)$, and the modified Bessel functions of the second kind, $K_\nu(z)$:

$$w(z) = C_1 I_\nu(z) + C_2 K_\nu(z)$$

This formula is the integral representation of the modified Bessel functions of the first kind:

$$I_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi e^{z\cos(t)} \sin(t)^{2\nu}\, dt$$

Examples

Solve this second-order differential equation. The solutions are the modified Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) -(z^2 + nu^2)*w == 0)

ans =
C2*besseli(nu, z) + C3*besselk(nu, z)
```

Verify that the modified Bessel function of the first kind is a valid solution of the modified Bessel differential equation.

```
syms nu z
simplify(z^2*diff(besseli(nu, z), z,
2) + z*diff(besseli(nu, z), z) - (z^2 +
nu^2)*besseli(nu, z)) == 0

ans =
     1
```

Compute the modified Bessel functions of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besseli(0, 5), besseli(-1, 2), besseli(1/3, 7/4),
besseli(1, 3/2 + 2*i)]

ans =
  27.2399             1.5906              1.7951
-0.1523 + 1.0992i
```

Compute the modified Bessel functions of the first kind for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `besseli` returns unresolved symbolic calls.

```
[besseli(sym(0), 5), besseli(sym(-1), 2), besseli(1/3,
sym(7/4)),  besseli(sym(1), 3/2 + 2*i)]

ans =
[ besseli(0, 5), besseli(1, 2), besseli(1/3, 7/4),
besseli(1, 3/2 + 2*i)]
```

For symbolic variables and expressions, `besseli` also returns unresolved symbolic calls:

```
syms x y
[besseli(x, y), besseli(1, x^2), besseli(2, x -
y), besseli(x^2, x*y)]

ans =
[ besseli(x, y), besseli(1, x^2), besseli(2, x -
y), besseli(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besseli` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besseli(1/2, x)

ans =
(2^(1/2)*sinh(x))/(pi^(1/2)*x^(1/2))

besseli(-1/2, x)
```

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

besseli(-3/2, x)

ans =
(2^(1/2)*(sinh(x) - cosh(x)/x))/(pi^(1/2)*x^(1/2))

besseli(5/2, x)

ans =
-(2^(1/2)*((3*cosh(x))/x - sinh(x)*(3/x^2 +
1)))/(pi^(1/2)*x^(1/2))
```

Differentiate the expressions involving the modified Bessel functions of the first kind:

```
syms x y
diff(besseli(1, x))
diff(diff(besseli(0, x^2 + x*y -y^2), x), y)

ans =
besseli(0, x) - besseli(1, x)/x

ans =
besseli(1, x^2 + x*y - y^2) +...
(2*x + y)*(besseli(0, x^2 + x*y - y^2)*(x - 2*y) -...
(besseli(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

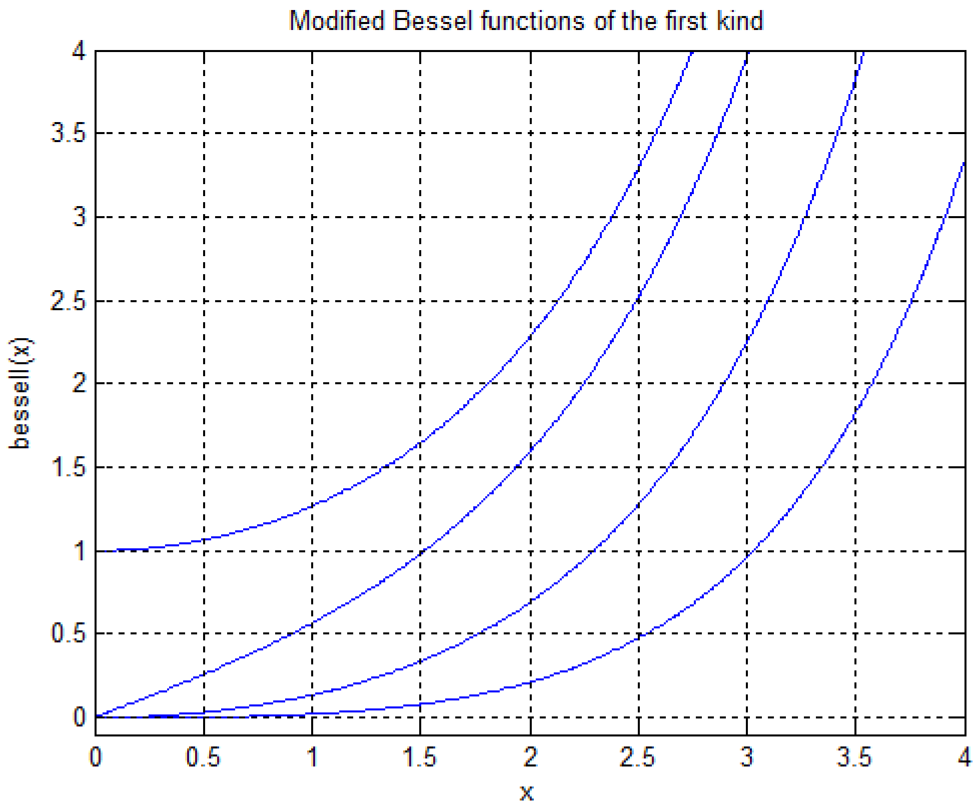
Call `besseli` for the matrix `A` and the value 1/2. The result is a matrix of the modified Bessel functions `besseli(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
besseli(1/2, A)
```

```
ans =
[         (2^(1/2)*sinh(1)*i)/pi^(1/2), (2^(1/2)*sinh(pi))/pi]
[ (2^(1/2)*sinh(x))/(pi^(1/2)*x^(1/2)),
0]
```

Plot the modified Bessel functions of the first kind for v = 0, 1, 2, 3:

```
syms x y
for nu =[0, 1, 2, 3]
  ezplot(besseli(nu, x) - y, [0, 4, -0.1, 4])
  colormap([0 0 1])
  hold on
end
title('Modified Bessel functions of the first kind')
ylabel('besselI(x)')
grid
hold off
```



References

[1] Olver, F. W. J. "Bessel Functions of Integer Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

[2] Antosiewicz, H. A. "Bessel Functions of Fractional Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`airy` | `besselj` | `besselk` | `bessely` | `mfun` | `mfunlist`

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Bessel function of the first kind

Syntax

```
besselj(nu,z)
besselj(nu,A)
```

Description

`besselj(nu,z)` returns the Bessel function of the first kind, $J_v(z)$.

`besselj(nu,A)` returns the Bessel function of the first kind for each element of `A`.

Tips

- Calling `besselj` for a number that is not a symbolic object invokes the MATLAB `besselj` function.

Input Arguments

nu

Symbolic number, variable, or expression.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Bessel Functions of the First Kind

The Bessel differential equation

$$z^2 \frac{d^2w}{dz^2} + z\frac{dw}{dz} + \left(z^2 - \nu^2\right)w = 0$$

has two linearly independent solutions. These solutions are represented by the Bessel functions of the first kind, $J_v(z)$, and the Bessel functions of the second kind, $Y_v(z)$:

$$w(z) = C_1 J_\nu(z) + C_2 Y_\nu(z)$$

This formula is the integral representation of the Bessel functions of the first kind:

$$J_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi \cos(z\cos(t))\sin(t)^{2\nu}\,dt$$

Examples

Solve this second-order differential equation. The solutions are the Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) +(z^2 - nu^2)*w == 0)

ans =
C2*besselj(nu, z) + C3*bessely(nu, z)
```

Verify that the Bessel function of the first kind is a valid solution of the Bessel differential equation:

```
syms nu z
simplify(z^2*diff(besselj(nu, z), z,
2) + z*diff(besselj(nu, z), z) + (z^2 -
nu^2)*besselj(nu, z)) == 0

ans =
     1
```

Compute the Bessel functions of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besselj(0, 5), besselj(-1, 2), besselj(1/3, 7/4),
besselj(1, 3/2 + 2*i)]

ans =
  -0.1776            -0.5767            0.5496
1.6113 + 0.3982i
```

Compute the Bessel functions of the first kind for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `besselj` returns unresolved symbolic calls.

```
[besselj(sym(0), 5), besselj(sym(-1), 2), besselj(1/3,
sym(7/4)),  besselj(sym(1), 3/2 + 2*i)]

ans =
[ besselj(0, 5), -besselj(1, 2), besselj(1/3, 7/4),
besselj(1, 3/2 + 2*i)]
```

For symbolic variables and expressions, `besselj` also returns unresolved symbolic calls:

```
syms x y
[besselj(x, y), besselj(1, x^2), besselj(2, x -
y), besselj(x^2, x*y)]

ans =
[ besselj(x, y), besselj(1, x^2), besselj(2, x -
y), besselj(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besselj` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besselj(1/2, x)

ans =
(2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2))

besselj(-1/2, x)
```

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

besselj(-3/2, x)

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

besselj(5/2, x)

ans =
-(2^(1/2)*((3*cos(x))/x - sin(x)*(3/x^2 -
1)))/(pi^(1/2)*x^(1/2))
```

Differentiate the expressions involving the Bessel functions of the first kind:

```
syms x y
diff(besselj(1, x))
diff(diff(besselj(0, x^2 + x*y -y^2), x), y)

ans =
besselj(0, x) - besselj(1, x)/x

ans =
- besselj(1, x^2 + x*y - y^2) -...
(2*x + y)*(besselj(0, x^2 + x*y - y^2)*(x - 2*y) -...
(besselj(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

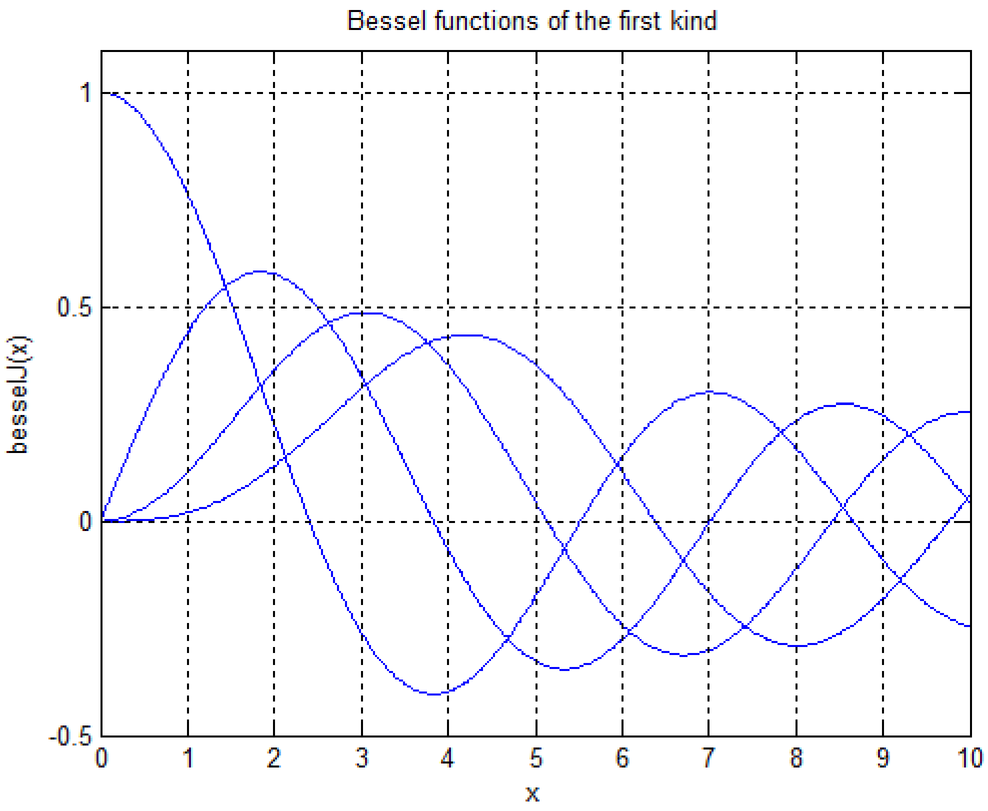
Call `besselj` for the matrix `A` and the value 1/2. The result is a matrix of the Bessel functions `besselj(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
besselj(1/2, A)
```

```
ans =
[         (2^(1/2)*sin(1)*i)/pi^(1/2), 0]
[ (2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2)), 0]
```

Plot the Bessel functions of the first kind for v = 0, 1, 2, 3:

```
syms x y
for nu =[0, 1, 2, 3]
  ezplot(besselj(nu, x) - y, [0, 10, -0.5, 1.1])
  colormap([0 0 1])
  hold on
end
title('Bessel functions of the first kind')
ylabel('besselJ(x)')
grid
hold off
```



References

[1] Olver, F. W. J. "Bessel Functions of Integer Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

[2] Antosiewicz, H. A. "Bessel Functions of Fractional Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`airy` | `besseli` | `besselk` | `bessely` | `mfun` | `mfunlist`

How To

- “Special Functions of Applied Mathematics” on page 2-142

besselk

Purpose

Modified Bessel function of the second kind

Syntax

```
besselk(nu,z)
besselk(nu,A)
```

Description

`besselk(nu,z)` returns the modified Bessel function of the second kind, $K_v(z)$.

`besselk(nu,A)` returns the modified Bessel function of the second kind for each element of `A`.

Tips

- Calling `besselk` for a number that is not a symbolic object invokes the MATLAB `besselk` function.

Input Arguments

nu

Symbolic number, variable, or expression.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Modified Bessel Functions of the Second Kind

The modified Bessel differential equation

$$z^2 \frac{d^2w}{dz^2} + z\frac{dw}{dz} - \left(z^2 + \nu^2\right)w = 0$$

has two linearly independent solutions. These solutions are represented by the modified Bessel functions of the first kind, $I_v(z)$, and the modified Bessel functions of the second kind, $K_v(z)$:

$$w(z) = C_1 I_\nu(z) + C_2 K_\nu(z)$$

The modified Bessel functions of the second kind are defined via the modified Bessel functions of the first kind:

$$K_\nu(z) = \frac{\pi/2}{\sin(\nu\pi)}\left(I_{-\nu}(z) - I_\nu(z)\right)$$

Here $I_\nu(z)$ are the modified Bessel functions of the first kind:

$$I_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)}\int_0^\pi e^{z\cos(t)}\sin(t)^{2\nu}\,dt$$

Examples

Solve this second-order differential equation. The solutions are the modified Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) -(z^2 + nu^2)*w == 0)

ans =
C2*besseli(nu, z) + C3*besselk(nu, z)
```

Verify that the modified Bessel function of the second kind is a valid solution of the modified Bessel differential equation:

```
syms nu z
simplify(z^2*diff(besselk(nu, z), z,
2) + z*diff(besselk(nu, z), z) - (z^2 +
nu^2)*besselk(nu, z)) == 0

ans =
     1
```

Compute the modified Bessel functions of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besselk(0, 5), besselk(-1, 2), besselk(1/3, 7/4),
besselk(1, 3/2 + 2*i)]

ans =
   0.0037             0.1399             0.1594
-0.1620 - 0.1066i
```

Compute the modified Bessel functions of the second kind for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `besselk` returns unresolved symbolic calls.

```
[besselk(sym(0), 5), besselk(sym(-1), 2), besselk(1/3,
sym(7/4)),  besselk(sym(1), 3/2 + 2*i)]

ans =
[ besselk(0, 5), besselk(1, 2), besselk(1/3, 7/4),
besselk(1, 3/2 + 2*i)]
```

For symbolic variables and expressions, `besselk` also returns unresolved symbolic calls:

```
syms x y
[besselk(x, y), besselk(1, x^2), besselk(2, x -
y), besselk(x^2, x*y)]

ans =
[ besselk(x, y), besselk(1, x^2), besselk(2, x -
y), besselk(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besselk` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besselk(1/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2))

besselk(-1/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2))

besselk(-3/2, x)

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

besselk(5/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x)*(3/x + 3/x^2 + 1))/(2*x^(1/2))
```

Differentiate the expressions involving the modified Bessel functions of the second kind:

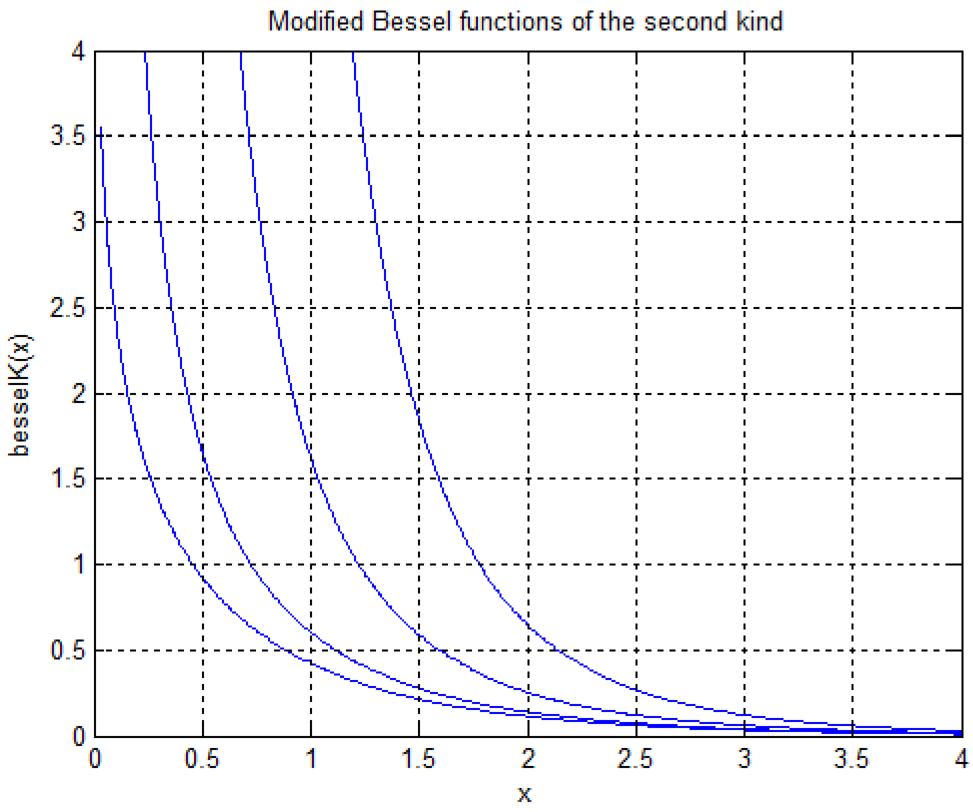
```
syms x y
diff(besselk(1, x))
diff(diff(besselk(0, x^2 + x*y -y^2), x), y)

ans =
- besselk(1, x)/x - besselk(0, x)

ans =
(2*x + y)*(besselk(0, x^2 + x*y - y^2)*(x - 2*y) +...
(besselk(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2)) -...
besselk(1, x^2 + x*y - y^2)
```

Call `besselk` for the matrix `A` and the value 1/2. The result is a matrix of the modified Bessel functions `besselk(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
besselk(1/2, A)

ans =
[         -(2^(1/2)*pi^(1/2)*exp(1)*i)/2, (2^(1/2)*exp(-pi))/2]
[ (2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2)),
Inf]
```

Plot the modified Bessel functions of the second kind for v = 0, 1, 2, 3:

```
syms x y
for nu =[0, 1, 2, 3]
  ezplot(besselk(nu, x) - y, [0, 4, 0, 4])
  colormap([0 0 1])
  hold on
end
title('Modified Bessel functions of the second kind')
ylabel('besselK(x)')
grid
hold off
```



References

[1] Olver, F. W. J. "Bessel Functions of Integer Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

[2] Antosiewicz, H. A. "Bessel Functions of Fractional Order." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`airy` | `besseli` | `besselj` | `bessely` | `mfun` | `mfunlist`

How To

- "Special Functions of Applied Mathematics" on page 2-142

Purpose

Bessel function of the second kind

Syntax

```
bessely(nu,z)
bessely(nu,A)
```

Description

`bessely(nu,z)` returns the Bessel function of the second kind, $Y_v(z)$.

`bessely(nu,A)` returns the Bessel function of the second kind for each element of `A`.

Tips

- Calling `bessely` for a number that is not a symbolic object invokes the MATLAB `bessely` function.

Input Arguments

nu

Symbolic number, variable, or expression representing a real number.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Bessel Functions of the Second Kind

The Bessel differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + \left(z^2 - v^2\right) w = 0$$

has two linearly independent solutions. These solutions are represented by the Bessel functions of the first kind, $J_v(z)$, and the Bessel functions of the second kind, $Y_v(z)$:

$$w(z) = C_1 J_v(z) + C_2 Y_v(z)$$

The Bessel functions of the second kind are defined via the Bessel functions of the first kind:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

Here J$_\nu$(z) are the Bessel function of the first kind:

$$J_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi \cos(z\cos(t))\sin(t)^{2\nu}\,dt$$

Examples

Solve this second-order differential equation. The solutions are the Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) +(z^2 - nu^2)*w == 0)

ans =
C2*besselj(nu, z) + C3*bessely(nu, z)
```

Verify that the Bessel function of the second kind is a valid solution of the Bessel differential equation:

```
syms nu z
simplify(z^2*diff(bessely(nu, z), z,
2) + z*diff(bessely(nu, z), z) + (z^2 -
nu^2)*bessely(nu, z)) == 0

ans =
     1
```

Compute the Bessel functions of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[bessely(0, 5), bessely(-1, 2), bessely(1/3, 7/4),
bessely(1, 3/2 + 2*i)]

ans =
  -0.3085             0.1070             0.2358
-0.4706 + 1.5873i
```

Compute the Bessel functions of the second kind for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `bessely` returns unresolved symbolic calls.

```
[bessely(sym(0), 5), bessely(sym(-1), 2), bessely(1/3,
sym(7/4)),  bessely(sym(1), 3/2 + 2*i)]

ans =
[ bessely(0, 5), -bessely(1, 2), bessely(1/3, 7/4),
bessely(1, 3/2 + 2*i)]
```

For symbolic variables and expressions, `bessely` also returns unresolved symbolic calls:

```
syms x y
[bessely(x, y), bessely(1, x^2), bessely(2, x -
y), bessely(x^2, x*y)]

ans =
[ bessely(x, y), bessely(1, x^2), bessely(2, x -
y), bessely(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besseli` rewrites the Bessel functions in terms of elementary functions:

```
syms x
bessely(1/2, x)

ans =
-(2^(1/2)*cos(x))/(pi^(1/2)*x^(1/2))

bessely(-1/2, x)

ans =
(2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2))

bessely(-3/2, x)

ans =
(2^(1/2)*(cos(x) - sin(x)/x))/(pi^(1/2)*x^(1/2))

bessely(5/2, x)

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

Differentiate the expressions involving the Bessel functions of the second kind:

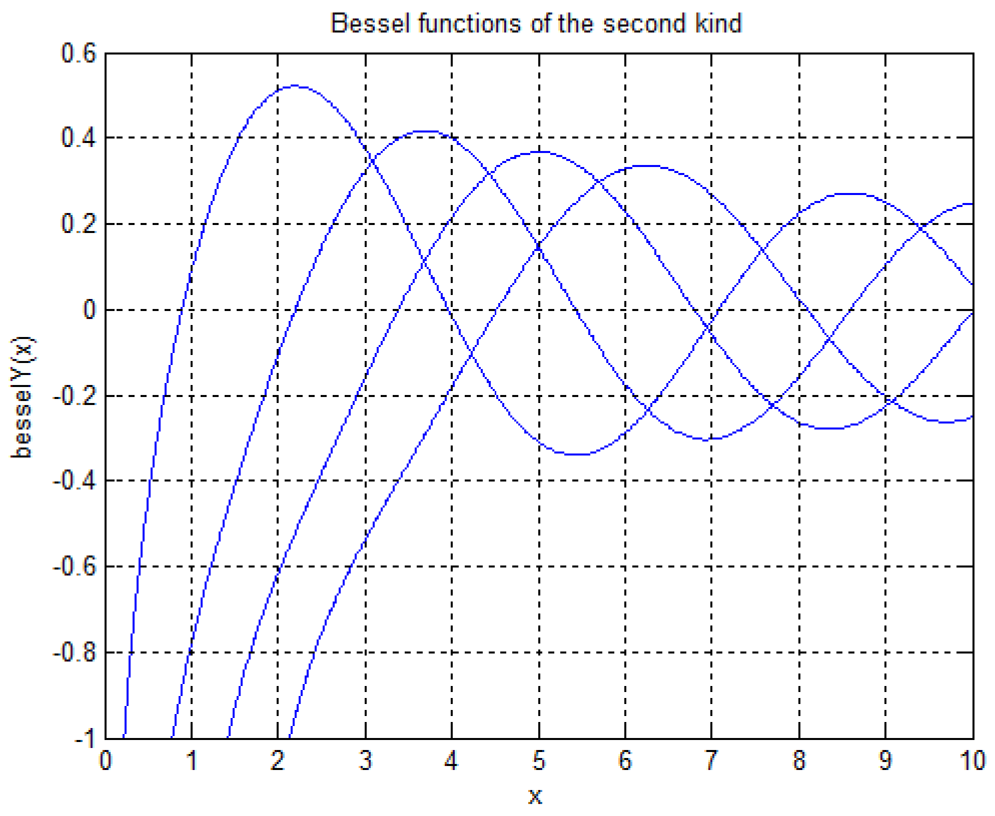
```
syms x y
diff(bessely(1, x))
diff(diff(bessely(0, x^2 + x*y -y^2), x), y)

ans =
bessely(0, x) - bessely(1, x)/x

ans =
- bessely(1, x^2 + x*y - y^2) -...
(2*x + y)*(bessely(0, x^2 + x*y - y^2)*(x - 2*y) -...
(bessely(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

Call `bessely` for the matrix `A` and the value 1/2. The result is a matrix of the Bessel functions `bessely(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
bessely(1/2, A)

ans =
[          (2^(1/2)*cos(1)*i)/pi^(1/2), 2^(1/2)/pi]
[ -(2^(1/2)*cos(x))/(pi^(1/2)*x^(1/2)),        Inf]
```

Plot the Bessel functions of the second kind for v = 0, 1, 2, 3:

```
syms x y
for nu =[0, 1, 2, 3]
  ezplot(bessely(nu, x) - y, [0, 10, -1, 0.6])
  colormap([0 0 1])
  hold on
end
title('Bessel functions of the second kind')
ylabel('besselY(x)')
grid
hold off
```



References

[1] Olver, F. W. J. “Bessel Functions of Integer Order.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

[2] Antosiewicz, H. A. “Bessel Functions of Fractional Order.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`airy` | `besseli` | `besselj` | `besselk` | `mfun` | `mfunlist`

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Beta function

Syntax

```
beta(x,y)
beta(x,A)
```

Description

`beta(x,y)` returns the beta function of `x` and `y`.

`beta(x,A)` returns the beta functions of `x` and each element of `A`.

Tips

- The beta function is uniquely defined for positive numbers and complex numbers with positive real parts. It is approximated for other numbers.
- Calling `beta` for numbers that are not symbolic objects invokes the MATLAB `beta` function. This function accepts real arguments only. If you want to compute the beta function for complex numbers, use `sym` to convert the numbers to symbolic objects, and then call `beta` for those symbolic objects.
- If one or both parameters are negative numbers, convert these numbers to symbolic objects using `sym`, and then call `beta` for those symbolic objects.
- If the beta function has a singularity, `beta` returns the positive infinity `Inf`.
- `beta(0, 0)` returns `NaN`.
- `beta(x,y) = beta(y,x)` and `beta(x,A) = beta(A,x)`.

Input Arguments

x

Symbolic number, variable, or expression.

y

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Beta Function

This integral defines the beta function:

$$B(x,y) = \int_0^1 t^{x-1} (1-t)^{y-1}\, dt = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

Examples

Compute the beta function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[beta(1, 5), beta(3, sqrt(2)), beta(pi,
exp(1)), beta(0, 1)]

ans =
    0.2000    0.1716    0.0379       Inf
```

Compute the beta function for the numbers converted to symbolic objects:

```
[beta(sym(1), 5), beta(3, sym(2)), beta(sym(4), sym(4))]

ans =
[ 1/5, 1/12, 1/140]
```

If one or both parameters are complex numbers, convert these numbers to symbolic objects:

```
[beta(sym(i), 3/2), beta(sym(i), i), beta(sym(i
+ 2), 1 - i)]

ans =
[ (pi^(1/2)*gamma(i))/(2*gamma(3/2 + i)),
gamma(i)^2/gamma(2*i), (pi*(1/2 + i/2))/sinh(pi)]
```

Compute the beta function for negative parameters. If one or both arguments are negative numbers, convert these numbers to symbolic objects:

```
[beta(sym(-3), 2), beta(sym(-1/3), 2), beta(sym(-3),
4),  beta(sym(-3), -2)]

ans =
[ 1/6, -9/2, Inf, Inf]
```

Call `beta` for the matrix `A` and the value `1`. The result is a matrix of the beta functions `beta(A(i,j),1)`:

```
A = sym([1 2; 3 4]);
beta(A,1)

ans =
[   1, 1/2]
[ 1/3, 1/4]
```

Differentiate the beta function, then substitute the variable *t* with the value 2/3 and approximate the result using `vpa`:

```
syms t
u = diff(beta(t^2 + 1, t))
vpa(subs(u, t, 2/3), 10)

u =
beta(t, t^2 + 1)*(psi(t) + 2*t*psi(t^2 + 1) -...
psi(t^2 + t + 1)*(2*t + 1))

ans =
-2.836889094
```

Expand these beta functions:

```
syms x y
expand(beta(x, y))
expand(beta(x + 1, y - 1))

ans =
(gamma(x)*gamma(y))/gamma(x + y)

ans =
-(x*gamma(x)*gamma(y))/(gamma(x + y) - y*gamma(x + y))
```

References

Zelen, M. and N. C. Severo. “Probability Functions.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`gamma` | `factorial` | `mfun` | `mfunlist` | `nchoosek` | `psi`

How To

- “Special Functions of Applied Mathematics” on page 2-142

ccode

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`

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`.

Examples

```
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)
```

Description

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

Tips

- `char` can change term ordering in an expression.

Input Arguments

A

Symbolic scalar or symbolic array.

Examples

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

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

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

Note that `char` changes the order of the terms in the resulting string.

Convert a symbolic matrix to a string:

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

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`

Purpose

Characteristic polynomial of matrix

Syntax

```
charpoly(A)
charpoly(A,var)
```

Description

`charpoly(A)` returns a vector of the coefficients of the characteristic polynomial of `A`. If `A` is a symbolic matrix, `charpoly` returns a symbolic vector. Otherwise, it returns a vector of double-precision values.

`charpoly(A,var)` returns the characteristic polynomial of `A` in terms of `var`.

Input Arguments

A

Matrix.

var

Free symbolic variable.

> **Default:** If you do not specify `var`, `charpoly` returns a vector of coefficients of the characteristic polynomial instead of returning the polynomial itself.

Definitions

Characteristic Polynomial of a Matrix

The characteristic polynomial of an n-by-n matrix `A` is the polynomial $p_A(x)$, such that

$$p_A(x) = \det(xI_n - A)$$

Here I_n is the n-by-n identity matrix.

Examples

Compute the characteristic polynomial of the matrix `A` in terms of the variable `x`:

```
syms x;
A = sym([1 1 0; 0 1 0; 0 0 1]);
```

```
charpoly(A, x)

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

To find the coefficients of the characteristic polynomial of `A`, call `charpoly` with one argument:

```
A = sym([1 1 0; 0 1 0; 0 0 1]);
charpoly(A)

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

Find the coefficients of the characteristic polynomial of the symbolic matrix `A`. For this matrix, `charpoly` returns the symbolic vector of coefficients:

```
A = sym([1 2; 3 4]);
P = charpoly(A)

P =
[ 1, -5, -2]
```

Now find the coefficients of the characteristic polynomial of the matrix `B`, all elements of which are double-precision values. Note that in this case `charpoly` returns coefficients as double-precision values:

```
B = ([1 2; 3 4]);
P = charpoly(B)

P =
     1    -5    -2
```

References

[1] Cohen, H. “A Course in Computational Algebraic Number Theory.” *Graduate Texts in Mathematics* (Axler, Sheldon and Ribet, Kenneth A., eds.). Vol. 138, Springer, 1993.

[2] Abdeljaoued, J. “The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring.” MapleTech, Vol. 4, Number 3, pp 21–32, Birkhauser, 1997.

See Also

`det` | `eig` | `jordan` | `minpoly` | `poly2sym` | `sym2poly`

children

Purpose

Subexpressions or terms of symbolic expression

Syntax

```
children(expr)
children(A)
```

Description

`children(expr)` returns a vector containing the child subexpressions of the symbolic expression `expr`. For example, the child subexpressions of a sum are its terms.

`children(A)` returns a cell array containing the child subexpressions of each expression in `A`.

Input Arguments

expr

Symbolic expression, equation, or inequality.

A

Vector or matrix of symbolic expressions, equations, or inequalities.

Examples

Find the child subexpressions of this expression. Child subexpressions of a sum are its terms.

```
syms x y
children(x^2 + x*y + y^2)

ans =
[ x*y, x^2, y^2]
```

Find the child subexpressions of this expression. This expression is also a sum, only some terms of that sum are negative.

```
children(x^2 - x*y - y^2)

ans =
[ -x*y, x^2, -y^2]
```

The child subexpression of a variable is the variable itself:

```
children(x)

ans =
x
```

Create the symbolic expression using `sym`. With this approach, you do not create symbolic variables corresponding to the terms of the expression. Nevertheless, `children` finds the terms of the expression:

```
children(sym('a + b + c'))

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

Find the child subexpressions of this equation. The child subexpressions of an equation are the left and right sides of that equation.

```
syms x y
children(x^2 + x*y == y^2 + 1)

ans =
[ x^2 + y*x, y^2 + 1]
```

Find the child subexpressions of this inequality. The child subexpressions of an inequality are the left and right sides of that inequality.

```
children(sin(x) < cos(x))

ans =
[ sin(x), cos(x)]
```

Call the `children` function for this matrix. The result is the cell array containing the child subexpressions of each element of the matrix.

```
syms x y
s = children([x + y, sin(x)*cos(y); x^3 - y^3, exp(x*y^2)])
```

```
s =
    [1x2 sym]    [1x2 sym]
    [1x2 sym]    [1x1 sym]
```

To access the contents of cells in the cell array, use braces:

```
s{1:4}
```

```
ans =
[ x, y]

ans =
[ x^3, -y^3]

ans =
[ cos(y), sin(x)]

ans =
x*y^2
```

See Also

`coeffs` | `numden` | `subs`

Concepts

- “Create Symbolic Expressions” on page 1-9

Purpose

Cholesky factorization

Syntax

```
T = chol(A)
[T,p] = chol(A)
[T,p,S] = chol(A)
[T,p,s] = chol(A,'vector')
___ = chol(A,'lower')
___ = chol(A,'noCheck')
___ = chol(A,'real')
___ = chol(A,'lower','noCheck','real')
[T,p,s] = chol(A,'lower','vector','noCheck','real')
```

Description

`T = chol(A)` returns an upper triangular matrix `T`, such that `T'*T = A`. `A` must be a Hermitian positive definite matrix. Otherwise, this syntax throws an error.

`[T,p] = chol(A)` computes the Cholesky factorization of `A`. This syntax does not error if `A` is not a Hermitian positive definite matrix. If `A` is a Hermitian positive definite matrix, then `p` is 0. Otherwise, `T` is `sym([])`, and `p` is a positive integer (typically, `p = 1`).

`[T,p,S] = chol(A)` returns a permutation matrix `S`, such that `T'*T = S'*A*S`, and the value `p = 0` if matrix `A` is Hermitian positive definite. Otherwise, it returns a positive integer `p` and an empty symbolic object `S = sym([])`.

`[T,p,s] = chol(A,'vector')` returns the permutation information as a vector `s`, such that `A(s,s) = T'*T`. If `A` is not recognized as a Hermitian positive definite matrix, then `p` is a positive integer and `s = sym([])`.

`___ = chol(A,'lower')` returns a lower triangular matrix `T`, such that `T*T' = A`.

`___ = chol(A,'noCheck')` skips checking whether matrix `A` is Hermitian positive definite. `'noCheck'` lets you compute Cholesky factorization of a matrix that contains symbolic parameters without setting additional assumptions on those parameters.

___ = `chol(A,'real')` computes the Cholesky factorization of `A` using real arithmetic. In this case, `chol` computes a symmetric factorization `A = T.'*T` instead of a Hermitian factorization `A = T'*T`. This approach is based on the fact that if `A` is real and symmetric, then `T'*T = T.'*T`. Use `'real'` to avoid complex conjugates in the result.

___ = `chol(A,'lower','noCheck','real')` computes the Cholesky factorization of `A` with one or more of these optional arguments: `'lower'`, `'noCheck'`, and `'real'`. These optional arguments can appear in any order.

`[T,p,s] = chol(A,'lower','vector','noCheck','real')` computes the Cholesky factorization of `A` and returns the permutation information as a vector `s`. You can use one or more of these optional arguments: `'lower'`, `'noCheck'`, and `'real'`. These optional arguments can appear in any order.

Tips

- Calling `chol` for numeric arguments that are not symbolic objects invokes the MATLAB `chol` function.
- If you use `'noCheck'`, then the identities `T'*T = A` (for an upper triangular matrix `T`) and `T*T' = A` (for a lower triangular matrix `T`) are not guaranteed to hold.
- If you use `'real'`, then the identities `T'*T = A` (for an upper triangular matrix `T`) and `T*T' = A` (for a lower triangular matrix `T`) are only guaranteed to hold for a real symmetric positive definite `A`.
- To use `'vector'`, you must specify three output arguments. Other flags do not require a particular number of output arguments.
- If you use `'matrix'` instead of `'vector'`, then `chol` returns permutation matrices, as it does by default.
- If you use `'upper'` instead of `'lower'`, then `chol` returns an upper triangular matrix, as it does by default.
- If `A` is not a Hermitian positive definite matrix, then the syntaxes containing the argument `p` typically return `p = 1` and an empty symbolic object `T`.

- To check whether a matrix is Hermitian, use the operator `'` (or its functional form `ctranspose`). Matrix `A` is Hermitian if and only if `A'= A`, where `A'` is the conjugate transpose of `A`.

Input Arguments

A

Symbolic matrix.

'lower'

Flag that prompts `chol` to return a lower triangular matrix instead of an upper triangular matrix.

'vector'

Flag that prompts `chol` to return the permutation information in the form of a vector. To use this flag, you must specify three output arguments.

'noCheck'

Flag that prompts `chol` to avoid checking whether matrix `A` is Hermitian positive definite. Use this flag if `A` contains symbolic parameters, and you want to avoid additional assumptions on these parameters.

'real'

Flag that prompts `chol` to use real arithmetic. Use this flag if `A` contains symbolic parameters, and you want to avoid complex conjugates.

Output Arguments

T

Upper triangular matrix, such that `T'*T = A`, or lower triangular matrix, such that `T*T' = A`.

p

Value `0` if `A` is Hermitian positive definite or if you use `'noCheck'`.

If `chol` does not identify `A` as a Hermitian positive definite matrix, then `p` is a positive integer. `R` is an upper triangular matrix of order `q = p - 1`, such that `R'*R = A(1:q,1:q)`.

S

Permutation matrix.

s

Permutation vector.

Definitions

Hermitian Positive Definite Matrix

A square complex matrix *A* is Hermitian positive definite if `v'*A*v` is real and positive for all nonzero complex vectors `v`, where `v'` is the conjugate transpose (Hermitian transpose) of `v`.

Cholesky Factorization of a Matrix

The Cholesky factorization of a Hermitian positive definite *n*-by-*n* matrix `A` is defined by an upper or lower triangular matrix with positive entries on the main diagonal. The Cholesky factorization of matrix `A` can be defined as `T'*T = A`, where `T` is an upper triangular matrix. Here `T'` is the conjugate transpose of `T`. The Cholesky factorization also can be defined as `T*T' = A`, where `T` is a lower triangular matrix. `T` is called the Cholesky factor of `A`.

Examples

Compute the Cholesky factorization of the 3-by-3 Hilbert matrix. Because these numbers are not symbolic objects, you get floating-point results.

```
chol(hilb(3))
```

```
ans =
    1.0000    0.5000    0.3333
         0    0.2887    0.2887
         0         0    0.0745
```

Now convert this matrix to a symbolic object, and compute the Cholesky factorization:

```
chol(sym(hilb(3)))
```

```
ans =
[ 1,       1/2,        1/3]
[ 0, 3^(1/2)/6,  3^(1/2)/6]
[ 0,         0, 5^(1/2)/30]
```

Compute the Cholesky factorization of the 3-by-3 Pascal matrix returning a lower triangular matrix as a result:

```
chol(sym(pascal(3)), 'lower')
```

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

Try to compute the Cholesky factorization of this matrix. Because this matrix is not Hermitian positive definite, `chol` used without output arguments or with one output argument throws an error:

```
A = sym([1 1 1; 1 2 3; 1 3 5]);
```

```
T = chol(A)
```

```
Error using sym/chol (line 132)
Cannot prove that input matrix is Hermitian positive definite.
Define a Hermitian positive definite matrix by setting
appropriate assumptions on matrix components, or use 'noCheck'
to skip checking whether the matrix is Hermitian positive definite.
```

To suppress the error, use two output arguments, T and p. If the matrix is not recognized as Hermitian positive definite, then this syntax assigns an empty symbolic object to T and the value 1 to p:

```
[T,p] = chol(A)

T =
[ empty sym ]

p =
1
```

For a Hermitian positive definite matrix, p is 0:

```
[T,p] = chol(sym(pascal(3)))

T =
[ 1, 1, 1]
[ 0, 1, 2]
[ 0, 0, 1]

p =
0
```

Compute the Cholesky factorization of the 3-by-3 inverse Hilbert matrix returning the permutation matrix:

```
A = sym(invhilb(3));
[T, p, S] = chol(A)

T =
[ 3,       -12,          10]
[ 0, 4*3^(1/2), -5*3^(1/2)]
[ 0,         0,     5^(1/2)]

p =
0
```

```
S =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]
```

Compute the Cholesky factorization of the 3-by-3 inverse Hilbert matrix returning the permutation information as a vector:

```
A = sym(invhilb(3));
[T, p, S] = chol(A, 'vector')

T =
[ 3,       -12,         10]
[ 0, 4*3^(1/2), -5*3^(1/2)]
[ 0,         0,    5^(1/2)]

p =
0

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

Compute the Cholesky factorization of matrix `A` containing symbolic parameters. Without additional assumptions on the parameter `a`, this matrix is not Hermitian:

```
syms a
A = [a 0; 0 a];
isAlways(A == A')

ans =
     0     1
     1     0
```

By setting assumptions on a and b, you can define A to be Hermitian positive definite. Therefore, you can compute the Cholesky factorization of A:

```
assume(a > 0)
chol(A)

ans =
[ a^(1/2),       0]
[       0, a^(1/2)]
```

For further computations, remove the assumptions:

```
syms a clear
```

'noCheck' lets you skip checking whether A is a Hermitian positive definite matrix. Thus, this flag lets you compute the Cholesky factorization of a symbolic matrix without setting additional assumptions on its components:

```
A = [a 0; 0 a];
chol(A,'noCheck')

ans =
[ a^(1/2),       0]
[       0, a^(1/2)]
```

If you use 'noCheck' for computing the Cholesky factorization of a matrix that is not Hermitian positive definite, chol can return a matrix T for which the identity T'*T = A does not hold:

```
T = chol(sym([1 1; 2 1]), 'noCheck')

T =
[ 1,          2]
[ 0, 3^(1/2)*i]

isAlways(A == T'*T)
```

```
ans =
     0     0
     0     0
```

Compute the Cholesky factorization of this matrix. To skip checking whether it is Hermitian positive definite, use `'noCheck'`. By default, `chol` computes a Hermitian factorization `A = T'*T`. Thus, the result contains complex conjugates.

```
syms a b
A = [a b; b a];
T = chol(A, 'noCheck')

T =
[ a^(1/2),                      conj(b)/conj(a^(1/2))]
[       0, (a*abs(a) - abs(b)^2)^(1/2)/abs(a)^(1/2)]
```

To avoid complex conjugates in the result, use `'real'`:

```
T = chol(A, 'noCheck', 'real')

T =
[ a^(1/2),             b/a^(1/2)]
[       0, ((a^2 - b^2)/a)^(1/2)]
```

When you use this flag, `chol` computes a symmetric factorization `A = T.'*T` instead of a Hermitian factorization `A = T'*T`:

```
isAlways(A == T.'*T)

ans =
     1     1
     1     1

isAlways(A == T'*T)

ans =
     0     0
     0     0
```

See Also

chol | ctranspose | eig | isAlways | lu | svd | transpose | vpalinalg::factorCholesky | linalg::isHermitian | linalg::isPosDef

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`. For multivariate polynomials, specify `x` as a vector of indeterminates.

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 this polynomial expression with respect to the variable `x`:

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

The results are:

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

t =
[ x^2, x]
```

Display the list of the coefficients and the list of the terms of this polynomial expression with respect to `x` and `y`:

```
[c,t] = coeffs(z, [x y])
```

The results are:

```
c =
[ 3, 5]

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

See Also

`sym2poly`

Purpose

Collect coefficients

Syntax

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

Description

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

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

Examples

The following statements

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

return

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

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

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

See Also

`expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

colspace

Purpose

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` can be symbolic or numeric.

Examples

Find the basis for the column space of this 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 =
exp(-z/u)^t

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

See Also

finverse | subs | syms

Purpose

Condition number of matrix

Syntax

```
cond(A)
cond(A,P)
```

Description

`cond(A)` returns the `2`-norm condition number of matrix `A`.

`cond(A,P)` returns the `P`-norm condition number of matrix `A`.

Tips

- Calling `cond` for a numeric matrix that is not a symbolic object invokes the MATLAB `cond` function.

Input Arguments

A

Symbolic matrix.

P

One of these values `1`, `2`, `inf`, or `'fro'`.

- `cond(A,1)` returns the `1`-norm condition number.
- `cond(A,2)` or `cond(A)` returns the `2`-norm condition number.
- `cond(A,inf)` returns the infinity norm condition number.
- `cond(A,'fro')` returns the Frobenius norm condition number.

Default: `2`

Definitions

Condition Number of a Matrix

Condition number of a matrix is the ratio of the largest singular value of that matrix to the smallest singular value. The `P`-norm condition number of the matrix `A` is defined as `norm(A,P)*norm(inv(A),P)`, where `norm` is the norm of the matrix `A`.

Examples

Compute the `2`-norm condition number of the inverse of the 3-by-3 magic square `A`:

```
A = inv(sym(magic(3)));
condN2 = cond(A)

condN2 =
(5*3^(1/2))/2
```

Use vpa to approximate the result with 20-digit accuracy:

```
vpa(condN2, 20)

ans =
4.3301270189221932338
```

Compute the 1-norm condition number, the Frobenius condition number, and the infinity condition number of the inverse of the 3-by-3 magic square A:

```
A = inv(sym(magic(3)));
condN1 = cond(A, 1)
condNf = cond(A, 'fro')
condNi = cond(A, inf)

condN1 =
16/3

condNf =
(285^(1/2)*391^(1/2))/60

condNi =
16/3
```

Use vpa to approximate these condition numbers with 20-digit accuracy:

```
vpa(condN1, 20)
vpa(condNf, 20)
vpa(condNi, 20)
```

```
ans =
5.3333333333333333333

ans =
5.5636468855119361059

ans =
5.3333333333333333333
```

Compute the condition numbers of the 3-by-3 Hilbert matrix `H` approximating the results with 30-digit accuracy:

```
H = sym(hilb(3));
condN2 = vpa(cond(H), 30)
condN1 = vpa(cond(H, 1), 30)
condNf = vpa(cond(H, 'fro'), 30)
condNi = vpa(cond(H, inf), 30)
```

```
condN2 =
524.056777586060817870782845928 +...
1.42681147881398269481283800423e-38*i

condN1 =
748.0

condNf =
526.158821079719236517033364845

condNi =
748.0
```

Hilbert matrices are classic examples of ill-conditioned matrices.

See Also

`equationsToMatrix | inv | linsolve | norm | rank`

conj

Purpose

Symbolic complex conjugate

Syntax

```
conj(X)
```

Description

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

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

See Also

`real` | `imag`

Purpose

Cosine integral

Syntax

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

Description

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

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

where γ is Euler's constant 0.577215664...

Examples

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

Curl of vector field

Syntax

```
curl(V,X)
```

Description

`curl(V,X)` returns the curl of the vector field `V` with respect to the vector `X`. The vector field `V` and the vector `X` are both three-dimensional.

Input Arguments

V

Three-dimensional vector of symbolic expressions or symbolic functions.

X

Three-dimensional vector with respect to which you compute the curl.

Definitions

Curl of a Vector Field

The curl of the vector field $V = (V_1, V_2, V_3)$ with respect to the vector $X = (X_1, X_2, X_3)$ in Cartesian coordinates is the vector

$$curl(V) = \nabla \times V = \begin{pmatrix} \frac{\partial V_3}{\partial X_2} - \frac{\partial V_2}{\partial X_3} \\ \frac{\partial V_1}{\partial X_3} - \frac{\partial V_3}{\partial X_1} \\ \frac{\partial V_2}{\partial X_1} - \frac{\partial V_1}{\partial X_2} \end{pmatrix}$$

Examples

Compute the curl of this vector field with respect to vector $X = (x, y, z)$ in Cartesian coordinates:

```
syms x y z
curl([x^3*y^2*z, y^3*z^2*x, z^3*x^2*y], [x, y, z])

ans =
   x^2*z^3 - 2*x*y^3*z
   x^3*y^2 - 2*x*y*z^3
```

```
 - 2*x^3*y*z + y^3*z^2
```

Compute the curl of the gradient of this scalar function. The curl of the gradient of any scalar function is the vector of 0s:

```
syms x y z
f = x^2 + y^2 + z^2;
curl(gradient(f, [x, y, z]), [x, y, z])

ans =
 0
 0
 0
```

The vector Laplacian of a vector field V is defined as:

$$\nabla^2 V = \nabla(\nabla \cdot V) - \nabla \times (\nabla \times V)$$

Compute the vector Laplacian of this vector field using the `curl`, `divergence`, and `gradient` functions:

```
syms x y z
V = [x^2*y, y^2*z, z^2*x];
gradient(divergence(V, [x, y, z])) - curl(curl(V,
[x, y, z]), [x, y, z])

ans =
 2*y
 2*z
 2*x
```

See Also

`diff` | `divergence` | `gradient` | `jacobian` | `hessian` | `laplacian` | `potential` | `vectorPotential`

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`

diag

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`

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`

How To

- "Differentiation" on page 2-3

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.

If the value `d` is not an integer, `digits` rounds it to the nearest integer.

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 =
0.000000000000000000086736173798840354720600815844403
```

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 =
0.000000000000000000086736173798840354720600815844403
```

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
```

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`

How To

- "Variable-Precision Arithmetic" on page 2-50

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`.

Examples

`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`

Purpose

Divergence of vector field

Syntax

```
divergence(V,X)
```

Description

`divergence(V,X)` returns the divergence of the vector field `V` with respect to the vector `X` in Cartesian coordinates. Vectors `V` and `X` must have the same length.

Input Arguments

V

Vector of symbolic expressions or symbolic functions.

X

Vector with respect to which you compute the divergence.

Definitions

Divergence of a Vector Field

The divergence of the vector field $V = (V_1,...,V_n)$ with respect to the vector $X = (X_1,...,X_n)$ in Cartesian coordinates is the sum of partial derivatives of V with respect to $X_1,...,X_n$:

$$div(V) = \nabla \cdot V = \sum_{i=1}^{n} \frac{\partial V_i}{\partial x_i}$$

Examples

Compute the divergence of the vector field $V(x, y, z) = (x, 2y^2, 3z^3)$ with respect to vector $X = (x, y, z)$ in Cartesian coordinates:

```
syms x y z
divergence([x, 2*y^2, 3*z^3], [x, y, z])

ans =
9*z^2 + 4*y + 1
```

Compute the divergence of the curl of this vector field. The divergence of the curl of any vector field is 0.

```
syms x y z
divergence(curl([x, 2*y^2, 3*z^3], [x, y, z]), [x, y, z])

ans =
0
```

Compute the divergence of the gradient of this scalar function. The result is the Laplacian of the scalar function:

```
syms x y z
f = x^2 + y^2 + z^2;
divergence(gradient(f, [x, y, z]), [x, y, z])

ans =
6
```

See Also

`curl` | `diff` | `gradient` | `jacobian` | `hessian` | `laplacian` | `potential` | `vectorPotential`

Purpose

Get help for MuPAD functions

Syntax

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

Description

`doc(symengine)` opens “Getting Started with MuPAD”.

`doc(symengine,'MuPAD_function_name')` opens the documentation page for `MuPAD_function_name`.

Examples

`doc(symengine,'simplify')` opens the documentation page for the MuPAD `simplify` function.

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 `r`.

Tips

- The working precision for `double` depends on the input argument. It is also ultimately limited by 664 digits. If your computation requires a larger working precision, specify the number of digits explicitly using the `digits` function.

Input Arguments

S

Symbolic constant, constant expression, or symbolic matrix whose entries are constants or constant expressions.

Output Arguments

r

If `S` is a symbolic constant or constant expression, `r` is a double-precision floating-point number representing the value of `S`. If `S` is a symbolic matrix whose entries are constants or constant expressions, `r` is a matrix of double precision floating-point numbers representing the values of the entries of `S`.

Examples

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

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

```
1.6180
```

Find the numeric value for the entries of this matrix `T`:

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

```
double(T)

ans =
    2.8284    0.5359
    1.5157    0.1895
```

Find the numeric value for this expression. By default, double uses a new upper limit of 664 digits for the working precision and returns the value 0:

```
x = sym('((exp(200) + 1)/(exp(200) - 1)) - 1');
double(x)

ans =
     0
```

To get a more accurate result, increase the precision of computations:

```
digits(1000);
double(x)

ans =
   2.7678e-87
```

See Also

sym | vpa

dsolve

Purpose

Ordinary differential equation and system solver

Syntax

```
S = dsolve(eqn)
S = dsolve(eqn,cond)
S = dsolve(eqn,cond,Name,Value)
Y = dsolve(eqns)
Y = dsolve(eqns,conds)
Y = dsolve(eqns,conds,Name,Value)
[y1,...,yN] = dsolve(eqns)
[y1,...,yN] = dsolve(eqns,conds)
[y1,...,yN] = dsolve(eqns,conds,Name,Value)
```

Description

`S = dsolve(eqn)` solves the ordinary differential equation `eqn`. Here `eqn` is a symbolic equation containing `diff` to indicate derivatives. Alternatively, you can use a string with the letter `D` indicating derivatives. For example, `syms y(x); dsolve(diff(y) == y + 1)` and `dsolve('Dy = y + 1','x')` both solve the equation `dy/dx = y + 1` with respect to the variable `x`. Also, `eqn` can be an array of such equations or strings.

`S = dsolve(eqn,cond)` solves the ordinary differential equation `eqn` with the initial or boundary condition `cond`.

`S = dsolve(eqn,cond,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`Y = dsolve(eqns)` solves the system of ordinary differential equations `eqns` and returns a structure array that contains the solutions. The number of fields in the structure array corresponds to the number of independent variables in the system.

`Y = dsolve(eqns,conds)` solves the system of ordinary differential equations `eqns` with the initial or boundary conditions `conds`.

`Y = dsolve(eqns,conds,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[y1,...,yN] = dsolve(eqns)` solves the system of ordinary differential equations `eqns` and assigns the solutions to the variables `y1,...,yN`.

`[y1,...,yN] = dsolve(eqns,conds)` solves the system of ordinary differential equations `eqns` with the initial or boundary conditions `conds`.

`[y1,...,yN] = dsolve(eqns,conds,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Tips

- The names of symbolic variables used in differential equations should not contain the letter `D` because `dsolve` assumes that `D` is a differential operator and any character immediately following `D` is a dependent variable.
- If `dsolve` cannot find a closed-form (explicit) solution, it attempts to find an implicit solution. When `dsolve` returns an implicit solution, it issues this warning:

  ```
  Warning: Explicit solution could not be found;
  implicit solution returned.
  ```

- If `dsolve` can find neither an explicit nor an implicit solution, then it issues a warning and returns the empty `sym`. In this case, try to find a numeric solution using the MATLAB `ode23` or `ode45` function. In some cases, the output is an equivalent lower-order differential equation or an integral.

Input Arguments

eqn

Symbolic equation, string representing an ordinary differential equation, or array of symbolic equations or strings.

When representing `eqn` as a symbolic equation, you must create a symbolic function, for example `y(x)`. Here `x` is an independent variable for which you solve an ordinary differential equation. Use the `==` operator to create an equation. Use the `diff` function to indicate differentiation. For example, to solve $d^2y(x)/dx^2 = x*y(x)$, enter:

```
syms y(x)
dsolve(diff(y, 2) == x*y)
```

When representing `eqn` as a string, use the letter `D` to indicate differentiation. By default, `dsolve` assumes that the independent variable is `t`. Thus, `Dy` means `dy/dt`. You can specify the independent variable. The letter `D` followed by a digit indicates repeated differentiation. Any character immediately following a differentiation operator is a dependent variable. For example, to solve `y''(x) = x*y(x)`, enter:

```
dsolve('D2y = x*y','x')
```

or

```
dsolve('D2y == x*y','x')
```

cond

Equation or string representing an initial or boundary condition. If you use equations, assign expressions with `diff` to some intermediate variables. For example, use `Dy`, `D2y`, and so on as intermediate variables:

```
Dy = diff(y);
D2y = diff(y, 2);
```

Then define initial conditions using symbolic equations, such as `y(a) == b` and `Dy(a) == b`. Here `a` and `b` are constants.

If you represent initial and boundary conditions as strings, you do not need to create intermediate variables. In this case, follow the same rules as you do when creating an equation `eqn` as a string. For example, specify `'y(a) = b'` and `'Dy(a) = b'`. When using strings, you can use = or == in equations.

eqns

Symbolic equations or strings separated by commas and representing a system of ordinary differential equations. Each equation or string represents an ordinary differential equation.

conds

Symbolic equations or strings separated by commas and representing initial or boundary conditions or both types of conditions. Each equation or string represents an initial or boundary condition. If the number of the specified conditions is less than the number of dependent variables, the resulting solutions contain arbitrary constants `C1`, `C2`,....

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'IgnoreAnalyticConstraints'

By default, the solver applies the purely algebraic simplifications to the expressions on both sides of equations. These simplifications might not be generally valid. Therefore, by default the solver does not guarantee general correctness and completeness of the results. To solve ordinary differential equations without additional assumptions, set `IgnoreAnalyticConstraints` to `false`. The results obtained with `IgnoreAnalyticConstraints` set to `false` are correct for all values of the arguments.

If you do not set `IgnoreAnalyticConstraints` to `false`, always verify results returned by the `dsolve` command.

Default: `true`

'MaxDegree'

Do not use explicit formulas that involve radicals when solving polynomial equations of degrees larger than the specified value. This value must be a positive integer smaller than 5.

Default: 2

Output Arguments

S

Symbolic array that contains solutions of an equation. The size of a symbolic array corresponds to the number of the solutions.

Y

Structure array that contains solutions of a system of equations. The number of fields in the structure array corresponds to the number of independent variables in a system.

y1,...,yN

Variables to which the solver assigns the solutions of a system of equations. The number of output variables or symbolic arrays must equal the number of independent variables in a system. The toolbox sorts independent variables alphabetically, and then assigns the solutions for these variables to output variables or symbolic arrays.

Examples

Solve these ordinary differential equations. Use == to create an equation, and `diff` to indicate differentiation:

```
syms a x(t)
dsolve(diff(x) == -a*x)

ans =
C2*exp(-a*t)

syms f(t)
dsolve(diff(f) == f + sin(t))

ans =
C4*exp(t) - sin(t)/2 - cos(t)/2
```

Solve this ordinary differential equation with the initial condition `y(0) = b`:

```
syms a b y(t)
```

```
dsolve(diff(y) == a*y, y(0) == b)
```

Specifying the initial condition lets you eliminate arbitrary constants, such as `C1`, `C2,...`:

```
ans =
b*exp(a*t)
```

Solve this ordinary differential equation with the initial and boundary conditions. To specify a condition that contains a derivative, assign the derivative to a variable:

```
syms a y(t)
Dy = diff(y);
dsolve(diff(y, 2) == -a^2*y, y(0) == 1, Dy(pi/a) == 0)
```

Because the equation contains the second-order derivative d^2y/dt^2, specifying two conditions lets you eliminate arbitrary constants in the solution:

```
ans =
exp(-a*t*i)/2 + exp(a*t*i)/2
```

Solve this system of ordinary differential equations:

```
syms x(t) y(t)
z = dsolve(diff(x) == y, diff(y) == -x)
```

When you assign the solution of a system of equations to a single output, `dsolve` returns a structure containing the solutions:

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

To see the results, enter `z.x` and `z.y`:

```
z.x
```

```
ans =
C12*cos(t) + C11*sin(t)

z.y

ans =
C11*cos(t) - C12*sin(t)
```

By default, the solver applies a set of purely algebraic simplifications that are not correct in general, but that can produce simple and practical solutions:

```
syms y(t)
dsolve(diff(y) == 1/sqrt(y), y(0) == 1)

ans =
((3*t)/2 + 1)^(2/3)
```

To obtain complete and generally correct solutions, set the value of `IgnoreAnalyticConstraints` to `false`:

```
dsolve(diff(y) == 1/sqrt(y), y(0) == 1,
'IgnoreAnalyticConstraints', false)

Warning: Explicit solution could not be found;
implicit solution returned.
Warning: The solutions are parametrized by the symbols:
l = Z_ intersect Dom::Interval([-(3*(PI/2 - angle(C19 + t)/3))/(2*PI)],..
(3*(PI/2 + angle(C19 + t)/3))/(2*PI)) intersect...
solve([C21 in Dom::Interval([-(2*(PI/2 - (3*angle(exp((4*PI*X319*I)/3)))/
(2*(PI/2 + (3*angle(exp((4*PI*X319*I)/3)))/4))/(3*PI)), C21 in Z_], X319,

ans =
exp(-(pi*l*4*i)/3)*((3*t)/2 +
exp(-C21*pi*3*i)*exp((pi*l*4*i)/3)^(3/2))^(2/3)
```

If you apply algebraic simplifications, you can get explicit solutions for some equations for which the solver cannot compute them using strict mathematical rules:

```
syms y(t)
dsolve(sqrt(diff(y)) == sqrt(y) + 1/y)

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

versus

```
dsolve(sqrt(diff(y)) == sqrt(y) + 1/y,
'IgnoreAnalyticConstraints', false)

Warning: Explicit solution could not be found;
implicit solution returned.

ans =
solve(signIm(((y(t)^(3/2) + 1)*i)/y(t)) == 1, y(t)) intersect...
Dom::ImageSet(exp(pi*l*(-(4*i)/3))*(exp((3*C28)/2 +...
(3*t)/2)*exp(wrightOmega(- (3*C28)/2 + pi*i -...
(3*t)/2)) - 1)^(2/3), l, Z_ intersect...
Dom::Interval([-(3*(pi/2 - angle(exp((3*C28)/2 +...
(3*t)/2)*exp(wrightOmega(- (3*C28)/2 + pi*i -...
(3*t)/2)) - 1)/3))/(2*pi)], (3*(pi/2 + angle(exp((3*C28)/2 +...
(3*t)/2)*exp(wrightOmega(- (3*C28)/2 + pi*i -
(3*t)/2)) - 1)/3))/(2*pi)))
```

When you solve a higher-order polynomial equation, the solver sometimes uses `RootOf` to return the results:

```
syms a y(x)
dsolve(diff(y) == a/(y^2 + 1))
```

```
Warning: Explicit solution could not be found;
implicit solution returned.

ans =
RootOf(z^3 + 3*z - 3*a*x - 3*C32, z)
```

To get an explicit solution for such equations, try calling the solver with `MaxDegree`. The option specifies the maximum degree of polynomials for which the solver tries to return explicit solutions. The default value is 2. By increasing this value, you can get explicit solutions for higher-order polynomials. For example, increase the value of `MaxDegree` to 4 and get explicit solutions instead of `RootOf` for this equation:

```
s = dsolve(diff(y) == a/(y^2 + 1), 'MaxDegree', 4);
pretty(s)


  +-                                 -+
  |                  1                |
  |             #1 - --               |
  |                  #1               |
  |                                   |
  |              1/2 / 1       \      |
  |             3    | -- + #1 | i    |
  |   1     #1       \ #1      /      |
  |  ---- - -- + -----------------    |
  |  2 #1   2            2            |
  |                                   |
  |              1/2 / 1       \      |
  |             3    | -- + #1 | i    |
  |   1     #1       \ #1      /      |
  |  ---- - -- - -----------------    |
  |  2 #1   2            2            |
  +-                                 -+

  where

           /                 /              2     \1/2 \1/3
           | 3 C36   3 a x   | 9 (C36 + a x)      |    |
     #1 == | ----- + ----- + | -------------- + 1 |    |
           \   2       2     \       4            /    /
```

If `dsolve` can find neither an explicit nor an implicit solution, then it issues a warning and returns the empty `sym`:

```
syms y(x)
dsolve(exp(diff(y)) == 0)

 Warning: Explicit solution could not be found.
```

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

Returning the empty symbolic object does not prove that there are no solutions.

Solve this equation specifying it as a string. By default, `dsolve` assumes that the independent variable is `t`:

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

ans =
              1
             -1
 cosh(C45 + t*i)
 cosh(C41 - t*i)
```

Now solve this equation with respect to the variable `s`:

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

ans =
              1
             -1
 cosh(C53 + s*i)
 cosh(C49 - s*i)
```

Algorithms

If you do not set the value of `IgnoreAnalyticConstraints` to `false`, the solver applies these rules to the expressions on both sides of an equation:

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

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

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

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

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
  - $\log(e^x) = x$
  - $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
  - $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
  - $\mathrm{W}_k(x \cdot e^x) = x$ for all values of k
- The solver can multiply both sides of an equation by any expression except `0`.
- The solutions of polynomial equations must be complete.

See Also

`ode23` | `ode45` | `odeToVectorField` | `solve` | `syms`

How To

- “Solve a Single Differential Equation” on page 2-88
- “Solve a System of Differential Equations” on page 2-92

Purpose

One-argument exponential integral function

Syntax

`ei(x)`

Description

`ei(x)` returns the one-argument exponential integral defined as follows:

$$\mathrm{ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt$$

Tips

- The one-argument exponential integral is singular at `x = 0`. The toolbox uses this special value: `ei(0) = Inf`.

Input Arguments

x - Input

floating-point number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input specified as a floating-point or symbolic number, variable, expression, function, vector, or matrix.

Examples

Exponential Integral for Floating-Point and Symbolic Numbers

Compute the exponential integrals for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ei(-2), ei(-1/2), ei(1), ei(sqrt(2))]
```

```
s =
   -0.0489   -0.5598    1.8951    3.0485
```

Compute the exponential integrals for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ei` returns unresolved symbolic calls.

```
s = [ei(sym(-2)), ei(sym(-1/2)), ei(sym(1)),
ei(sqrt(sym(2)))]
```

```
s =
[ ei(-2), ei(-1/2), ei(1), ei(2^(1/2))]
```

Use vpa to approximate this result with the 10 digits accuracy:

```
vpa(s, 10)

ans =
[ -0.04890051071, -0.5597735948, 1.895117816, 3.048462479]
```

Branch Cut at the Negative Real Axis

Compute the exponential integrals for these numbers. The negative real axis is a branch cut. The exponential integral has a jump of height 2πi when crossing this cut:

```
[ei(-1), ei(-1 + 10^(-10)*i), ei(-1 - 10^(-10)*i)]

ans =
  -0.2194 + 0.0000i  -0.2194 + 3.1416i  -0.2194 - 3.1416i
```

Derivatives of the Exponential Integral

Compute the first, second, and third derivatives of the one-argument exponential integral:

```
syms x
diff(ei(x), x)
diff(ei(x), x, 2)
diff(ei(x), x, 3)

ans =
exp(x)/x

ans =
exp(x)/x - exp(x)/x^2

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

Limits of the Exponential Integral

Compute the limits of this one-argument exponential integral:

```
syms x
limit(ei(2*x^2/(1+x)), x, -Inf)
limit(ei(2*x^2/(1+x)), x, 0)
limit(ei(2*x^2/(1+x)), x, Inf)
```

```
ans =
0

ans =
-Inf

ans =
Inf
```

References

[1] Gautschi, W., and W. F. Gahill "Exponential Integral and Related Functions." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`expint | expintEi | int | vpa`

Purpose

Eigenvalues and eigenvectors of symbolic matrix

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
  -13.1262809307092188025256430859493
  -21.276765471473795530626426697974
```

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

`charpoly` | `jordan` | `svd` | `vpa`

How To

- “Eigenvalues” on page 2-64

ellipke

Purpose

Complete elliptic integrals of the first and second kinds

Syntax

```
[K,E] = ellipke(m)
```

Description

`[K,E] = ellipke(m)` returns the complete elliptic integrals of the first and second kinds.

Tips

- Calling `ellipke` for numbers that are not symbolic objects invokes the MATLAB `ellipke` function. This function accepts only `0 <= x <= 1`. To compute the complete elliptic integrals of the first and second kinds for the values out of this range, use `sym` to convert the numbers to symbolic objects, and then call `ellipke` for those symbolic objects. Alternatively, use the `ellipticK` and `ellipticE` functions to compute the integrals separately.
- For most symbolic (exact) numbers, `ellipke` returns results using the `ellipticK` and `ellipticE` functions. You can approximate such results with floating-point numbers using `vpa`.
- If `m` is a vector or a matrix, then `[K,E] = ellipke(m)` returns the complete elliptic integrals of the first and second kinds, evaluated for each element of `m`.

Input Arguments

m

Symbolic number, variable, expression, or function. This argument also can be a vector or matrix of symbolic numbers, variables, expressions, or functions.

Output Arguments

K

Complete elliptic integral of the first kind.

E

Complete elliptic integral of the second kind.

Definitions

Complete Elliptic Integral of the First Kind

The complete elliptic integral of the first kind is defined as follows:

$$K(m) = F\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - m\sin^2\theta}} d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Complete Elliptic Integral of the Second Kind

The complete elliptic integral of the second kind is defined as follows:

$$E(m) = E\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta} d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complete elliptic integrals of the first and second kinds for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[K0, E0] = ellipke(0)
[K05, E05] = ellipke(1/2)

K0 =
    1.5708

E0 =
    1.5708

K05 =
    1.8541

E05 =
    1.3506
```

Compute the complete elliptic integrals for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipke` returns results using the `ellipticK` and `ellipticE` functions.

```
[K0, E0] = ellipke(sym(0))
[K05, E05] = ellipke(sym(1/2))

K0 =
pi/2

E0 =
pi/2

K05 =
ellipticK(1/2)

E05 =
ellipticE(1/2)
```

Use `vpa` to approximate `K05` and `E05` with floating-point numbers:

```
vpa([K05, E05], 10)

ans =
[ 1.854074677, 1.350643881]
```

If the argument does not belong to the range from 0 to 1, then convert that argument to a symbolic object before using `ellipke`:

```
[K, E] = ellipke(sym(pi/2))

K =
ellipticK(pi/2)

E =
ellipticE(pi/2)
```

Alternatively, use `ellipticK` and `ellipticE` to compute the integrals of the first and the second kinds separately:

```
K = ellipticK(sym(pi/2))
E = ellipticE(sym(pi/2))

K =
ellipticK(pi/2)

E =
ellipticE(pi/2)
```

Call `ellipke` for this symbolic matrix. When the input argument is a matrix, `ellipke` computes the complete elliptic integrals of the first and second kinds for each element.

```
[K, E] = ellipke(sym([-1 0; 1/2 1]))

K =
[  ellipticK(-1), pi/2]
[ ellipticK(1/2),  Inf]

E =
[  ellipticE(-1), pi/2]
[ ellipticE(1/2),    1]
```

References

[1] Milne-Thomson, L. M. “Elliptic Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

Alternatives

You can use `ellipticK` and `ellipticE` to compute elliptic integrals of the first and second kinds separately.

See Also

`ellipke | ellipticE | ellipticKellipticE | ellipticK | vpa`

ellipticCE

Purpose

Complementary complete elliptic integral of the second kind

Syntax

```
ellipticCE(m)
```

Description

`ellipticCE(m)` returns the complementary complete elliptic integral of the second kind.

Tips

- `ellipticCE` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticCE` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If `m` is a vector or a matrix, then `ellipticCE(m)` returns the complementary complete elliptic integral of the second kind, evaluated for each element of `m`.

Input Arguments

m

Number, symbolic number, variable, expression, or function. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Complementary Complete Elliptic Integral of the Second Kind

The complementary complete elliptic integral of the second kind is defined as $E'(m) = E(1-m)$, where $E(m)$ is the complete elliptic integral of the second kind:

$$E(m) = E\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complementary complete elliptic integrals of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticCE(0), ellipticCE(pi/4), ellipticCE(1),
ellipticCE(pi/2)]

s =
    1.0000    1.4828    1.5708    1.7753
```

Compute the complementary complete elliptic integrals of the second kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticCE` returns unresolved symbolic calls.

```
s = [ellipticCE(sym(0)), ellipticCE(sym(pi/4)),
ellipticCE(sym(1)), ellipticCE(sym(pi/2))]

s =
[ 1, ellipticCE(pi/4), pi/2, ellipticCE(pi/2)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.0, 1.482786927, 1.570796327, 1.775344699]
```

Differentiate these expressions involving the complementary complete elliptic integral of the second kind:

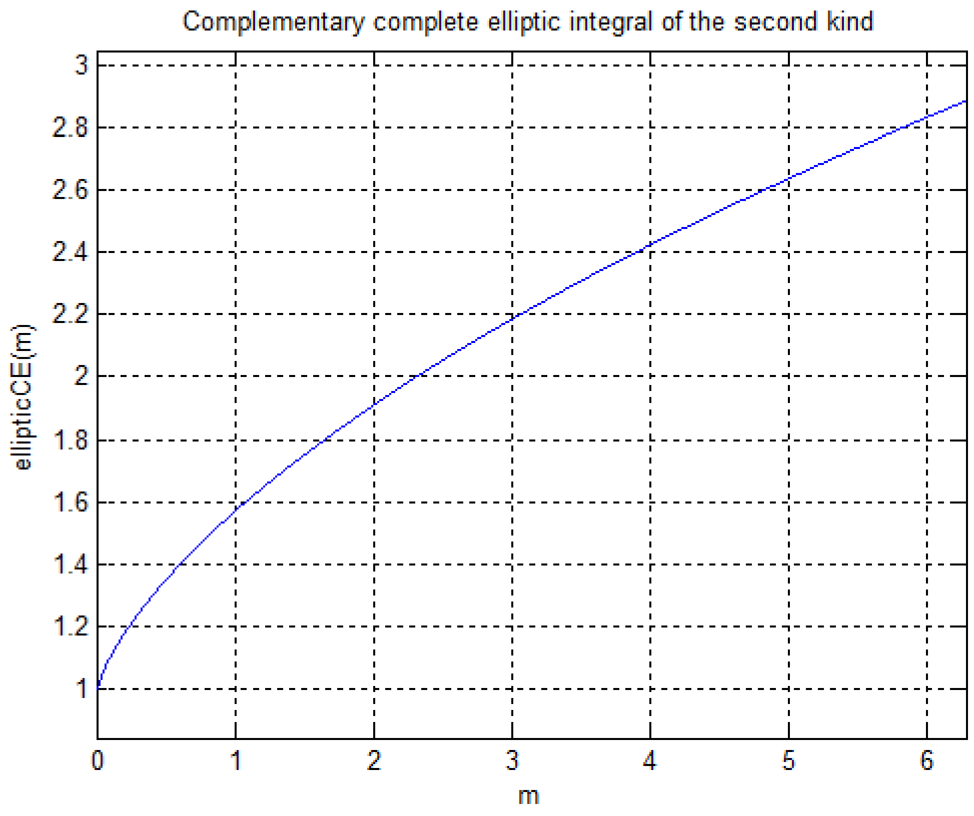
```
syms m
diff(ellipticCE(m))
diff(ellipticCE(m^2), m, 2)

ans =
ellipticCE(m)/(2*m - 2) - ellipticCK(m)/(2*m - 2)
```

```
ans =
(2*ellipticCE(m^2))/(2*m^2 - 2) -...
(2*ellipticCK(m^2))/(2*m^2 - 2) +...
2*m*(((2*m*ellipticCK(m^2))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m*(m^2 - 1)))/(2*m^2 - 2) +...
(2*m*(ellipticCE(m^2)/(2*m^2 - 2) -...
ellipticCK(m^2)/(2*m^2 - 2)))/(2*m^2 - 2) -...
(4*m*ellipticCE(m^2))/(2*m^2 - 2)^2 +...
(4*m*ellipticCK(m^2))/(2*m^2 - 2)^2)
```

Here, `ellipticCK` represents the complementary complete elliptic integral of the first kind.

Plot the complementary complete elliptic integral of the second kind:

```
syms m
ezplot(ellipticCE(m))
hold on

colormap([0 0 1])
title('Complementary complete elliptic integral of the second kind')
xlabel('m')
ylabel('ellipticCE(m)')
grid
hold off
```



Call `ellipticCE` for this symbolic matrix. When the input argument is a matrix, `ellipticCE` computes the complementary complete elliptic integral of the second kind for each element.

```
ellipticCE(sym([pi/6 pi/4; pi/3 pi/2]))
```

```
ans =
[ ellipticCE(pi/6), ellipticCE(pi/4)]
[ ellipticCE(pi/3), ellipticCE(pi/2)]
```

References

[1] Milne-Thomson, L. M. “Elliptic Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`ellipke` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticK` | `ellipticF` | `ellipticPiellipticCE` | `vpa`

Purpose

Complementary complete elliptic integral of the first kind

Syntax

```
ellipticCK(m)
```

Description

`ellipticCK(m)` returns the complementary complete elliptic integral of the first kind.

Tips

- `ellipticK` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticCK` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using the `vpa` function.
- If `m` is a vector or a matrix, then `ellipticCK(m)` returns the complementary complete elliptic integral of the first kind, evaluated for each element of `m`.

Input Arguments

m

Number, symbolic number, variable, expression, or function. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Complementary Complete Elliptic Integral of the First Kind

The complementary complete elliptic integral of the first kind is defined as $K'(m) = K(1-m)$, where $K(m)$ is the complete elliptic integral of the first kind:

$$K(m) = F\left(\frac{\pi}{2} \,|\, m\right) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - m\sin^2\theta}}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complementary complete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticCK(1/2), ellipticCK(pi/4), ellipticCK(1),
ellipticCK(inf)]

s =
    1.8541    1.6671    1.5708       NaN
```

Compute the complete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticCK` returns unresolved symbolic calls.

```
s = [ellipticCK(sym(1/2)), ellipticCK(sym(pi/4)),
ellipticCK(sym(1)), ellipticCK(sym(inf))]

s =
[ ellipticCK(1/2), ellipticCK(pi/4), pi/2, ellipticCK(Inf)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.854074677, 1.667061338, 1.570796327, NaN]
```

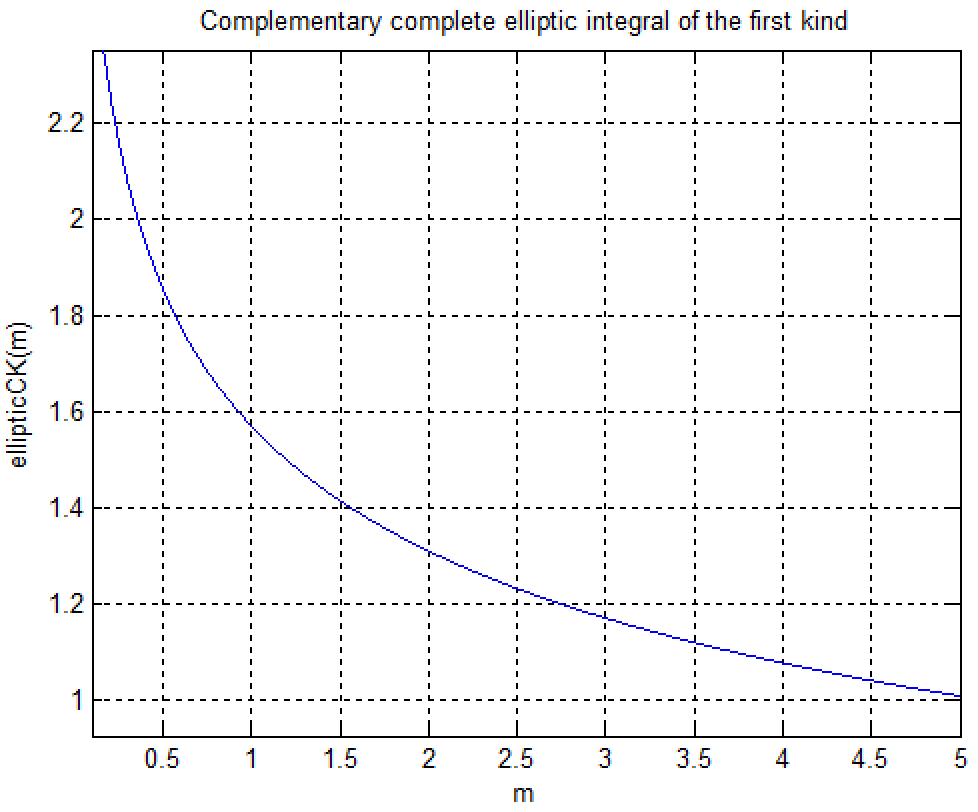
Differentiate these expressions involving the complementary complete elliptic integral of the first kind:

```
syms m
diff(ellipticCK(m))
diff(ellipticCK(m^2), m, 2)

ans =
ellipticCE(m)/(2*m*(m - 1)) - ellipticCK(m)/(2*m - 2)

ans =
(2*(ellipticCE(m^2)/(2*m^2 - 2) -...
ellipticCK(m^2)/(2*m^2 - 2)))/(m^2 - 1) -...
(2*ellipticCE(m^2))/(m^2 - 1)^2 -...
(2*ellipticCK(m^2))/(2*m^2 - 2) +...
(8*m^2*ellipticCK(m^2))/(2*m^2 - 2)^2 +...
(2*m*((2*m*ellipticCK(m^2))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m*(m^2 - 1)))))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m^2*(m^2 - 1))
```

Here, `ellipticCE` represents the complementary complete elliptic integral of the second kind.

Plot the complementary complete elliptic integral of the first kind:

```
syms m
ezplot(ellipticCK(m), [0.1, 5])
hold on

colormap([0 0 1])
title('Complementary complete elliptic integral of the first kind')
xlabel('m')
ylabel('ellipticCK(m)')
grid
hold off
```



Call `ellipticCK` for this symbolic matrix. When the input argument is a matrix, `ellipticCK` computes the complementary complete elliptic integral of the first kind for each element.

```
ellipticCK(sym([pi/6 pi/4; pi/3 pi/2]))
```

```
ans =
[ ellipticCK(pi/6), ellipticCK(pi/4)]
[ ellipticCK(pi/3), ellipticCK(pi/2)]
```

References

[1] Milne-Thomson, L. M. “Elliptic Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`ellipke` | `ellipticCE` | `ellipticCPi` | `ellipticE` | `ellipticK` | `ellipticF` | `ellipticPiellipticCK` | `vpa`

Purpose

Complementary complete elliptic integral of the third kind

Syntax

`ellipticCPi(n,m)`

Description

`ellipticCPi(n,m)` returns the complementary complete elliptic integral of the third kind.

Tips

- `ellipticCPi` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticCPi` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If one input argument is a scalar and the other one is a vector or a matrix, then `ellipticCPi` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

Input Arguments

n

Number, symbolic number, variable, expression, or function specifying the characteristic. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

m

Number, symbolic number, variable, expression, or function specifying the parameter. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Complementary Complete Elliptic Integral of the Third Kind

The complementary complete elliptic integral of the third kind is defined as $\Pi'(m) = \Pi(n, 1-m)$, where $\Pi(n,m)$ is the complete elliptic integral of the third kind:

$$\Pi(n,m) = \Pi\left(n;\frac{\pi}{2}\,|\,m\right) = \int_0^{\pi/2} \frac{1}{\left(1-n\sin^2\theta\right)\sqrt{1-m\sin^2\theta}}\,d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complementary complete elliptic integrals of the third kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticCPi(-1, 1/3), ellipticCPi(0, 1/2),
ellipticCPi(9/10, 1), ellipticCPi(-1, 0)]

s =
    1.3703    1.8541    4.9673       Inf
```

Compute the complementary complete elliptic integrals of the third kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticCPi` returns unresolved symbolic calls.

```
s = [ellipticCPi(-1, sym(1/3)), ellipticCPi(sym(0), 1/2),
ellipticCPi(sym(9/10), 1), ellipticCPi(-1, sym(0))]

s =
[ ellipticCPi(-1, 1/3), ellipticCK(1/2),
(pi*10^(1/2))/2, Inf]
```

Here, `ellipticCK` represents the complementary complete elliptic integrals of the first kind.

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.370337322, 1.854074677, 4.967294133, Inf]
```

Differentiate these expressions involving the complementary complete elliptic integral of the third kind:

```
syms n m
diff(ellipticCPi(n, m), n)
diff(ellipticCPi(n, m), m)
```

```
ans =
ellipticCK(m)/(2*n*(n - 1)) -...
ellipticCE(m)/(2*(n - 1)*(m + n - 1)) -...
(ellipticCPi(n, m)*(n^2 + m - 1))/(2*n*(n - 1)*(m + n - 1))

ans =
ellipticCE(m)/(2*m*(m + n - 1)) - ellipticCPi(n,
m)/(2*(m + n - 1))
```

Here, `ellipticCK` and `ellipticCE` represent the complementary complete elliptic integrals of the first and second kinds.

References

[1] Milne-Thomson, L. M. "Elliptic Integrals." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`ellipke` | `ellipticCE` | `ellipticCK` | `ellipticE` | `ellipticK` | `ellipticF` | `ellipticPiellipticCPi` | `vpa`

Purpose

Elliptic integral of the second kind

Syntax

```
ellipticE(m)
ellipticE(phi,m)
```

Description

`ellipticE(m)` returns the complete elliptic integral of the second kind.

`ellipticE(phi,m)` returns the incomplete elliptic integral of the second kind.

Tips

- `ellipticE` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticE` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If `m` is a vector or a matrix, then `ellipticE(m)` returns the complete elliptic integral of the second kind, evaluated for each element of `m`.
- If one input argument is a scalar and the other one is a vector or a matrix, then `ellipticE` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.
- `ellipticE(pi/2, m) = ellipticE(m)`.

Input Arguments

m

Number, symbolic number, variable, expression, or function specifying the parameter. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

phi

Number, symbolic number, variable, expression, or function specifying the amplitude. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Incomplete Elliptic Integral of the Second Kind

The incomplete elliptic integral of the second kind is defined as follows:

$$E(\varphi \mid m) = \int_0^{\varphi} \sqrt{1 - m\sin^2\theta}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Complete Elliptic Integral of the Second Kind

The complete elliptic integral of the second kind is defined as follows:

$$E(m) = E\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complete elliptic integrals of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticE(-10.5), ellipticE(-pi/4),
ellipticE(0),  ellipticE(1)]

s =
    3.7096    1.8443    1.5708    1.0000
```

Compute the complete elliptic integral of the second kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticE` returns unresolved symbolic calls.

```
s = [ellipticE(sym(-10.5)), ellipticE(sym(-pi/4)),
ellipticE(sym(0)),  ellipticE(sym(1))]

s =
[ ellipticE(-21/2), ellipticE(-pi/4), pi/2, 1]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 3.70961391, 1.844349247, 1.570796327, 1.0]
```

Differentiate these expressions involving elliptic integrals of the second kind:

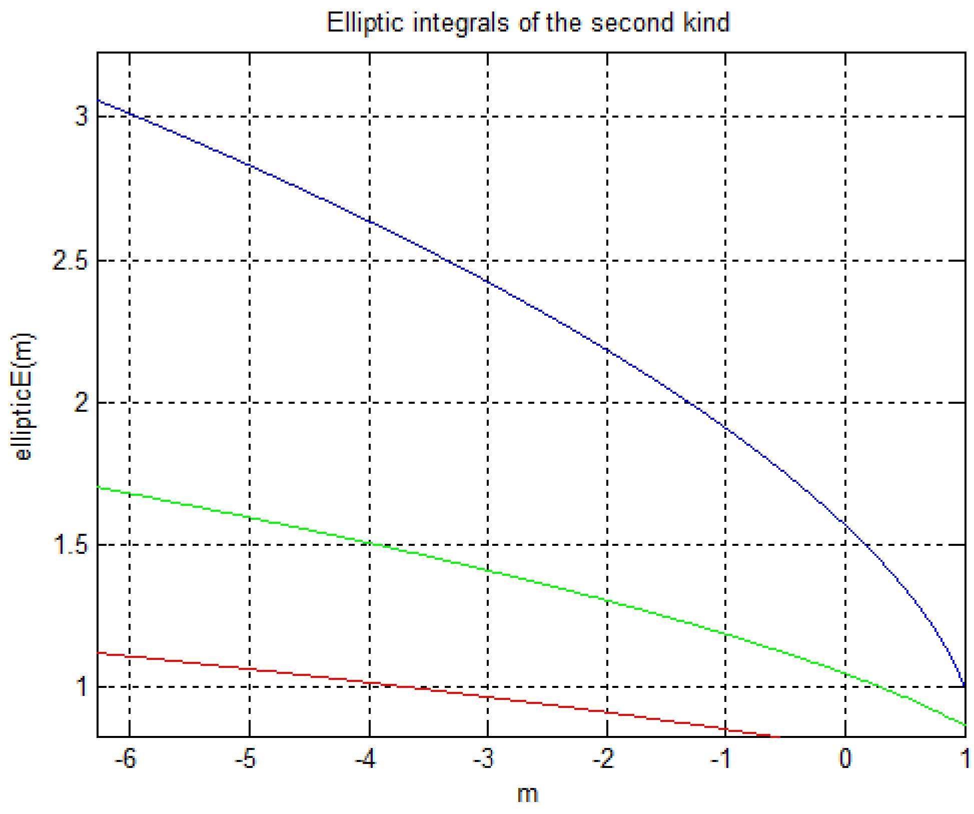
```
syms m
diff(ellipticE(pi/3, m))
diff(ellipticE(m^2), m, 2)

ans =
ellipticE(pi/3, m)/(2*m) - ellipticF(pi/3, m)/(2*m)

ans =
2*m*((ellipticE(m^2)/(2*m^2) -...
ellipticK(m^2)/(2*m^2))/m - ellipticE(m^2)/m^3 +...
ellipticK(m^2)/m^3 + (ellipticK(m^2)/m +...
ellipticE(m^2)/(m*(m^2 - 1)))/(2*m^2)) +...
ellipticE(m^2)/m^2 - ellipticK(m^2)/m^2
```

Here, `ellipticK` and `ellipticF` represent the complete and incomplete elliptic integrals of the first kind, respectively.

Plot the incomplete elliptic integrals `ellipticE(phi,m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticE(m)`:

```
syms m
p1 = ezplot(ellipticE(pi/4, m));
hold on
p2 = ezplot(ellipticE(pi/3, m));
```

```
p3 = ezplot(ellipticE(m));

set(p1,'Color','red')
set(p2,'Color','green')

title('Elliptic integrals of the second kind')
xlabel('m')
ylabel('ellipticE(m)')
grid
hold off
```



Call `ellipticE` for this symbolic matrix. When the input argument is a matrix, `ellipticE` computes the complete elliptic integral of the second kind for each element.

```
ellipticE(sym([1/3 1; 1/2 0]))
```

```
ans =
[ ellipticE(1/3),    1]
[ ellipticE(1/2), pi/2]
```

References

[1] Milne-Thomson, L. M. “Elliptic Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

See Also

`ellipke` | `ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticF` | `ellipticK` | `ellipticPiellipticE` | `vpa`

ellipticF

Purpose

Incomplete elliptic integral of the first kind

Syntax

`ellipticF(phi,m)`

Description

`ellipticF(phi,m)` returns the complete elliptic integral of the first kind.

Tips

- `ellipticF` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticF` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If one input argument is a scalar and the other one is a vector or a matrix, `ellipticF` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.
- `ellipticF(pi/2, m) = ellipticK(m)`.

Input Arguments

m

Number, symbolic number, variable, expression, or function specifying the parameter. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

phi

Number, symbolic number, variable, expression, or function specifying the amplitude. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Incomplete Elliptic Integral of the First Kind

The complete elliptic integral of the first kind is defined as follows:

$$F(\varphi \mid m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - m \sin^2 \theta}} d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the incomplete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticF(pi/3, -10.5), ellipticF(pi/4, -pi),
ellipticF(1, -1),  ellipticF(pi/2, 0)]
```

```
s =
    0.6184    0.6486    0.8964    1.5708
```

Compute the incomplete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticF` returns unresolved symbolic calls.

```
s = [ellipticF(sym(pi/3), -10.5),
ellipticF(sym(pi/4), -pi),...
ellipticF(sym(1), -1),  ellipticF(pi/6, sym(0))]
```

```
s =
[ ellipticF(pi/3, -21/2), ellipticF(pi/4, -pi),
ellipticF(1, -1), pi/6]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)
```

```
ans =
[ 0.6184459461, 0.6485970495, 0.8963937895, 0.5235987756]
```

Differentiate this expression involving the incomplete elliptic integral of the first kind:

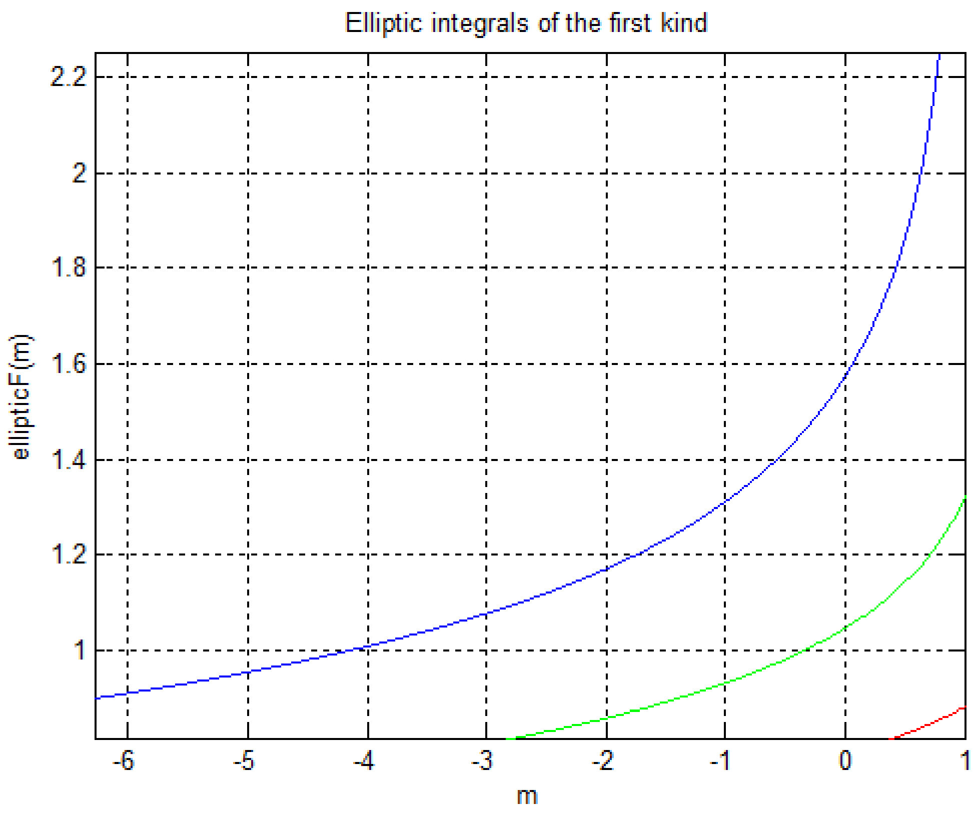
```
syms m
diff(ellipticF(pi/4, m))
```

```
ans =
1/(4*(1 - m/2)^(1/2)*(m - 1)) - ellipticF(pi/4, m)/(2*m)
- ellipticE(pi/4, m)/(2*m*(m - 1))
```

Here, `ellipticE` represents the incomplete elliptic integral of the second kind.

Plot the incomplete elliptic integrals `ellipticF(phi, m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticK(m)`:

```
syms m
p1 = ezplot(ellipticF(pi/4, m))
hold on
p2 = ezplot(ellipticF(pi/3, m))
p3 = ezplot(ellipticK(m))

set(p1,'Color','red')
set(p2,'Color','green')

colormap([0 0 1])
title('Elliptic integrals of the first kind')
xlabel('m')
ylabel('ellipticF(m)')
grid
hold off
```



References

[1] Milne-Thomson, L. M. “Elliptic Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`ellipke` | `ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticK` | `ellipticPiellipticF` | `vpa`

ellipticK

Purpose

Complete elliptic integral of the first kind

Syntax

```
ellipticK(m)
```

Description

`ellipticK(m)` returns the complete elliptic integral of the first kind.

Tips

- `ellipticK` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticK` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If `m` is a vector or a matrix, then `ellipticK(m)` returns the complete elliptic integral of the first kind, evaluated for each element of `m`.

Input Arguments

m

Number, symbolic number, variable, expression, or function. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Complete Elliptic Integral of the First Kind

The complete elliptic integral of the first kind is defined as follows:

$$K(m) = F\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - m\sin^2\theta}}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the complete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticK(1/2), ellipticK(pi/4), ellipticK(1),
ellipticK(-5.5)]

s =
    1.8541    2.2253       Inf    0.9325
```

Compute the complete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticK` returns unresolved symbolic calls.

```
s = [ellipticK(sym(1/2)), ellipticK(sym(pi/4)),
ellipticK(sym(1)),  ellipticK(sym(-5.5))]

s =
[ ellipticK(1/2), ellipticK(pi/4), Inf, ellipticK(-11/2)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.854074677, 2.225253684, Inf, 0.9324665884]
```

Differentiate these expressions involving the complete elliptic integral of the first kind:

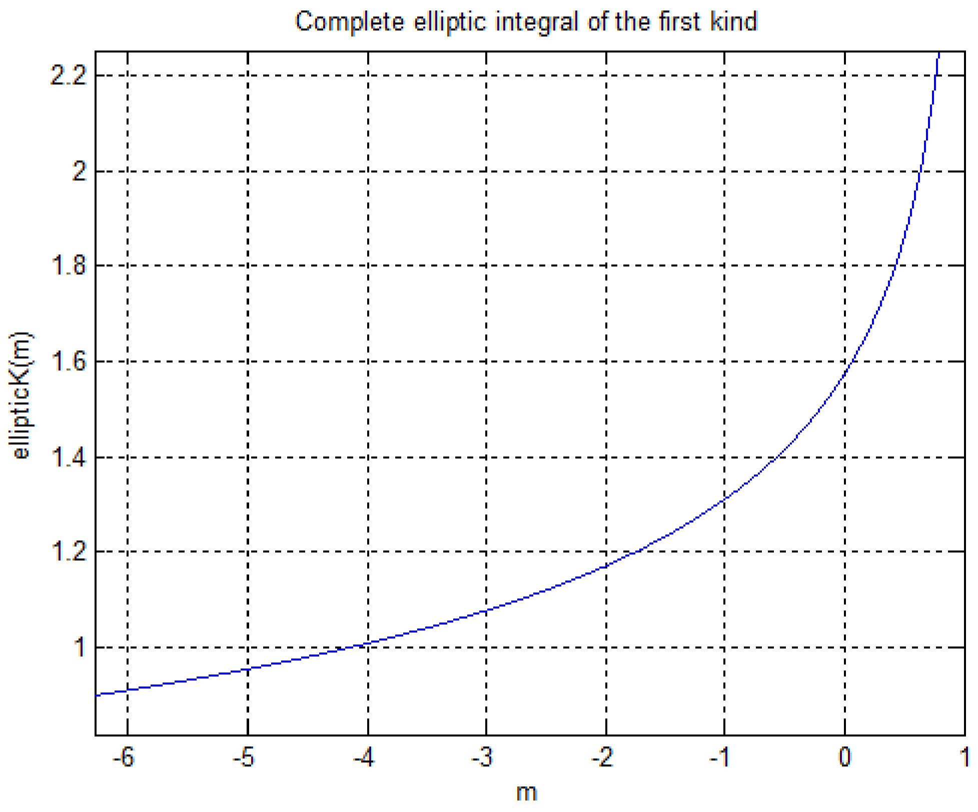
```
syms m
diff(ellipticK(m))
diff(ellipticK(m^2), m, 2)

ans =
- ellipticK(m)/(2*m) - ellipticE(m)/(2*m*(m - 1))

ans =
(2*ellipticE(m^2))/(m^2 - 1)^2 - (2*(ellipticE(m^2)/(2*m^2) -...
ellipticK(m^2)/(2*m^2)))/(m^2 - 1) + ellipticK(m^2)/m^2 +...
(ellipticK(m^2)/m + ellipticE(m^2)/(m*(m^2 - 1)))/m +...
```

```
ellipticE(m^2)/(m^2*(m^2 - 1))
```

Here, `ellipticE` represents the complete elliptic integral of the second kind.

Plot the complete elliptic integral of the first kind:

```
syms m
ezplot(ellipticK(m))

colormap([0 0 1])
title('Complete elliptic integral of the first kind')
xlabel('m')
ylabel('ellipticK(m)')
grid
hold off
```



Call `ellipticK` for this symbolic matrix. When the input argument is a matrix, `ellipticK` computes the complete elliptic integral of the first kind for each element.

```
ellipticK(sym([-2*pi -4; 0 1]))
```

```
ans =
[ ellipticK(-2*pi), ellipticK(-4)]
[             pi/2,           Inf]
```

References

[1] Milne-Thomson, L. M. "Elliptic Integrals." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

See Also

`ellipke` | `ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticPiellipticK` | `vpa`

Purpose

Elliptic integral of the third kind

Syntax

```
ellipticPi(n,m)
ellipticPi(n,phi,m)
```

Description

`ellipticPi(n,m)` returns the complete elliptic integral of the third kind.

`ellipticPi(n,phi,m)` returns the incomplete elliptic integral of the third kind.

Tips

- `ellipticPi` returns floating-point results for numeric arguments that are not symbolic objects.
- For most symbolic (exact) numbers, `ellipticPi` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- If one input argument is a vector or a matrix, and the other two arguments are scalars, then `ellipticPi` expands the scalars into vectors or matrices of the same size as the non-scalar argument, with all elements equal to the corresponding scalar.
- `ellipticPi(n, pi/2, m) = ellipticPi(n, m)`.

Input Arguments

n

Number, symbolic number, variable, expression, or function specifying the characteristic. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

m

Number, symbolic number, variable, expression, or function specifying the parameter. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

phi

Number, symbolic number, variable, expression, or function specifying the amplitude. This argument also can be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Incomplete Elliptic Integral of the Third Kind

The incomplete elliptic integral of the third kind is defined as follows:

$$\Pi(n;\varphi\,|\,m)=\int_0^{\varphi}\frac{1}{\left(1-n\sin^2\theta\right)\sqrt{1-m\sin^2\theta}}\,d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Complete Elliptic Integral of the Third Kind

The complete elliptic integral of the third kind is defined as follows:

$$\Pi(n,m)=\Pi\left(n;\frac{\pi}{2}\,|\,m\right)=\int_0^{\pi/2}\frac{1}{\left(1-n\sin^2\theta\right)\sqrt{1-m\sin^2\theta}}\,d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

Examples

Compute the incomplete elliptic integrals of the third kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticPi(-2.3, pi/4, 0), ellipticPi(1/3,
pi/3, 1/2),...
ellipticPi(-1, 0, 1),  ellipticPi(2, pi/6, 2)]
```

```
s =
    0.5877    1.2850         0    0.7507
```

Compute the incomplete elliptic integrals of the third kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticPi` returns unresolved symbolic calls.

```
s = [ellipticPi(-2.3, sym(pi/4), 0), ellipticPi(sym(1/3),
pi/3, 1/2),...
ellipticPi(-1, sym(0), 1),  ellipticPi(2, pi/6, sym(2))]
```

```
s =
[ ellipticPi(-23/10, pi/4, 0), ellipticPi(1/3, pi/3, 1/2),...
0, (2^(1/2)*3^(1/2))/2 - ellipticE(pi/6, 2)]
```

Here, `ellipticE` represents the incomplete elliptic integral of the second kind.

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)
```

```
ans =
[ 0.5876852228, 1.285032276, 0, 0.7507322117]
```

Differentiate these expressions involving the complete elliptic integral of the third kind:

```
syms n m
diff(ellipticPi(n, m), n)
diff(ellipticPi(n, m), m)
```

```
ans =
ellipticK(m)/(2*n*(n - 1)) + ellipticE(m)/(2*(m - n)*(n - 1)) -...
(ellipticPi(n, m)*(- n^2 + m))/(2*n*(m - n)*(n - 1))

ans =
- ellipticPi(n, m)/(2*(m - n)) - ellipticE(m)/(2*(m
- n)*(m - 1))
```

Here, `ellipticK` and `ellipticE` represent the complete elliptic integrals of the first and second kinds.

Call `ellipticPi` for the scalar and the matrix. When one input argument is a matrix, `ellipticPi` expands the scalar argument to a matrix of the same size with all its elements equal to the scalar.

```
ellipticPi(sym(0), sym([1/3 1; 1/2 0]))
```

```
ans =
[ ellipticK(1/3),  Inf]
[ ellipticK(1/2), pi/2]
```

Here, `ellipticK` represents the complete elliptic integral of the first kind.

References

[1] Milne-Thomson, L. M. "Elliptic Integrals." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`ellipke` | `ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticKellipticPi` | `vpa`

Purpose

Define equation

Note In previous releases, `eq` evaluated equations and returned logical `1` or `0`. Now it returns unevaluated equations letting you create equations that you can pass to `solve`, `assume`, and other functions. To obtain the same results as in previous releases, wrap equations in `logical` or `isAlways`. For example, use `logical(A == B)`.

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` creates a symbolic equation.

`eq(A,B)` is equivalent to `A == B`.

Tips

- If `A` and `B` are both numbers, then `A == B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A == B` returns a symbolic equation. You can use that equation as an argument for such functions as `solve`, `assume`, `ezplot`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A == B` returns an array of equations `A(i,j,...)==B(i,j,...)`
- If one input is scalar and the other an array, then == expands the scalar into an array of the same dimensions as the input array. In other words, if `A` is a variable (for example, `x`), and `B` is an *m*-by-*n* matrix, then `A` is expanded into *m*-by-*n* matrix of elements, each set to `x`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

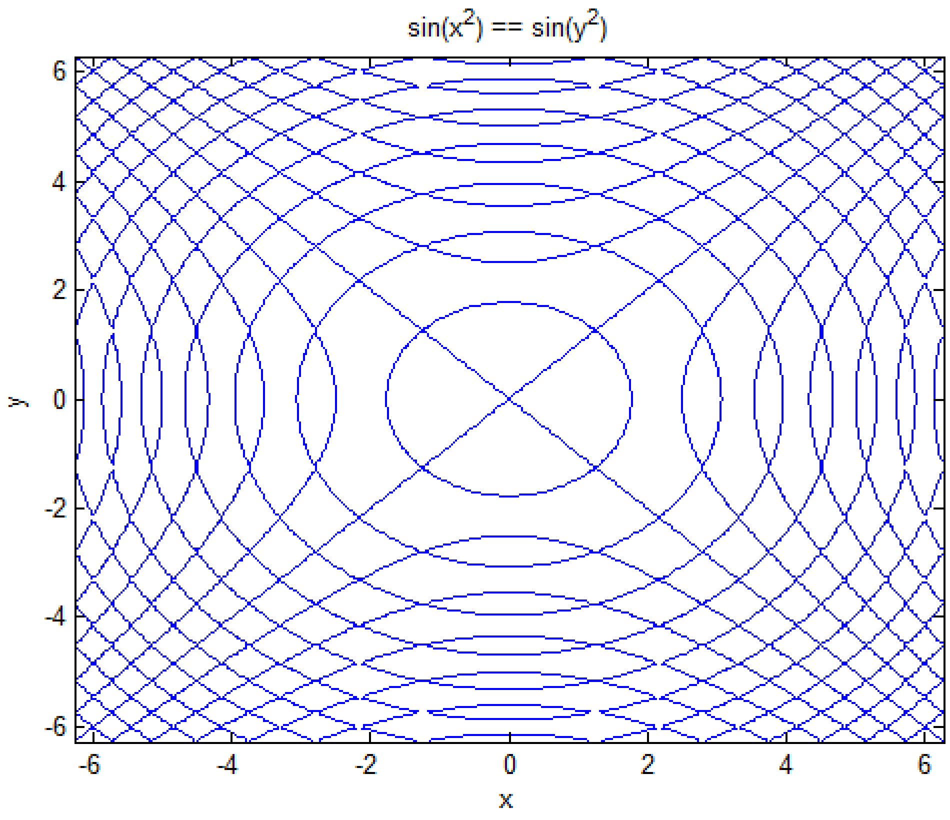
Solve this trigonometric equation. To define the equation, use the relational operator ==.

```
syms x
solve(sin(x) == cos(x), x)
```

```
ans =
pi/4
```

Plot this trigonometric equation. To define the equation, use the relational operator ==.

```
syms x y
ezplot(sin(x^2) == sin(y^2))
```



Check the equality of two symbolic matrices. Because the elements of both matrices are numbers, == returns logical 1s and 0s:

```
A = sym(hilb(3));
B = sym([1, 1/2, 5; 1/2, 2, 1/4; 1/3, 1/8, 1/5]);
A == B

ans =
     1     1     0
     1     0     1
     1     0     1
```

If you use == to compare a matrix and a scalar, then == expands the scalar into a matrix of the same dimensions as the input matrix:

```
A = sym(hilb(3));
B = sym(1/2);
A == B

ans =
     0     1     0
     1     0     0
     0     0     0
```

If the input arguments are symbolic variables or expression, == does not return logical `1`s and `0`s. Instead, it creates equations:

```
syms x
x + 1 == x + 1
sin(x)/cos(x) == tan(x)

ans =
x + 1 == x + 1

ans =
sin(x)/cos(x) == tan(x)
```

To test the equality of two symbolic expressions, use `logical` or `isAlways`. Use `logical` when expressions on both sides of the equation do not require simplification or transformation:

```
logical(x + 1 == x + 1)

ans =
     1
```

Use `isAlways` when expressions need to be simplified or transformed or when you use assumptions on variables:

```
isAlways(sin(x)/cos(x) == tan(x))


ans =
     1
```

See Also

`ge` | `gt` | `isAlways` | `le` | `logical` | `lt` | `ne` | `solve`

Concepts

- “Solve Equations” on page 1-28
- “Set Assumptions” on page 1-35

equationsToMatrix

Purpose

Convert set of linear equations to matrix form

Syntax

```
[A,b] = equationsToMatrix(eqns,vars)
[A,b] = equationsToMatrix(eqns)
A = equationsToMatrix(eqns,vars)
A = equationsToMatrix(eqns)
```

Description

`[A,b] = equationsToMatrix(eqns,vars)` converts `eqns` to the matrix form. Here `eqns` must be linear equations in `vars`.

`[A,b] = equationsToMatrix(eqns)` converts `eqns` to the matrix form. Here `eqns` must be a linear system of equations in all variables that `symvar` finds in these equations.

`A = equationsToMatrix(eqns,vars)` converts `eqns` to the matrix form and returns only the coefficient matrix. Here `eqns` must be linear equations in `vars`.

`A = equationsToMatrix(eqns)` converts `eqns` to the matrix form and returns only the coefficient matrix. Here `eqns` must be a linear system of equations in all variables that `symvar` finds in these equations.

Tips

- If you specify equations and variables all together, without dividing them into two vectors, specify all equations first, and then specify variables. If input arguments are not vectors, `equationsToMatrix` searches for variables starting from the last input argument. When it finds the first argument that is not a single variable, it assumes that all remaining arguments are equations, and therefore stops looking for variables.

Input Arguments

eqns

Vector of equations or equations separated by commas. Each equation is either a symbolic equation defined by the relation operator == or a symbolic expression. If you specify a symbolic expression (without the right side), `equationsToMatrix` assumes that the right side is 0.

Equations must be linear in terms of `vars`.

vars

Independent variables of `eqns`. You can specify `vars` as a vector. Alternatively, you can list variables separating them by commas.

> **Default:** Variables determined by `symvar`

Output Arguments

A

Coefficient matrix of the system of linear equations.

b

Vector containing the right sides of equations.

Definitions

Matrix Representation of a System of Linear Equations

A system of linear equations

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ &\cdots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \end{aligned}$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

and $\vec{b}$ is the vector containing the right sides of equations:

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Examples

Convert this system of linear equations to the matrix form. To get the coefficient matrix and the vector of the right sides of equations, assign the result to a vector of two output arguments:

```
syms x y z;
[A, b] = equationsToMatrix([x + y - 2*z == 0, x + y +
z == 1, 2*y - z + 5 == 0], [x, y, z])

A =
[ 1, 1, -2]
[ 1, 1,  1]
[ 0, 2, -1]

b =
  0
  1
 -5
```

Convert this system of linear equations to the matrix form. Assigning the result of the `equationsToMatrix` call to a single output argument, you get the coefficient matrix. In this case, `equationsToMatrix` does not return the vector containing the right sides of equations:

```
syms x y z;
A = equationsToMatrix([x + y - 2*z == 0, x + y + z ==
1, 2*y - z + 5 == 0], [x, y, z])

A =
[ 1, 1, -2]
[ 1, 1,  1]
[ 0, 2, -1]
```

Convert this linear system of equations to the matrix form without specifying independent variables. The toolbox uses `symvar` to identify variables:

```
syms s t;
[A, b] = equationsToMatrix([s - 2*t + 1 ==
0, 3*s - t == 10])

A =
[ 1, -2]
[ 3, -1]

b =
 -1
 10
```

If the system is only linear in some variables, specify those variables explicitly:

```
syms a s t;
[A, b] = equationsToMatrix([s - 2*t + a == 0, 3*s
- a*t == 10], [t, s])

A =
[ -2, 1]
[ -a, 3]

b =
 -a
 10
```

You also can specify equations and variables all together, without using vectors and simply separating each equation or variable by a comma. Specify all equations first, and then specify variables:

```
syms x y;
[A, b] = equationsToMatrix(x + y == 1, x - y + 1, x, y)

A =
[ 1,  1]
[ 1, -1]
```

```
b =
  1
 -1
```

Now change the order of the input arguments as follows. `equationsToMatrix` finds the variable `y`, then it finds the expression `x     y + 1`. After that, it assumes that all remaining arguments are equations, and stops looking for variables. Thus, `equationsToMatrix` finds the variable `y` and the system of equations `x + y = 1, x = 0, x - y + 1 = 0`:

```
[A, b] = equationsToMatrix(x + y == 1, x, x - y + 1, y)
```

```
A =
  1
  0
 -1

b =
   1 - x
      -x
 - x - 1
```

If you try to convert a nonlinear system of equations, `equationsToMatrix` throws an error:

```
syms x y;
[A, b] = equationsToMatrix(x^2 + y^2 == 1, x - y + 1, x, y)
```

```
Error using symengine (line 56)
Cannot convert to matrix form because
the system does not seem to be linear.
```

See Also

`linsolve` | `odeToVectorField` | `solve` | `symvar`

Related Examples

- "Solve a System of Differential Equations" on page 2-92

Purpose

Error function

Syntax

```
erf(x)
erf(A)
```

Description

`erf(x)` computes the error function of `x`.

`erf(A)` computes the error function of each element of `A`.

Tips

- Calling `erf` for a number that is not a symbolic object invokes the MATLAB `erf` function. This function accepts real arguments only. If you want to compute the error function for a complex number, use `sym` to convert that number to a symbolic object, and then call `erf` for that symbolic object.

Input Arguments

x

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Error Function

The following integral defines the error function:

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

Examples

Compute the error function for these numbers. Because these numbers are not symbolic objects, you get the floating-point results:

```
[erf(1/2), erf(1.41), erf(sqrt(2))]
```

```
ans =
    0.5205    0.9539    0.9545
```

Compute the error function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `erf` returns unresolved symbolic calls:

```
[erf(sym(1/2)), erf(sym(1.41)), erf(sqrt(sym(2)))]

ans =
[ erf(1/2), erf(141/100), erf(2^(1/2))]
```

Compute the error function for $x = 0$, $x = \infty$, and $x = -\infty$. Use `sym` to convert `0` and infinities to symbolic objects. The error function has special values for these parameters:

```
[erf(sym(0)), erf(sym(inf)), erf(sym(-inf))]

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

Compute the error function for complex infinities. Use `sym` to convert complex infinities to symbolic objects:

```
[erf(sym(i*inf)), erf(sym(-i*inf))]

ans =
[ Inf*i, -Inf*i]
```

Compute the error function for `x` and `sin(x) + x*exp(x)`. For most symbolic variables and expressions, `erf` returns unresolved symbolic calls:

```
syms x
f = sin(x) + x*exp(x);
erf(x)
erf(f)
```

```
ans =
erf(x)

ans =
erf(sin(x) + x*exp(x))
```

Now compute the derivatives of these expressions:

```
diff(erf(x), x, 2)
diff(erf(f), x)
```

```
ans =
-(4*x*exp(-x^2))/pi^(1/2)

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

Compute the error function for elements of matrix `M` and vector `V`:

```
M =sym([0 inf; 1/3 -inf]);
V = sym([1; -i*inf]);
erf(M)
erf(V)
```

```
ans =
[        0,  1]
[ erf(1/3), -1]

ans =
 erf(1)
 -Inf*i
```

Algorithms

The toolbox can simplify expressions that contain error functions and their inverses. For real values `x`, the toolbox applies these simplification rules:

- `erfinv(erf(x)) = erfinv(1 - erfc(x)) = erfcinv(1 - erf(x)) = erfcinv(erfc(x)) = x`
- `erfinv(-erf(x)) = erfinv(erfc(x) - 1) = erfcinv(1 + erf(x)) = erfcinv(2 - erfc(x)) = -x`

For any value `x`, the system applies these simplification rules:

- `erfcinv(x) = erfinv(1 - x)`
- `erfinv(-x) = -erfinv(x)`
- `erfcinv(2 - x) = -erfcinv(x)`
- `erf(erfinv(x)) = erfc(erfcinv(x)) = x`
- `erf(erfcinv(x)) = erfc(erfinv(x)) = 1 - x`

References

Gautschi, W. "Error Function and Fresnel Integrals." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`erfc` | `erfcinv` | `erfi` | `erfinv`

How To

- "Special Functions of Applied Mathematics" on page 2-142

Purpose

Complementary error function

Syntax

```
erfc(x)
erfc(A)
```

Description

`erfc(x)` computes the complementary error function of `x`.

`erfc(A)` computes the complementary error function of each element of `A`.

Tips

- Calling `erfc` for a number that is not a symbolic object invokes the MATLAB `erfc` function. This function accepts real arguments only. If you want to compute the complementary error function for a complex number, use `sym` to convert that number to a symbolic object, and then call `erfc` for that symbolic object.

Input Arguments

x

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Complementary Error Function

The following integral defines the complementary error function:

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

Here `erf(x)` is the error function.

Examples

Compute the complementary error function for these numbers. Because these numbers are not symbolic objects, you get the floating-point results:

```
[erfc(1/2), erfc(1.41), erfc(sqrt(2))]
```

```
ans =
    0.4795    0.0461    0.0455
```

Compute the complementary error function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfc` returns unresolved symbolic calls:

```
[erfc(sym(1/2)), erfc(sym(1.41))]

ans =
[ erfc(1/2), erfc(141/100)]
```

Compute the complementary error function for $x = 0$, $x = \infty$, and $x = -\infty$. The complementary error function has special values for these parameters:

```
[erfc(0), erfc(inf), erfc(-inf)]

ans =
     1     0     2
```

Compute the complementary error function for complex infinities. Use `sym` to convert complex infinities to symbolic objects:

```
[erfc(sym(i*inf)), erfc(sym(-i*inf))]

[ 1 - Inf*i, Inf*i + 1]
```

Compute the complementary error function for `x` and `sin(x) + x*exp(x)`. For most symbolic variables and expressions, `erfc` returns unresolved symbolic calls:

```
syms x
f = sin(x) + x*exp(x);
erfc(x)
erfc(f)

ans =
erfc(x)

ans =
erfc(sin(x) + x*exp(x))
```

Now compute the derivatives of these expressions:

```
diff(erfc(x), x, 2)
diff(erfc(f), x)

ans =
(4*x*exp(-x^2))/pi^(1/2)

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

Compute the complementary error function for elements of matrix M and vector V:

```
M = sym([0 inf; 1/3 -inf]);
V = sym([1; -i*inf]);
erfc(M)
erfc(V)

ans =
[         1, 0]
[ erfc(1/3), 2]

ans =
   erfc(1)
 Inf*i + 1
```

Algorithms

The toolbox can simplify expressions that contain error functions and their inverses. For real values `x`, the toolbox applies these simplification rules:

- `erfinv(erf(x)) = erfinv(1 - erfc(x)) = erfcinv(1 - erf(x)) = erfcinv(erfc(x)) = x`
- `erfinv(-erf(x)) = erfinv(erfc(x) - 1) = erfcinv(1 + erf(x)) = erfcinv(2 - erfc(x)) = -x`

For any value `x`, the system applies these simplification rules:

- `erfcinv(x) = erfinv(1 - x)`
- `erfinv(-x) = -erfinv(x)`
- `erfcinv(2 - x) = -erfcinv(x)`
- `erf(erfinv(x)) = erfc(erfcinv(x)) = x`
- `erf(erfcinv(x)) = erfc(erfinv(x)) = 1 - x`

References

Gautschi, W. "Error Function and Fresnel Integrals." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`erf` | `erfcinv` | `erfi` | `erfinv`

How To

- "Special Functions of Applied Mathematics" on page 2-142

Purpose

Inverse complementary error function

Syntax

```
erfcinv(x)
erfcinv(A)
```

Description

`erfcinv(x)` computes the inverse complementary error function of `x`.

`erfcinv(A)` computes the inverse complementary error function of each element of `A`.

Tips

- Calling `erfcinv` for a number that is not a symbolic object invokes the MATLAB `erfcinv` function. This function accepts real arguments only. If you want to compute the inverse complementary error function for a complex number, use `sym` to convert that number to a symbolic object, and then call `erfcinv` for that symbolic object.
- If $x < 0$ or $x > 2$, the MATLAB `erfcinv` function returns `NaN`. The symbolic `erfcinv` function returns unresolved symbolic calls for such numbers. To call the symbolic `erfcinv` function, convert its argument to a symbolic object using `sym`.

Input Arguments

x

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Inverse Complementary Error Function

The inverse complementary error function is defined as erfc^{-1}(x), such that erfc(erfc^{-1}(x)) = x. Here

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

is the complementary error function.

Examples

Compute the inverse complementary error function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[erfcinv(1/2), erfcinv(1.33), erfcinv(3/2),
erfcinv(-1), erfcinv(15)]

ans =
    0.4769   -0.3013   -0.4769       NaN       NaN
```

Compute the inverse complementary error function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfcinv` returns unresolved symbolic calls:

```
[erfcinv(sym(1/2)), erfcinv(sym(1.33)), erfcinv(sym(-2))]

ans =
[ -erfcinv(3/2), erfcinv(133/100), -erfcinv(4)]
```

Compute the inverse complementary error function for $x = 0$, $x = 1$, and $x = 2$. The inverse complementary error function has special values for these parameters:

```
[erfcinv(0), erfcinv(1), erfcinv(2)]

ans =
   Inf     0  -Inf
```

Compute the inverse complementary error function for complex numbers. Use `sym` to convert complex numbers to symbolic objects:

```
[erfcinv(sym(2 + 3*i)), erfcinv(sym(1 - i))]

ans =
[ erfcinv(2 + 3*i), -erfcinv(1 + i)]
```

Compute the inverse complementary error function for `x` and `sin(x) + x*exp(x)`. For most symbolic variables and expressions, `erfcinv` returns unresolved symbolic calls:

```
syms x
f = sin(x) + x*exp(x);
erfcinv(x)
erfcinv(f)

ans =
erfcinv(x)

ans =
erfcinv(sin(x) + x*exp(x))
```

Now compute the derivatives of these expressions:

```
diff(erfcinv(x), x, 2)
diff(erfcinv(f), x)

ans =
(pi*exp(2*erfcinv(x)^2)*erfcinv(x))/2

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

Compute the inverse complementary error function for elements of matrix `M` and vector `V`:

```
M = sym([0 1 + i; 1/3 1]);
V = sym([2; inf]);
erfcinv(M)
erfcinv(V)
```

```
ans =
[          Inf, erfcinv(1 + i)]
[ -erfcinv(5/3),             0]

ans =
         -Inf
 erfcinv(Inf)
```

Algorithms

The toolbox can simplify expressions that contain error functions and their inverses. For real values `x`, the toolbox applies these simplification rules:

- `erfinv(erf(x)) = erfinv(1 - erfc(x)) = erfcinv(1 - erf(x)) = erfcinv(erfc(x)) = x`
- `erfinv(-erf(x)) = erfinv(erfc(x) - 1) = erfcinv(1 + erf(x)) = erfcinv(2 - erfc(x)) = -x`

For any value `x`, the toolbox applies these simplification rules:

- `erfcinv(x) = erfinv(1 - x)`
- `erfinv(-x) = -erfinv(x)`
- `erfcinv(2 - x) = -erfcinv(x)`
- `erf(erfinv(x)) = erfc(erfcinv(x)) = x`
- `erf(erfcinv(x)) = erfc(erfinv(x)) = 1 - x`

References

Gautschi, W. “Error Function and Fresnel Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`erf` | `erfc` | `erfi` | `erfinv`

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Imaginary error function

Syntax

```
erfi(x)
```

Description

`erfi(x)` returns the imaginary error function of `x`. If `x` is a vector or a matrix, `erfi(x)` returns the imaginary error function of each element of `x`.

Tips

- `erfi` returns special values for these parameters:
  - `erfi(0) = 0`
  - `erfi(inf) = inf`
  - `erfi(-inf) = -inf`
  - `erfi(i*inf) = i`
  - `erfi(-i*inf) = -i`

Input Arguments

x - Input
floating-point number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input specified as a floating-point or symbolic number, variable, expression, function, vector, or matrix.

Definitions

Imaginary Error Function

The imaginary error function is defined as:

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

Examples

Imaginary Error Function for Floating-Point and Symbolic Numbers

Compute the imaginary error function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [erfi(1/2), erfi(1.41), erfi(sqrt(2))]

s =
    0.6150    3.7382    3.7731
```

Compute the imaginary error function for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfi` returns unresolved symbolic calls.

```
s = [erfi(sym(1/2)), erfi(sym(1.41)), erfi(sqrt(sym(2)))]

s =
[ erfi(1/2), erfi(141/100), erfi(2^(1/2))]
```

Use `vpa` to approximate this result with the 10-digit accuracy:

```
vpa(s, 10)

ans =
[ 0.6149520947, 3.738199581, 3.773122512]
```

Special Values of Imaginary Error Function

Compute the imaginary error function for $x = 0$, $x = \infty$, and $x = -\infty$. Use `sym` to convert `0` and infinities to symbolic objects. The imaginary error function has special values for these parameters:

```
[erfi(sym(0)), erfi(sym(inf)), erfi(sym(-inf))]

ans =
[ 0, Inf, -Inf]
```

Compute the imaginary error function for complex infinities. Use `sym` to convert complex infinities to symbolic objects:

```
[erfi(sym(i*inf)), erfi(sym(-i*inf))]

ans =
[ i, -i]
```

Imaginary Error Function for Variables and Expressions

Compute the imaginary error function for `x` and `sin(x) + x*exp(x)`. For most symbolic variables and expressions, `erfi` returns unresolved symbolic calls.

```
syms x
f = sin(x) + x*exp(x);
erfi(x)
erfi(f)

ans =
erfi(x)

ans =
erfi(sin(x) + x*exp(x))
```

Now, compute the derivatives of these expressions:

```
diff(erfi(x), x, 2)
diff(erfi(f), x)

ans =
(4*x*exp(x^2))/pi^(1/2)

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

Imaginary Error Function for Matrices and Vectors

Compute the imaginary error function for elements of matrix `M` and vector `V`:

```
M =sym([0 inf; 1/3 -inf]);
V = sym([1; -i*inf]);
erfi(M)
erfi(V)

ans =
[         0,  Inf]
[ erfi(1/3), -Inf]

ans =
 erfi(1)
      -i
```

See Also

`erf` | `erfc` | `erfcinv` | `erfinv` | `vpa`

erfinv

Purpose

Inverse error function

Syntax

```
erfinv(x)
erfinv(A)
```

Description

`erfinv(x)` computes the inverse error function of `x`.

`erfinv(A)` computes the inverse error function of each element of `A`.

Tips

- Calling `erfinv` for a number that is not a symbolic object invokes the MATLAB `erfinv` function. This function accepts real arguments only. If you want to compute the inverse error function for a complex number, use `sym` to convert that number to a symbolic object, and then call `erfinv` for that symbolic object.
- If $x < -1$ or $x > 1$, the MATLAB `erfinv` function returns `NaN`. The symbolic `erfinv` function returns unresolved symbolic calls for such numbers. To call the symbolic `erfinv` function, convert its argument to a symbolic object using `sym`.

Input Arguments

x

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Inverse Error Function

The inverse error function is defined as $\mathrm{erf}^{-1}(x)$, such that $\mathrm{erf}(\mathrm{erf}^{-1}(x)) = \mathrm{erf}^{-1}(\mathrm{erf}(x)) = x$. Here

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

is the error function.

Examples

Compute the inverse error function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[erfinv(1/2), erfinv(0.33), erfinv(-1/3),
erfinv(-2), erfinv(15)]

ans =
    0.4769    0.3013   -0.3046       NaN       NaN
```

Compute the inverse error function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfinv` returns unresolved symbolic calls:

```
[erfinv(sym(1/2)), erfinv(sym(0.33)), erfinv(sym(-2))]

ans =
[ erfinv(1/2), erfinv(33/100), -erfinv(2)]
```

Compute the inverse error function for $x = –1$, $x = 0$, and $x = 1$. The inverse error function has special values for these parameters:

```
[erfinv(-1), erfinv(0), erfinv(1)]

ans =
  -Inf     0   Inf
```

Compute the inverse error function for complex numbers. Use `sym` to convert complex numbers to symbolic objects:

```
[erfinv(sym(2 + 3*i)), erfinv(sym(1 - i))]

ans =
[ erfinv(2 + 3*i), erfinv(1 - i)]
```

Compute the inverse error function for `x` and `sin(x) + x*exp(x)`. For most symbolic variables and expressions, `erfinv` returns unresolved symbolic calls:

```
syms x
f = sin(x) + x*exp(x);
erfinv(x)
erfinv(f)

ans =
erfinv(x)

ans =
erfinv(sin(x) + x*exp(x))
```

Now compute the derivatives of these expressions:

```
diff(erfinv(x), x, 2)
diff(erfinv(f), x)

ans =
(pi*exp(2*erfinv(x)^2)*erfinv(x))/2

ans =
(pi^(1/2)*exp(erfinv(sin(x) +...
x*exp(x))^2)*(cos(x) + exp(x) + x*exp(x)))/2
```

Compute the inverse error function for elements of matrix `M` and vector `V`:

```
M = sym([0 1 + i; 1/3 1]);
V = sym([-1; inf]);
erfinv(M)
erfinv(V)

ans =
[            0, erfinv(1 + i)]
[ erfinv(1/3),          Inf]

ans =
        -Inf
 erfinv(Inf)
```

Algorithms

The toolbox can simplify expressions that contain error functions and their inverses. For real values `x`, the toolbox applies these simplification rules:

- `erfinv(erf(x)) = erfinv(1 - erfc(x)) = erfcinv(1 - erf(x)) = erfcinv(erfc(x)) = x`
- `erfinv(-erf(x)) = erfinv(erfc(x) - 1) = erfcinv(1 + erf(x)) = erfcinv(2 - erfc(x)) = -x`

For any value `x`, the toolbox applies these simplification rules:

- `erfcinv(x) = erfinv(1 - x)`
- `erfinv(-x) = -erfinv(x)`
- `erfcinv(2 - x) = -erfcinv(x)`
- `erf(erfinv(x)) = erfc(erfcinv(x)) = x`
- `erf(erfcinv(x)) = erfc(erfinv(x)) = 1 - x`

References

Gautschi, W. “Error Function and Fresnel Integrals.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`erf` | `erfc` | `erfcinv` | `erfi`

How To

- “Special Functions of Applied Mathematics” on page 2-142

evalin

Purpose

Evaluate MuPAD expressions without specifying their arguments

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. If `MuPAD_expression` throws an error in MuPAD, then this syntax throws an error in MATLAB.

`[result,status] = evalin(symengine,MuPAD_expression)` lets you catch errors thrown by MuPAD. This syntax 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.

Tips

- Results returned by `evalin` can differ from the results that you get using a MuPAD notebook directly. The reason is that `evalin` sets a lower level of evaluation to achieve better performance.

Input Arguments

MuPAD_expression

String containing a MuPAD expression.

Output Arguments

result

Symbolic object or string containing a MuPAD error message.

status

Integer indicating the error status. If `MuPAD_expression` executes without errors, the error status is 0.

Examples

Compute the discriminant of the following polynomial:

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

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

Try using `polylib::discrim` to compute the discriminant of the following nonpolynomial expression:

```
[result, status] =
evalin(symengine,'polylib::discrim(a*x^2+b*x+c*ln(x),x)')

result =
Error: An arithmetical expression is expected. [normal]

status =
     2
```

Alternatives

`feval` lets you evaluate MuPAD expressions with arguments. When using `feval`, you must explicitly specify the arguments of the MuPAD expression.

See Also

`feval | read | symengine`

Related Examples

- “Call Built-In MuPAD Functions from MATLAB Command Window” on page 3-32

Concepts

- “Evaluations in Symbolic Computations”
- “Level of Evaluation”

expand

Purpose

Symbolic expansion of polynomials and elementary functions

Syntax

```
expand(S)
expand(S,Name,Value)
```

Description

`expand(S)` expands the symbolic expression `S`. `expand` is often used with polynomials. It also expands trigonometric, exponential, and logarithmic functions.

`expand(S,Name,Value)` expands `S` using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

S

Symbolic expression or symbolic matrix.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'ArithmeticOnly'

If the value is `true`, expand the arithmetic part of an expression without expanding trigonometric, hyperbolic, logarithmic, and special functions. This option does not prevent expansion of powers and roots.

Default: `false`

'IgnoreAnalyticConstraints'

If the value is `true`, apply purely algebraic simplifications to an expression. With `IgnoreAnalyticConstraints`, `expand` can return simpler results for the expressions for which it would return more complicated results otherwise. Using `IgnoreAnalyticConstraints` also can lead to results that are not equivalent to the initial expression.

Default: `false`

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]
```

Expand this expression:

```
syms x
expand((sin(3*x) - 1)^2)
```

By default, `expand` works on all subexpressions including trigonometric subexpressions:

```
ans =
2*sin(x) + sin(x)^2 - 8*cos(x)^2*sin(x) - 8*cos(x)^2*sin(x)^2
+ 16*cos(x)^4*sin(x)^2 + 1
```

To prevent expansion of trigonometric, hyperbolic, and logarithmic subexpressions and subexpressions involving special functions, use `ArithmeticOnly`:

```
expand((sin(3*x) - 1)^2, 'ArithmeticOnly', true)
```

The result is the expression with expanded arithmetical parts:

```
ans =
sin(3*x)^2 - 2*sin(3*x) + 1
```

Expand this logarithm:

```
syms a b c
expand(log((a*b/c)^2))
```

By default, the `expand` function does not expand logarithms because expanding logarithms is not valid for generic complex values:

```
ans =
log((a^2*b^2)/c^2)
```

To apply the simplification rules that let the `expand` function expand logarithms, use `IgnoreAnalyticConstraints`:

```
expand(log((a*b/c)^2), 'IgnoreAnalyticConstraints', true)
```

The result is:

```
ans =
 2*log(a) + 2*log(b) - 2*log(c)
```

Algorithms

When you use `IgnoreAnalyticConstraints`, `expand` applies these rules:

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

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

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

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

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:

  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

See Also

`collect` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

How To

- “Simplifications” on page 2-33

Purpose

Exponential integral function

Syntax

```
expint(x)
expint(n,x)
```

Description

`expint(x)` returns the one-argument exponential integral function defined as follows:

$$\operatorname{expint}(x) = \int_x^\infty \frac{e^{-t}}{t}\,dt$$

`expint(n,x)` returns the two-argument exponential integral function defined as follows:

$$\operatorname{expint}(n,x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt$$

Tips

- `expint(x)` is uniquely defined for positive numbers. It is approximated for the rest of the complex plane.
- Calling `expint` for numbers that are not symbolic objects invokes the MATLAB `expint` function. This function accepts one argument only. To compute the two-argument exponential integral, use `sym` to convert the numbers to symbolic objects, and then call `expint` for those symbolic objects. You can approximate the results with floating-point numbers using `vpa`.
- The following values of the exponential integral differ from those returned by the MATLAB `expint` function: `expint(sym(Inf)) = 0`, `expint(-sym(Inf)) = -Inf`, `expint(sym(NaN)) = NaN`.
- For positive `x`, `expint(x) = -ei(-x)`. For negative `x`, `expint(x) = -pi*i - ei(-x)`.
- If one input argument is a scalar and the other one is a vector or a matrix, `expint(n,x)` expands the scalar into a vector or matrix of

the same size as the other argument with all elements equal to that scalar.

Input Arguments

x - Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input specified as a symbolic number, variable, expression, function, vector, or matrix.

n - Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input specified as a symbolic number, variable, expression, function, vector, or matrix. When you compute the two-argument exponential integral function, at least one argument must be a scalar.

Examples

One-Argument Exponential Integral for Floating-Point and Symbolic Numbers

Compute the exponential integrals for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [expint(1/3), expint(1), expint(-2)]
```

```
s =
   0.8289 + 0.0000i   0.2194 + 0.0000i  -4.9542 - 3.1416i
```

Compute the exponential integrals for the same numbers converted to symbolic objects. For positive values `x`, `expint(x)` returns `-ei(-x)`. For negative values `x`, it returns `-pi*i - ei(-x)`.

```
s = [expint(sym(1)/3), expint(sym(1)), expint(sym(-2))]
```

```
s =
[ -ei(-1/3), -ei(-1), - pi*i - ei(2)]
```

Use `vpa` to approximate this result with the 10-digit accuracy:

```
vpa(s, 10)

ans =
[ 0.8288877453, 0.2193839344, - 4.954234356
- 3.141592654*i]
```

Two-Argument Exponential Integral for Floating-Point and Symbolic Numbers

When computing two-argument exponential integrals, convert numbers to symbolic objects:

```
s = [expint(2, sym(1)/3), expint(sym(1), Inf),
expint(-1, sym(-2))]

s =
[ expint(2, 1/3), 0, -exp(2)/4]
```

Use `vpa` to approximate this result with the 25- digit accuracy:

```
vpa(s, 25)

ans =
[ 0.4402353954575937050522018, 0,
-1.847264024732662556807607]
```

Two-Argument Exponential Integral with a Nonpositive First Argument

Compute these two-argument exponential integrals. If `n` is a nonpositive integer, then `expint(n, x)` returns an explicit expression in the form `exp(-x)*p(1/x)`, where `p` is a polynomial of degree `1 - n`.

```
syms x
expint(0, x)
expint(-1, x)
expint(-2, x)

ans =
exp(-x)/x
```

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

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

Derivatives of the Exponential Integral

Compute the first, second, and third derivatives of the one-argument exponential integral:

```
syms x
diff(expint(x), x)
diff(expint(x), x, 2)
diff(expint(x), x, 3)

ans =
-exp(-x)/x

ans =
exp(-x)/x + exp(-x)/x^2

ans =
- exp(-x)/x - (2*exp(-x))/x^2 - (2*exp(-x))/x^3
```

Compute the first derivatives of the two-argument exponential integral:

```
syms n x
diff(expint(n, x), x)
diff(expint(n, x), n)

ans =
-expint(n - 1, x)

ans =
- hypergeom([1 - n, 1 - n], [2 - n, 2 - n], -x)/(n - 1)^2 -...
(pi*x^(n - 1)*(psi(n) - log(x) +
pi*cot(pi*n)))/(sin(pi*n)*gamma(n))
```

See Also

`ei` | `expintEi` | `vpa`

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`

ezcontour

Purpose

Contour plotter

Syntax

```
ezcontour(f)
ezcontour(f,domain)
ezcontour(...,n)
```

Description

`ezcontour(f)` plots the contour lines of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezcontour(f,domain)` plots $f(x,y)$ over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezcontour(u^2 - v^3,[0,1],[3,6])` plots the contour lines for u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

`ezcontour(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour` automatically adds a title and axis labels.

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

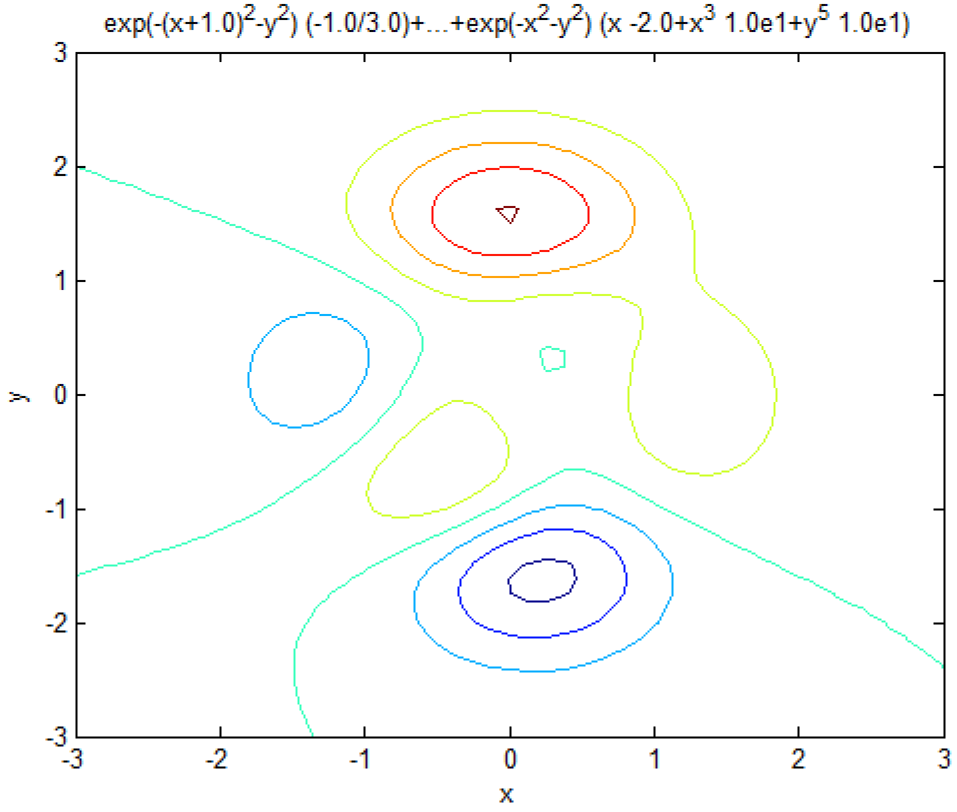
`ezcontour` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

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

For convenience, this expression is written on three lines.

Pass the `sym` `f` to `ezcontour` along with a domain ranging from `-3` to `3` and specify a computational grid of 49-by-49.

```
ezcontour(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB abbreviates the string.

See Also

`contour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc`

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*.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

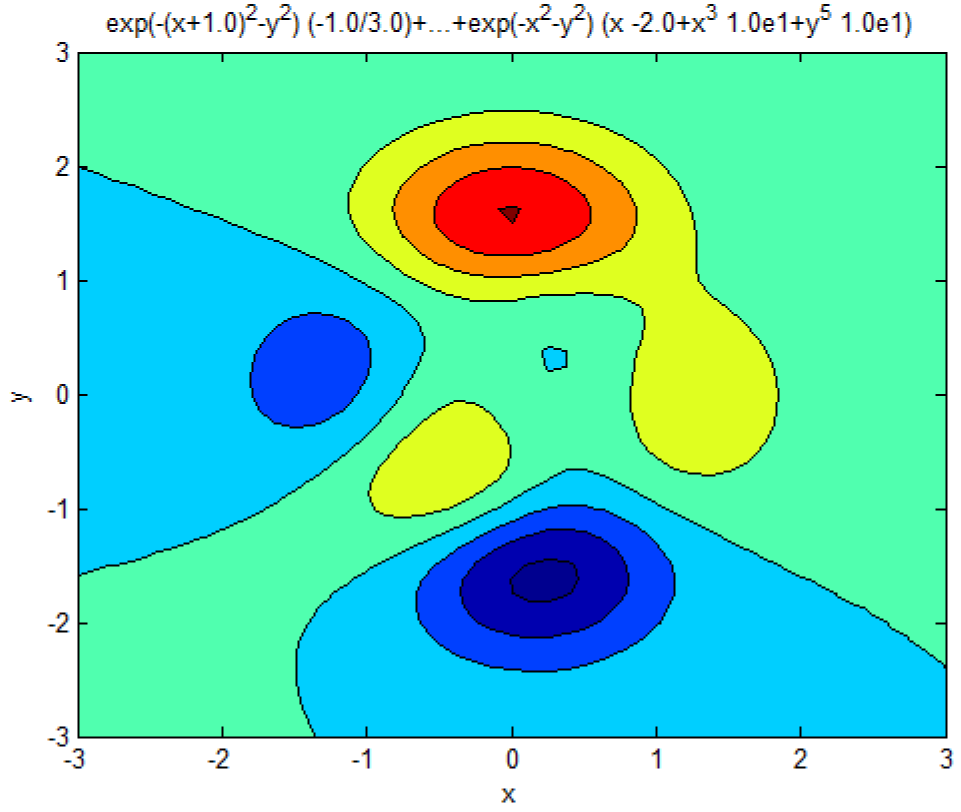
`ezcontourf` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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`

ezmesh

Purpose

3-D mesh plotter

Syntax

```
ezmesh(f)
ezmesh(f, domain)
ezmesh(x,y,z)
ezmesh(x,y,z,[smin,smax,tmin,tmax])
ezmesh(x,y,z,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
```

Description

`ezmesh(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmesh(f, domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezmesh(u^2 - v^3,[0,1],[3,6])` plots u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

`ezmesh(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmesh(x,y,z,[smin,smax,tmin,tmax])` or `ezmesh(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots f over a disk centered on the domain.

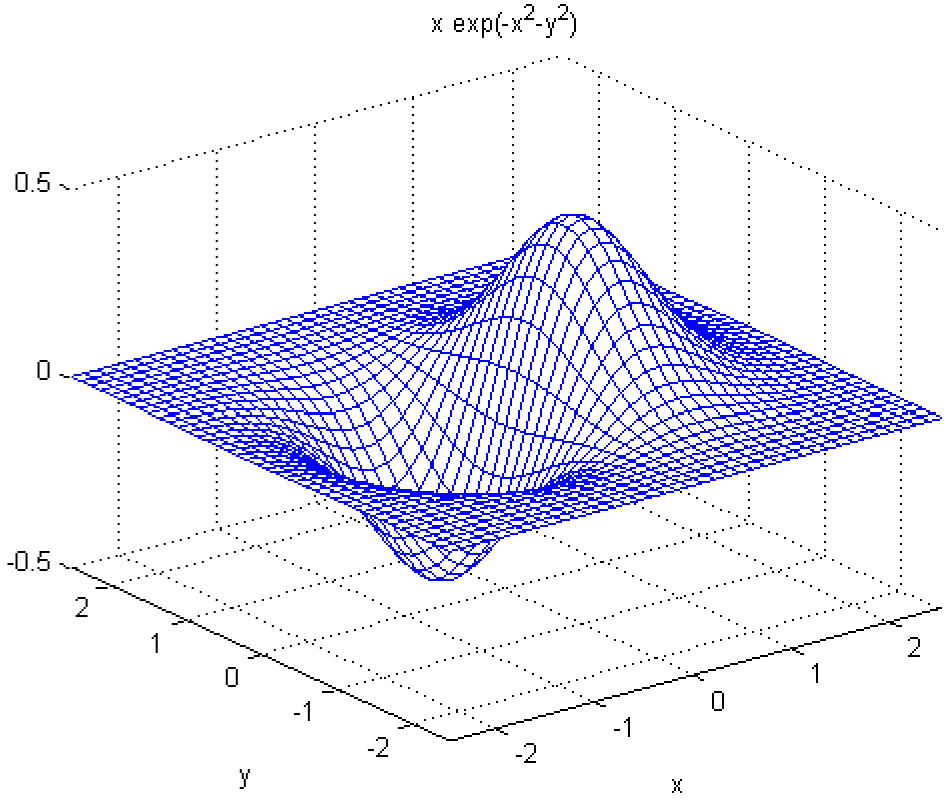
Examples

This example visualizes the function,

$$f(x, y) = xe^{-x^2 - y^2},$$

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color.

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



See Also ezcontour | ezcontourf | ezmeshc | ezplot | ezplot3 | ezpolar | ezsurf | ezsurfc | mesh

Purpose

Combined mesh and contour plotter

Syntax

```
ezmeshc(f)
ezmeshc(f,domain)
ezmeshc(x,y,z)
ezmeshc(x,y,z,[smin,smax,tmin,tmax])
ezmeshc(x,y,z,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
```

Description

`ezmeshc(f)` creates a graph of $f(x,y)$, where f is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmeshc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezmeshc(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezmeshc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmeshc(x,y,z,[smin,smax,tmin,tmax])` or `ezmeshc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots f over a disk centered on the domain.

Examples

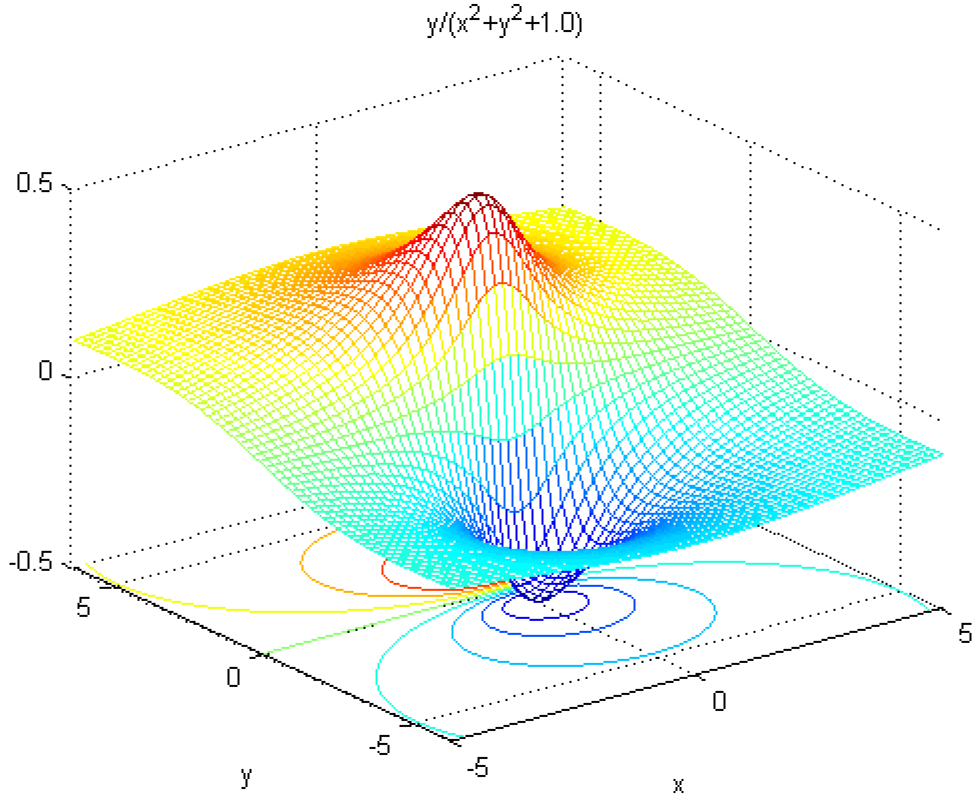
Create a mesh/contour graph of the expression,

$$f(x,y) = \frac{y}{1+x^2+y^2},$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$.

```
syms x y
ezmeshc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi])
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = –65 and elevation = 26).



See Also

`ezcontour` | `ezcontourf` | `ezmesh` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `meshc`

ezplot

Purpose

Plot symbolic expression, equation, or function

Syntax

```
ezplot(f)
ezplot(f,[min,max])
ezplot(f,[xmin,xmax,ymin,ymax])
ezplot(f,fign)
ezplot(x,y)
ezplot(x,y,[tmin,tmax])
ezplot(f,figure_handle)
```

Description

`ezplot(f)` plots a symbolic expression, equation, or function `f`. By default, `ezplot` plots a univariate expression or function over the range [–2*π* 2*π*] or over a subinterval of this range. If `f` is an equation or function of two variables, the default range for both variables is [–2*π* 2*π*] or over a subinterval of this range.

`ezplot(f,[min,max])` plots `f` over the specified range. If `f` is a univariate expression or function, then `[min,max]` specifies the range for that variable. This is the range along the abscissa (horizontal axis). If `f` is an equation or function of two variables, then `[min,max]` specifies the range for both variables, that is the ranges along both the abscissa and the ordinate.

`ezplot(f,[xmin,xmax,ymin,ymax])` plots `f` over the specified ranges along the abscissa and the ordinate. For this syntax, `f` needs two variables. If `f` is univariate, this syntax throws an error.

`ezplot(f,fign)` displays the plot in the plot window with the number `fign`. The title of each plot window contains the word Figure and the number, for example, **Figure 1**, **Figure 2**, and so on. If the plot window with the number `fign` is already opened, `ezplot` overwrites the content of that window with the new plot.

`ezplot(x,y)` plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default range $0 <= t <= 2\pi$ or over a subinterval of this range.

`ezplot(x,y,[tmin,tmax])` plots $x = x(t)$ and $y = y(t)$ over the specified range $tmin <= t <= tmax$.

`ezplot(f,figure_handle)` plots `f` in the plot window identified by the handle `figure_handle`.

Tips

- If you do not specify a plot range, `ezplot` uses the interval [–2π 2π] as a starting point. Then it can choose to display a part of the plot over a subinterval of [–2π 2π] where the plot has significant variation. Also, when selecting the plotting range, `ezplot` omits extreme values associated with singularities.
- `ezplot` open a plot window and displays a plot there. If any plot windows are already open, `ezplot` does not create a new window. Instead, it displays the new plot in the currently active window. (Typically, it is the window with the highest number.) To display the new plot in a new plot window or in an existing window other than that with highest number, use `fign`.
- If `f` is an equation or function of two variables, then the alphabetically first variable defines the abscissa (horizontal axis) and the other variable defines the ordinate (vertical axis). Thus, `ezplot(x^2 == a^2,[-3,3,-2,2])` creates the plot of the equation $x^2 = a^2$ with $-3 <= a <= 3$ along the horizontal axis, and $-2 <= x <= 2$ along the vertical axis.

Input Arguments

f

Symbolic expression, equation, or function.

[min,max]

Numbers specifying the plotting range. For a univariate expression or function, the plotting range applies to that variable. For an equation or function of two variables, the plotting range applies to both variables. In this case, the range is the same for the abscissa and the ordinate.

> **Default:** `[-2*pi,2*pi]` or its subinterval.

[xmin,xmax,ymin,ymax]

Numbers specifying the plotting range along the abscissa (first two numbers) and the ordinate (last two numbers).

> **Default:** `[-2*pi,2*pi,-2*pi,2*pi]` or its subinterval.

fign

Number of the figure window where you want to display a plot.

> **Default:** If no plot windows are open, then 1. If one plot window is open, then the number in the title of that window. If more than one plot window is open, then the highest number in the titles of open windows.

x,y

Symbolic expressions or functions defining a parametric curve $x = x(t)$ and $y = y(t)$.

[tmin,tmax]

Numbers specifying the plotting range for a parametric curve.

> **Default:** `[0,2*pi]` or its subinterval.

figure_handle

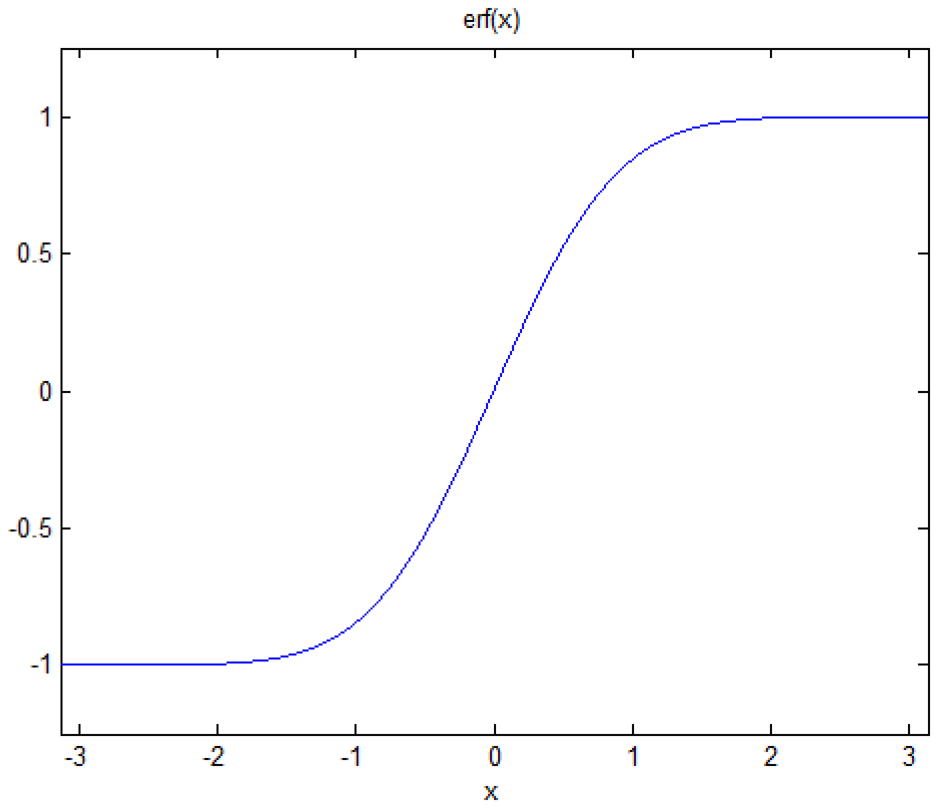
Figure handle specifying the plot window in which you create or modify a plot.

> **Default:** Current figure handle returned by `gcf`.

Examples

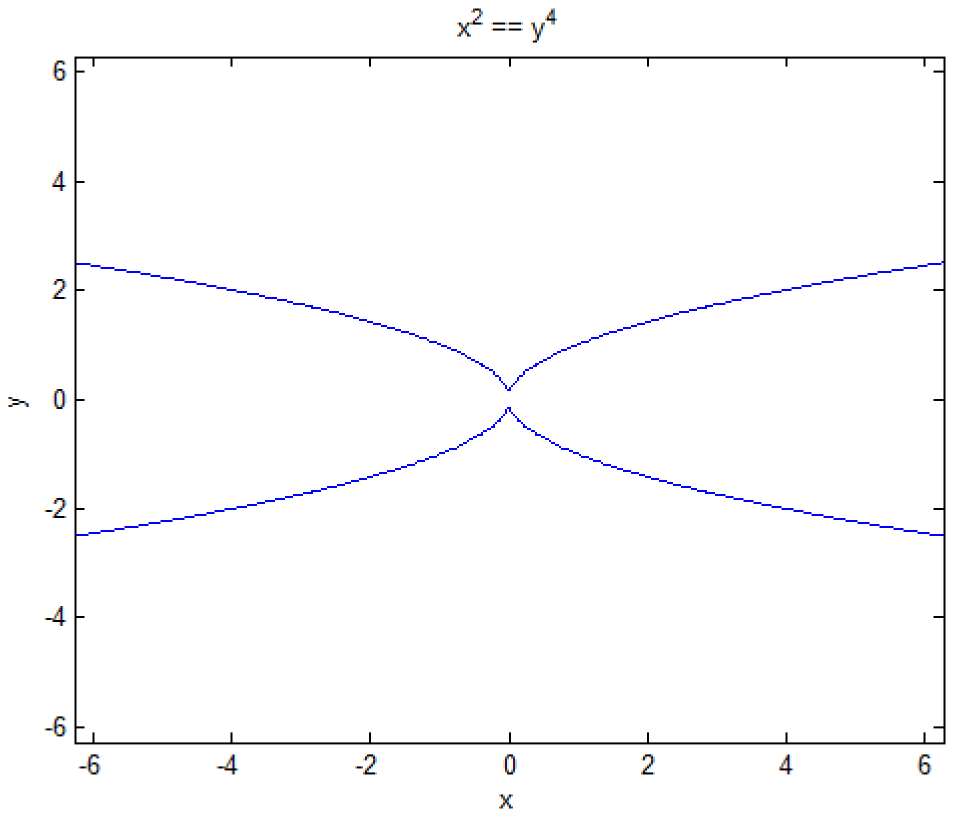
Plot the expression `erf(x)*sin(x)` over the range $[-\pi, \pi]$:

```
syms x
ezplot(erf(x), [-pi, pi])
```



Plot this equation over the default range:

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

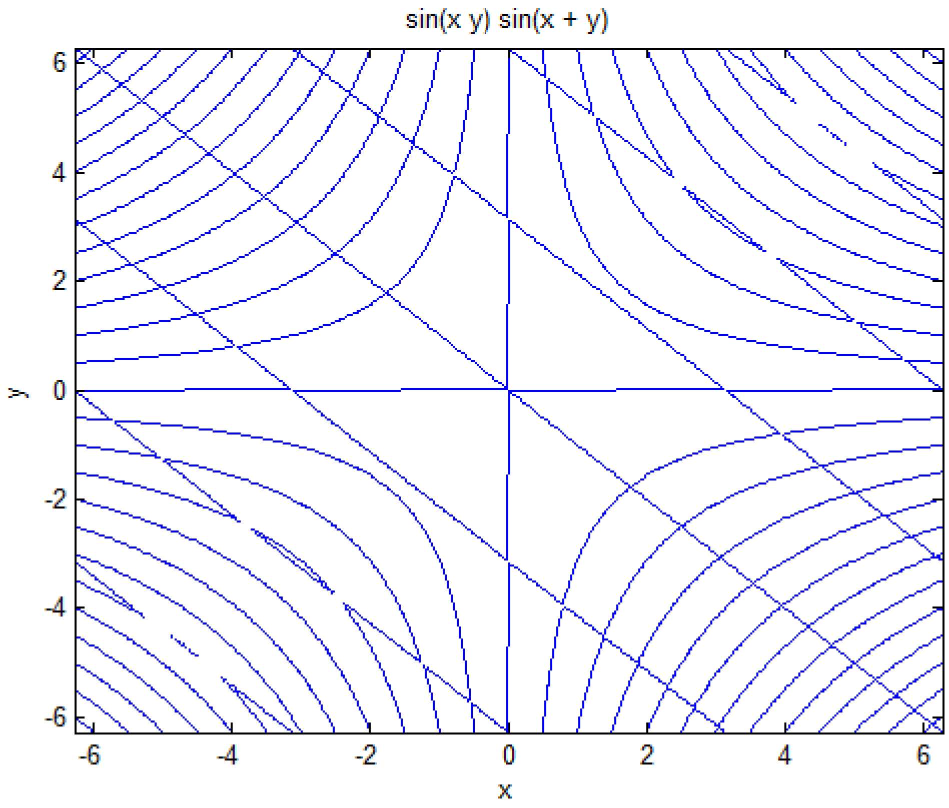


Create this symbolic function `f(x, y)`:

```
syms x y
f(x, y) = sin(x + y)*sin(x*y);
```

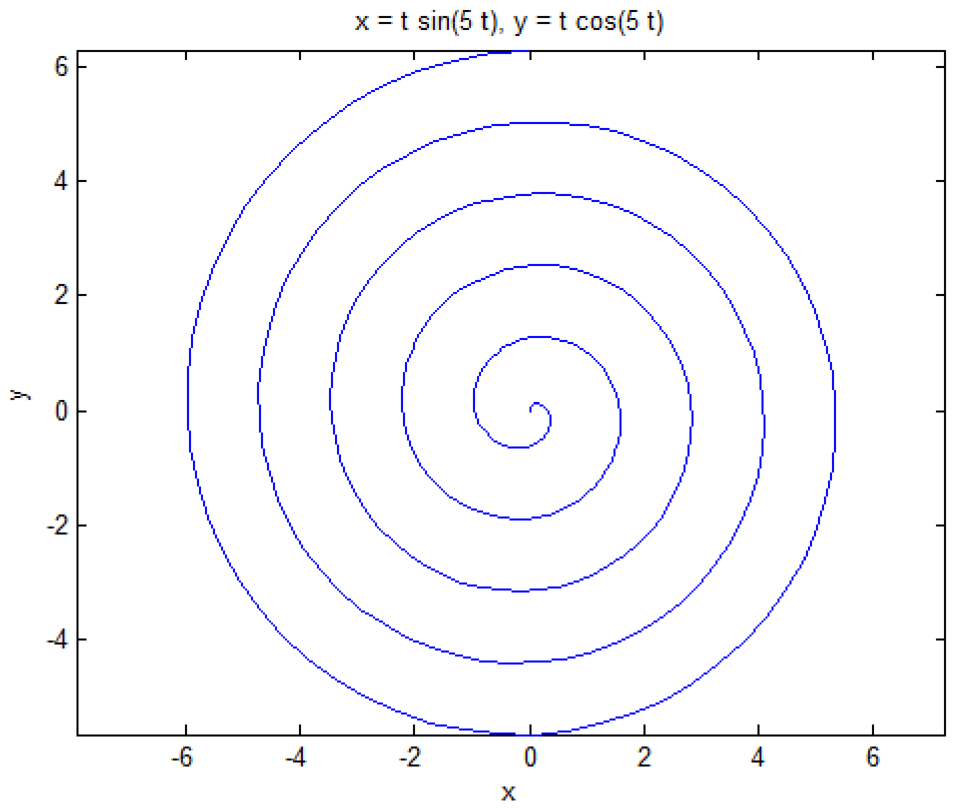
Plot this function over the default range:

```
ezplot(f)
```



Plot this parametric curve:

```
syms t
x = t*sin(5*t);
y = t*cos(5*t);
ezplot(x, y)
```



See Also

`ezcontour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `plot`

Concepts

- “Create Plots” on page 2-112

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 $0 < t < 2\pi$.

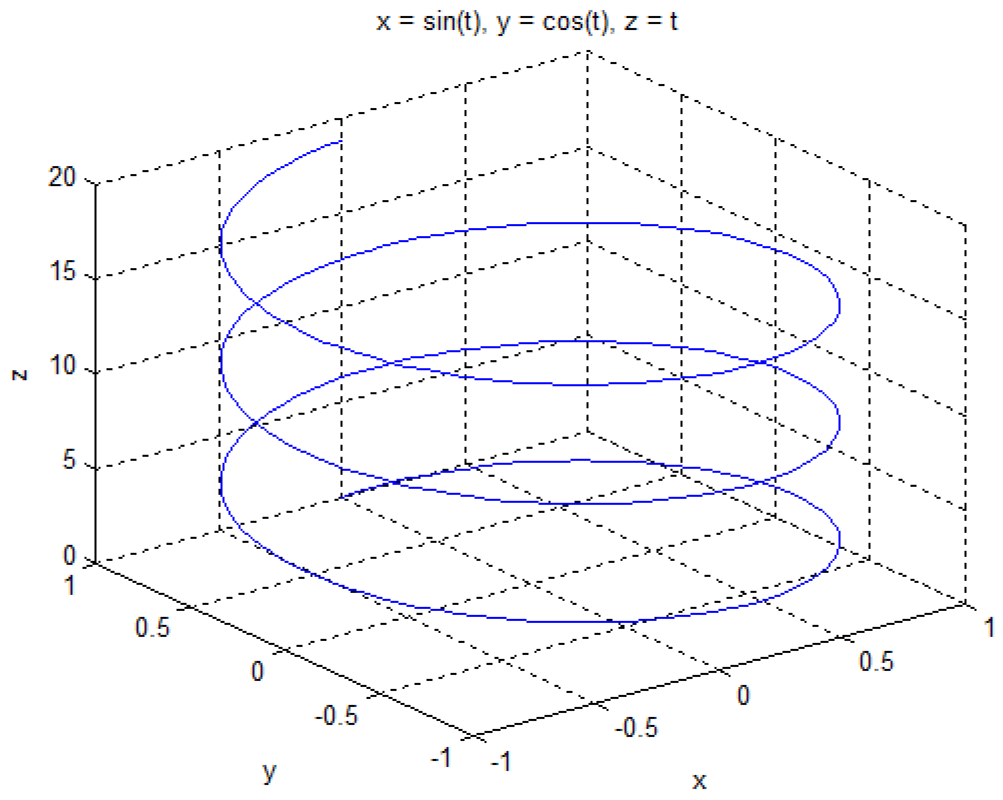
`ezplot3(x,y,z,[tmin,tmax])` plots the curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the domain $tmin < t < tmax$.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

Examples

Plot 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`.

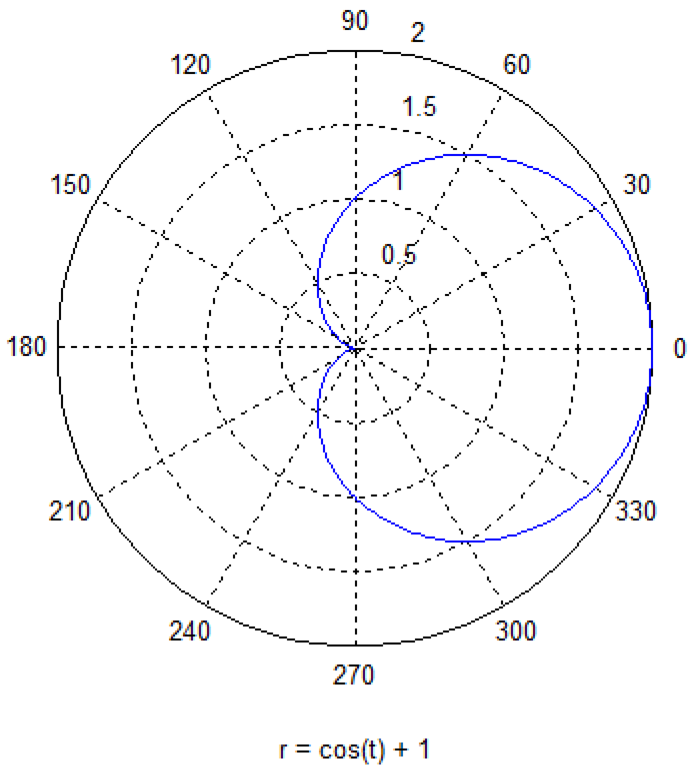
Examples

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 = cos(t) + 1

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.

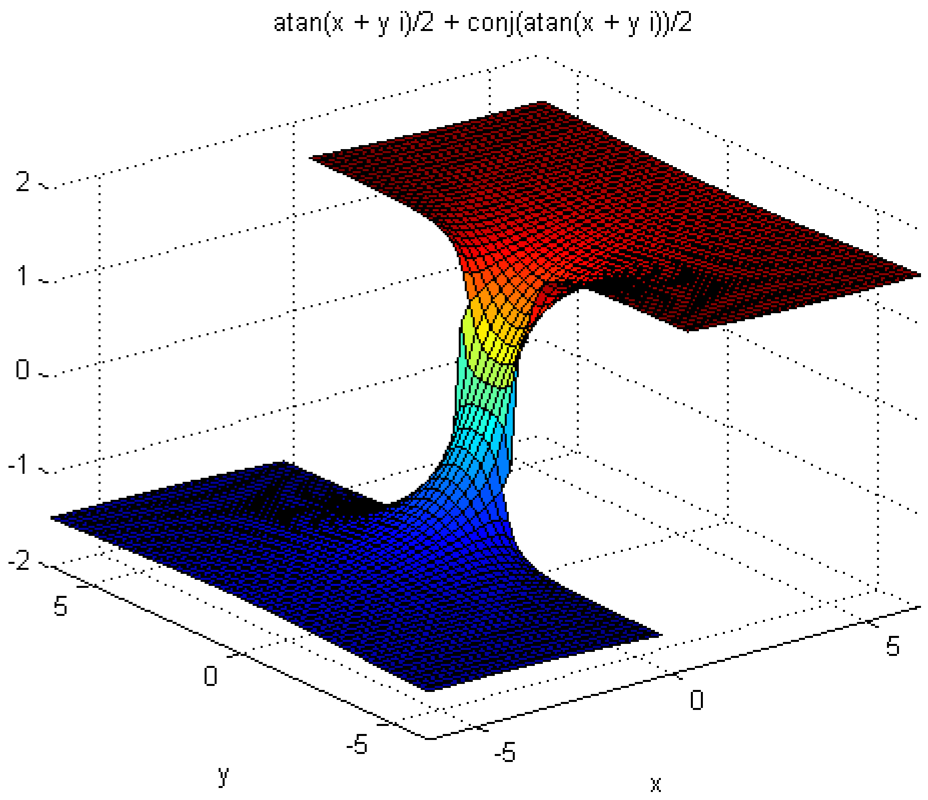
Examples

`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`

ezsurfc

Purpose

Combined surface and contour plotter

Syntax

```
ezsurfc(f)
ezsurfc(f,domain)
ezsurfc(x,y,z)
ezsurfc(x,y,z,[smin,smax,tmin,tmax])
ezsurfc(x,y,z,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
```

Description

`ezsurfc(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezsurfc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector *[xmin, xmax, ymin, ymax]* or a 2-by-1 vector *[min, max]* (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezsurfc(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezsurfc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurfc(x,y,z,[smin,smax,tmin,tmax])` or `ezsurfc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots f over a disk centered on the domain.

Examples

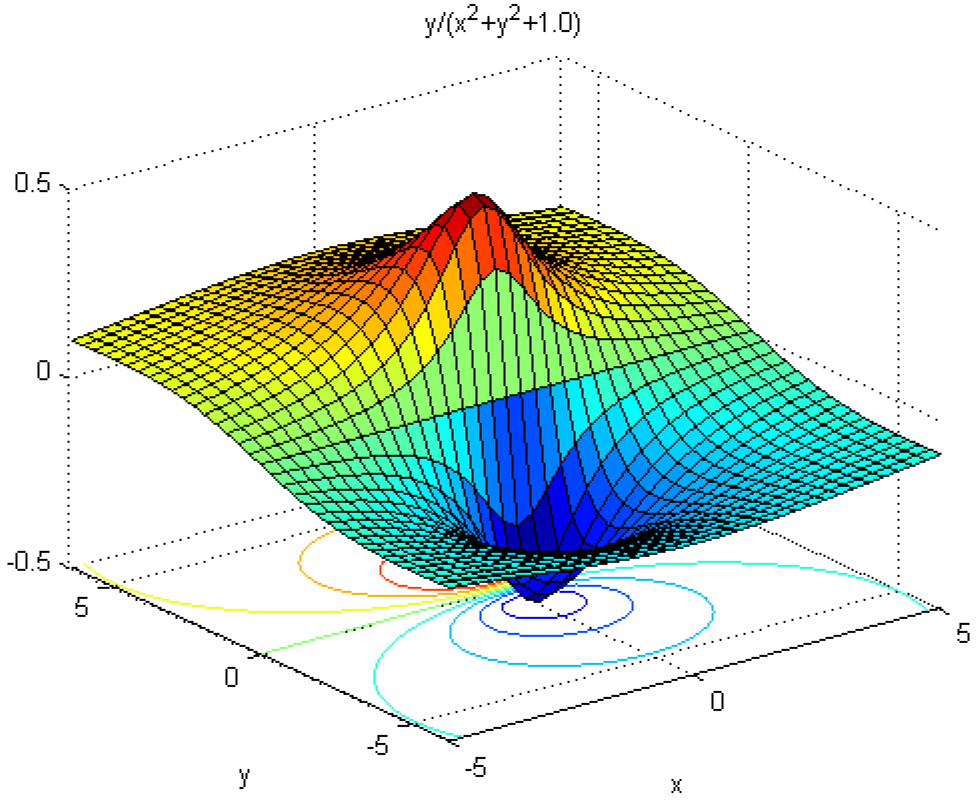
Create a surface/contour plot of the expression,

$$f(x, y) = \frac{y}{1 + x^2 + y^2},$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$, with a computational grid of size 35-by-35

```
syms x y
ezsurfc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi],35)
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65 and elevation = 26).



See Also

`ezcontour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezpolar` | `ezsurf` | `surfc`

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)

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])

ans =
[ (a - b)*(a + b), (a + b)*(a^2 - a*b + b^2)]
```

Factorize the symbolic number:

```
factor(sym('12345678901234567890'))

ans =
2*3^2*5*101*3541*3607*3803*27961
```

See Also

`collect` | `expand` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

factorial

Purpose

Factorial function

Syntax

```
factorial(n)
factorial(A)
```

Description

`factorial(n)` returns the factorial of `n`.

`factorial(A)` returns the factorials of each element of `A`.

Tips

- Calling `factorial` for a number that is not a symbolic object invokes the MATLAB `factorial` function.

Input Arguments

n

Symbolic variable or expression representing a nonnegative integer.

A

Vector or matrix of symbolic variables or expressions representing nonnegative integers.

Definitions

Factorial Function

This product defines the factorial function of a positive integer:

$$n! = \prod_{k=1}^{n} k$$

The factorial function 0! = 1.

Examples

Compute the factorial function for these expressions:

```
syms n
f = factorial(n^2 + 1)

f =
factorial(n^2 + 1)
```

Now substitute the variable n with the value 3:

```
subs(f, n, 3)

ans =
     3628800
```

Differentiate the expression involving the factorial function:

```
syms n
diff(factorial(n^2 + n + 1))

ans =
factorial(n^2 + n + 1)*psi(n^2 + n + 2)*(2*n + 1)
```

Expand the expression involving the factorial function:

```
syms n
expand(factorial(n^2 + n + 1))

ans =
factorial(n^2 + n)*(n^2 + n + 1)
```

Compute the limit for the expression involving the factorial function:

```
syms n
limit(factorial(n)/exp(n), n, inf)

ans =
Inf
```

Call `factorial` for the matrix A. The result is a matrix of the factorial functions:

```
A = sym([1 2; 3 4]);
factorial(A)

ans =
[ 1,  2]
[ 6, 24]
```

See Also

beta | gamma | mfun | mfunlist | nchoosek | psi

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Evaluate MuPAD expressions specifying their arguments

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. If `F` with the arguments `x1,...,xn` throws an error in MuPAD, then this syntax throws an error in MATLAB.

`[result,status] = feval(symengine,F,x1,...,xn)` lets you catch errors thrown by MuPAD. This syntax 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.

Tips

- Results returned by `feval` can differ from the results that you get using a MuPAD notebook directly. The reason is that `feval` sets a lower level of evaluation to achieve better performance.

Input Arguments

F

MuPAD function name or symbolic object.

x1,...,xn

Arguments of `F`.

Output Arguments

result

Symbolic object or string containing a MuPAD error message.

status

Integer indicating the error status. If `F` with the arguments `x1,...,xn` executes without errors, the error status is 0.

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
```

Try using `polylib::discrim` to compute the discriminant of the following nonpolynomial expression:

```
[result, status] = feval(symengine,'polylib::discrim',
'a*x^2 + b*x + c*ln(x)', 'x')

result =
Error: An arithmetical expression is expected. [normal]

status =
     2
```

Alternatives

`evalin` lets you evaluate MuPAD expressions without explicitly specifying their arguments.

See Also

`evalin` | `read` | `symengine`

Related Examples

- “Call Built-In MuPAD Functions from MATLAB Command Window” on page 3-32

Concepts

- "Evaluations in Symbolic Computations"
- "Level of Evaluation"

findsym

Purpose

Find symbolic variables in symbolic expression, matrix, or function

Note `findsym` is not recommended. Use `symvar` instead.

Syntax

```
findsym(s)
findsym(s,n)
```

Description

`findsym(s)` returns a string containing all symbolic variables in `s` in alphabetical order, separated by commas. If `s` does not contain any variables, `findsym` returns an empty string.

`findsym(s,n)` returns `n` symbolic variables in `s` alphabetically closest to `x`. If `s` is a symbolic function, `findsym(s,n)` returns the input arguments of `s` in front of other free variables in `s`.

Tips

- `findsym(s)` can return variables in a different order than `findsym(s,n)`.
- `findsym` does treat the constants `pi`, `i`, and `j` as variables.
- If there are no symbolic variables in `s`, `findsym` returns the empty vector.

Input Arguments

s

Symbolic expression, matrix, or function.

n

Integer.

Algorithms

When sorting the symbolic variables by their proximity to `x`, `findsym` uses this algorithm:

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.

See Also `symvar`

finverse

Purpose

Functional inverse

Syntax

```
g = finverse(f)
g = finverse(f,var)
```

Description

`g = finverse(f)` returns the functional inverse of `f`. Here `f` is an expression or function of one symbolic variable, for example, `x`. Then `g` is an expression or function, such that `f(g(x)) = x`. That is, `finverse(f)` returns f^{-1}, provided f^{-1} exists.

`g = finverse(f,var)` uses the symbolic variable `var` as the independent variable. Then `g` is an expression or function, such that `f(g(var)) = var`. Use this form when `f` contains more than one symbolic variable.

Tips

- `finverse` does not issue a warning when the inverse is not unique.

Input Arguments

f

Symbolic expression or function.

var

Symbolic variable.

Output Arguments

g

Symbolic expression or function.

Examples

Compute functional inverse for this trigonometric function:

```
syms x
f(x) = 1/tan(x);
g = finverse(f)

g(x) =
atan(1/x)
```

Compute functional inverse for this exponent function:

```
syms u v
finverse(exp(u - 2*v), u)

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`.

Examples

```
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`

formula

Purpose

Mathematical expression defining symbolic function

Syntax

```
formula(f)
```

Description

`formula(f)` returns the mathematical expression that defines `f`.

Input Arguments

f

Symbolic function.

Examples

Create this symbolic function:

```
syms x y
f(x, y) = x + y;
```

Use `formula` to find the mathematical expression that defines `f`:

```
formula(f)

ans =
x + y
```

Create this symbolic function:

```
syms f(x, y)
```

If you do not specify a mathematical expression for the symbolic function, `formula` returns the symbolic function definition as follows:

```
formula(f)

ans =
f(x, y)
```

See Also

`argnames` | `sym` | `syms` | `symvar`

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`

Purpose

Fourier transform

Syntax

`fourier(f,trans_var,eval_point)`

Description

`fourier(f,trans_var,eval_point)` computes the Fourier transform of `f` with respect to the transformation variable `trans_var` at the point `eval_point`.

Tips

- If you call `fourier` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `f` is a matrix, `fourier` applies the Fourier transform to all components of the matrix.
- To compute the inverse Fourier transform, use `ifourier`.

Input Arguments

f

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

trans_var

Symbolic variable representing the transformation variable. This variable is often called the “time variable” or the “space variable”.

> **Default:** The variable determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the “frequency variable”.

> **Default:** The variable `w`. If `w` is the transformation variable of `f`, then the default evaluation point is the variable `v`.

Definitions

Fourier Transform

The Fourier transform of the expression $f = f(x)$ with respect to the variable x at the point w is defined as follows:

$$F(w) = c \int_{-\infty}^{\infty} f(x) e^{iswx} dx.$$

Here c and s are parameters of the Fourier transform. The `fourier` function uses $c = 1$, $s = -1$.

Examples

Compute the Fourier transform of this expression with respect to the variable `x` at the evaluation point `y`:

```
syms x y
f = exp(-x^2);
fourier(f, x, y)
```

```
ans =
pi^(1/2)*exp(-y^2/4)
```

Compute the Fourier transform of this expression calling the `fourier` function with one argument. If you do not specify the transformation variable, it is determined by `symvar`. For this expression, `symvar` chooses `x` as the transformation variable.

```
syms x t y
f = exp(-x^2)*exp(-t^2);
fourier(f, y)
```

```
ans =
pi^(1/2)*exp(-t^2)*exp(-y^2/4)
```

If you also do not specify the evaluation point, `fourier` uses the variable `w`:

```
fourier(f)

ans =
pi^(1/2)*exp(-t^2)*exp(-w^2/4)
```

Compute the following Fourier transforms that involve the Dirac, Heaviside, and piecewise functions:

```
syms t w
fourier(t^3, t, w)

ans =
-pi*dirac(w, 3)*2*i

syms t0
fourier(heaviside(t - t0), t, w)

ans =
exp(-t0*w*i)*(pi*dirac(w) - i/w)

assume(x,'real')
f = exp(-x^2*abs(t))*sin(t)/t;
fourier(f, t, w)

ans =
piecewise([x ~= 0, atan((w + 1)/x^2) - atan((w - 1)/x^2)])
```

If `fourier` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(t) w
F = fourier(f, t, w)

F(w) =
fourier(f(t), t, w)
```

`ifourier` returns the original expression:

```
ifourier(F, w, t)

ans(t) =
f(t)
```

The Fourier transform of a function is related to the Fourier transform of its derivative:

```
syms f(t) w
fourier(diff(f(t), t), t, w)

ans =
w*fourier(f(t), t, w)*i
```

References

Oberhettinger F., "Tables of Fourier Transforms and Fourier Transforms of Distributions", Springer, 1990.

See Also

`ifourier` | `ilaplace` | `iztrans` | `laplace` | `ztrans`

Concepts

- "Compute Fourier and Inverse Fourier Transforms" on page 2-94

Purpose

Symbolic matrix element-wise fractional parts

Syntax

```
frac(X)
```

Description

`frac(X)` is the matrix of the fractional parts of the elements: `frac(X) = X - fix(X)`.

Examples

```
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`

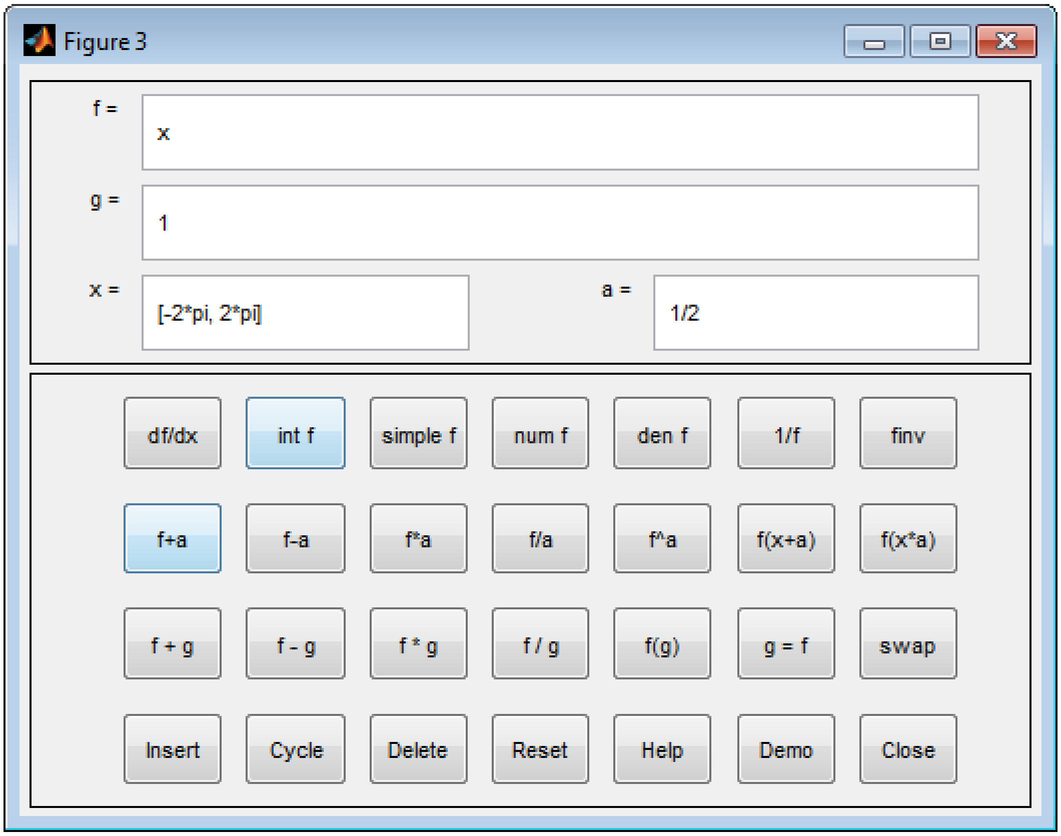
funtool

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`
simplify 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 **int f** and **finv** can 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`

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

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, 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))
u1 = subs(u, 1)

u =
3*t^2*gamma(t^3 + 1)*psi(t^3 + 1)

u1 =
3 - 3*eulergamma
```

Approximate the result using vpa:

```
vpa(u1, 10)

ans =
1.268353005
```

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)
```

See Also

beta | factorial | mfun | mfunlist | nchoosek | psi

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Define greater than or equal to relation

Syntax

```
A >= B
ge(A,B)
```

Description

`A >= B` creates a greater than or equal to relation.

`ge(A,B)` is equivalent to `A >= B`.

Tips

- If `A` and `B` are both numbers, then `A >= B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A >= B` returns a symbolic greater than or equal to relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A >= B` returns an array of relations `A(i,j,...)>=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if `A` is a variable (for example, `x`), and `B` is an *m*-by-*n* matrix, then `A` is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x >= i` becomes `x >= 0`, and `x >= 3 + 2*i` becomes `x >= 3`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

Use `assume` and the relational operator >= to set the assumption that `x` is greater than or equal to 3:

```
syms x
assume(x >= 3)
```

Solve this equation. The solver takes into account the assumption on variable `x`, and therefore returns these two solutions.

```
solve((x - 1)*(x - 2)*(x - 3)*(x - 4) == 0, x)

ans =
 3
 4
```

Use the relational operator >= to set this condition on variable `x`:

```
syms x
cond = (abs(sin(x)) >= 1/2);

for i = 0:sym(pi/12):sym(pi)
  if subs(cond, x, i)
    disp(i)
  end
end
```

Use the `for` loop with step $\pi/24$ to find angles from 0 to π that satisfy that condition:

```
pi/6
pi/4
pi/3
(5*pi)/12
```

```
pi/2
(7*pi)/12
(2*pi)/3
(3*pi)/4
(5*pi)/6
```

Alternatives

You can also define this relation by combining an equation and a greater than relation. Thus, `A >= B` is equivalent to `(A > B) & (A == B)`.

See Also

`eq | gt | isAlways | le | logical | lt | ne`

Concepts

- "Set Assumptions" on page 1-35

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.

Examples

Start a new MuPAD notebook and define a handle `mpnb` to that notebook:

```
mpnb = mupad;
```

In the MuPAD notebook, enter the command `f:=x^2`. This command creates the variable `f` and assigns the value `x^2` to this variable. At this point, the variable and its value exist only in MuPAD. Now, return to the MATLAB Command Window and use the `getVar` function:

```
f = getVar(mpnb,'f')
```

After you use `getVar`, the variable `f` appears in the MATLAB workspace. The value of the variable `f` is `x^2`.

See Also

`mupad` | `setVar`

Purpose

Gradient vector of scalar function

Syntax

```
gradient(f,x)
gradient(f)
```

Description

`gradient(f,x)` computes the gradient vector of the scalar function `f` with respect to vector `x` in Cartesian coordinates.

`gradient(f)` computes the gradient vector of the scalar function `f` with respect to a vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Tips

- If `x` is a scalar, `gradient(f, x) = diff(f, x)`.

Input Arguments

f

Scalar function represented by symbolic expression or symbolic function.

x

Vector with respect to which you compute the gradient vector.

> **Default:** Vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Definitions

Gradient Vector

The gradient vector of *f*(*x*) with respect to the vector *x* is the vector of the first partial derivatives of `f`:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Examples

Compute the gradient vector of `f(x, y, z)` with respect to vector `[x, y, z]`:

```
syms x y z
```

```
f = 2*y*z*sin(x) + 3*x*sin(z)*cos(y);
gradient(f, [x, y, z])
```

The gradient is a vector with these components:

```
ans =
 3*cos(y)*sin(z) + 2*y*z*cos(x)
 2*z*sin(x) - 3*x*sin(y)*sin(z)
 2*y*sin(x) + 3*x*cos(y)*cos(z)
```

Compute the gradient vector of `f(x, y, z)` with respect to vector `[x, y]`:

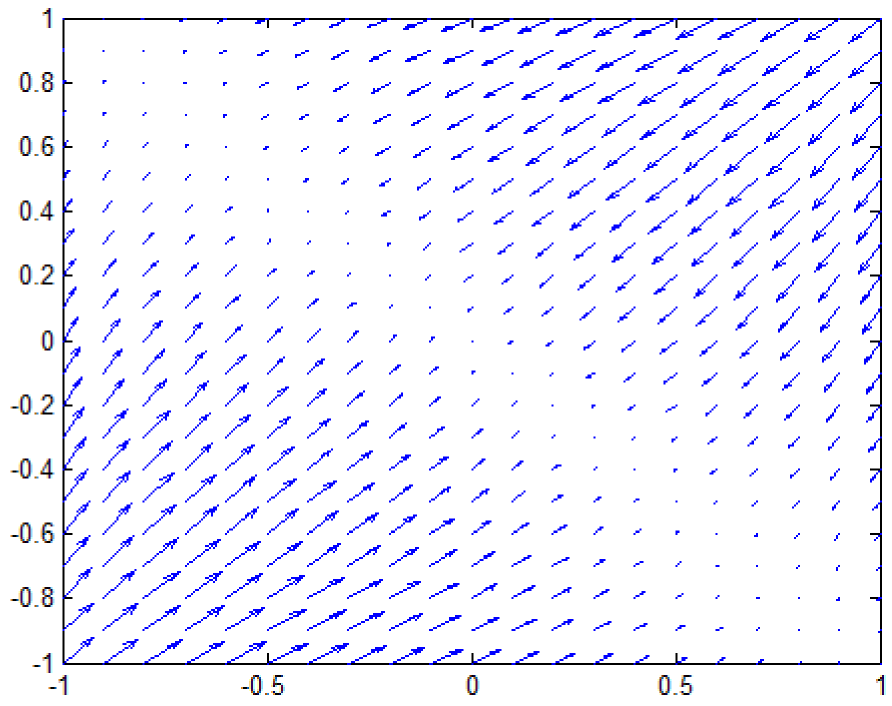
```
syms x y
f = -(sin(x) + sin(y))^2;
g = gradient(f, [x, y])
```

The gradient is vector `g` with these components:

```
g =
 -2*cos(x)*(sin(x) + sin(y))
 -2*cos(y)*(sin(x) + sin(y))
```

Now plot the vector field defined by these components. MATLAB provides the `quiver` plotting function for this task. The function does not accept symbolic arguments. First, replace symbolic variables in expressions for components of `g` with numeric values. Then use `quiver`:

```
[X, Y] = meshgrid(-1:.1:1,-1:.1:1);
G1 = subs(g(1), [x y], {X,Y}); G2 = subs(g(2), [x y], {X,Y});
quiver(X, Y, G1, G2)
```



See Also

`curl` | `divergence` | `diff` | `hessian` | `jacobian` | `laplacian` | `potential` | `quiver` | `vectorPotential`

Purpose

Define greater than relation

Syntax

```
A > B
gt(A,B)
```

Description

`A > B` creates a greater than relation.

`gt(A,B)` is equivalent to `A > B`.

Tips

- If `A` and `B` are both numbers, then `A > B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A > B` returns a symbolic greater than relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A > B` returns an array of relations `A(i,j,...)>B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if `A` is a variable (for example, `x`), and `B` is an m-by-n matrix, then `A` is expanded into m-by-n matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x > i` becomes `x > 0`, and `x > 3 + 2*i` becomes `x > 3`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

Use `assume` and the relational operator > to set the assumption that x is greater than 3:

```
syms x
assume(x > 3)
```

Solve this equation. The solver takes into account the assumption on variable x, and therefore returns this solution.

```
solve((x - 1)*(x - 2)*(x - 3)*(x - 4) == 0, x)

ans =
4
```

Use the relational operator > to set this condition on variable x:

```
syms x
cond = abs(sin(x)) + abs(cos(x)) > 7/5;

for i= 0:sym(pi/24):sym(pi)
  if subs(cond, x, i)
    disp(i)
  end
end
```

Use the `for` loop with step $\pi/24$ to find angles from 0 to π that satisfy that condition:

```
(5*pi)/24
pi/4
(7*pi)/24
(17*pi)/24
(3*pi)/4
(19*pi)/24
```

See Also

`eq | ge | isAlways | le | logical | lt | ne`

Concepts

- “Set Assumptions” on page 1-35

Purpose

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`

hessian

Purpose

Hessian matrix of scalar function

Syntax

```
hessian(f,x)
hessian(f)
```

Description

`hessian(f,x)` computes the Hessian matrix of the scalar function `f` with respect to vector `x` in Cartesian coordinates.

`hessian(f)` computes the Hessian matrix of the scalar function `f` with respect to a vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Input Arguments

f

Scalar function represented by symbolic expression or symbolic function.

x

Vector with respect to which you compute the Hessian matrix.

> **Default:** Vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Definitions

Hessian Matrix

The Hessian matrix of *f*(*x*) is the square matrix of the second partial derivatives of *f*(*x*):

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Examples

Compute the Hessian of this function of three variables:

```
syms x y z
f = x*y + 2*z*x;
hessian(f)

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

You also can compute the Hessian matrix of a scalar function as the Jacobian of the gradient of that function:

```
syms x y z
f = x*y + 2*z*x;
jacobian(gradient(f))

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

See Also

`curl` | `divergence` | `diff` | `gradient` | `jacobian` | `laplacian` | `potential` | `vectorPotential`

horner

Purpose

Horner nested polynomial representation

Syntax

```
horner(P)
```

Description

Suppose `P` is a matrix of symbolic polynomials. `horner(P)` transforms each element of `P` into its Horner, or nested, representation.

Examples

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

`collect` | `expand` | `factor` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

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 point-wise. The result is numeric if all the arguments are numeric and symbolic if any of the arguments is symbolic.

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
```

References

Oberhettinger, F. “Hypergeometric Functions.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

Purpose

Inverse Fourier transform

Syntax

`ifourier(F,trans_var,eval_point)`

Description

`ifourier(F,trans_var,eval_point)` computes the inverse Fourier transform of `F` with respect to the transformation variable `trans_var` at the point `eval_point`.

Tips

- If you call `ifourier` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `F` is a matrix, `ifourier` applies the inverse Fourier transform to all components of the matrix.
- The toolbox computes the inverse Fourier transform via the direct Fourier transform:

  $$ifourier(F,w,t) = \frac{1}{2\pi} fourier(F,w,-t)$$

  If `ifourier` cannot find an explicit representation of the inverse Fourier transform, it returns results in terms of the direct Fourier transform.
- To compute the direct Fourier transform, use `fourier`.

Input Arguments

F

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

trans_var

Symbolic variable representing the transformation variable. This variable is often called the “frequency variable”.

> **Default:** The variable `w`. If `F` does not contain `w`, then the default variable is determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the “time variable” or the “space variable”.

> **Default:** The variable `x`. If `x` is the transformation variable of `F`, then the default evaluation point is the variable `t`.

Definitions

Inverse Fourier Transform

The inverse Fourier transform of the expression $F = F(w)$ with respect to the variable w at the point x is defined as follows:

$$f(x) = \frac{|s|}{2\pi c} \int_{-\infty}^{\infty} F(w) e^{-iswx} dw.$$

Here c and s are parameters of the inverse Fourier transform. The `ifourier` function uses $c = 1$, $s = –1$.

Examples

Compute the inverse Fourier transform of this expression with respect to the variable `y` at the evaluation point `x`:

```
syms x y
F = sqrt(sym(pi))*exp(-y^2/4);
ifourier(F, y, x)

ans =
exp(-x^2)
```

Compute the inverse Fourier transform of this expression calling the `ifourier` function with one argument. If you do not specify the transformation variable, `ifourier` uses the variable `w`:

```
syms a w t real
F = exp(-w^2/(4*a^2));
ifourier(F, t)
```

```
ans =
exp(-a^2*t^2)/(2*pi^(1/2)*(1/(4*a^2))^(1/2))
```

If you also do not specify the evaluation point, `ifourier` uses the variable `x`:

```
ifourier(F)

ans =
exp(-a^2*x^2)/(2*pi^(1/2)*(1/(4*a^2))^(1/2))
```

Compute the following inverse Fourier transforms that involve the Dirac and Heaviside functions:

```
syms t w
ifourier(dirac(w), w, t)

ans =
1/(2*pi)

ifourier(2*exp(-abs(w)) - 1, w, t)

ans =
-(2*pi*dirac(t) - 4/(t^2 + 1))/(2*pi)

ifourier(1/(w^2 + 1), w, t)

ans =
(pi*exp(-t)*heaviside(t) + pi*heaviside(-t)*exp(t))/(2*pi)
```

If `ifourier` cannot find an explicit representation of the transform, it returns results in terms of the direct Fourier transform:

```
syms F(w) t
f = ifourier(F, w, t)
```

```
f(t) =
fourier(F(w), w, -t)/(2*pi)
```

References

Oberhettinger, F. “Tables of Fourier Transforms and Fourier Transforms of Distributions”, Springer, 1990.

See Also

`fourier` | `ilaplace` | `iztrans` | `laplace` | `ztrans`

Concepts

- “Compute Fourier and Inverse Fourier Transforms” on page 2-94

Purpose

Inverse Laplace transform

Syntax

```
ilaplace(F,trans_var,eval_point)
```

Description

`ilaplace(F,trans_var,eval_point)` computes the inverse Laplace transform of `F` with respect to the transformation variable `trans_var` at the point `eval_point`.

Tips

- If you call `ilaplace` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `F` is a matrix, `ilaplace` applies the inverse Laplace transform to all components of the matrix.
- To compute the direct Laplace transform, use `laplace`.

Input Arguments

F

Symbolic expression or function, vector or matrix of symbolic expressions or functions.

trans_var

Symbolic variable representing the transformation variable. This variable is often called the "complex frequency variable".

> **Default:** The variable `s`. If `F` does not contain `s`, then the default variable is determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the "time variable".

> **Default:** The variable `t`. If `t` is the transformation variable of `F`, then the default evaluation point is the variable `x`.

Definitions

Inverse Laplace Transform

The inverse Laplace transform is defined by a contour integral in the complex plane:

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

Here c is a suitable complex number.

Examples

Compute the inverse Laplace transform of this expression with respect to the variable `y` at the evaluation point `x`:

```
syms x y
F = 1/y^2;
ilaplace(F, y, x)

ans =
x
```

Compute the inverse Laplace transform of this expression calling the `ilaplace` function with one argument. If you do not specify the transformation variable, `ilaplace` uses the variable `s`:

```
syms a s x
F = 1/(s - a)^2;
ilaplace(F, x)

ans =
x*exp(a*x)
```

If you also do not specify the evaluation point, `ilaplace` uses the variable `t`:

```
ilaplace(F)

ans =
t*exp(a*t)
```

Compute the following inverse Laplace transforms that involve the Dirac and Heaviside functions:

```
syms s t
ilaplace(1, s, t)

ans =
dirac(t)

ilaplace(exp(-2*s)/(s^2 + 1) + s/(s^3 + 1), s, t)

ans =
heaviside(t - 2)*sin(t - 2) - exp(-t)/3 +...
(exp(t/2)*(cos((3^(1/2)*t)/2) +
3^(1/2)*sin((3^(1/2)*t)/2)))/3
```

If `ilaplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms F(s) t
f = ilaplace(F, s, t)

f(t) =
ilaplace(F(s), s, t)
```

`laplace` returns the original expression:

```
laplace(f, t, s)

ans(s) =
F(s)
```

See Also `fourier` | `ifourier` | `iztrans` | `laplace` | `ztrans`

Concepts

- “Compute Laplace and Inverse Laplace Transforms” on page 2-101

Purpose

Imaginary part of complex number

Syntax

```
imag(z)
imag(A)
```

Description

`imag(z)` returns the imaginary part of `z`.

`imag(A)` returns the imaginary part of each element of `A`.

Tips

- Calling `imag` for a number that is not a symbolic object invokes the MATLAB `imag` function.

Input Arguments

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Examples

Find the imaginary parts of these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[imag(2 + 3/2*i), imag(sin(5*i)), imag(2*exp(1 + i))]
```

```
ans =
    1.5000   74.2032    4.5747
```

Compute the imaginary parts of the numbers converted to symbolic objects:

```
[imag(sym(2) + 3/2*i), imag(4/(sym(1) + 3*i)),
imag(sin(sym(5)*i))]
```

```
ans =
[ 3/2, -6/5, sinh(5)]
```

Compute the imaginary part of this symbolic expression:

```
imag(sym('2*exp(1 + i)'))

ans =
2*exp(1)*sin(1)
```

In general, `imag` cannot extract the entire imaginary parts from symbolic expressions containing variables. However, `imag` can rewrite and sometimes simplify the input expression:

```
syms a x y
imag(a + 2)
imag(x + y*i)

ans =
imag(a)

ans =
imag(x) + real(y)
```

If you assign numeric values to these variables or if you specify that these variables are real, `imag` can extract the imaginary part of the expression:

```
syms a
 a = 5 + 3*i;
imag(a + 2)

ans =
     3

syms x y real
imag(x + y*i)

ans =
y
```

Clear the assumption that `x` and `y` are real:

```
syms x y clear
```

Find the imaginary parts of the elements of matrix `A`:

```
A = sym('[-1 + i, sinh(x); exp(10 + 7*i), exp(pi*i)]');
imag(A)

ans =
[              1, imag(sinh(x))]
[ exp(10)*sin(7),             0]
```

Alternatives

You can compute the imaginary part of `z` via the conjugate: `imag(z)= (z - conj(z))/2i`.

See Also

`conj` | `real`

Purpose

Symbolic integration

Syntax

```
int(expr,var)
int(expr,var,Name,Value)
int(expr,var,a,b)
int(expr,var,a,b,Name,Value)
```

Description

`int(expr,var)` computes the indefinite integral of `expr` with respect to the symbolic scalar variable `var`. Specifying the variable `var` is optional. If you do not specify it, `int` uses the default variable determined by `symvar`.

`int(expr,var,Name,Value)` computes the indefinite integral of `expr` with respect to the symbolic scalar variable `var` with additional options specified by one or more `Name,Value` pair arguments. If you do not specify it, `int` uses the default variable determined by `symvar`.

`int(expr,var,a,b)` computes the definite integral of `expr` with respect to `var` from `a` to `b`. If you do not specify it, `int` uses the default variable determined by `symvar`.

`int(expr,var,a,b,Name,Value)` computes the definite integral of `expr` with respect to `var` from `a` to `b` with additional options specified by one or more `Name,Value` pair arguments. If you do not specify it, `int` uses the default variable determined by `symvar`.

Tips

- In contrast to differentiation, symbolic integration is a more complicated task. If `int` cannot compute an integral of an expression, one of the following reasons might apply:
  - The antiderivative does not exist in a closed form.
  - The antiderivative exists, but `int` cannot find it.

  If `int` cannot compute a closed form of an integral, it issues a warning and returns an unresolved integral.

  Try to approximate such integrals by using one of the following methods:

  - For indefinite integrals, use series expansions. Use this method to approximate an integral around a particular value of the variable.
  - For definite integrals, use numeric approximations.
- Results returned by `int` do not include integration constants.

Input Arguments

expr

Symbolic expression or matrix of symbolic expressions.

var

Differentiation variable.

> **Default:** Variable determined by `symvar`.

a

Number or symbolic expression, including expressions with infinities.

b

Number or symbolic expression, including expressions with infinities.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'IgnoreAnalyticConstraints'

If the value is `true`, apply purely algebraic simplifications to the integrand. This option can provide simpler results for expressions, for which the direct use of the integrator returns complicated results. In some cases, it also enables `int` to compute integrals that cannot be computed otherwise. Note that using this option can lead to wrong or incomplete results.

Default: `false`

'IgnoreSpecialCases'

If the value is `true` and integration requires case analysis, ignore cases that require one or more parameters to be elements of a comparatively small set, such as a fixed finite set or a set of integers

Default: `false`

'PrincipalValue'

If the value is `true`, compute the Cauchy principal value of the integral

Default: `false`

Examples

Find an 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 an indefinite integral of the following multivariate expression with respect to `z`:

```
syms x z
int(x/(1 + z^2), z)
```

The result is:

```
ans =
x*atan(z)
```

Integrate the following expression from 0 to 1:

```
syms x
int(x*log(1 + x), 0, 1)
```

The result is:

```
ans =
1/4
```

Integrate 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]
```

Compute this indefinite integral:

```
syms x
```

```
int(acos(sin(x)), x)
```

By default, int uses strict mathematical rules. These rules do not let int rewrite asin(sin(x)) and acos(cos(x)) as x. Therefore, int returns this result:

```
ans =
x*acos(sin(x)) + (x^2*sign(cos(x)))/2
```

If you want a simple practical solution, try IgnoreAnalyticConstraints:

```
int(acos(sin(x)), x, 'IgnoreAnalyticConstraints', true)

ans =
(x*(pi - x))/2
```

Compute this integral with respect to the variable x:

```
syms x t
int(x^t, x)
```

By default, int returns the integral as a piecewise object where every branch corresponds to a particular value (or a range of values) of the symbolic parameter t:

```
ans =
piecewise([t == -1, log(x)], [t ~= -1, x^(t + 1)/(t + 1)])
```

To ignore special cases of parameter values, use IgnoreSpecialCases:

```
int(x^t, x, 'IgnoreSpecialCases', true)
```

With this option, int ignores the special case t=-1 and returns only the branch where t<> 1:

```
ans =
x^(t + 1)/(t + 1)
```

Compute this definite integral, where the integrand has a pole in the interior of the interval of integration:

```
syms x
int(1/(x - 1), x, 0, 2)
```

Mathematically, this integral is not defined:

```
ans =
NaN
```

However, the Cauchy principal value of the integral exists. Use `PrincipalValue` to compute the Cauchy principal value of the integral:

```
int(1/(x - 1), x, 0, 2, 'PrincipalValue', true)
```

The result is:

```
ans =
0
```

If `int` cannot compute a closed form of an integral, it issues a warning and returns an unresolved integral:

```
syms x
F = sin(sinh(x));
int(F, x)

Warning: Explicit integral could not be found.

ans =
int(sin(sinh(x)), x)
```

If `int` cannot compute a closed form of an indefinite integral, try to approximate the expression around some point using `taylor`, and then compute the integral. For example, approximate the expression around $x = 0$:

```
int(taylor(F, x, 'ExpansionPoint', 0, 'Order', 10), x)

ans =
x^10/56700 - x^8/720 - x^6/90 + x^2/2
```

Compute this definite integral:

```
syms x
F = int(cos(x)/sqrt(1 + x^2), x, 0, 10)

Warning: Explicit integral could not be found.

F =
int(cos(x)/(x^2 + 1)^(1/2), x == 0..10)
```

If `int` cannot compute a closed form of a definite integral, try approximating that integral numerically using `vpa`. For example, approximate `F` with 5 significant digits:

```
vpa(F, 5)

ans =
0.37571
```

Algorithms

When you use `IgnoreAnalyticConstraints`, `int` applies these rules:

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

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

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

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

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

See Also

`diff` | `symsum` | `symvar`

How To

- “Integration” on page 2-13

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

Check whether equation or inequality holds for all values of its variables

Syntax

```
isAlways(cond)
isAlways(cond,Name,Value)
```

Description

`isAlways(cond)` checks whether the condition `cond` is valid for all possible values of symbolic variables in `cond`. When verifying the validity of `cond`, `isAlways` takes into account all assumptions set on the variables in `cond`. If the condition holds, `isAlways` returns logical `1` (`true`). Otherwise it returns logical `0` (`false`).

`isAlways(cond,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

cond

Equation, inequality, or vector or matrix of equations or inequalities. You also can combine several conditions by using the logical operators `and`, `or`, `xor`, `not`, or their shortcuts.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Unknown'

One of these strings: `false`, `true`, or `error`. If `isAlways` cannot determine whether the specified condition holds for all values of its variables and at the same time cannot prove that the condition does not hold, then the function can return logical 0 or 1 or throw an error. By default, it returns logical 0 (`false`). If you specify `true`, then `isAlways` will return logical 1 (`true`) when it cannot decide whether the condition holds or not. If you specify `error`, `isAlways` will throw an error.

Default: `false`

Examples

Check whether this inequality is valid for all values of `x`:

```
syms x
isAlways(abs(x) >= 0)
```

```
ans =
     1
```

Now check whether this equation is valid for all values of `x`:

```
isAlways(sin(x)^2 + cos(x)^2 == 1)
```

```
ans =
     1
```

Check if at least one of the following two conditions is valid. To check if at least one of several conditions is valid, combine these conditions by using the logical operator `or` or its shortcut `|`.

```
syms x
isAlways(sin(x)^2 + cos(x)^2 == 1 | x^2 > 0)
```

```
ans =
     1
```

Check the validity of this inequality. When `isAlways` cannot determine whether the condition is valid, it returns logical `0` by default:

```
syms x
isAlways(2*x >= x)
```

```
ans =
     0
```

To change this default behavior, use `Unknown`. For example, specify that `isAlways` must return logical `1` if it cannot determine the validity of this inequality:

```
isAlways(2*x >= x,'Unknown','true')
```

```
ans =
     1
```

Instead of `true`, you can also specify `error`. In this case, `isAlways` will throw an error if it cannot determine the validity of the condition.

Check validity of this inequality under the assumption that `x` is positive. When `isAlways` determines validity of an equation or inequality, it takes into account assumptions on variables in that equation or inequality:

```
syms x
assume(x < 0)
isAlways(2*x < x)
```

```
ans =
     1
```

For further computations, clear the assumption on `x`:

```
syms x clear
```

See Also

`assume` | `assumeAlso` | `assumptions` | `isequaln` | `logical` | `sym` | `syms`

Concepts

- "Assumptions on Symbolic Objects" on page 1-35
- "Clear Assumptions and Reset the Symbolic Engine" on page 3-43

isequaln

Purpose

Test symbolic objects for equality, treating `NaN` values as equal

Syntax

```
isequaln(A,B)
isequaln(A1,A2,...,An)
```

Description

`isequaln(A,B)` returns logical `1` (true) if `A` and `B` are the same size and their contents are of equal value. Otherwise, `isequaln` returns logical `0` (false). All `NaN` (not a number) values are considered to be equal to each other. `isequaln` recursively compares the contents of symbolic data structures and the properties of objects. If all contents in the respective locations are equal, `isequaln` returns logical `1` (true).

`isequaln(A1,A2,...,An)` returns logical `1` (true) if all the inputs are equal.

Tips

- Calling `isequaln` for arguments that are not symbolic objects invokes the MATLAB `isequaln` function. If one of the arguments is symbolic, then all other arguments are converted to symbolic objects before comparison.

Input Arguments

A,B - Inputs to compare

symbolic numbers | symbolic variables | symbolic expressions | symbolic functions | symbolic vectors | symbolic matrices

Inputs to compare, specified as symbolic numbers, variables, expressions, functions, vectors, or matrices. If one of the arguments is a symbolic object and the other one is numeric, the toolbox converts the numeric object to symbolic before comparing them.

A1,A2,...,An - Series of inputs to compare

symbolic numbers | symbolic variables | symbolic expressions | symbolic functions | symbolic vectors | symbolic matrices

Series of inputs to compare, specified as symbolic numbers, variables, expressions, functions, vectors, or matrices. If at least one of the arguments is a symbolic object, the toolbox converts all other arguments to symbolic objects before comparing them.

Examples

Compare Two Expressions

Use isequaln to compare these two expressions:

```
syms x
isequaln(abs(x), x)

ans =
     0
```

For positive x, these expressions are identical:

```
assume(x > 0)
isequaln(abs(x), x)

ans =
     1
```

For further computations, remove the assumption:

```
syms x clear
```

Compare Two Matrices

Use isequaln to compare these two matrices:

```
A = hilb(3);
B = sym(A);
isequaln(A, B)

ans =
     1
```

Compare Vectors Containing NaN Values

Use isequaln to compare these vectors:

```
syms x
A1 = [x NaN NaN];
A2 = [x NaN NaN];
A3 = [x NaN NaN];
isequaln(A1, A2, A3)
```

```
ans =
     1
```

See Also `isAlways | isequaln | logical`

Purpose

Inverse Z-transform

Syntax

```
iztrans(F,trans_index,eval_point)
```

Description

`iztrans(F,trans_index,eval_point)` computes the inverse Z-transform of `F` with respect to the transformation index `trans_index` at the point `eval_point`.

Tips

- If you call `iztrans` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `F` is a matrix, `iztrans` applies the Z-transform to all components of the matrix.
- To compute the direct Z-transform, use `ztrans`.

Input Arguments

F

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

trans_index

Symbolic variable representing the transformation index. This variable is often called the “complex frequency variable”.

> **Default:** The variable `z`. If `F` does not contain `z`, then the default variable is determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the “discrete time variable”.

> **Default:** The variable `n`. If `n` is the transformation index of `F`, then the default evaluation point is the variable `k`.

Definitions

Inverse Z-Transform

If R is a positive number, such that the function $F(z)$ is analytic on and outside the circle $|z| = R$, then the inverse Z-transform is defined as follows:

$$f(n) = \frac{1}{2\pi i} \oint_{|z|=R} F(z) z^{n-1} dz, \quad n = 0,1,2...$$

Examples

Compute the inverse Z-transform of this expression with respect to the transformation index `x` at the evaluation point `k`:

```
syms k x
F =  2*x/(x - 2)^2;
iztrans(F, x, k)

ans =
2^k + 2^k*(k - 1)
```

Compute the inverse Z-transform of this expression calling the `iztrans` function with one argument. If you do not specify the transformation index, `iztrans` uses the variable `z`:

```
syms z a k
F = exp(a/z);
iztrans(F, k)

ans =
a^k/factorial(k)
```

If you also do not specify the evaluation point, `iztrans` uses the variable `n`:

```
iztrans(F)

ans =
a^n/factorial(n)
```

Compute the inverse Z-transforms of these expressions. The results involve the Kronecker's delta function:

```
syms n z
iztrans(1/z, z, n)

ans =
kroneckerDelta(n - 1, 0)

iztrans((z^3 + 3*z^2 + 6*z + 5)/z^5, z, n)

ans =
kroneckerDelta(n - 2, 0) + 3*kroneckerDelta(n - 3, 0) +...
6*kroneckerDelta(n - 4, 0) + 5*kroneckerDelta(n - 5, 0)
```

If `iztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms F(z) n
f = iztrans(F, z, n)

f(n) =
iztrans(F(z), z, n)
```

`ztrans` returns the original expression:

```
ztrans(f, n, z)

ans(z) =
F(z)
```

See Also

`fourier` | `ifourier` | `ilaplace` | `laplace` | `ztrans`

Concepts

- “Compute Z-Transforms and Inverse Z-Transforms” on page 2-108

Purpose

Jacobian matrix

Syntax

```
jacobian(f,v)
```

Description

`jacobian(f,v)` computes the Jacobian matrix 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 a scalar, the Jacobian matrix of `f` is the gradient of `f`. If `v` is a scalar, the result equals to `diff(f, v)`.

Examples

Compute the Jacobian matrix for each of these 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)

R =
[ y*z, x*z, x*y]
[   0,   1,   0]
[   1,   0,   1]

b =
[ 1, 0, 1]
```

See Also

`curl` | `divergence` | `diff` | `gradient` | `hessian` | `laplacian` | `potential` | `vectorPotential`

jordan

Purpose

Jordan form of matrix

Syntax

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

Description

`J = jordan(A)` computes the Jordan canonical form (also called Jordan 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 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 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 form and the similarity transform for this 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 `charpoly | eig | inv`

lambertw

Purpose

Lambert W function

Syntax

```
W = lambertw(X)
W = lambertw(K,X)
```

Description

`W = lambertw(X)` evaluates the Lambert W function at the elements of `X`, a numeric matrix or a symbolic matrix. The Lambert W function 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 the Lambert W function:

```
lambertw([0 exp(2); pi 1])
```

The result is:

```
ans =
         0    1.5571
    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,trans_var,eval_point)
```

Description

`laplace(f,trans_var,eval_point)` computes the Laplace transform of `f` with respect to the transformation variable `trans_var` at the point `eval_point`.

Tips

- If you call `laplace` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `f` is a matrix, `laplace` applies the Laplace transform to all components of the matrix.
- To compute the inverse Laplace transform, use `ilaplace`.

Input Arguments

f

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

trans_var

Symbolic variable representing the transformation variable. This variable is often called the "time variable".

> **Default:** The variable `t`. If `f` does not contain `t`, then the default variable is determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the "complex frequency variable".

> **Default:** The variable `s`. If `s` is the transformation variable of `f`, then the default evaluation point is the variable `z`.

Definitions

Laplace Transform

The Laplace transform is defined as follows:

$$F(s) = \int_{0}^{\infty} f(t)\, e^{-st} dt.$$

Examples

Compute the Laplace transform of this expression with respect to the variable `x` at the evaluation point `y`:

```
syms x y
f = 1/sqrt(x);
laplace(f, x, y)

ans =
pi^(1/2)/y^(1/2)
```

Compute the Laplace transform of this expression calling the `laplace` function with one argument. If you do not specify the transformation variable, `laplace` uses the variable `t`:

```
syms a t y
f = exp(-a*t);
laplace(f, y)

ans =
1/(a + y)
```

If you also do not specify the evaluation point, `laplace` uses the variable `s`:

```
laplace(f)

ans =
1/(a + s)
```

Compute the following Laplace transforms that involve the Dirac and Heaviside functions:

```
syms t s
laplace(dirac(t - 3), t, s)

ans =
exp(-3*s)

laplace(heaviside(t - pi), t, s)

ans =
exp(-pi*s)/s
```

If `laplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(t) s
F = laplace(f, t, s)

F(s) =
laplace(f(t), t, s)
```

`ilaplace` returns the original expression:

```
ilaplace(F, s, t)

ans(t) =
f(t)
```

The Laplace transform of a function is related to the Laplace transform of its derivative:

```
syms f(t) s
laplace(diff(f(t), t), t, s)
```

```
ans =
s*laplace(f(t), t, s) - f(0)
```

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `ztrans`

Concepts

- "Compute Laplace and Inverse Laplace Transforms" on page 2-101

laplacian

Purpose

Laplacian of scalar function

Syntax

```
laplacian(f,x)
laplacian(f)
```

Description

`laplacian(f,x)` computes the Laplacian of the scalar function or functional expression `f` with respect to the vector `x` in Cartesian coordinates.

`laplacian(f)` computes the gradient vector of the scalar function or functional expression `f` with respect to a vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Tips

- If `x` is a scalar, `gradient(f, x) = diff(f, 2, x)`.

Input Arguments

f

Symbolic expression or symbolic function.

x

Vector with respect to which you compute the Laplacian.

> **Default:** Vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

Definitions

Laplacian of a Scalar Function

The Laplacian of the scalar function or functional expression f with respect to the vector $X = (X_1,...,X_n)$ is the sum of the second derivatives of f with respect to $X_1,...,X_n$:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f_i}{\partial x_i^2}$$

Examples

Compute the Laplacian of this symbolic expression. By default, `laplacian` computes the Laplacian of an expression with respect to a vector of all variables found in that expression. The order of variables is defined by `symvar`.

```
syms x y t
laplacian(1/x^3 + y^2 - log(t))

ans =
1/t^2 + 12/x^5 + 2
```

Create this symbolic function:

```
syms x y z
f(x, y, z) = 1/x + y^2 + z^3;
```

Compute the Laplacian of this function with respect to the vector `[x, y, z]`:

```
L = laplacian(f, [x y z])

L(x, y, z) =
6*z + 2/x^3 + 2
```

Alternatives

The Laplacian of a scalar function or functional expression is the divergence of the gradient of that function or expression:

$$\Delta f = \nabla \cdot (\nabla f)$$

Therefore, you can compute the Laplacian using the `divergence` and `gradient` functions:

```
syms f(x, y)
divergence(gradient(f(x, y)), [x y])
```

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `jacobian` | `potential` | `vectorPotential`

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`

le

Purpose

Define less than or equal to relation

Syntax

```
A <= B
le(A,B)
```

Description

`A <= B` creates a less than or equal to relation.

`le(A,B)` is equivalent to `A <= B`.

Tips

- If `A` and `B` are both numbers, then `A <= B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A <= B` returns a symbolic less than or equal to relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A <= B` returns an array of relations `A(i,j,...)<=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if `A` is a variable (for example, `x`), and `B` is an *m*-by-*n* matrix, then `A` is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x <= i` becomes `x <= 0`, and `x <= 3 + 2*i` becomes `x <= 3`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

Use `assume` and the relational operator <= to set the assumption that `x` is less than or equal to 3:

```
syms x
assume(x <= 3)
```

Solve this equation. The solver takes into account the assumption on variable `x`, and therefore returns these three solutions.

```
solve((x - 1)*(x - 2)*(x - 3)*(x - 4) == 0, x)

ans =
 1
 2
 3
```

Use the relational operator <= to set this condition on variable `x`:

```
syms x
cond = (abs(sin(x)) <= 1/2);

for i = 0:sym(pi/12):sym(pi)
  if subs(cond, x, i)
    disp(i)
  end
end
```

Use the `for` loop with step $\pi/24$ to find angles from 0 to π that satisfy that condition:

```
0
pi/12
pi/6
```

```
(5*pi)/6
(11*pi)/12
pi
```

Alternatives

You can also define this relation by combining an equation and a less than relation. Thus, `A <= B` is equivalent to `(A < B) & (A == B)`.

See Also

`eq | ge | gt | isAlways | lt | logical | ne`

Concepts

- "Set Assumptions" on page 1-35

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)

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')

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)

ans =
[ exp(a), 0]
```

See Also

diff | taylor

Purpose

Solve linear system of equations given in matrix form

Syntax

```
X = linsolve(A,B)
[X,R] = linsolve(A,B)
```

Description

`X = linsolve(A,B)` solves the matrix equation $AX = B$. In particular, if `B` is a column vector, `linsolve` solves a linear system of equations given in the matrix form.

`[X,R] = linsolve(A,B)` solves the matrix equation $AX = B$ and returns the reciprocal of the condition number of `A` if `A` is a square matrix, and the rank of `A` otherwise.

Tips

- If the solution is not unique, `linsolve` issues a warning, chooses one solution and returns it.
- If the system does not have a solution, `linsolve` issues a warning and returns `X` with all elements set to `Inf`.
- Calling `linsolve` for numeric matrices that are not symbolic objects invokes the MATLAB `linsolve` function. This function accepts real arguments only. If your system of equations uses complex numbers, use `sym` to convert at least one matrix to a symbolic matrix, and then call `linsolve`.

Input Arguments

A

Coefficient matrix.

B

Matrix or column vector containing the right sides of equations.

Output Arguments

X

Matrix or vector representing the solution.

R

Reciprocal of the condition number of A if A is a square matrix. Otherwise, the rank of A.

Definitions

Matrix Representation of a System of Linear Equations

A system of linear equations

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ &\ldots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \end{aligned}$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

and $\vec{b}$ is the vector containing the right sides of equations:

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Examples

Define the matrix equation using the following matrices A and B:

```
syms x y z;
A = [x 2*x y; x*z 2*x*z y*z+z; 1 0 1];
B = [z y; z^2 y*z; 0 0];
```

Use `linsolve` to solve this equation. Assigning the result of the `linsolve` call to a single output argument, you get the matrix of solutions:

```
X = linsolve(A, B)

X =
[       0,       0]
[ z/(2*x), y/(2*x)]
[       0,       0]
```

To return the solution and the reciprocal of the condition number of the square coefficient matrix, assign the result of the `linsolve` call to a vector of two output arguments:

```
syms a x y z;
A = [a 0 0; 0 a 0; 0 0 1];
B = [x; y; z];
[X, R] = linsolve(A, B)

X =
 x/a
 y/a
   z

R =
1/(max(abs(a), 1)*max(1/abs(a), 1))
```

If the coefficient matrix is rectangular, `linsolve` returns the rank of the coefficient matrix as the second output argument:

```
syms a b x y;
A = [a 0 1; 1 b 0];
B = [x; y];
[X, R] = linsolve(A, B)

Warning: System is rank deficient. Solution is not unique.

X =
               x/a
 -(x - a*y)/(a*b)
                 0

R =
2
```

See Also

`cond` | `equationsToMatrix` | `inv` | `norm` | `odeToVectorField` | `rank` | `solve` | `symvar`

Related Examples

- "Solve a System of Algebraic Equations" on page 2-85

Purpose

Natural logarithm of entries of symbolic matrix

Syntax

```
Y = log(X)
```

Description

`Y = log(X)` returns the natural logarithm of `X`.

Input Arguments

X

Symbolic variable, expression, function, or matrix

Output Arguments

Y

Number, variable, expression, function, or matrix. 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`.

Examples

Compute the natural logarithm of each entry of this symbolic matrix:

```
syms x
M = x*hilb(2);
log(M)

ans =
[   log(x), log(x/2)]
[ log(x/2), log(x/3)]
```

Differentiate this symbolic expression:

```
syms x
diff(log(x^3), x)

ans =
3/x
```

See Also

`log2` | `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

`log` | `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

`log` | `log10`

Purpose

Check validity of equation or inequality

Syntax

```
logical(cond)
```

Description

`logical(cond)` checks whether the condition `cond` is valid.

Tips

- For symbolic equations, `logical` returns logical `1` (`true`) only if the left and right sides are identical. Otherwise, it returns logical `0` (`false`).
- For symbolic inequalities constructed with `~=`, `logical` returns logical `0` (`false`) only if the left and right sides are identical. Otherwise, it returns logical `1` (`true`).
- For all other inequalities (constructed with `<`, `<=`, `>`, or `>=`), `logical` returns logical `1` if it can prove that the inequality is valid and logical `0` if it can prove that the inequality is invalid. If `logical` cannot determine whether such inequality is valid or not, it throws an error.
- `logical` evaluates expressions on both sides of an equation or inequality, but does not simplify or mathematically transform them. To compare two expressions applying mathematical transformations and simplifications, use `isAlways`.
- `logical` typically ignores assumptions on variables.

Input Arguments

cond

Equation, inequality, or vector or matrix of equations or inequalities. You also can combine several conditions by using the logical operators `and`, `or`, `xor`, `not`, or their shortcuts.

Examples

Use `logical` to check whether `1` if less than `2`:

```
logical(1 < 2)
```

```
ans =
     1
```

Check if the following two conditions are both valid. To check if several conditions are valid at the same time, combine these conditions by using the logical operator `and` or its shortcut `&`.

```
syms x
logical(1 < 2 & x == x)

ans =
     1
```

Check this inequality. Note that `logical` evaluates the left side of the inequality.

```
logical(4 - 1 > 2)

ans =
     1
```

`logical` also evaluates more complicated symbolic expressions on both sides of equations and inequalities. For example, it evaluates the integral on the left side of this equation:

```
syms x
logical(int(x, x, 0, 2) - 1 == 1)

ans =
     1
```

Check the validity of this equation using `logical`. Without an additional assumption that `x` is nonnegative, this equation is invalid.

```
syms x
logical(x == sqrt(x^2))

ans =
     0
```

Use `assume` to set an assumption that `x` is nonnegative. Now the expression `sqrt(x^2)` evaluates to `x`, and `logical` returns `1`:

```
assume(x >= 0)
logical(x == sqrt(x^2))

ans =
     1
```

Note that `logical` typically ignores assumptions on variables:

```
syms x
assume(x == 5)
logical(x == 5)

ans =
     0
```

To compare expressions taking into account assumptions on their variables, use `isAlways`:

```
isAlways(x == 5)

ans =
     1
```

For further computations, clear the assumption on `x`:

```
syms x clear
```

Do not use `logical` to check equations and inequalities that require simplification or mathematical transformations. For such equations and inequalities, `logical` might return unexpected results. For example, `logical` does not recognize mathematical equivalence of these expressions:

```
syms x
logical(sin(x)/cos(x) == tan(x))

ans =
     0
```

`logical` also does not realize that this inequality is invalid:

```
logical(sin(x)/cos(x) ~= tan(x))

ans =
     1
```

To test the validity of equations and inequalities that require simplification or mathematical transformations, use `isAlways`:

```
isAlways(sin(x)/cos(x) == tan(x))

ans =
     1

isAlways(sin(x)/cos(x) ~= tan(x))

ans =
     0
```

See Also

`assume` | `assumeAlso` | `assumptions` | `isAlways` | `isequaln` | `sym` | `syms`

Concepts

- "Assumptions on Symbolic Objects" on page 1-35
- "Clear Assumptions and Reset the Symbolic Engine" on page 3-43

lt

Purpose

Define less than relation

Syntax

```
A < B
lt(A,B)
```

Description

`A < B` creates a less than relation.

`lt(A,B)` is equivalent to `A < B`.

Tips

- If `A` and `B` are both numbers, then `A < B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A < B` returns a symbolic less than relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A < B` returns an array of relations `A(i,j,...)<B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if `A` is a variable (for example, `x`), and `B` is an *m*-by-*n* matrix, then `A` is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x < i` becomes `x < 0`, and `x < 3 + 2*i` becomes `x < 3`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

Use `assume` and the relational operator < to set the assumption that `x` is less than 3:

```
syms x
assume(x < 3)
```

Solve this equation. The solver takes into account the assumption on variable `x`, and therefore returns these two solutions.

```
solve((x - 1)*(x - 2)*(x - 3)*(x - 4) == 0, x)
```

```
ans =
 1
 2
```

Use the relational operator < to set this condition on variable `x`:

```
syms x
cond = abs(sin(x)) + abs(cos(x)) < 6/5;
```

Use the `for` loop with step $\pi/24$ to find angles from 0 to π that satisfy that condition:

```
for i = 0:sym(pi/24):sym(pi)
  if subs(cond, x, i)
    disp(i)
  end
end
```

```
0
pi/24
(11*pi)/24
pi/2
(13*pi)/24
(23*pi)/24
pi
```

See Also `eq` | `ge` | `gt` | `isAlways` | `le` | `logical` | `ne`

Concepts

- “Set Assumptions” on page 1-35

Purpose

LU factorization

Syntax

```
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,p] = lu(A,'vector')
[L,U,p,q] = lu(A,'vector')
[L,U,P,Q,R] = lu(A)
[L,U,p,q,R] = lu(A,'vector')
lu(A)
```

Description

`[L,U] = lu(A)` returns an upper triangular matrix `U` and a matrix `L`, such that `A = L*U`. Here, `L` is a product of the inverse of the permutation matrix and a lower triangular matrix.

`[L,U,P] = lu(A)` returns an upper triangular matrix `U`, a lower triangular matrix `L`, and a permutation matrix `P`, such that `P*A = L*U`.

`[L,U,p] = lu(A,'vector')` returns the permutation information as a vector `p`, such that `A(p,:)  = L*U`.

`[L,U,p,q] = lu(A,'vector')` returns the permutation information as two row vectors `p` and `q`, such that `A(p,q) = L*U`.

`[L,U,P,Q,R] = lu(A)` returns an upper triangular matrix `U`, a lower triangular matrix `L`, permutation matrices `P` and `Q`, and a scaling matrix `R`, such that `P*(R\A)*Q = L*U`.

`[L,U,p,q,R] = lu(A,'vector')` returns the permutation information in two row vectors `p` and `q`, such that `R(:,p)\A(:,q) = L*U`.

`lu(A)` returns the matrix that contains the strictly lower triangular matrix `L` (the matrix without its unit diagonal) and the upper triangular matrix `U` as submatrices. Thus, `lu(A)` returns the matrix `U + L - eye(size(A))`, where `L` and `U` are defined as `[L,U,P] = lu(A)`. The matrix `A` must be square.

Tips

- Calling `lu` for numeric arguments that are not symbolic objects invokes the MATLAB `lu` function.

- The `thresh` option supported by the MATLAB `lu` function does not affect symbolic inputs.
- If you use `'matrix'` instead of `'vector'`, then `lu` returns permutation matrices, as it does by default.
- `L` and `U` are nonsingular if and only if `A` is nonsingular. `lu` also can compute the LU factorization of a singular matrix `A`. In this case, `L` or `U` is a singular matrix.
- Most algorithms for computing LU factorization are variants of Gaussian elimination.

Input Arguments

A

Square or rectangular symbolic matrix.

'vector'

Flag that prompts `lu` to return the permutation information in row vectors.

Output Arguments

L

Lower triangular matrix or a product of the inverse of the permutation matrix and a lower triangular matrix.

U

Upper triangular matrix.

P

Permutation matrix.

p

Row vector.

q

Row vector.

Q

Permutation matrix.

R

Diagonal scaling matrix.

Definitions

LU Factorization of a Matrix

LU factorization expresses an m-by-n matrix A as $P*A = L*U$. Here, L is an m-by-m lower triangular matrix, U is an m-by-n upper triangular matrix, and P is a permutation matrix.

Permutation Vector

Permutation vector p contains numbers corresponding to row exchanges in the matrix A. For an m-by-m matrix, p represents the following permutation matrix with indices i and j ranging from 1 to m:

$$P_{ij} = \delta_{p_i,j} = \begin{cases} 1 \text{ if } j = p_i \\ 0 \text{ if } j \neq p_i \end{cases}$$

Examples

Compute the LU factorization of this matrix. Because these numbers are not symbolic objects, you get floating-point results.

```
[L, U] = lu([2 -3 -1; 1/2 1 -1; 0 1 -1])

L =
    1.0000         0         0
    0.2500    1.0000         0
         0    0.5714    1.0000

U =
    2.0000   -3.0000   -1.0000
         0    1.7500   -0.7500
         0         0   -0.5714
```

Now convert this matrix to a symbolic object, and compute the LU factorization:

```
[L, U] = lu(sym([2 -3 -1; 1/2 1 -1; 0 1 -1]))

L =
[   1,   0, 0]
[ 1/4,   1, 0]
[   0, 4/7, 1]

U =
[ 2,  -3,   -1]
[ 0, 7/4, -3/4]
[ 0,   0, -4/7]
```

Compute the LU factorization returning the lower and upper triangular matrices and the permutation matrix:

```
syms a;
[L, U, P] = lu(sym([0 0 a; a 2 3; 0 a 2]))

L =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]

U =
[ a, 2, 3]
[ 0, a, 2]
[ 0, 0, a]

P =
[ 0, 1, 0]
[ 0, 0, 1]
[ 1, 0, 0]
```

Use the `'vector'` flag to return the permutation information as a vector:

```
syms a;
A = [0 0 a; a 2 3; 0 a 2];
[L, U, p] = lu(A, 'vector')

L =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]

U =
[ a, 2, 3]
[ 0, a, 2]
[ 0, 0, a]

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

Use `isAlways` to check that `A(p,:)  = L*U`:

```
isAlways(A(p,:) == L*U)

ans =
     1     1     1
     1     1     1
     1     1     1
```

Restore the permutation matrix `P` from the vector `p`:

```
P = zeros(3, 3);
for i = 1:3
    P(i, p(i)) = 1;
end;
P

P =
     0     1     0
     0     0     1
     1     0     0
```

Compute the LU factorization of this matrix returning the permutation information in the form of two vectors `p` and `q`:

```
syms a
A = [a, 2, 3*a; 2*a, 3, 4*a; 4*a, 5, 6*a];
[L, U, p, q] = lu(A, 'vector')

L =
[ 1, 0, 0]
[ 2, 1, 0]
[ 4, 3, 1]

U =
[ a,  2,  3*a]
[ 0, -1, -2*a]
[ 0,  0,    0]

p =
[ 1, 2, 3]

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

Use `isAlways` to check that `A(p, q) = L*U`:

```
isAlways(A(p, q) == L*U)

ans =
     1     1     1
     1     1     1
     1     1     1
```

Compute the LU factorization of this matrix returning the lower and upper triangular matrices, permutation matrices, and the scaling matrix:

```
syms a;
A = [0, a; 1/a, 0; 0, 1/5; 0,-1];
[L, U, P, Q, R] = lu(A)

L =
[ 1,        0, 0, 0]
[ 0,        1, 0, 0]
[ 0, 1/(5*a), 1, 0]
[ 0,     -1/a, 0, 1]

U =
[ 1/a, 0]
[   0, a]
[   0, 0]
[   0, 0]

P =
[ 0, 1, 0, 0]
[ 1, 0, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]

Q =
[ 1, 0]
[ 0, 1]

R =
[ 1, 0, 0, 0]
[ 0, 1, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]
```

Use `isAlways` to check that `P*(R\A)*Q = L*U`:

```
isAlways(P*(R\A)*Q == L*U)

ans =
     1     1
     1     1
     1     1
     1     1
```

Compute the LU factorization of this matrix using the `'vector'` flag to return the permutation information as vectors `p` and `q`. Also compute the scaling matrix `R`:

```
syms a;
A = [0, a; 1/a, 0; 0, 1/5; 0,-1];
[L, U, p, q, R] = lu(A, 'vector')

L =
[ 1,        0, 0, 0]
[ 0,        1, 0, 0]
[ 0, 1/(5*a), 1, 0]
[ 0,     -1/a, 0, 1]

U =
[ 1/a, 0]
[   0, a]
[   0, 0]
[   0, 0]

p =
[ 2, 1, 3, 4]

q =
[ 1, 2]

R =
[ 1, 0, 0, 0]
[ 0, 1, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]
```

Use `isAlways` to check that `R(:,p)\A(:,q) = L*U`:

```
isAlways(R(:,p)\A(:,q) == L*U)

ans =
     1     1
     1     1
     1     1
     1     1
```

Call the `lu` function for this matrix:

```
syms a;
A = [0 0 a; a 2 3; 0 a 2];
lu(A)

ans =
[ a, 2, 3]
[ 0, a, 2]
[ 0, 0, a]
```

Verify that the resulting matrix is equal to `U + L - eye(size(A))`, where `L` and `U` are defined as `[L,U,P] = lu(A)`:

```
[L,U,P] = lu(A);
U + L - eye(size(A))

ans =
[ a, 2, 3]
[ 0, a, 2]
[ 0, 0, a]
```

See Also

`chol | eig | isAlwayslinalg::factorLU | lu | svd | vpa`

matlabFunction

Purpose

Convert symbolic expression to function handle or file

Syntax

```
g = matlabFunction(f)
g = matlabFunction(f1,...,fN)
g = matlabFunction(f,Name,Value)
g = matlabFunction(f1,...,fN,Name,Value)
```

Description

`g = matlabFunction(f)` converts the symbolic expression or function `f` to a MATLAB function with the handle `g`.

`g = matlabFunction(f1,...,fN)` converts a vector of the symbolic expressions or functions `f1,...,fN` to a MATLAB function with multiple outputs. The function handle is `g`.

`g = matlabFunction(f,Name,Value)` converts the symbolic expression or function `f` to a MATLAB function using additional options specified by one or more `Name,Value` pair arguments.

`g = matlabFunction(f1,...,fN,Name,Value)` converts a vector of the symbolic expressions or functions `f1,...,fN` to a MATLAB function with multiple outputs using additional options specified by one or more `Name,Value` pair arguments.

Tips

- 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 conversion results. To verify the results, execute the resulting function.
- When you use the `file` argument, use `rehash` to make the generated function available immediately. `rehash` updates the MATLAB list of known files for directories on the search path.

Input Arguments

f

Symbolic expression or function.

f1,...,fN

Vector of symbolic expressions or functions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'file'

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 up 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.

> **Default:** If the value string is empty, `matlabFunction` generates an anonymous function. If the string does not end in `.m`, the function appends `.m`.

'outputs'

Specify the names of output variables. The value must be a cell array of strings.

> **Default:** The 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 a number, for example, `out3`.

'vars'

Specify the order of the input variables or symbolic vectors in the resulting function handle or the file. 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 must equal or exceed the number of free variables in `f`.

> **Default:** When converting symbolic expressions, the order is alphabetical. When converting symbolic functions, the input arguments appear in front of other variables. Other variables are sorted alphabetically.

Output Arguments

g

MATLAB function handle.

Examples

Convert this symbolic expression to a MATLAB function with the handle `ht`:

```
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)
```

Create this symbolic function:

```
syms x y
f(x, y) = x^3 + y^3;
```

Convert `f` to a MATLAB function:

```
ht = matlabFunction(f)

ht =
    @(x,y)x.^3+y.^3
```

Convert this expression to a MATLAB function generating the file `myfile` that contains the function:

```
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);
```

Convert this expression to a MATLAB function using an empty string to represent a path to the file. An empty string causes `matlabFunction` to generate an anonymous function:

```
syms x y z
r = x^2 + y^2 + z^2;
f = matlabFunction(log(r)+r^(-1/2),'file','')

f =
    @(x,y,z)log(x.^2+y.^2+z.^2)+1.0./sqrt(x.^2+y.^2+z.^2)
```

When converting this expression to a MATLAB function, specify the order of the input variables:

```
syms x y z
r = x^2 + y^2 + z^2;
matlabFunction(r, 'file', 'my_function',...
'vars', [y z x]);
```

The created `my_function` accepts variables in the required order:

```
function r = my_function(y,z,x)
%MY_FUNCTION
%    R = MY_FUNCTION(Y,Z,X)

r = x.^2 + y.^2 + z.^2;
```

When converting this expression to a MATLAB function, specify its second input argument as a vector:

```
syms x y z t
r = (x^2 + y^2 + z^2)*exp(-t);
matlabFunction(r, 'file', 'my_function',...
'vars', {t, [x y z]});
```

The resulting function operates on vectors:

```
function r = my_function(t,in2)
%MY_FUNCTION
%    R = MY_FUNCTION(T,IN2)

x = in2(:,1);
y = in2(:,2);
z = in2(:,3);
r = exp(-t).*(x.^2+y.^2+z.^2);
```

When converting this expression to a MATLAB function, 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', 'my_function',...
'outputs', {'name1','name2'});
```

The generated function returns `name1` and `name2`:

```
function [name1,name2] = my_function(x,y,z)
%MY_FUNCTION
%    [NAME1,NAME2] = MY_FUNCTION(X,Y,Z)

t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
name1 = t2+t3+t4;
if nargout > 1
    name2 = t2-t3-t4;
end
```

Convert this MuPAD expression to a MATLAB function:

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

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

```
function f = my_function(x,y)
%MY_FUNCTION
%    F = MY_FUNCTION(X,Y)
```

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

See Also

`ccode` | `evalin` | `feval` | `fortran` | `rehash` | `matlabFunctionBlock` | `simscapeEquation` | `subs` | `sym2poly`

Concepts

- "Generate MATLAB Functions" on page 2-131
- "Create MATLAB Functions from MuPAD Expressions" on page 3-48

Purpose

Convert symbolic expression to MATLAB Function block

Note `emlBlock` will be removed in a future version. Use `matlabFunctionBlock` instead.

Syntax

```
matlabFunctionBlock(block,f)
matlabFunctionBlock(block,f1,...,fN)
matlabFunctionBlock(block,f,Name,Value)
matlabFunctionBlock(block,f1,...,fN,Name,Value)
```

Description

`matlabFunctionBlock(block,f)` converts the symbolic expression or function `f` to a MATLAB Function block that you can use in Simulink models. Here, `block` specifies the name of the block that you create or modify.

`matlabFunctionBlock(block,f1,...,fN)` converts a vector of the symbolic expressions or functions `f1,...,fN` to a MATLAB Function block with multiple outputs.

`matlabFunctionBlock(block,f,Name,Value)` converts the symbolic expression or function `f` to a MATLAB Function block using additional options specified by one or more `Name,Value` pair arguments.

`matlabFunctionBlock(block,f1,...,fN,Name,Value)` converts a vector of the symbolic expressions or functions `f` to a MATLAB Function block with multiple outputs using additional options specified by one or more `Name,Value` pair arguments.

Tips

- To convert a MuPAD expression or function to a MATLAB Function block, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine,'MuPAD_Function',x1,...,xn)`. `matlabFunctionBlock` 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 conversion results. 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 MATLAB Functions Supported for Code Generation.

Input Arguments

f

Symbolic expression or function.

f1,...,fN

Vector of symbolic expressions or functions.

block

String specifying the block name that you create or modify.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'functionName'

Specify the name of the function. The value must be a string.

> **Default:** The value coincides with the block name.

'outputs'

Specify the names of output ports. The value must be a cell array of strings. The number of entries must equal or exceed the number of free variables in `f`.

> **Default:** The name of an output port consists of the word `out` followed by the output port number, for example, `out3`.

'vars'

Specify the order of the variables and the corresponding input ports of a block. The value must be either a cell array of strings or symbolic arrays, or a vector of symbolic variables. The number of entries must equal or exceed the number of free variables in `f`.

> **Default:** When converting symbolic expressions, the order is alphabetical. When converting symbolic functions, the input arguments appear in front of other variables. Other variables are sorted alphabetically.

Examples

Before you can convert a symbolic expression to a MATLAB Function block, create an empty model or open an existing one:

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

Use `matlabFunctionBlock` to create the block `my_block` containing the symbolic expression:

```
syms x y z
f = x^2 + y^2 + z^2;
matlabFunctionBlock('my_system/my_block',f);
```

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

You can open and edit the resulting block. To open a block, double-click it:

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

f = x.^2 + y.^2 + z.^2;
```

Save and close `my_system`:

```
save_system('my_system');
close_system('my_system');
```

Create this symbolic function:

```
syms x y z
f(x, y, z) = x^2 + y^2 + z^2;
```

Convert `f` to a MATLAB Function block:

```
new_system('my_system');
open_system('my_system');
matlabFunctionBlock('my_system/my_block',f);
```

Generate a block and set the function name to `my_function`:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system');
open_system('my_system');
matlabFunctionBlock('my_system/my_block', x, y, z,'functionName', 'my_fur
```

When generating a block, specify the order of the input variables:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system');
open_system('my_system');
matlabFunctionBlock('my_system/my_block', x, y, z, 'vars', [y z x])
```

When generating a block, rename the output variables and the corresponding ports:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system');
```

```
open_system('my_system');
matlabFunctionBlock('my_system/my_block', x, y, z, 'outputs',{'name1',
```

Call `matlabFunctionBlock` using several options simultaneously:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system');
open_system('my_system');
matlabFunctionBlock('my_system/my_block', x, y, z,...
'functionName', 'my_function','vars', [y z x],...
'outputs',{'name1','name2','name3'})
```

Convert this MuPAD expression to a MATLAB Function block:

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

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

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

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

See Also

`ccode` | `evalin` | `feval` | `fortran` | `matlabFunction` | `simscapeEquation` | `subs` | `sym2poly`

Concepts

- "Generate MATLAB Function Blocks" on page 2-136
- "Create MATLAB Function Blocks from MuPAD Expressions" on page 3-52

mfun

Purpose

Numeric evaluation of special mathematical function

Syntax

```
mfun('function',par1,par2,par3,par4)
```

Description

`mfun('function',par1,par2,par3,par4)` numerically evaluates one of the special mathematical functions listed in "Syntax and Definitions of mfun Special Functions" on page 2-143. 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 4-389.

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 the Fresnel cosine integral:

```
mfun('FresnelC',0:4)
```

The result is:

```
ans =
0    0.7799    0.4883    0.6057    0.4984
```

Evaluate the 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)`
			$n \geq 0$
			 $\lvert t \rvert < \frac{\pi}{2}$

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)`
			$n \geq 0$
			 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)`	$0 < x \le \infty$. `a` is real, $-\infty < a < \infty$. `k` is real, $0 < k < 1$.
Incomplete gamma function	$\Gamma(a,z)=\int_z^{\infty} e^{-t}\cdot t^{a-1}dt$	`GAMMA(z1,z2)` `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.

References

[1] Abramowitz, M. and I.A., Stegun, *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables.* New York: Dover, 1972.

See Also

`mfun`

Purpose

Minimal polynomial of matrix

Syntax

```
minpoly(A)
minpoly(A,var)
```

Description

`minpoly(A)` returns a vector of the coefficients of the minimal polynomial of `A`. If `A` is a symbolic matrix, `minpoly` returns a symbolic vector. Otherwise, it returns a vector with elements of type `double`.

`minpoly(A,var)` returns the minimal polynomial of `A` in terms of `var`.

Input Arguments

A

Matrix.

var

Free symbolic variable.

> **Default:** If you do not specify `var`, `minpoly` returns a vector of coefficients of the minimal polynomial instead of returning the polynomial itself.

Definitions

Minimal Polynomial of a Matrix

The minimal polynomial of a square matrix `A` is the monic polynomial $p(x)$ of the least degree, such that $p(A) = 0$.

Examples

Compute the minimal polynomial of the matrix `A` in terms of the variable `x`:

```
syms x;
A = sym([1 1 0; 0 1 0; 0 0 1]);
minpoly(A, x)

ans =
x^2 - 2*x + 1
```

To find the coefficients of the minimal polynomial of `A`, call `minpoly` with one argument:

```
A = sym([1 1 0; 0 1 0; 0 0 1]);
minpoly(A)

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

Find the coefficients of the minimal polynomial of the symbolic matrix `A`. For this matrix, `minpoly` returns the symbolic vector of coefficients:

```
A = sym([0 2 0; 0 0 2; 2 0 0]);
P = minpoly(A)

P =
[ 1, 0, 0, -8]
```

Now find the coefficients of the minimal polynomial of the matrix `B`, all elements of which are double-precision values. Note that in this case `minpoly` returns coefficients as double-precision values:

```
B = [0 2 0; 0 0 2; 2 0 0];
P = minpoly(B)

P =
     1     0     0    -8
```

See Also

`charpoly` | `eig` | `jordan` | `poly2sym` | `sym2poly`

Purpose

Symbolic matrix element-wise 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 named `file` and keeps a handle (pointer) to the notebook in the variable `mphandle`. You also 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 (`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 “Work with Links”.

See Also `getVar` | `mupadwelcome` | `openmn` | `openmu` | `setVar`

Purpose

Start MuPAD interfaces

Syntax

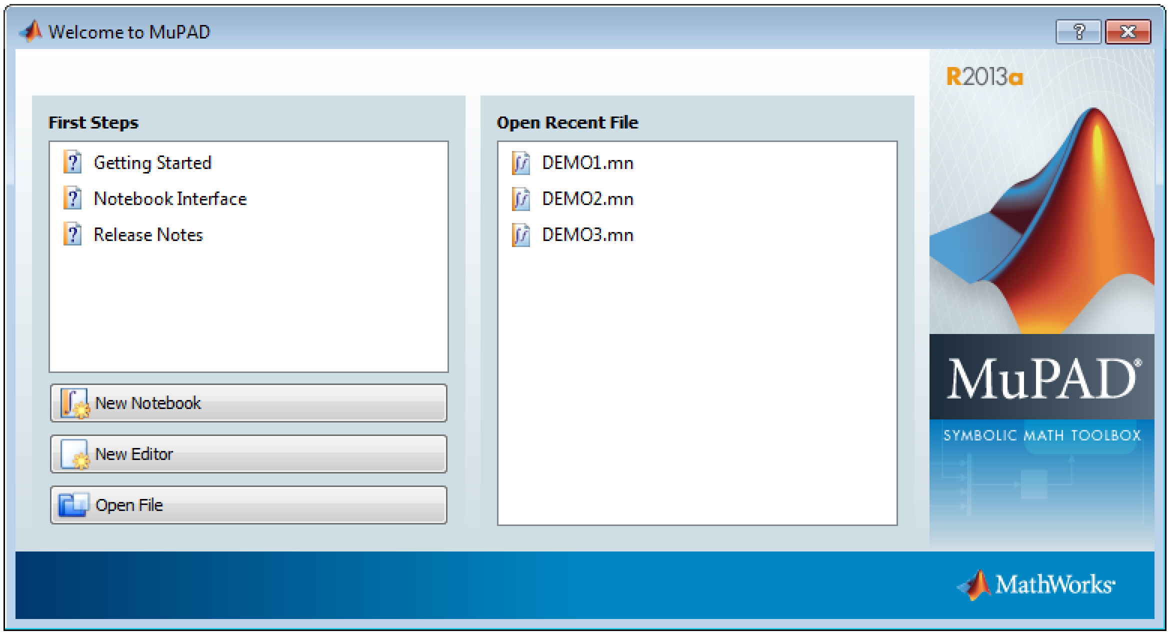
`mupadwelcome`

Description

`mupadwelcome` opens a window that enables you to start various interfaces:

- MuPAD Notebook Interface, for performing calculations
- MATLAB Editor, for writing programs and libraries
- Documentation in the **First Steps** pane, for information and examples

It also enables you to access recent MuPAD files or browse for files.



See Also

`mupad`

How To

- “Create, Open, and Save MuPAD Notebooks” on page 3-3

Purpose

Binomial coefficient

Syntax

```
nchoosek(n,k)
```

Description

`nchoosek(n,k)` returns the binomial coefficient of `n` and `k`.

Tips

- Calling `nchoosek` for numbers that are not symbolic objects invokes the MATLAB `nchoosek` function.
- If one or both parameters are complex or negative numbers, convert these numbers to symbolic objects using `sym`, and then call `nchoosek` for those symbolic objects.

Input Arguments

n

Symbolic number, variable or expression.

k

Symbolic number, variable or expression.

Definitions

Binomial Coefficient

If n and k are integers and $0 \le k \le n$, the binomial coefficient is defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

For complex numbers, the binomial coefficient is defined via the `gamma` function:

$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}$$

Examples

Compute the binomial coefficients for these expressions:

```
syms n
[nchoosek(n, n), nchoosek(n, n + 1), nchoosek(n, n - 1)]

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

If one or both parameters are negative numbers, convert these numbers to symbolic objects:

```
[nchoosek(sym(-1), 3), nchoosek(sym(-7), 2),
nchoosek(sym(-5), -5)]

ans =
[ -1, 28, 1]
```

If one or both parameters are complex numbers, convert these numbers to symbolic objects:

```
[nchoosek(sym(i), 3), nchoosek(sym(i), i),
nchoosek(sym(i), i + 1)]

ans =
[ 1/2 + i/6, 1, 0]
```

Differentiate the binomial coefficient:

```
syms n
diff(nchoosek(n, 2))

ans =
-(psi(n - 1) - psi(n + 1))*nchoosek(n, 2)
```

Expand the binomial coefficient:

```
syms n k
expand(nchoosek(n, k))

ans =
-(n*gamma(n))/(k^2*gamma(k)*gamma(n - k) -
k*n*gamma(k)*gamma(n - k))
```

Algorithms

If $k < 0$ or $n - k < 0$, `nchoosek(n,k)` returns 0.

If one or both arguments are complex, `nchoosek` uses the formula representing the binomial coefficient via the `gamma` function.

See Also

`beta` | `gamma` | `factorial` | `mfun` | `mfunlist` | `psi`

How To

- "Special Functions of Applied Mathematics" on page 2-142

ne

Purpose

Define inequality

Syntax

```
A ~= B
ne(A,B)
```

Description

`A ~= B` creates a symbolic inequality.

`ne(A,B)` is equivalent to `A ~= B`.

Tips

- If `A` and `B` are both numbers, then `A ~= B` compares `A` and `B` and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A ~= B` returns a symbolic inequality. You can use that inequality as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both `A` and `B` are arrays, then these arrays must have the same dimensions. `A ~= B` returns an array of inequalities `A(i,j,...)~=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if `A` is a variable (for example, `x`), and `B` is an *m*-by-*n* matrix, then `A` is expanded into *m*-by-*n* matrix of elements, each set to `x`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

Examples

Use `assume` and the relational operator `~=` to set the assumption that `x` does not equal to 5:

```
syms x
assume(x ~= 5)
```

Solve this equation. The solver takes into account the assumption on variable `x`, and therefore returns only one solution.

```
solve((x - 5)*(x - 6) == 0, x)
```

```
ans =
6
```

Alternatives

You can also define inequality using `eq` (or its shortcut `==`) and the logical negation `not` (or `~`). Thus, `A ~= B` is equivalent to `~(A == B)`.

See Also

`eq | ge | gt | isAlways | le | logical | lt`

Concepts

- "Set Assumptions" on page 1-35

norm

Purpose

Norm of matrix or vector

Syntax

```
norm(A)
norm(A,p)
norm(V)
norm(V,P)
```

Description

`norm(A)` returns the `2`-norm of matrix `A`.

`norm(A,p)` returns the `p`-norm of matrix `A`.

`norm(V)` returns the `2`-norm of vector `V`.

`norm(V,P)` returns the `P`-norm of vector `V`.

Tips

- Calling `norm` for a numeric matrix that is not a symbolic object invokes the MATLAB `norm` function.

Input Arguments

A

Symbolic matrix.

p

One of these values `1`, `2`, `inf`, or `'fro'`.

- `norm(A,1)` returns the `1`-norm of `A`.
- `norm(A,2)` or `norm(A)` returns the `2`-norm of `A`.
- `norm(A,inf)` returns the infinity norm of `A`.
- `norm(A,'fro')` returns the Frobenius norm of `A`.

Default: `2`

V

Symbolic vector.

P

- `norm(V,P)` is computed as `sum(abs(V).^P)^(1/P)` for `1<=P<inf`.
- `norm(V)` computes the `2`-norm of `V`.
- `norm(A,inf)` is computed as `max(abs(V))`.
- `norm(A,-inf)` is computed as `min(abs(V))`.

  Default: `2`

Definitions

1-norm of a Matrix

The `1`-norm of an m-by-n matrix A is defined as follows:

$$\|A\|_1 = \max_j\left(\sum_{i=1}^{m}|A_{ij}|\right), \quad \text{where } j = 1\ldots n$$

2-norm of a Matrix

The `2`-norm of an m-by-n matrix A is defined as follows:

$$\|A\|_2 = \sqrt{\text{max eigenvalue of } \mathrm{A}^H A}$$

The `2`-norm is also called the spectral norm of a matrix.

Frobenius Norm of a Matrix

The Frobenius norm of an m-by-n matrix A is defined as follows:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\left(\sum_{j=1}^{n}|A_{ij}|^2\right)}$$

Infinity Norm of a Matrix

The infinity norm of an m-by-n matrix A is defined as follows:

$$\|A\|_\infty = \max\left(\sum_{j=1}^{n}\left|A_{1j}\right|, \sum_{j=1}^{n}\left|A_{2j}\right|, \ldots, \sum_{j=1}^{n}\left|A_{mj}\right|\right)$$

P-norm of a Vector

The P-norm of a 1-by-*n* or *n*-by-1 vector *V* is defined as follows:

$$\|V\|_P = \left(\sum_{i=1}^{n}\left|V_i\right|^P\right)^{1/P}$$

Here *n* must be an integer greater than 1.

Frobenius Norm of a Vector

The Frobenius norm of a 1-by-*n* or *n*-by-1 vector *V* is defined as follows:

$$\|V\|_F = \sqrt{\sum_{i=1}^{n}\left|V_i\right|^2}$$

The Frobenius norm of a vector coincides with its 2-norm.

Infinity and Negative Infinity Norm of a Vector

The infinity norm of a 1-by-*n* or *n*-by-1 vector *V* is defined as follows:

$$\|V\|_\infty = \max\left(\left|V_i\right|\right), \text{ where } i = 1\ldots n$$

The negative infinity norm of a 1-by-*n* or *n*-by-1 vector *V* is defined as follows:

$$\|V\|_{-\infty} = \min\left(\left|V_i\right|\right), \text{ where } i = 1\ldots n$$

Examples

Compute the 2-norm of the inverse of the 3-by-3 magic square A:

```
A = inv(sym(magic(3)));
norm2 = norm(A)
```

```
norm2 =
3^(1/2)/6
```

Use vpa to approximate the result with 20-digit accuracy:

```
vpa(norm2, 20)

ans =
0.28867513459481288225
```

Compute the 1-norm, Frobenius norm, and infinity norm of the inverse of the 3-by-3 magic square A:

```
A = inv(sym(magic(3)));
norm1 = norm(A, 1)
normf = norm(A, 'fro')
normi = norm(A, inf)

norm1 =
16/45

normf =
391^(1/2)/60

normi =
16/45
```

Use vpa to approximate these results 20-digit accuracy:

```
vpa(norm1, 20)
vpa(normf, 20)
vpa(normi, 20)

ans =
0.35555555555555555556

ans =
0.32956199888808647519

ans =
0.35555555555555555556
```

Compute the 1-norm, 2-norm, and 3-norm of the column vector `V = [Vx; Vy; Vz]`:

```
syms Vx Vy Vz
V = [Vx; Vy; Vz];
norm1 = norm(V, 1)
norm2 = norm(V)
norm3 = norm(V, 3)

norm1 =
abs(Vx) + abs(Vy) + abs(Vz)

norm2 =
(abs(Vx)^2 + abs(Vy)^2 + abs(Vz)^2)^(1/2)

norm3 =
(abs(Vx)^3 + abs(Vy)^3 + abs(Vz)^3)^(1/3)
```

Compute the infinity norm, negative infinity norm, and Frobenius norm of `V`:

```
normi = norm(V, inf)
normni = norm(V, -inf)
normf = norm(V, 'fro')

normi =
max(abs(Vx), abs(Vy), abs(Vz))

normni =
min(abs(Vx), abs(Vy), abs(Vz))

normf =
(abs(Vx)^2 + abs(Vy)^2 + abs(Vz)^2)^(1/2)
```

See Also

`cond` | `equationsToMatrix` | `inv` | `linsolve` | `rank`

Purpose

Logical NOT for symbolic expressions

Syntax

```
~A
not(A)
```

Description

`~A` represents the logical negation. `~A` is true when `A` is false and vice versa.

`not(A)` is equivalent to `~A`.

Tips

- If you call `simplify` for a logical expression that contains symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.

Input Arguments

A

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

Examples

Create this logical expression using ~:

```
syms x y
xy = ~(x > y);
```

Use `assume` to set the corresponding assumption on variables `x` and `y`:

```
assume(xy);
```

Verify that the assumption is set:

```
assumptions

ans =
not y < x
```

Create this logical expression using logical operators ~ and `&`:

```
syms x
range = abs(x) < 1 & ~(abs(x) < 1/3);
```

Replace variable `x` with these numeric values. Note that `subs` does not evaluate these inequalities to logical `1` or `0`.

```
x1 = subs(range, x, 0)
x2 = subs(range, x, 2/3)

x1 =
0 < 1 and not 0 < 1/3

x2 =
2/3 < 1 and not 2/3 < 1/3
```

To evaluate these inequalities to logical `1` or `0`, use `logical` or `isAlways`:

```
logical(x1)
isAlways(x2)

ans =
     0

ans =
     1
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)

s1 =
FALSE

s2 =
TRUE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)

ans =
     0

ans =
     1
```

See Also all | and | any | isAlways | logical | or | xor

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)
nullityOfA = size(Z, 2)
A*Z
```

The results are:

```
Z =
 -1
 -3
  3
  1

nullityOfA =
     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`

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]
```

Purpose

Convert higher-order differential equations to systems of first-order differential equations

Syntax

```
V = odeToVectorField(eqn1,...,eqnN)
[V,Y] = odeToVectorField(eqn1,...,eqnN)
```

Description

`V = odeToVectorField(eqn1,...,eqnN)` converts higher-order differential equations `eqn1,...,eqnN` to a system of first-order differential equations. This syntax returns a symbolic vector representing the resulting system of first-order differential equations.

`[V,Y] = odeToVectorField(eqn1,...,eqnN)` converts higher-order differential equations `eqn1,...,eqnN` to a system of first-order differential equations. This syntax returns two symbolic vectors. The first vector represents the resulting system of first-order differential equations. The second vector shows the substitutions made during conversion.

Tips

- The names of symbolic variables used in differential equations should not contain the letter `D` because `odeToVectorField` assumes that `D` is a differential operator and any character immediately following `D` is a dependent variable.
- To generate a MATLAB function for the resulting system of first-order differential equations, use `matlabFunction` with `V` as an input. Then, you can use the generated MATLAB function as an input for the MATLAB numerical solvers `ode23` and `ode45`.
- The highest-order derivatives must appear in `eqn1,...,eqnN` linearly. For example, `odeToVectorField` can convert these equations:
  - $y''(t) = -t^2$
  - $y*y''(t) = -t^2$. `odeToVectorField` can convert this equation because it can be rewritten as $y''(t) = -t^2/y$.

  However, it cannot convert these equations:
  - $y''(t)^2 = -t^2$

- $\sin(y''(t)) = -t^2$

Input Arguments

eqn1,...,eqnN

Symbolic equations, strings separated by commas and representing a system of ordinary differential equations, or array of symbolic equations or strings. Each equation or string represents an ordinary differential equation.

When representing `eqn` as a symbolic equation, you must create a symbolic function, for example `y(x)`. Here `x` is an independent variable for which you solve an ordinary differential equation. Use the == operator to create an equation. Use the `diff` function to indicate differentiation. For example, to convert `d`^{2}`y(x)/dt`^{2} `= x*y(x)`, use:

```
syms y(x)
V = odeToVectorField(diff(y, 2) == x*y)
```

When representing `eqn` as a string, use the letter `D` to indicate differentiation. By default, `odeToVectorField` assumes that the independent variable is `t`. Thus, `Dy` means `dy/dt`. You can specify the independent variable. The letter `D` followed by a digit indicates repeated differentiation. Any character immediately following a differentiation operator is a dependent variable. For example, to convert `d`^{2}`y(x)/dt`^{2} `= x*y(x)`, enter:

```
V = odeToVectorField('D2y = x*y','x')
```

or

```
V = odeToVectorField('D2y == x*y','x')
```

Output Arguments

V

Symbolic vector representing the system of first-order differential equations. Each element of this vector is the right side of the first-order differential equation $Y[i]' = V[i]$.

Y

Symbolic vector representing the substitutions made when converting the input equations `eqn1,...,eqnN` to the elements of `V`.

Examples

Convert this fifth-order differential equation to a system of first-order differential equations:

```
syms y(t)
V = odeToVectorField(t^3*diff(y, 5) + 2*t*diff(y,
4) + diff(y, 2) + y^2 == -3*t)

V =
                                  Y[2]
                                  Y[3]
                                  Y[4]
                                  Y[5]
 -(3*t + Y[1]^2 + 2*t*Y[5] + Y[3])/t^3
```

Convert this system of first- and second-order differential equations to a system of first-order differential equations. To see the substitutions that `odeToVectorField` makes for this conversion, use two output arguments:

```
syms f(t) g(t)
[V,Y] = odeToVectorField(diff(f, 2) == f + g,
diff(g) == -f + g)

V =
 Y[1] - Y[2]
        Y[3]
 Y[1] + Y[2]

Y =
  g
  f
 Df
```

Convert this second-order differential equation to a system of first-order differential equations:

```
syms y(t)
V = odeToVectorField(diff(y, 2) == (1 - y^2)*diff(y) - y)
```

```
V =
                       Y[2]
 - (Y[1]^2 - 1)*Y[2] - Y[1]
```

Generate a MATLAB function from this system of first-order differential equations using `matlabFunction` with `V` as an input:

```
M = matlabFunction(V,'vars', {'t','Y'})
```

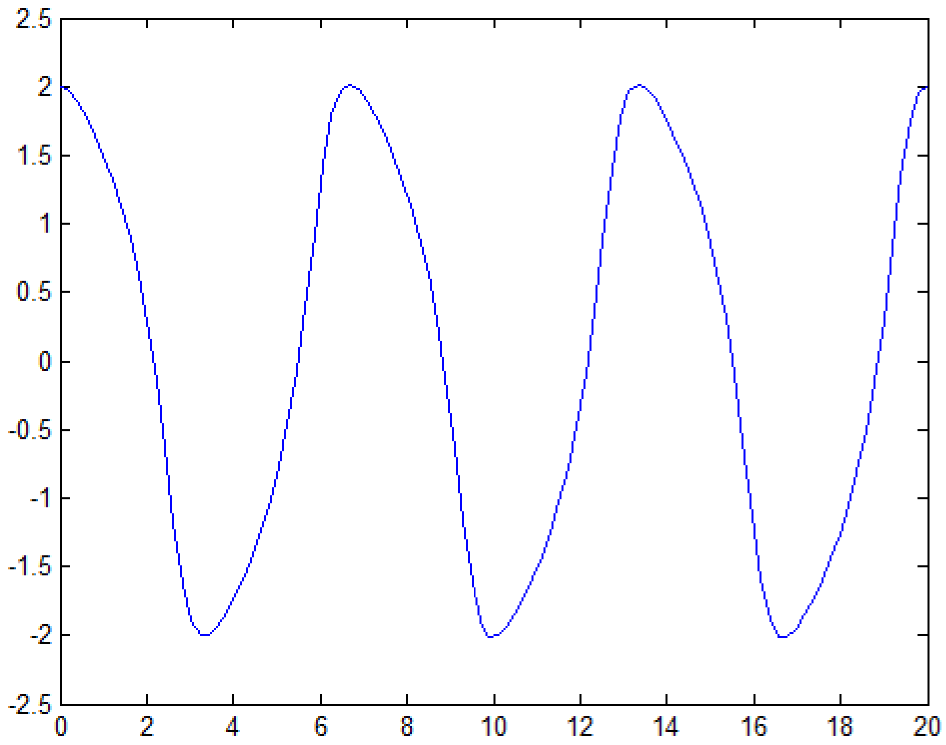
```
M =
    @(t,Y)[Y(2);-(Y(1).^2-1.0).*Y(2)-Y(1)]
```

To solve this system, call the MATLAB `ode45` numerical solver using the generated MATLAB function as an input:

```
sol = ode45(M,[0 20],[2 0]);
```

Plot the solution using `linspace` to generate 100 points in the interval [0,20] and `deval` to evaluate the solution for each point:

```
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y);
```



Convert the second-order differential equation $y''(x) = x$ with the initial condition $y(0) = t$ to a system. Specify the differential equation and initial condition as strings. Also specify that `x` is an independent variable:

```
V = odeToVectorField('D2y = x', 'y(0) = t', 'x')
```

```
V =
 Y[2]
    x
```

If you define equations by strings and do not specify the independent variable, `odeToVectorField` assumes that the independent variable is `t`. This assumption makes the equation $y''(t) = x$ inconsistent with the initial condition $y(0) = t$. In this case, $y''(t) = \mathrm{d}^2t/\mathrm{d}t^2 = 0$, and `odeToVectorField` errors.

Algorithms

To convert an nth-order differential equation

$$a_n(t)y^{(n)} + a_{n-1}(t)y^{(n-1)} + a_{n-2}(t)y^{(n-2)} + \ldots + a_2(t)y'' + a_1(t)y' + a_0(t)y + r(t) = 0$$

into a system of first-order differential equations, make these substitutions:

$$\begin{aligned}
&Y_1 = y\\
&Y_2 = y'\\
&Y_3 = y''\\
&\ldots\\
&Y_{n-1} = y^{(n-2)}\\
&Y_n = y^{(n-1)}
\end{aligned}$$

Using the new variables, you can rewrite the equation as a system of n first-order differential equations:

$$\begin{aligned}
&Y_1' = y' = Y_2\\
&Y_2' = y'' = Y_3\\
&\ldots\\
&Y_{n-1}' = y^{(n-1)} = Y_n\\
&Y_n' = -\frac{a_{n-1}(t)}{a_n(t)}Y_n - \frac{a_{n-2}(t)}{a_n(t)}Y_{n-1} - \ldots - \frac{a_1(t)}{a_n(t)}Y_2 - \frac{a_0(t)}{a_n(t)}Y_1 + \frac{r(t)}{a_n(t)}
\end{aligned}$$

`odeToVectorField` returns the right sides of these equations as the elements of vector `V`.

When you convert a system of higher-order differential equations to a system of first-order differential equations, it can be helpful to see the substitutions that `odeToVectorField` made during the conversion. These substitutions are listed as elements of vector `Y`.

See Also

`dsolve` | `matlabFunction` | `ode23` | `ode45` | `syms`

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.

Examples

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` | `openxvc` | `openxvz`

Purpose

Open MuPAD program file

Syntax

`openmu(file)`

Description

`openmu(file)` opens the MuPAD program file named `file` in the MATLAB Editor. The command `open(file)` accomplishes the same task.

Examples

To open a program file named `yyx.mu` located in the folder `\Documents\Notes` on drive `H:`, enter:

```
openmu('H:\Documents\Notes\yyx.mu');
```

This command opens `yyx.mu` in the MATLAB Editor.

See Also

`mupad` | `open` | `openmn` | `openxvc` | `openxvz`

openxvc

Purpose

Open MuPAD XVC graphics file

Syntax

```
openxvc(file)
```

Description

`openxvc(file)` opens the MuPAD XVC graphics file named `file`.

Input Arguments

file

MuPAD XVC graphics file.

Examples

To open a graphics file named `image1.xvc` in the folder `\Documents\Notes` of drive `H:`, enter:

```
openxvc('H:\Documents\Notes\image1.xvc');
```

See Also

`mupad` | `open` | `openmn` | `openmu` | `openxvz`

Purpose

Open MuPAD XVZ graphics file

Syntax

`openxvz(file)`

Description

`openxvz(file)` opens the MuPAD XVZ graphics file named `file`.

Input Arguments

file

MuPAD XVZ graphics file.

Examples

To open a graphics file named `image1.xvz` in the folder `\Documents\Notes` of drive `H:`, enter:

```
openxvz('H:\Documents\Notes\image1.xvz');
```

See Also

`mupad` | `open` | `openmn` | `openmu` | `openxvc`

Purpose

Logical OR for symbolic expressions

Syntax

```
A | B
or(A,B)
```

Description

`A | B` represents the logical disjunction. `A | B` is true when either `A` or `B` or both are true.

`or(A,B)` is equivalent to `A | B`.

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.

Input Arguments

A

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

B

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

Examples

Combine these symbolic inequalities into the logical expression using `|`:

```
syms x y
xy = x >= 0 | y >= 0;
```

Set the corresponding assumptions on variables `x` and `y` using `assume`:

```
assume(xy)
```

Verify that the assumptions are set:

```
assumptions
```

```
0 <= x or 0 <= y
```

Combine two symbolic inequalities into the logical expression using `|`:

```
syms x
range = x < -1 | x > 1;
```

Replace variable `x` with these numeric values. If you replace `x` with 10, one inequality is valid. If you replace `x` with 0, both inequalities are invalid. Note that `subs` does not evaluate these inequalities to logical `1` or `0`.

```
x1 = subs(range, x, 10)
x2 = subs(range, x, 0)

x1 =
1 < 10 or 10 < -1

x2 =
0 < -1 or 1 < 0
```

To evaluate these inequalities to logical `1` or `0`, use `logical` or `isAlways`:

```
logical(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)

s1 =
TRUE

s2 =
FALSE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)
```

```
ans =
     1

ans =
     0
```

See Also `all | and | any | isAlways | logical | not | xor`

Purpose

Orthonormal basis for range of symbolic matrix

Syntax

```
B = orth(A)
B = orth(A,'real')
B = orth(A,'skipNormalization')
B = orth(A,'real','skipNormalization')
```

Description

`B = orth(A)` computes an orthonormal basis for the range of `A`.

`B = orth(A,'real')` computes an orthonormal basis using a real scalar product in the orthogonalization process.

`B = orth(A,'skipNormalization')` computes a non-normalized orthogonal basis. In this case, the vectors forming the columns of `B` do not necessarily have length 1.

`B = orth(A,'real','skipNormalization')` computes a non-normalized orthogonal basis using a real scalar product in the orthogonalization process.

Tips

- Calling `orth` for numeric arguments that are not symbolic objects invokes the MATLAB `orth` function. Results returned by MATLAB `orth` can differ from results returned by `orth` because these two functions use different algorithms to compute an orthonormal basis. The Symbolic Math Toolbox `orth` function uses the classic Gram-Schmidt orthogonalization algorithm. The MATLAB `orth` function uses the modified Gram-Schmidt algorithm because the classic algorithm is numerically unstable.
- Using `'skipNormalization'` to compute an orthogonal basis instead of an orthonormal basis can speed up your computations.

Input Arguments

A

Symbolic matrix.

'real'

Flag that prompts `orth` to avoid using a complex scalar product in the orthogonalization process.

'skipNormalization'

Flag that prompts `orth` to skip normalization and compute an orthogonal basis instead of an orthonormal basis. If you use this flag, lengths of the resulting vectors (the columns of matrix `B`) are not required to be 1.

Output Arguments

B

Symbolic matrix.

Definitions

Orthonormal Basis

An orthonormal basis for the range of matrix `A` is matrix `B`, such that:

- `B'*B = I`, where `I` is the identity matrix.
- The columns of `B` span the same space as the columns of `A`.
- The number of columns of `B` is the rank of `A`.

Examples

Compute an orthonormal basis of the range of this matrix. Because these numbers are not symbolic objects, you get floating-point results.

```
A = [2 -3 -1; 1 1 -1; 0 1 -1];
B = orth(A)

B =
   -0.9859   -0.1195    0.1168
    0.0290   -0.8108   -0.5846
    0.1646   -0.5729    0.8029
```

Now, convert this matrix to a symbolic object, and compute an orthonormal basis:

```
A = sym([2 -3 -1; 1 1 -1; 0 1 -1]);
B = orth(A)
```

```
B =
[ (2*5^(1/2))/5, -6^(1/2)/6, -(2^(1/2)*15^(1/2))/30]
[     5^(1/2)/5,  6^(1/2)/3,  (2^(1/2)*15^(1/2))/15]
[             0,  6^(1/2)/6,  -(2^(1/2)*15^(1/2))/6]
```

You can use `double` to convert this result to the double-precision numeric form. The resulting matrix differs from the matrix returned by the MATLAB `orth` function because these functions use different versions of the Gram-Schmidt orthogonalization algorithm:

```
double(B)

ans =
    0.8944   -0.4082   -0.1826
    0.4472    0.8165    0.3651
         0    0.4082   -0.9129
```

Verify that `B'*B = I`, where `I` is the identity matrix:

```
B'*B

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

Now, verify that the 2-norm of each column of `B` is 1:

```
norm(B(:, 1))
norm(B(:, 2))
norm(B(:, 3))

ans =
1

ans =
1

ans =
1
```

Compute an orthonormal basis of this matrix using `'real'` to avoid complex conjugates:

```
syms a
A = [a 1; 1 a];
B = orth(A,'real')

B =
[ a/(abs(a)^2 + 1)^(1/2),
-(a^2 - 1)/((a^2 + 1)*(abs(a^2 - 1)^2/abs(a^2 + 1)^2 +...
abs(a*(a^2 - 1))^2/abs(a^2 + 1)^2)^(1/2))]
[ 1/(abs(a)^2 + 1)^(1/2),
(a*(a^2 - 1))/((a^2 + 1)*(abs(a^2 - 1)^2/abs(a^2 + 1)^2 +...
abs(a*(a^2 - 1))^2/abs(a^2 + 1)^2)^(1/2))]
```

Compute an orthogonal basis of this matrix using `'skipNormalization'`:

```
syms a
A = [a 1; 1 a];
B = orth(A,'skipNormalization')

B =
[ a,              -(a^2 - 1)/(a*conj(a) + 1)]
[ 1, -(conj(a) - a^2*conj(a))/(a*conj(a) + 1)]
```

Compute an orthogonal basis of this matrix using `'skipNormalization'` and `'real'`:

```
syms a
A = [a 1; 1 a];
B = orth(A,'skipNormalization','real')
```

```
B =
[ a,    -(a^2 - 1)/(a^2 + 1)]
[ 1, (a*(a^2 - 1))/(a^2 + 1)]
```

Algorithms

`orth` uses the classic Gram-Schmidt orthogonalization algorithm.

See Also

`norm` | `null` | `orth` | `rank` | `svd` `linalg::normalize` | `linalg::orthog`

pinv

Purpose

Moore-Penrose inverse (pseudoinverse) of symbolic matrix

Syntax

```
X = pinv(A)
```

Description

`X = pinv(A)` returns the pseudoinverse of `A`. Pseudoinverse is also called the Moore-Penrose inverse.

Tips

- Calling `pinv` for numeric arguments that are not symbolic objects invokes the MATLAB `pinv` function.
- For an invertible matrix `A`, the Moore-Penrose inverse `X` of `A` coincides with the inverse of `A`.

Input Arguments

A

Symbolic *m*-by-*n* matrix.

Output Arguments

X

Symbolic *n*-by-*m* matrix, such that `A*X*A = A` and `X*A*X = X`.

Definitions

Moore-Penrose Pseudoinverse

The pseudoinverse of an *m*-by-*n* matrix `A` is an *n*-by-*m* matrix `X`, such that `A*X*A = A` and `X*A*X = X`. The matrices `A*X` and `X*A` must be Hermitian.

Examples

Compute the pseudoinverse of this matrix. Because these numbers are not symbolic objects, you get floating-point results.

```
A = [1 1i 3; 1 3 2];
X = pinv(A)
```

```
X =
    0.0729 + 0.0312i   0.0417 - 0.0312i
   -0.2187 - 0.0521i   0.3125 + 0.0729i
    0.2917 + 0.0625i   0.0104 - 0.0938i
```

Now, convert this matrix to a symbolic object, and compute the pseudoinverse:

```
A = sym([1 1i 3; 1 3 2]);
X = pinv(A)

X =
[       7/96 + i/32,     1/24 - i/32]
[ - 7/32 - (5*i)/96, 5/16 + (7*i)/96]
[       7/24 + i/16, 1/96 - (3*i)/32]
```

Check that `A*X*A = A` and `X*A*X = X`:

```
logical(A*X*A == A)

ans =
     1     1     1
     1     1     1

logical(X*A*X == X)

ans =
     1     1
     1     1
     1     1
```

Now, verify that `A*X` and `X*A` are Hermitian matrices:

```
logical(A*X == (A*X)')

ans =
     1     1
     1     1

logical(X*A == (X*A)')

ans =
     1     1     1
     1     1     1
     1      1      1
```

Compute the pseudoinverse of this matrix:

```
syms a;
A = [1 a; -a 1];
X = pinv(A)

X =
[ (a*conj(a) + 1)/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a - conj(a)))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1),
- (a - conj(a))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a*conj(a) + 1))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1)]
[ (a - conj(a))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) +...
(conj(a)*(a*conj(a) + 1))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1),
(a*conj(a) + 1)/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a - conj(a)))/(a^2*conj(a)^2 + a^2
+ conj(a)^2 + 1)]
```

Now, compute the pseudoinverse of A assuming that a is real:

```
assume(a,'real')
A = [1 a; -a 1];
X = pinv(A)

X =
[ 1/(a^2 + 1), -a/(a^2 + 1)]
[ a/(a^2 + 1),  1/(a^2 + 1)]
```

For further computations, remove the assumption:

```
syms a clear
```

See Also

invlinalg::pseudoInverse | rank | pinv | svd

Purpose

Poles of expression or function

Syntax

```
poles(f,var)
P = poles(f,var)
[P,N] = poles(f,var)
[P,N,R] = poles(f,var)
poles(f,var,a,b)
P = poles(f,var,a,b)
[P,N] = poles(f,var,a,b)
[P,N,R] = poles(f,var,a,b)
```

Description

`poles(f,var)` finds nonremovable singularities of `f`. These singularities are called the poles of `f`. Here, `f` is a function of the variable `var`.

`P = poles(f,var)` finds the poles of `f` and assigns them to vector `P`.

`[P,N] = poles(f,var)` finds the poles of `f` and their orders. This syntax assigns the poles to vector `P` and their orders to vector `N`.

`[P,N,R] = poles(f,var)` finds the poles of `f` and their orders and residues. This syntax assigns the poles to vector `P`, their orders to vector `N`, and their residues to vector `R`.

`poles(f,var,a,b)` finds the poles in the interval (`a,b`).

`P = poles(f,var,a,b)` finds the poles of `f` in the interval (`a,b`) and assigns them to vector `P`.

`[P,N] = poles(f,var,a,b)` finds the poles of `f` in the interval (`a,b`) and their orders. This syntax assigns the poles to vector `P` and their orders to vector `N`.

`[P,N,R] = poles(f,var,a,b)` finds the poles of `f` in the interval (`a,b`) and their orders and residues. This syntax assigns the poles to vector `P`, their orders to vector `N`, and their residues to vector `R`.

Tips

- If `poles` cannot find all nonremovable singularities and cannot prove that they do not exist, it issues a warning and returns an empty symbolic object.

- If `poles` can prove that `f` has no poles (either in the specified interval (`a,b`) or in the complex plane), it returns an empty symbolic object without issuing a warning.
- `a` and `b` must be real numbers or infinities. If you provide complex numbers, `poles` uses an empty interval and returns an empty symbolic object.

Input Arguments

f

Symbolic expression or function.

var

Symbolic variable.

Default: Variable determined by `symvar`.

a,b

Real numbers (including infinities) that specify the search interval for function poles.

Default: Entire complex plane.

Output Arguments

P

Symbolic vector containing the values of poles.

N

Symbolic vector containing the orders of poles.

R

Symbolic vector containing the residues of poles.

Examples

Find the poles of these expressions:

```
syms x;
```

```
poles(1/(x - i))
poles(sin(x)/(x - 1))

ans =
i

ans =
1
```

Find the poles of this expression. If you do not specify a variable, `poles` uses the default variable determined by `symvar`:

```
syms x a;
poles(1/((x - 1)*(a - 2)))

ans =
1
```

To find the poles of this expression as a function of variable `a`, specify `a` as the second argument:

```
syms x a;
poles(1/((x - 1)*(a - 2)), a)

ans =
2
```

Find the poles of the tangent function in the interval `(-pi, pi)`:

```
syms x;
 poles(tan(x), x, -pi, pi)

ans =
 -pi/2
  pi/2
```

The tangent function has an infinite number of poles. If you do not specify the interval, `poles` cannot find all of them. It issues a warning and returns an empty symbolic object:

```
syms x;
 poles(tan(x))

Warning: Cannot determine the poles.
ans =
[ empty sym ]
```

If `poles` can prove that the expression or function does not have any poles in the specified interval, it returns an empty symbolic object without issuing a warning:

```
syms x;
 poles(tan(x), x, -1, 1)

ans =
[ empty sym ]
```

Use two output vectors to find the poles of this expression and their orders. Restrict the search interval to `(-pi, 10*pi)`:

```
syms x;
[Poles, Orders] = poles(tan(x)/(x - 1)^3, x, -pi, pi)

Poles =
 -pi/2
  pi/2
     1

Orders =
 1
 1
 3
```

Use three output vectors to find the poles of this expression and their orders and residues:

```
syms x a
[Poles, Orders, Residues] = poles(a/x^2/(x - 1), x)

Poles =
 1
 0

Orders =
 1
 2

Residues =
  a
 -a
```

See Also

`limit | solve | symvar | vpasolve`

poly

Purpose

Characteristic polynomial of matrix

Note `poly` has been removed. Use `charpoly` instead.

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)` returns the characteristic polynomial of a symbolic or numeric matrix `A` in terms of the variable `v`.

`poly(sym(A))` approximately equals `poly2sym(poly(A))` for numeric `A`. The approximation is due to round-off error.

See Also

`charpoly` | `eig` | `jordan` | `minpoly` | `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 form the numeric vector `c`. The default symbolic variable is `x`. The variable `v` can be specified as a second input argument. If `c = [c1 c2 ...  cn]`, `r = poly2sym(c)` has the form

$$c_1x^{n-1} + c_2x^{n-2} + \ldots + c_n$$

`poly2sym` uses `sym`'s default (rational) conversion mode to convert the numeric coefficients to symbolic constants. This mode expresses the symbolic coefficient approximately as a ratio of integers, if `sym` can find a simple ratio that approximates the numeric value, otherwise as an integer multiplied by a power of 2.

`r = poly2sym(c,v)` 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`

Purpose

Potential of vector field

Syntax

```
potential(V,X)
potential(V,X,Y)
```

Description

`potential(V,X)` computes the potential of the vector field `V` with respect to the vector `X` in Cartesian coordinates. The vector field `V` must be a gradient field.

`potential(V,X,Y)` computes the potential of vector field `V` with respect to `X` using `Y` as base point for the integration.

Tips

- If `potential` cannot verify that `V` is a gradient field, it returns `NaN`.
- Returning `NaN` does not prove that `V` is not a gradient field. For performance reasons, `potential` sometimes does not sufficiently simplify partial derivatives, and therefore, it cannot verify that the field is gradient.
- If `Y` is a scalar, then `potential` expands it into a vector of the same length as `X` with all elements equal to `Y`.

Input Arguments

V

Vector of symbolic expressions or functions.

X

Vector of symbolic variables with respect to which you compute the potential.

Y

Vector of symbolic variables, expressions, or numbers that you want to use as a base point for the integration. If you use this argument, `potential` returns `P(X)` such that `P(Y) = 0`. Otherwise, the potential is only defined up to some additive constant.

Definitions

Scalar Potential of a Gradient Vector Field

The potential of a gradient vector field $V(X) = [v_1(x_1,x_2,...),v_2(x_1,x_2,...),...]$ is the scalar $P(X)$ such that $V(X) = \nabla P(X)$.

The vector field is gradient if and only if the corresponding Jacobian is symmetrical:

$$\left(\frac{\partial v_i}{\partial x_j}\right) = \left(\frac{\partial v_j}{\partial x_i}\right)$$

The `potential` function represents the potential in its integral form:

$$P(X) = \int_0^1 (X-Y) \cdot V\big(Y + \lambda(X-Y)\big)\, d\lambda$$

Examples

Compute the potential of this vector field with respect to the vector `[x, y, z]`:

```
syms x y z
P = potential([x, y, z*exp(z)], [x y z])

P =
x^2/2 + y^2/2 + exp(z)*(z - 1)
```

Use the `gradient` function to verify the result:

```
simplify(gradient(P, [x y z]))

ans =
        x
        y
 z*exp(z)
```

Compute the potential of this vector field specifying the integration base point as `[0 0 0]`:

```
syms x y z
P = potential([x, y, z*exp(z)], [x y z], [0 0 0])
```

```
P =
x^2/2 + y^2/2 + exp(z)*(z - 1) + 1
```

Verify that `P([0 0 0]) = 0`:

```
subs(P, [x y z], [0 0 0])
```

```
ans =
     0
```

If a vector field is not gradient, `potential` returns `NaN`:

```
potential([x*y, y], [x y])
```

```
ans =
NaN
```

See Also

`curl` | `diff` | `divergence` | `gradient` | `jacobian` | `hessian` | `laplacian` | `vectorPotential`

pretty

Purpose

Prettyprint symbolic expressions

Syntax

```
pretty(X)
```

Description

`pretty(X)` prints symbolic output of `X` 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   |
  |        5      |
  |   3/2 - ----  |
  |          2    |
  |               |
  |    1/2        |
  |   5           |
  |   ---- + 3/2  |
  |    2          |
  +-             -+
```

Solve this equation, and then use `pretty` to represent the solutions in the format similar to typeset mathematics:

```
syms a b c d x
s = solve(a*x^3 + b*x^2 + c*x + d, x);
pretty(s)
```

For better readability, `pretty` uses abbreviations when representing long expressions:

```
  +-                                      -+
  |                    b    #2             |
  |              #1 - --- - --             |
  |                   3 a   #1             |
  |                                        |
  |          1/2 / #2      \               |
  |         3    | -- + #1 | i             |
  |   #2         \ #1      /      b    #1  |
  |  ---- + ----------------- - --- - --   |
  |  2 #1           2           3 a   2    |
  |                                        |
  |          1/2 / #2      \               |
  |         3    | -- + #1 | i             |
  |   #2         \ #1      /      b    #1  |
  |  ---- - ----------------- - --- - --   |
  |  2 #1           2           3 a   2    |
  +-                                      -+

  where

           / / /         3          \2      \1/2     3                \1/3
           | | |  d     b      b c  |      3 |       b       d    b c  |
     #1 == | | | --- + ----- - ---- |  + #2  |    - ----- - --- + ---- |
           | | | 2 a       3      2 |        |         3   2 a      2 |
           \ \ \       27 a    6 a  /        /      27 a          6 a  /

               2
```

```
              b         c
   #2 == -  ---- + ---
                 2    3  a
              9  a
```

Purpose

Digamma function

Syntax

```
psi(x)
psi(k,x)
psi(A)
psi(k,A)
```

Description

`psi(x)` computes the digamma function of `x`.

`psi(k,x)` computes the polygamma function of `x`, which is the `k`th derivative of the digamma function at `x`.

`psi(A)` computes the digamma function of each element of `A`.

`psi(k,A)` computes the polygamma function of `A`, which is the `k`th derivative of the digamma function at `A`.

Tips

- Calling `psi` for a number that is not a symbolic object invokes the MATLAB `psi` function. This function accepts real arguments only. If you want to compute the polygamma function for a complex number, use `sym` to convert that number to a symbolic object, and then call `psi` for that symbolic object.
- `psi(0, x)` is equivalent to `psi(x)`.

Input Arguments

x

Nonnegative symbolic number, variable, or expression.

k

Nonnegative integer.

A

Vector or matrix of nonnegative symbolic numbers, variables, or expressions.

Definitions

digamma Function

The digamma function is the first derivative of the logarithm of the gamma function:

$$\psi(x) = \frac{d}{dx} \ln \Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

polygamma Function

The polygamma function of the order *k* is the (*k* + 1)th derivative of the logarithm of the gamma function:

$$\psi^{(k)}(x) = \frac{d^{k+1}}{dx^{k+1}} \ln \Gamma(x) = \frac{d^k}{dx^k} \psi(x)$$

Examples

Compute the digamma and polygamma functions for these numbers. Because these numbers are not symbolic objects, you get the floating-point results:

```
[psi(1/2) psi(2, 1/2) psi(1.34) psi(1, sin(pi/3))]
```

The results are:

```
ans =
   -1.9635  -16.8288   -0.1248    2.0372
```

Compute the digamma and polygamma functions for the numbers converted to symbolic objects:

```
[psi(sym(1/2)), psi(1, sym(1/2)), psi(sym(1/4))]
```

```
ans =
[ - eulergamma - 2*log(2), pi^2/2, - eulergamma - pi/2 - 3*log(2)]
```

For some symbolic (exact) numbers, `psi` returns unresolved symbolic calls:

```
psi(sym(sqrt(2)))

ans =
psi(2^(1/2))
```

Compute the derivatives of these expressions containing the digamma and polygamma functions:

```
syms x
diff(psi(1, x^3 + 1), x)
diff(psi(sin(x)), x)

ans =
3*x^2*psi(2, x^3 + 1)

ans =
cos(x)*psi(1, sin(x))
```

Expand the expressions containing the digamma functions:

```
syms x
expand(psi(2*x + 3))
expand(psi(x + 2)*psi(x))

ans =
psi(x + 1/2)/2 + log(2) + psi(x)/2 +...
1/(2*x + 1) + 1/(2*x + 2) + 1/(2*x)

ans =
psi(x)/x + psi(x)^2 + psi(x)/(x + 1)
```

Compute the limits for expressions containing the digamma and polygamma functions:

```
syms x
limit(x*psi(x), x, 0)
limit(psi(3, x), x, inf)

ans =
-1

ans =
0
```

Compute the digamma function for elements of these matrix `M` and vector `V`:

```
M =sym([0 inf; 1/3 1/2]);
V = sym([1; inf]);
psi(M)
psi(V)

ans =
[                                              Inf,                          Inf]
[ - eulergamma - (3*log(3))/2 - (pi*3^(1/2))/6, - eulergamma - 2*log(2)]

ans =
 -eulergamma
         Inf
```

See Also

`beta` | `gamma` | `nchoosek` | `factorial` | `mfun` | `mfunlist`

How To

- "Special Functions of Applied Mathematics" on page 2-142

Purpose

Symbolic matrix element-wise 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)`.

Examples

```
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

Read MuPAD program file into symbolic engine

Syntax

```
read(symengine,filename)
```

Description

`read(symengine,filename)` reads the MuPAD program file `filename` into the symbolic engine. Reading a program file means finding and executing it.

Tips

- If you do not specify the file extension, `read` searches for the file `filename.mu`.
- If `filename` is a GNU® zip file with the extension `.gz`, `read` uncompresses it upon reading.
- `filename` can include full or relative path information. If `filename` does not have a path component, `read` uses the MATLAB function `which` to search for the file on the MATLAB path.
- `read` ignores any MuPAD aliases defined in the program file. If your program file contains aliases or uses the aliases predefined by MATLAB, see "Alternatives" on page 4-465.

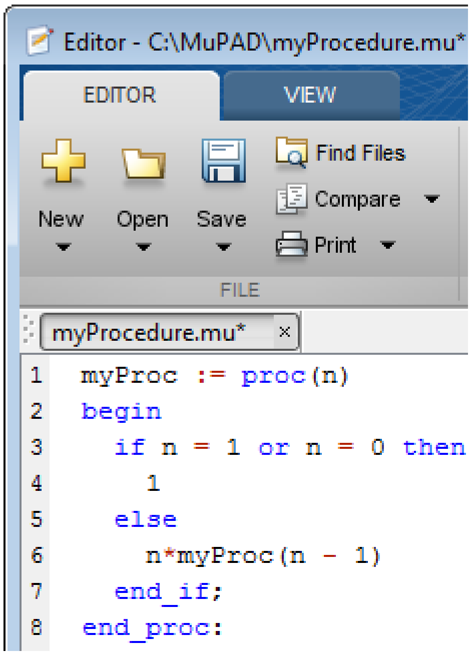
Input Arguments

filename

The name of a MuPAD program file that you want to read. This file must have the extension `.mu` or `.gz`.

Examples

Suppose you wrote the MuPAD procedure `myProc` and saved it in the file `myProcedure.mu`.



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

Before you can call this procedure at the MATLAB Command Window, you must read the file `myProcedure.mu` into the symbolic engine. To read a program file into the symbolic engine, use `read`:

```
read(symengine, 'myProcedure.mu');
```

If the file is not on the MATLAB path, specify the full path to this file. For example, if `myProcedure.mu` is in the `MuPAD` folder on disk `C`, enter:

```
read(symengine, 'C:/MuPAD/myProcedure.mu');
```

Now you can access the procedure `myProc` using `evalin` or `feval`. For example, compute the factorial of 10:

```
feval(symengine, 'myProc', 10)

ans =
3628800
```

Alternatives

You also can use `feval` to call the MuPAD `read` function. The `read` function available from the MATLAB Command Window is equivalent to calling the MuPAD `read` function with the `Plain` option. It ignores any MuPAD aliases defined in the program file:

```
eng=symengine;
eng.feval('read',' "myProcedure.mu" ', 'Plain');
```

If your program file contains aliases or uses the aliases predefined by MATLAB, do not use `Plain`:

```
eng=symengine;
eng.feval('read',' "myProcedure.mu" ');
```

See Also

`evalin` | `feval` | `symengine`

How To

- "Use Your Own MuPAD Procedures" on page 3-39
- "Conflicts Caused by Syntax Conversions" on page 3-27

real

Purpose

Real part of complex number

Syntax

```
real(z)
real(A)
```

Description

`real(z)` returns the real part of `z`.

`real(A)` returns the real part of each element of `A`.

Tips

- Calling `real` for a number that is not a symbolic object invokes the MATLAB `real` function.

Input Arguments

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Examples

Find the real parts of these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[real(2 + 3/2*i), real(sin(5*i)), real(2*exp(1 + i))]

ans =
    2.0000         0    2.9374
```

Compute the real parts of the numbers converted to symbolic objects:

```
[real(sym(2) + 3/2*i), real(4/(sym(1) + 3*i)),
real(sin(sym(5)*i))]

ans =
[ 2, 2/5, 0]
```

Compute the real part of this symbolic expression:

```
real(sym('2*exp(1 + i)'))

ans =
2*cos(1)*exp(1)
```

In general, `real` cannot extract the entire real parts from symbolic expressions containing variables. However, `real` can rewrite and sometimes simplify the input expression:

```
syms a x y
real(a + 2)
real(x + y*i)

ans =
real(a) + 2

ans =
real(x) - imag(y)
```

If you assign numeric values to these variables or specify that these variables are real, `real` can extract the real part of the expression:

```
syms a
a = 5 + 3*i;
real(a + 2)

ans =
     7

syms x y real
real(x + y*i)

ans =
x
```

Clear the assumption that `x` and `y` are real:

```
syms x y clear
```

Find the real parts of the elements of matrix A:

```
A = sym('[-1 + i, sinh(x); exp(10 + 7*i), exp(pi*i)]');
real(A)

ans =
[              -1, real(sinh(x))]
[ cos(7)*exp(10),             -1]
```

Alternatives

You can compute the real part of z via the conjugate: `real(z)= (z + conj(z))/2`.

See Also

`conj | imag`

Purpose

Rectangular pulse function

Syntax

```
rectangularPulse(a,b,x)
rectangularPulse(x)
```

Description

`rectangularPulse(a,b,x)` returns the rectangular pulse function.

`rectangularPulse(x)` is a shortcut for `rectangularPulse(-1/2,1/2,x)`.

Tips

- If `a` and `b` are variables or expressions with variables, `rectangularPulse` assumes that `a < b`. If `a` and `b` are numerical values, such that `a > b`, `rectangularPulse` throws an error.
- If `a = b`, `rectangularPulse` returns 0.

Input Arguments

a

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression. This argument specifies the rising edge of the rectangular pulse function.

Default: `-1/2`

b

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression. This argument specifies the falling edge of the rectangular pulse function.

Default: `1/2`

x

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression.

Definitions

Rectangular Pulse Function

The rectangular pulse function is defined as follows:

If `a < x < b`, then the rectangular pulse function equals 1. If `x = a` or `x = b` and `a <> b`, then the rectangular pulse function equals 1/2. Otherwise, it equals 0.

The rectangular pulse function is also called the rectangle function, box function, Π-function, or gate function.

Examples

Compute the rectangular pulse function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[rectangularPulse(-1, 1, -2)
rectangularPulse(-1, 1, -1)
rectangularPulse(-1, 1, 0)
rectangularPulse(-1, 1, 1)
rectangularPulse(-1, 1, 2)]

ans =
         0
    0.5000
    1.0000
    0.5000
         0
```

Compute the rectangular pulse function for the numbers converted to symbolic objects:

```
[rectangularPulse(sym(-1), 1, -2)
rectangularPulse(-1, sym(1), -1)
rectangularPulse(-1, 1, sym(0))
rectangularPulse(sym(-1), 1, 1)
rectangularPulse(sym(-1), 1, 2)]

ans =
   0
```

```
 1/2
   1
 1/2
   0
```

If `a < b`, the rectangular pulse function for `x = a` and `x = b` equals `1/2`:

```
syms a b x
assume(a < b)
rectangularPulse(a, b, a)
rectangularPulse(a, b, b)

ans =
1/2

ans =
1/2
```

For further computations, remove the assumption:

```
syms a b clear
```

For `a = b`, the rectangular pulse function returns `0`:

```
syms a x
rectangularPulse(a, a, x)

ans =
0
```

Use `rectangularPulse` with one input argument as a shortcut for computing `rectangularPulse(-1/2, 1/2, x)`:

```
syms x
```

```
rectangularPulse(x)
```

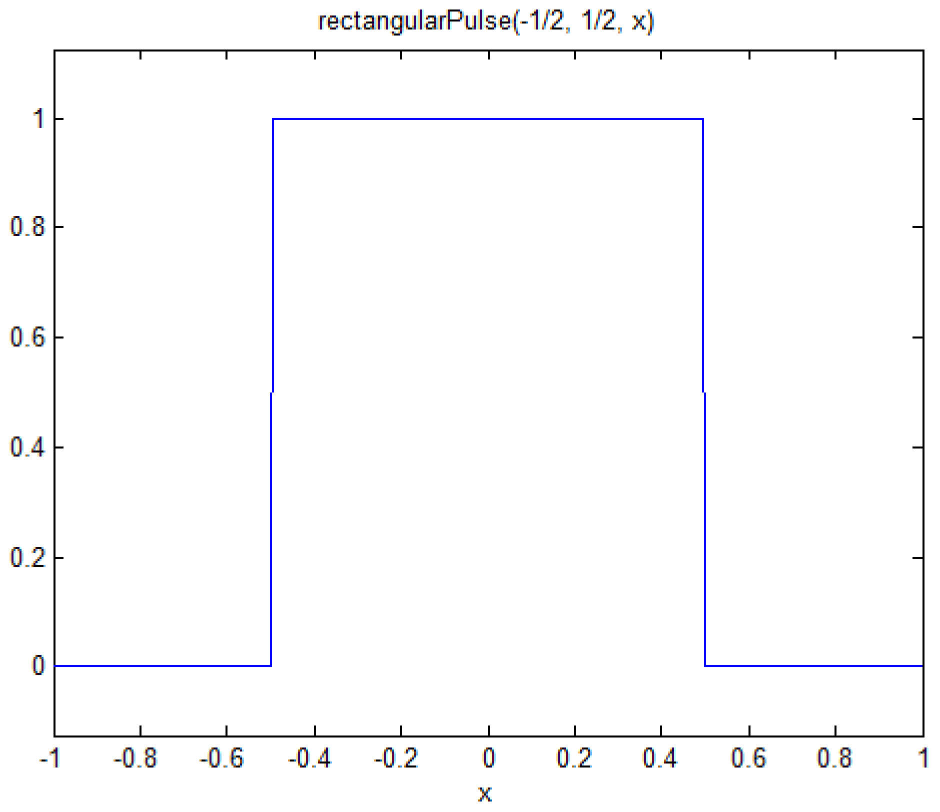
```
ans =
rectangularPulse(-1/2, 1/2, x)
```

```
[rectangularPulse(sym(-1))
rectangularPulse(sym(-1/2))
rectangularPulse(sym(0))
rectangularPulse(sym(1/2))
rectangularPulse(sym(1))]
```

```
ans =
   0
 1/2
   1
 1/2
   0
```

Plot the rectangular pulse function:

```
syms x
ezplot(rectangularPulse(x), [-1, 1])
```



Call `rectangularPulse` with infinities as its rising and falling edges:

```
syms x
rectangularPulse(-inf, 0, x)
rectangularPulse(0, inf, x)
rectangularPulse(-inf, inf, x)

ans =
heaviside(-x)
```

```
ans =
heaviside(x)

ans =
1
```

See Also `dirac | heaviside | triangularPulse`

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`

Purpose

Rewrite expression in new terms

Syntax

```
rewrite(expr,target)
rewrite(A,target)
```

Description

`rewrite(expr,target)` rewrites the symbolic expression `expr` in terms of `target`. The returned expression is mathematically equivalent to the original expression.

`rewrite(A,target)` rewrites each element of `A` in terms of `target`.

Tips

- `rewrite` replaces symbolic function calls in `expr` with the target function only if such replacement is mathematically valid. Otherwise, it keeps the original function calls.

Input Arguments

expr

Symbolic expression.

A

Vector or matrix of symbolic expressions.

target

One of these strings: `exp`, `log`, `sincos`, `sin`, `cos`, `tan`, `sqrt`, or `heaviside`.

Examples

Rewrite these trigonometric functions in terms of the exponential function:

```
syms x
rewrite(sin(x), 'exp')
rewrite(cos(x), 'exp')
rewrite(tan(x), 'exp')

ans =
(exp(-x*i)*i)/2 - (exp(x*i)*i)/2
```

```
ans =
exp(-x*i)/2 + exp(x*i)/2

ans =
-(exp(x*2*i)*i - i)/(exp(x*2*i) + 1)
```

Rewrite the tangent function in terms of the sine function:

```
syms x
rewrite(tan(x), 'sin')

ans =
-sin(x)/(2*sin(x/2)^2 - 1)
```

Rewrite the hyperbolic tangent function in terms of the sine function:

```
syms x
rewrite(tanh(x), 'sin')

ans =
(sin(x*i)*i)/(2*sin((x*i)/2)^2 - 1)
```

Rewrite these inverse trigonometric functions in terms of the natural logarithm:

```
syms x
rewrite(acos(x), 'log')
rewrite(acot(x), 'log')

ans =
-log(x + (1 - x^2)^(1/2)*i)*i

ans =
(log(1 - i/x)*i)/2 - (log(i/x + 1)*i)/2
```

Rewrite the rectangular pulse function in terms of the Heaviside step function:

```
syms a b x
rewrite(rectangularPulse(a, b, x), 'heaviside')

ans =
heaviside(x - a) - heaviside(x - b)
```

Rewrite the triangular pulse function in terms of the Heaviside step function:

```
syms a b c x
rewrite(triangularPulse(a, b, c, x), 'heaviside')

ans =
(heaviside(x - a)*(a - x))/(a - b) - (heaviside(x -
b)*(a - x))/(a - b) - (heaviside(x - b)*(c - x))/(b -
c) + (heaviside(x - c)*(c - x))/(b - c)
```

Call `rewrite` to rewrite each element of this matrix of symbolic expressions in terms of the exponential function:

```
syms x
A = [sin(x) cos(x); sinh(x) cosh(x)];
rewrite(A, 'exp')

ans =
[ (exp(-x*i)*i)/2 - (exp(x*i)*i)/2, exp(-x*i)/2 + exp(x*i)/2]
[             exp(x)/2 - exp(-x)/2,     exp(-x)/2
+ exp(x)/2]
```

Rewrite the cosine function in terms of sine function. Here `rewrite` replaces the cosine function using the identity `cos(2*x) = 1 sin(x)^2` which is valid for any `x`:

```
syms x
rewrite(cos(x),'sin')
```

```
ans =
1 - 2*sin(x/2)^2
```

`rewrite` does not replace the sine function with either $-\sqrt{1-\cos^2(x)}$ or $\sqrt{1-\cos^2(x)}$ because these expressions are only valid for `x` within particular intervals:

```
syms x
rewrite(sin(x),'cos')
```

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

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `simplify` | `simplifyFraction`

Concepts

- “Simplifications” on page 2-33

Purpose

Symbolic matrix element-wise 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.

Examples

```
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)))

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)

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

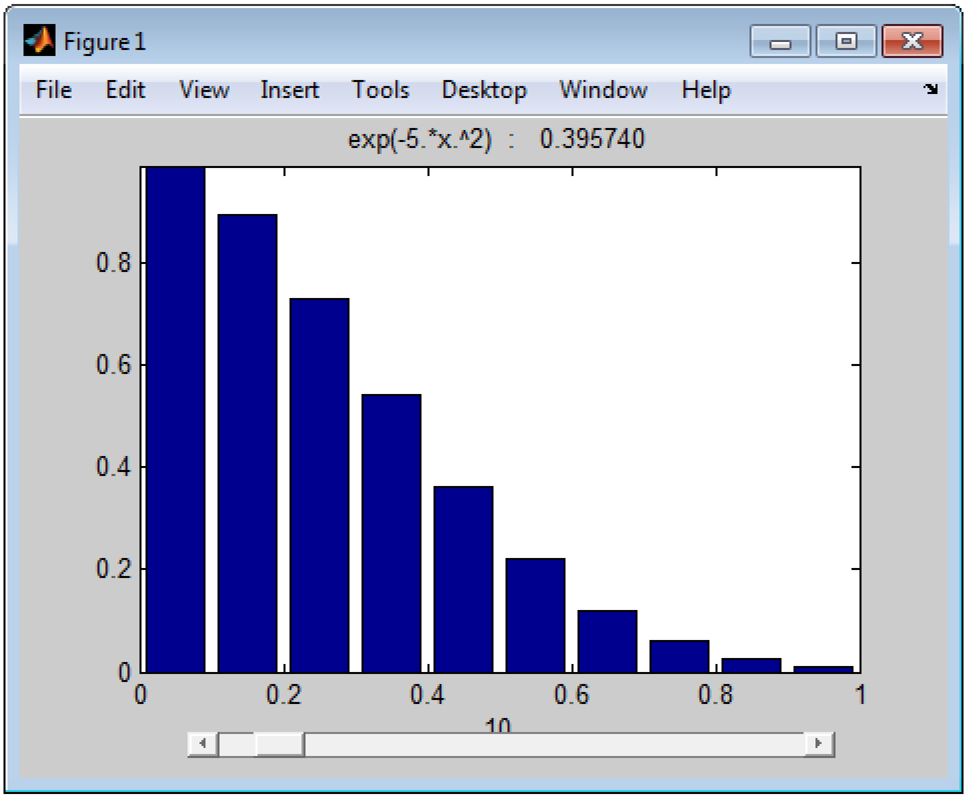


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
exp(-5.*x.^2) : 0.395740
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
10

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

Sign of real or complex value

Syntax

```
sign(z)
```

Description

`sign(z)` returns the sign of real or complex value `z`. The sign of a complex number `z` is defined as `z/abs(z)`. If `z` is a vector or a matrix, `sign(z)` returns the sign of each element of `z`.

Tips

- Calling `sign` for a number that is not a symbolic object invokes the MATLAB `sign` function.

Input Arguments

z - Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input specified as a symbolic number, variable, expression, function, vector, or matrix.

Examples

Signs of Real Numbers

Find the signs of these symbolic real numbers:

```
[sign(sym(1/2)), sign(sym(0)), sign(sym(pi) - 4)]

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

Signs of Matrix Elements

Find the signs of the real and complex elements of matrix `A`:

```
A = sym([(1/2 + i), -25; i*(i + 1), pi/6 - i*pi/2]);
sign(A)

ans =
[ 5^(1/2)*(1/5 + (2*i)/5),                            -1]
[   2^(1/2)*(- 1/2 + i/2), 5^(1/2)*18^(1/2)*(1/30 - i/10)]
```

Sign of Symbolic Expression

Find the sign of this expression assuming that the value `x` is negative:

```
syms x
assume(x < 0)
sign(5*x^3)

ans =
-1
```

For further computations, clear the assumption:

```
syms x clear
```

Definitions

Sign Function

The sign function of any number z is defined via the absolute value of z:

$$sign(z) = \frac{z}{|z|}$$

Thus, the sign function of a real number z can be defined as follows:

$$sign(z) = \begin{cases} -1 \text{ if } x < 0 \\ \ \ 0 \text{ if } x = 0 \\ \ \ 1 \text{ if } x > 0 \end{cases}$$

See Also

`abs` | `angle` | `imag` | `realsign`

Purpose

Search for simplest form of symbolic expression

Note `simple` will be removed in a future release. Use `simplify(S)` instead of `simple(S)`. There is no replacement for `[r, how] = simple(S)`.

Syntax

```
simple(S)
simple(S,Name,Value)
r = simple(S)
r = simple(S,Name,Value)
[r,how] = simple(S)
[r,how] = simple(S,Name,Value)
```

Description

`simple(S)` applies different algebraic simplification functions and displays all resulting forms of `S`, and then returns the shortest form.

`simple(S,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`r = simple(S)` tries different algebraic simplification functions without displaying the results, and then returns the shortest form of `S`.

`r = simple(S,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[r,how] = simple(S)` tries different algebraic simplification functions without displaying the results, and then returns the shortest form of `S` and a string describing the corresponding simplification method.

`[r,how] = simple(S,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Tips

- Simplification of mathematical expression is not a clearly defined subject. There is no universal idea as to which form of an expression is simplest. The form of a mathematical expression that is simplest for one problem might turn out to be complicated or even unsuitable for another problem.

- 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.

Input Arguments

S

Symbolic expression or symbolic matrix.

> **Default:** false

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'IgnoreAnalyticConstraints'

If the value is `true`, apply purely algebraic simplifications to an expression. With `IgnoreAnalyticConstraints`, `simple` can return simpler results for expressions for which it would return more complicated results otherwise. Using `IgnoreAnalyticConstraints` also can lead to results that are not equivalent to the initial expression.

> **Default:** `false`

Output Arguments

r

A symbolic object representing the shortest form of `S`

how

A string describing the simplification method that gives the shortest form of `S`

Algorithms

When you use `IgnoreAnalyticConstraints`, `simple` applies these rules:

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

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

  $(a^b)^c = a^{b \cdot c}$.
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:
  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify`

How To

- "Simplifications" on page 2-33

simplify

Purpose

Algebraic simplification

Syntax

```
simplify(S)
simplify(S,Name,Value)
```

Description

`simplify(S)` performs algebraic simplification of `S`. If `S` is a symbolic vector or matrix, this function simplifies each element of `S`.

`simplify(S,Name,Value)` performs algebraic simplification of `S` using additional options specified by one or more `Name,Value` pair arguments.

Tips

- Simplification of mathematical expression is not a clearly defined subject. There is no universal idea as to which form of an expression is simplest. The form of a mathematical expression that is simplest for one problem might be complicated or even unsuitable for another problem.

Input Arguments

S - Input expression.

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input expression, specified as a symbolic expression, function, vector, or matrix.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Criterion' - Simplification criterion

`'default'` (default) | `'preferReal'`

Simplification criterion, specified as the comma-separated pair consisting of `'Criterion'` and one of these strings.

`'default'`	Use the default (internal) simplification criteria.
`'preferReal'`	Favor the forms of `S` containing real values over the forms containing complex values. If any form of `S` contains complex values, the simplifier disfavors the forms where complex values appear inside subexpressions. In case of nested subexpressions, the deeper the complex value appears inside an expression, the least preference this form of an expression gets.

'IgnoreAnalyticConstraints' - Simplification rules

`false` (default) | `true`

Simplification rules, specified as the comma-separated pair consisting of `'IgnoreAnalyticConstraints'` and one of these values.

`false`	Use strict simplification rules. `simplify` always returns results equivalent to the initial expression.
`true`	Apply purely algebraic simplifications to an expression. `simplify` can return simpler results for expressions for which it would return more complicated results otherwise. Setting `IgnoreAnalyticConstraints` to `true` can lead to results that are not equivalent to the initial expression.

'Seconds' - Time limit for the simplification process

`Inf` (default) | `positive number`

Time limit for the simplification process, specified as the comma-separated pair consisting of `'Seconds'` and a positive value that denotes the maximal time in seconds.

'Steps' - Number of simplification steps

`1` (default) | `positive number`

Number of simplification steps, specified as the comma-separated pair consisting of `'Steps'` and a positive value that denotes the maximal number of internal simplification steps. Note that increasing the number of simplification steps can slow down your computations.

`simplify(S,'Steps',n)` is equivalent to `simplify(S,n)`, where `n` is the number of simplification steps.

Examples

Simplify Expressions

Simplify these symbolic expressions:

```
syms x a b c
simplify(sin(x)^2 + cos(x)^2)
simplify(exp(c*log(sqrt(a+b))))

ans =
1

ans =
(a + b)^(c/2)
```

Simplify Elements of a Symbolic Matrix

Call `simplify` for this symbolic matrix. When the input argument is a vector or matrix, `simplify` tries to find a simpler form of each element of the vector or matrix.

```
syms x
simplify([(x^2 + 5*x + 6)/(x + 2), sin(x)*sin(2*x) + cos(x)*cos(2*x);
(exp(-x*i)*i)/2 - (exp(x*i)*i)/2, sqrt(16)])

ans =
[  x + 3, cos(x)]
[ sin(x),      4]
```

Get Simpler Results Using IgnoreAnalyticConstraints

Try to simplify this expression. By default, `simplify` does not combine powers and logarithms because combining them is not valid for generic complex values.

```
syms x
s = (log(x^2 + 2*x + 1) - log(x + 1))*sqrt(x^2);
simplify(s)

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

To apply the simplification rules that let the `simplify` function combine powers and logarithms, set `IgnoreAnalyticConstraints` to `true`:

```
simplify(s, 'IgnoreAnalyticConstraints', true)

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

Get Simpler Results Using Steps

Simplify this expression:

```
syms x
f = ((exp(-x*i)*i)/2 - (exp(x*i)*i)/2)/(exp(-x*i)/2 + exp(x*i)/2);
simplify(f)

ans =
-(exp(x*2*i)*i - i)/(exp(x*2*i) + 1)
```

By default, `simplify` uses one internal simplification step. You can get different, often shorter, simplification results by increasing the number of simplification steps:

```
simplify(f, 'Steps', 10)
simplify(f, 'Steps', 30)
simplify(f, 'Steps', 50)

ans =
(2*i)/(exp(x*2*i) + 1) - i

ans =
((cos(x) - sin(x)*i)*i)/cos(x) - i
```

```
ans =
tan(x)
```

Simplify Favoring Real Numbers

To force `simplify` favor real values over complex values, set the value of `Criterion` to `preferReal`:

```
syms x
f = (exp(x + exp(-x*i)/2 - exp(x*i)/2)*i)/2 - (exp(- x - exp(-x*i)/2 + ex
simplify(f, 'Criterion','preferReal', 'Steps', 100)

ans =
cos(sin(x))*sinh(x)*i + sin(sin(x))*cosh(x)
```

If `x` is a real value, then this form of expression explicitly shows the real and imaginary parts.

Although the result returned by `simplify` with the default setting for `Criterion` is shorter, here the complex value is a parameter of the sine function:

```
simplify(f, 'Steps', 100)

ans =
sin(x*i + sin(x))
```

When you set `Criterion` to `preferReal`, the simplifier disfavors expression forms where complex values appear inside subexpressions. In case of nested subexpressions, the deeper the complex value appears inside an expression, the least preference this form of an expression gets.

Simplify Expressions with Complex Arguments in Exponents

Setting `Criterion` to `preferReal` helps you avoid complex arguments in exponents.

Simplify these symbolic expressions:

```
simplify(sym(i)^i, 'Steps', 100)
simplify(sym(i)^(i+1), 'Steps', 100)

ans =
exp(-pi/2)

ans =
(-1)^(1/2 + i/2)
```

Now, simplify the second expression with the `Criterion` set to `preferReal`:

```
simplify(sym(i)^(i+1), 'Criterion', 'preferReal',
'Steps', 100)

ans =
exp(-pi/2)*i
```

Algorithms

When you use `IgnoreAnalyticConstraints`, `simplify` follows these rules:

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

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

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

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

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex values of x. In particular:

  - $\log(e^x) = x$
  - $\operatorname{asin}(\sin(x)) = x$, $\operatorname{acos}(\cos(x)) = x$, $\operatorname{atan}(\tan(x)) = x$
  - $\operatorname{asinh}(\sinh(x)) = x$, $\operatorname{acosh}(\cosh(x)) = x$, $\operatorname{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

Alternative Functionality

Besides the general simplification function (`simplify`), the toolbox provides a set of functions for transforming mathematical expressions to particular forms. For example, you can use particular functions to expand or factor expressions, collect terms with the same powers, find a nested (Horner) representation of an expression, or quickly simplify fractions. If the problem that you want to solve requires a particular form of an expression, the best approach is to choose the appropriate simplification function. These simplification functions are often faster than `simplify`.

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplifyFraction`

Related Examples

- “Simplifications” on page 2-33

simplifyFraction

Purpose

Symbolic simplification of fractions

Syntax

```
simplifyFraction(expr)
simplifyFraction(expr,Name,Value)
```

Description

`simplifyFraction(expr)` represents the expression `expr` as a fraction where both the numerator and denominator are polynomials whose greatest common divisor is 1.

`simplifyFraction(expr,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Tips

- `expr` can contain irrational subexpressions, such as `sin(x)`, `x^(-1/3)`, and so on. As a first step, `simplifyFraction` replaces these subexpressions with auxiliary variables. Before returning results, `simplifyFraction` replaces these variables with the original subexpressions.
- `simplifyFraction` ignores algebraic dependencies of irrational subexpressions.

Input Arguments

expr

Symbolic expression or matrix (or vector) of symbolic expressions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Expand'

Expand the numerator and denominator of the resulting fraction

Default: `false`

Examples

Simplify these fractions:

```
syms x y
simplifyFraction((x^2 - 1)/(x + 1))
simplifyFraction(((y + 1)^3*x)/((x^3 - x*(x
+ 1)*(x - 1))*y))

ans =
x - 1

ans =
(y + 1)^3/y
```

Use `Expand` to expand the numerator and denominator in the resulting fraction:

```
syms x y
simplifyFraction(((y + 1)^3*x)/((x^3 - x*(x + 1)*(x - 1))*y),...
'Expand', true)

ans =
(y^3 + 3*y^2 + 3*y + 1)/y
```

Use `simplifyFraction` to simplify rational subexpressions of irrational expressions:

```
syms x
simplifyFraction(((x^2 + 2*x + 1)/(x + 1))^(1/2))

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

Also, use `simplifyFraction` to simplify rational expressions containing irrational subexpressions:

```
simplifyFraction((1 - sin(x)^2)/(1 - sin(x)))
```

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

When you call `simplifyFraction` for an expression that contains irrational subexpressions, the function ignores algebraic dependencies of irrational subexpressions:

```
simplifyFraction((1 - cos(x)^2)/sin(x))
```

```
ans =
-(cos(x)^2 - 1)/sin(x)
```

Alternatives

You also can simplify fractions using the general simplification function `simplify`. Note that in terms of performance, `simplifyFraction` is significantly more efficient for simplifying fractions than `simplify`.

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify`

How To

- "Simplifications" on page 2-33

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 == sin(time)+y*5.0+x.der;

ans =
y.der == sin(time)+y*5.0+x.der;
```

See Also

`matlabFunctionBlock` | `matlabFunction` | `ccode` | `fortran`

How To

- “Generate Simscape Equations” on page 2-139

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`

Purpose

Equations and systems solver

Syntax

```
S = solve(eqn)
S = solve(eqn,var,Name,Value)

Y = solve(eqns)
Y = solve(eqns,vars,Name,Value)

[y1,...,yN] = solve(eqns)
[y1,...,yN] = solve(eqns,vars,Name,Value)
```

Description

`S = solve(eqn)` solves the equation `eqn` for the default variable determined by `symvar`. You can specify the independent variable. For example, `solve(x + 1 == 2, x)` solves the equation $x + 1 = 2$ with respect to the variable x.

`S = solve(eqn,var,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. If you do not specify `var`, the solver uses the default variable determined by `symvar`.

`Y = solve(eqns)` solves the system of equations `eqns` for the variables determined by `symvar` and returns a structure array that contains the solutions. The number of fields in the structure array corresponds to the number of independent variables in a system.

`Y = solve(eqns,vars,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. If you do not specify `vars`, the solver uses the default variables determined by `symvar`.

`[y1,...,yN] = solve(eqns)` solves the system of equations `eqns` for the variables determined by `symvar` and assigns the solutions to the variables `y1,...,yN`.

`[y1,...,yN] = solve(eqns,vars,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. If you specify the variables `vars`, `solve` returns the results in the same order in which you specify `vars`. If you do not specify `vars`, the solver uses the default variables determined by `symvar`.

Tips

- If the symbolic solver cannot find a solution of an equation or a system of equations, the toolbox internally calls the numeric solver that tries to find a numeric approximation. For polynomial equations and systems without symbolic parameters, the numeric solver returns all solutions. For nonpolynomial equations and systems without symbolic parameters, the solver returns only one solution (if a solution exists).
- If the solution of an equation or a system of equations contains parameters, the solver can choose one or more values of the parameters and return the results corresponding to these values. For some equations and systems, the solver returns parameterized solutions without choosing particular values. In this case, the solver also issues a warning indicating the values of parameters in the returned solutions.
- To solve differential equations, use the `dsolve` function.
- When solving a system of equations, always assign the result to output arguments. Output arguments let you access the values of the solutions of a system.
- `MaxDegree` only accepts positive integers smaller than 5 because, in general, there are no explicit expressions for the roots of polynomials of degrees higher than 4.
- The output variables `y1,...,yN` do not specify the variables for which `solve` solves equations or systems. If `y1,...,yN` are the variables that appear in `eqns`, that does not guarantee that `solve(eqns)` will assign the solutions to `y1,...,yN` using the correct order. Thus, for the call `[b,a] = solve(eqns)`, you might get the solutions for `a` assigned to `b` and vice versa.

To ensure the order of the returned solutions, specify the variables `vars`. For example, the call `[b,a] = solve(eqns,b,a)` assigns the solutions for `a` assigned to `a` and the solutions for `b` assigned to `b`.

Input Arguments

eqn - Equation to solve

symbolic expression | symbolic equation

Equation to solve, specified as a symbolic expression or symbolic equation. Symbolic equations are defined by the relation operator `==`. If `eqn` is a symbolic expression (without the right side), the solver assumes that the right side is 0, and solves the equation `eqn == 0`.

var - Variable for which you solve an equation

symbolic variable

Variable for which you solve an equation, specified as a symbolic variable. By default, `solve` uses the variable determined by `symvar`.

eqns - System of equations

symbolic expressions | symbolic equations

System of equations, specified as symbolic expressions or symbolic equations. If any of `eqns` are symbolic expressions (without the right side), the solver assumes that the right sides of those equations are 0s.

vars - Variables for which you solve an equation or a system of equations

symbolic variables

Variables for which you solve an equation or system of equations, specified as symbolic variables. By default, `solve` uses the variables determined by `symvar`.

The order in which you specify these variables defines the order in which the solver returns the solutions.

Name-Value Pair Arguments

'IgnoreAnalyticConstraints' - Simplification rules applied to expressions and equations

`false` (default) | `true`

Simplification rules applied to expressions and equations, specified as the comma-separated pair consisting of `'IgnoreAnalyticConstraints'` and one of these values.

`false`	Use strict simplification rules.
`true`	Apply purely algebraic simplifications to expressions and equations. Setting `IgnoreAnalyticConstraints` to `true` can give you simple solutions for the equations for which the direct use of the solver returns complicated results. In some cases, it also enables `solve` to solve equations and systems that cannot be solved otherwise. Note that setting `IgnoreAnalyticConstraints` to `true` can lead to wrong or incomplete results.

'IgnoreProperties' - Flag for returning solutions inconsistent with the properties of variables

`false` (default) | `true`

Flag for returning solutions inconsistent with the properties of variables, specified as the comma-separated pair consisting of `'IgnoreProperties'` and one of these values.

`false`	Do not exclude solutions inconsistent with the properties of variables.
`true`	Exclude solutions inconsistent with the properties of variables.

'MaxDegree' - Maximal degree of polynomial equations for which the solver uses explicit formulas

`3` (default) | positive integer smaller than 5

Maximal degree of polynomial equations for which the solver uses explicit formulas, specified as a positive integer smaller than 5. The solver does not use explicit formulas that involve radicals when solving polynomial equations of a degree larger than the specified value.

'PrincipalValue' - Flag for returning only one solution
`false` (default) | `true`

Flag for returning only one solution, specified as the comma-separated pair consisting of `'PrincipalValue'` and one of these values.

`false`	Return all solutions.
`true`	Return only one solution. If an equation or a system of equations does not have a solution, the solver returns an empty symbolic object.

'Real' - Flag for returning only real solutions
`false` (default) | `true`

Flag for returning only real solutions, specified as the comma-separated pair consisting of `'Real'` and one of these values.

`false`	Return all solutions.
`true`	Return only those solutions for which every subexpression of the original equation represents a real number. Also, assume that all symbolic parameters of an equation represent real numbers.

Output Arguments

S - Solutions of an equation
symbolic array

Solutions of an equation, returned as a symbolic array. The size of a symbolic array corresponds to the number of the solutions.

Y - Solutions of a system of equations

structure array

Solutions of a system of equations, returned as a structure array. The number of fields in the structure array corresponds to the number of independent variables in a system.

y1,...,yN - Solutions of a system of equations
symbolic variables

Solutions of a system of equations, returned as symbolic variables. The number of output variables or symbolic arrays must be equal to the number of independent variables in a system. If you explicitly specify independent variables `vars`, then the solver uses the same order to return the solutions. If you do not specify `vars`, the toolbox sorts independent variables alphabetically, and then assigns the solutions for these variables to the output variables.

Examples

Solve Univariate Equations

If the right side of an equation is 0, you can specify either a symbolic expression without the left side or an equation with the == operator:

```
syms x
solve(x^2 - 1)
solve(x^2 + 4*x + 1 == 0)
```

```
ans =
  1
 -1

ans =
   3^(1/2) - 2
 - 3^(1/2) - 2
```

If the right side of an equation is not 0, specify the equation using ==:

```
syms x
solve(x^4 + 1 == 2*x^2 - 1)
```

```
ans =
  (1 + i)^(1/2)
  (1 - i)^(1/2)
 -(1 + i)^(1/2)
 -(1 - i)^(1/2)
```

Solve Multivariate Equations

To avoid ambiguities when solving equations with symbolic parameters, specify the variable for which you want to solve an equation:

```
syms a b c x
solve(a*x^2 + b*x + c == 0, a)
solve(a*x^2 + b*x + c == 0, b)
```

The result is:

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

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

If you do not specify the variable for which you want to solve the equation, the toolbox chooses a variable by using the `symvar` function. Here, the solver chooses the variable *x*:

```
syms a b c x
solve(a*x^2 + b*x + c == 0)
```

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

Solve a System of Equations Returning Solutions as a Structure Array

When solving a system of equations, use one output argument to return the solutions in the form of a structure array:

```
syms x y
S = solve(x + y == 1, x - 11*y == 5)

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

To display the solutions, access the elements of the structure array `S`:

```
S = [S.x S.y]

S =
[ 4/3, -1/3]
```

Solve a System of Equations Assigning the Solutions to Variables

When solving a system of equations, use multiple output arguments to assign the solutions directly to output variables:

```
syms a u v
[solutions_a, solutions_u, solutions_v] =...
 solve(a*u^2 + v^2 == 0, u - v == 1, a^2 + 6 == 5*a)
```

The solver returns a symbolic array of solutions for each independent variable:

```
solutions_a =
 3
 2
 2
 3

solutions_u =
 (3^(1/2)*i)/4 + 1/4
 (2^(1/2)*i)/3 + 1/3
 1/3 - (2^(1/2)*i)/3
 1/4 - (3^(1/2)*i)/4
```

```
solutions_v =
   (3^(1/2)*i)/4 - 3/4
   (2^(1/2)*i)/3 - 2/3
 - (2^(1/2)*i)/3 - 2/3
 - (3^(1/2)*i)/4 - 3/4
```

Entries with the same index form the solutions of a system:

```
solutions = [solutions_a, solutions_u, solutions_v]

solutions =
[ 3, (3^(1/2)*i)/4 + 1/4,   (3^(1/2)*i)/4 - 3/4]
[ 2, (2^(1/2)*i)/3 + 1/3,   (2^(1/2)*i)/3 - 2/3]
[ 2, 1/3 - (2^(1/2)*i)/3, - (2^(1/2)*i)/3 - 2/3]
[ 3, 1/4 - (3^(1/2)*i)/4, - (3^(1/2)*i)/4 - 3/4]
```

Specify the Order of Returned Solutions

Solve this system of equations and assign the solutions to variables `b` and `a`. To ensure the correct order of the returned solutions, specify the variables explicitly. The order in which you specify the variables defines the order in which the solver returns the solutions.

```
syms a b
[b, a] = solve(a + b == 1, 2*a - b == 4, b, a)

b =
-2/3

a =
5/3
```

Return Numeric Solutions

Solve the following equation:

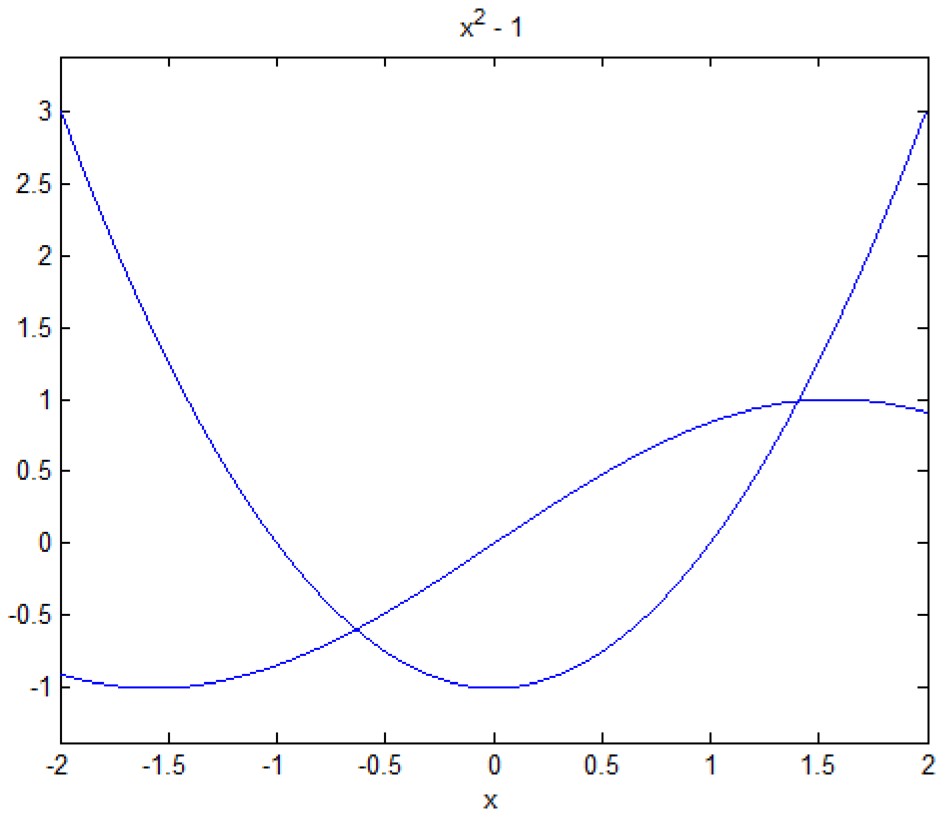
```
syms x
solve(sin(x) == x^2 - 1)
```

The symbolic solver cannot find an exact symbolic solution for this equation, and therefore, it calls the numeric solver. Because the equation is not polynomial, an attempt to find all possible solutions can take a long time. The numeric solver does not try to find all numeric solutions for this equation. Instead, it returns only the first solution that it finds:

```
ans =
-0.63673265080528201088799090383828
```

Plotting the left and the right sides of the equation in one graph shows that the equation also has a positive solution:

```
ezplot(sin(x), -2, 2)
hold on
ezplot(x^2 - 1, -2, 2)
hold off
```



You can find this solution by calling the MuPAD numeric solver directly and specifying the interval where this solution can be found. To call MuPAD commands from the MATLAB Command Window, use the `evalin` or `feval` function:

```
evalin(symengine, 'numeric::solve(sin(x) =
x^2 - 1, x = 0..2)')

ans =
1.4096240040025962492355939705895
```

Return Parameterized Solutions

Solve these trigonometric equations:

```
syms x
solve(sin(1/sqrt(x)) == 0, x)
solve(sin(1/x) == 0, x)
```

For the first equation, the solver returns the solution with one parameter and issues a warning indicating the values of the parameter. For the second equation, the solver chooses one value of the parameter and returns the solution corresponding to this value:

```
Warning: The solutions are parametrized by the symbols:
k = Z_ intersect Dom::Interval([0], infinity)

ans =
1/(pi^2*k^2)

ans =
1/pi
```

Return Real Solutions

Solve this equation:

```
syms x
solve(x^5 == 3125, x)
```

This equation has five solutions:

```
ans =
                                                                   5
   (5*5^(1/2))/4 + (2^(1/2)*(5^(1/2) + 5)^(1/2)*5*i)/4 - 5/4
   (5*5^(1/2))/4 - (2^(1/2)*(5^(1/2) + 5)^(1/2)*5*i)/4 - 5/4
   (2^(1/2)*(5 - 5^(1/2))^(1/2)*5*i)/4 - (5*5^(1/2))/4 - 5/4
 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*5*i)/4 - (5*5^(1/2))/4 - 5/4
```

If you need a solution in real numbers, use `Real`. The only real solution of this equation is 5:

```
solve(x^5 == 3125, x, 'Real', true)

ans =
5
```

Return One Solution

Solve this equation:

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

Instead of returning an infinite set of periodic solutions, the solver picks these three solutions that it considers to be most practical:

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

To pick only one solution, use `PrincipalValue`:

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

ans =
0
```

Apply Simplification Rules That Shorten the Result

Solve this equation. By default, the solver returns a complete, but rather long and complicated solution:

```
syms x
solve(x^(7/2) + 1/x^(7/2) == 1, x)

ans =
                 1/((3^(1/2)*i)/2 + 1/2)^(2/7)
                 1/(1/2 - (3^(1/2)*i)/2)^(2/7)
   exp((pi*4*i)/7)/(3^(1/2)*(i/2) + 1/2)^(2/7)
  exp((pi*4*i)/7)/(3^(1/2)*(-i/2) + 1/2)^(2/7)
  -exp((pi*3*i)/7)/(3^(1/2)*(i/2) + 1/2)^(2/7)
```

```
 -exp((pi*3*i)/7)/(3^(1/2)*(-i/2) + 1/2)^(2/7)
```

To apply the simplification rules that shorten the result, use `IgnoreAnalyticConstraints`:

```
solve(x^(7/2) + 1/x^(7/2) == 1, x,...
'IgnoreAnalyticConstraints', true)

ans =
 1/((3^(1/2)*i)/2 + 1/2)^(2/7)
 1/(1/2 - (3^(1/2)*i)/2)^(2/7)
```

Ignore Assumptions on Variables

The `sym` and `syms` functions let you set assumptions for symbolic variables. For example, declare that the variable x can have only positive values:

```
syms x positive
```

When you solve an equation or a system of equations with respect to such a variable, the solver verifies the results against the assumptions and returns only the solutions consistent with the assumptions:

```
solve(x^2 + 5*x - 6 == 0, x)

ans =
1
```

To ignore the assumptions and return all solutions, use `IgnoreProperties`:

```
solve(x^2 + 5*x - 6 == 0, x, 'IgnoreProperties', true)

ans =
  1
 -6
```

For further computations, clear the assumption that you set for the variable x:

```
syms x clear
```

Specify Maximal Degree of Polynomials for Which the Solver Uses Explicit Formulas

When you solve a higher-order polynomial equation, the solver sometimes uses `RootOf` to return the results:

```
syms x a
solve(x^4 + 2*x + a == 0, x)

ans =
RootOf(z^4 + 2*z + a, z)
```

To get an explicit solution for such equations, try calling the solver with `MaxDegree`. The option specifies the maximal degree of polynomials for which the solver tries to return explicit solutions. The default value is 3. Increasing this value, you can get explicit solutions for higher-order polynomials. For example, increase the value of `MaxDegree` to 4 and get explicit solutions instead of `RootOf` for the fourth-order polynomial:

```
s = solve(x^4 + 2*x + a == 0, x, 'MaxDegree', 4);
pretty(s)
```

```
+-           -+
|   #2 - #3   |
|             |
|   #3 + #1   |
|             |
|   #3 - #1   |
|             |
|  - #3 - #2  |
+-           -+

where

          1/2       1/2            1/2           1/2  1/2   1/2           3 1/2     1/2 1/2
         3    (- 3 3    #4 #5 - 4 3    a #4 - 4 3    6    (3    (27 - 16 a )    + 9)   )
```

```
#1 == -----------------------------------------------------------------------------------
                                        /    1/2           3 1/2     \1/6
                                    1/4 | 2 3    (27 - 16 a )        |
                     6 (12 a + 9 #5)    | ---------------------- + 2 |
                                        \           9                /

       1/2     1/2  1/2   1/2           3 1/2     1/2       1/2           1/2       1/2
      3    (4 3    6    (3    (27 - 16 a )    + 9)     - 4 3    a #4 - 3 3    #4 #5)
#2 == ---------------------------------------------------------------------------------
                                       /    1/2           3 1/2     \1/6
                                   1/4 | 2 3    (27 - 16 a )        |
                    6 (12 a + 9 #5)    | ---------------------- + 2 |
                                       \           9                /

                     1/2
                    3    #4
#3 == -----------------------------------
        /    1/2           3 1/2     \1/6
        | 2 3    (27 - 16 a )        |
      6 | ---------------------- + 2 |
        \           9                /

                  1/2
#4 == (4 a + 3 #5)

      /    1/2           3 1/2     \2/3
      | 2 3    (27 - 16 a )        |
#5 == | ---------------------- + 2 |
      \           9                /
```

Algorithms

When you use `IgnoreAnalyticConstraints`, the solver applies these rules to the expressions on both sides of an equation:

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

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

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

  $(a^b)^c = a^{b \cdot c}$.
- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex values x. In particular:
  - $\log(e^x) = x$
  - $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
  - $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k
- The solver can multiply both sides of an equation by any expression except `0`.
- The solutions of polynomial equations must be complete.

See Also

`dsolve | symvar | vpasolve`

Related Examples

- "Solve an Algebraic Equation" on page 2-83
- "Solve a System of Algebraic Equations" on page 2-85

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

Purpose

Matrix square root

Syntax

```
X = sqrtm(A)
[X,resnorm] = sqrtm(A)
```

Description

`X = sqrtm(A)` returns a matrix `X`, such that $X^2 = A$ and the eigenvalues of `X` are the square roots of the eigenvalues of `A`.

`[X,resnorm] = sqrtm(A)` returns a matrix `X` and the residual `norm(A-X^2,'fro')/norm(A,'fro')`.

Tips

- Calling `sqrtm` for a matrix that is not a symbolic object invokes the MATLAB `sqrtm` function.
- Matrix `A` must have a Jordan canonical form. Otherwise, `sqrtm` cannot compute the square root of `A`.
- If `A` has an eigenvalue 0 of geometric multiplicity higher than 1, the square root of `A` does not exist.

Input Arguments

A

Symbolic matrix.

Output Arguments

X

Matrix, such that $X^2 = A$.

resnorm

Residual computed as `norm(A-X^2,'fro')/norm(A,'fro')`.

Definitions

Square Root of a Matrix

The square root of a matrix `A` is the matrix `X`, such that $X^2 = A$ and the eigenvalues of `X` are the square roots of the eigenvalues of `A`.

Examples

Compute the square root of this matrix. Because these numbers are not symbolic objects, you get floating-point results.

```
A = [2 -2 0; -1 3 0; -1/3 5/3 2];
X = sqrtm(A)

X =
    1.3333   -0.6667    0.0000
   -0.3333    1.6667   -0.0000
   -0.0572    0.5286    1.4142
```

Now, convert this matrix to a symbolic object, and compute its square root again:

```
A = sym([2 -2 0; -1 3 0; -1/3 5/3 2]);
X = sqrtm(A)

X =
[               4/3,           -2/3,       0]
[              -1/3,            5/3,       0]
[ (2*2^(1/2))/3 - 1, 1 - 2^(1/2)/3, 2^(1/2)]
```

Check the correctness of the result:

```
isAlways(X^2 == A)

ans =
     1     1     1
     1     1     1
     1     1     1
```

Use the syntax with two output arguments to return the square root of a matrix and the residual:

```
A = vpa(sym([0 0; 0 5/3]), 100);
[X,resnorm] = sqrtm(A)

X =
[ 0,                                 0]
[ 0, 1.2909944487358056283930884665941]
```

```
resnorm =
2.938735877055718769921841343O556e-40
```

See Also

`cond` | `eig` | `expm` | `jordanlinalg::sqrtMatrix` | `norm` | `sqrtm`

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) - (- b^2/(9*a^2) + c/(3*a))/s^(1/3)
(- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 +...
(3^(1/2)*(s^(1/3) + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)
(- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 -...
(3^(1/2)*(s^(1/3) + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)

s =
((d/(2*a) + b^3/(27*a^3) - (b*c)/(6*a^2))^2 +...
(- b^2/(9*a^2) + c/(3*a))^3)^(1/2) - b^3/(27*a^3) -...
d/(2*a) + (b*c)/(6*a^2)
```

See Also

`pretty` | `simple` | `subs`

How To

- "Substitute with subexpr" on page 2-41

Purpose

Symbolic substitution

Syntax

```
subs(s,old,new)
subs(s,new)
subs(s)
```

Description

`subs(s,old,new)` returns a copy of `s` replacing all occurrences of `old` with `new`, and then evaluating `s`.

`subs(s,new)` returns a copy of `s` replacing all occurrences of the default variable in `s` with `new`, and then evaluating `s`. The default variable is defined by `symvar`.

`subs(s)` returns a copy of `s` replacing symbolic variables in `s` with their values obtained from the calling function and the MATLAB workspace, and then evaluating `s`. Variables with no assigned values remain as variables.

Compatibility

`subs(s,old,new,0)` will not accept `0` in a future release. Use `subs(s,old,new)` instead.

In previous releases, `subs(s,old,new,0)` prevented switching the arguments `old` and `new` if `subs(s,old,new)` returned `s`. The `subs` function does not switch `old` and `new` anymore.

Tips

- `subs(s,old,new)` does not modify `s`. To modify `s`, use `s = subs(s,old,new)`.
- If `old` and `new` are both vectors or cell arrays of the same size, `subs` replaces each element of `old` by the corresponding element of `new`.
- If `old` is a scalar, and `new` is a vector or matrix, then `subs(s,old,new)` replaces all instances of `old` in `s` with `new`, performing all operations elementwise. All constant terms in `s` are replaced with the constant times a vector or matrix of all 1s.

- If s is a univariate polynomial and new is a numeric matrix, use polyvalm(sym2poly(s), new) to evaluate s in the matrix sense. All constant terms are replaced with the constant times an identity matrix.

Input Arguments

s - Input

symbolic variable | symbolic expression | symbolic equation | symbolic function | symbolic array | symbolic vector | symbolic matrix

Input specified as a symbolic variable, expression, equation, function, array, vector, or matrix.

old - Existing element that needs to be replaced

symbolic variable | symbolic expression | string representing variable or expression | symbolic array | symbolic vector | symbolic matrix | array of strings | vector of strings | matrix of strings

Existing element that needs to be replaced specified as a symbolic variable, expression, string, array, vector, or matrix.

new - New element

number | symbolic variable | symbolic expression | string representing variable or expression | symbolic array | symbolic vector | symbolic matrix | array of strings | vector of strings | matrix of strings | structure array

New element specified as a number, variable, expression, string, array, vector, matrix, or structure array.

Examples

Single Substitution

Replace a with 4 in this expression:

```
syms a b
subs(a + b, a, 4)
```

```
ans =
b + 4
```

Replace a*b with 5 in this expression:

```
subs(a*b^2, a*b, 5)

ans =
5*b
```

Value That Gets Substituted by Default

Substitute the default value in this expression with a. If you do not specify which variable or expression that you want to replace, subs uses symvar to find the default variable. For x + y, the default variable is x:

```
syms x y a
symvar(x + y, 1)

ans =
x
```

Therefore, subs replaces x with a:

```
subs(x + y, a)

ans =
a + y
```

Single Input

Solve this ordinary differential equation:

```
syms a y(t)
y = dsolve(diff(y) == -a*y)

y =
C2*exp(-a*t)
```

Now, specify the values of the symbolic parameters a and C2:

```
a = 980; C2 = 3;
```

Although the values `a` and `C2` are now in the MATLAB workspace, `y` is not evaluated with the account of these values:

```
y

y =
C2*exp(-a*t)
```

To evaluate `y` taking into account the new values of `a` and `C2`, use `subs`:

```
subs(y)

ans =
3*exp(-980*t)
```

Multiple Substitutions

Make multiple substitutions by specifying the old and new values as vectors:

```
syms a b
subs(cos(a) + sin(b), [a, b], [sym('alpha'), 2])

ans =
sin(2) + cos(alpha)
```

You also can use cell arrays for that purpose:

```
subs(cos(a) + sin(b), {a, b}, {sym('alpha'), 2})

ans =
sin(2) + cos(alpha)
```

Scalar Expansion

Replace variable `a` in this expression with the 3-by-3 magic square matrix. Note that the constant `1` expands to the 3-by-3 matrix with all its elements equal to `1`:

```
syms a t
subs(exp(a*t) + 1, a, -magic(3))
```

```
ans =
[ exp(-8*t) + 1,   exp(-t) + 1, exp(-6*t) + 1]
[ exp(-3*t) + 1, exp(-5*t) + 1, exp(-7*t) + 1]
[ exp(-4*t) + 1, exp(-9*t) + 1, exp(-2*t) + 1]
```

Multiple Scalar Expansion

Replace variables `x` and `y` with these 2-by-2 matrices. When you make multiple substitutions involving vectors or matrices, use cell arrays to specify the old and new values:

```
syms x y
subs(x*y, {x, y}, {[0 1; -1 0], [1 -1; -2 1]})

ans =
[ 0, -1]
[ 2,  0]
```

Note that these substitutions are elementwise:

```
[0 1; -1 0].*[1 -1; -2 1]

ans =
     0    -1
     2     0
```

Substitutions in Equations

Replace `sin(x + 1)` with `a` in this equation:

```
syms x a
subs(sin(x + 1) + 1 == x, sin(x + 1), a)

ans =
a + 1 == x
```

Substitutions in Functions

Replace `x` with `a` in this symbolic function:

```
syms x y a
syms f(x, y);
```

```
f(x, y) = x + y;
f = subs(f, x, a)

f(x, y) =
a + y
```

`subs` replaces the values in the symbolic function formula, but does not replace input arguments of the function:

```
formula(f)
argnames(f)

ans =
a + y

ans =
[ x, y]
```

You can replace the arguments of a symbolic function explicitly:

```
syms x y
f(x, y) = x + y;
f(x, a) = subs(f, x, a);
f

f(x, a) =
a + y
```

Original Expression

Assign the expression `x + y` to `s`:

```
syms x y
s = x + y;
```

Replace `y` in this expression with the value `1`. Here, `s` itself does not change:

```
subs(s, y, 1); s
```

```
s =
x + y
```

To replace the value of `s` with the new expression, assign the result returned by `subs` to `s`:

```
s = subs(s, y, 1); s

s =
x + 1
```

Structure Array

Suppose you want to verify the solutions of this system of equations:

```
syms x y
eqs = [x^2 + y^2 == 1, x == y];
S = solve(eqs, x, y);
S.x
S.y

ans =
  2^(1/2)/2
 -2^(1/2)/2

ans =
  2^(1/2)/2
 -2^(1/2)/2
```

To verify the correctness of the returned solutions, substitute the solutions into the original system:

```
logical(subs(eqs, S))

ans =
     1     1
     1     1
```

See Also

`double` | `evalevalAt` | `simplify` | `subexpr` | `subs` | `subset` | `subsex` | `subsop` | `vpa`

Related Examples

- “Substitutions in Symbolic Expressions” on page 1-19
- “Substitute with subs” on page 2-43
- “Combine subs and double for Numeric Evaluations” on page 2-47

Purpose

Singular value decomposition of symbolic matrix

Syntax

```
sigma = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X,'econ')
```

Description

`sigma = svd(X)` returns a vector `sigma` containing the singular values of a symbolic matrix `A`.

`[U,S,V] = svd(X)` returns 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 the condition `A = U*S*V'`, where `V'` is the Hermitian transpose (the complex conjugate of the transpose) of `V`. The singular vector computation uses variable-precision arithmetic. `svd` does not compute symbolic singular vectors. Therefore, the input matrix `X` must be a matrix of symbolic numbers.

`[U,S,V] = svd(X,0)` produces the "economy size" decomposition. If `X` is an `m`-by-`n` matrix with `m > n`, then `svd` computes only the first `n` columns of `U`. In this case, `S` is an `n`-by-`n` matrix. For `m <= n`, this syntax is equivalent to `svd(X)`.

`[U,S,V] = svd(X,'econ')` also produces the "economy size" decomposition. If `X` is an `m`-by-`n` matrix with `m >= n`, then this syntax is equivalent to `svd(X,0)`. For `m < n`, `svd` computes only the first `m` columns of `V`. In this case, `S` is an `m`-by-`m` matrix.

Tips

- The second arguments `0` and `'econ'` only affect the shape of the returned matrices. These arguments do not affect the performance of the computations.
- Calling `svd` for numeric matrices that are not symbolic objects invokes the MATLAB `svd` function.

Input Arguments

X - Input matrix
symbolic matrix

Input matrix specified as a symbolic matrix. For syntaxes with one output argument, the elements of `X` can be symbolic numbers, variables, expressions, or functions. For syntaxes with three output arguments, the elements of `X` must be numeric.

Output Arguments

sigma - Singular values
symbolic vector | vector of symbolic numbers

Singular values of a matrix, returned as a symbolic or numeric vector. If `sigma` is a numeric vector, then its elements are sorted in descending order.

U - Singular vectors
matrix of symbolic numbers

Singular vectors, returned as a numeric unitary matrix. Each column of this matrix is a singular vector.

S - Singular values
matrix of symbolic numbers

Singular values, returned as a diagonal matrix. Diagonal elements of this matrix appear in descending order.

V - Singular vectors
matrix of symbolic numbers

Singular vectors, returned as a numeric unitary matrix. Each column of this matrix is a singular vector.

Examples

Symbolic Singular Values

Compute the singular values of the symbolic 4-by-4 magic square:

```
A = sym(magic(4));
sigma = svd(A)
sigma =
        34
 8*5^(1/2)
 2*5^(1/2)
         0
```

Now, compute singular values of the matrix whose elements are symbolic expressions:

```
syms t real
A = [0 1; -1 0];
E = expm(t*A)
sigma = svd(E)

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

sigma =
 (cos(t)^2 + sin(t)^2)^(1/2)
 (cos(t)^2 + sin(t)^2)^(1/2)
```

Simplify the result:

```
sigma = simplify(sigma)

sigma =
 1
 1
```

For further computations, remove the assumption:

```
syms t clear
```

Floating-Point Singular Values

Convert the elements of the symbolic 4-by-4 magic square to floating-point numbers, and compute the singular values of the matrix:

```
A = sym(magic(4));
```

```
sigma = svd(vpa(A))

sigma =
                                                 34.0
                           17.88854381999831757127338934985
                          4.4721359549995793928183473374626
 0.0000000000000000000042127245515076439434819165724023*i
```

Singular Values and Singular Vectors

Compute the singular values and singular vectors of the 4-by-4 magic square:

```
old = digits(10);
A = sym(magic(4))
[U, S, V] = svd(A)
digits(old);

A =
[ 16,  2,  3, 13]
[  5, 11, 10,  8]
[  9,  7,  6, 12]
[  4, 14, 15,  1]

U =
[ 0.5,  0.6708203932,  0.5, -0.2236067977]
[ 0.5, -0.2236067977, -0.5, -0.6708203932]
[ 0.5,  0.2236067977, -0.5,  0.6708203932]
[ 0.5, -0.6708203932,  0.5,  0.2236067977]

S =
[ 34.0,           0,           0,              0]
[    0, 17.88854382,           0,              0]
[    0,           0, 4.472135955,              0]
[    0,           0,           0, 1.108401846e-15]

V =
[ 0.5,  0.5,  0.6708203932,  0.2236067977]
[ 0.5, -0.5, -0.2236067977,  0.6708203932]
[ 0.5, -0.5,  0.2236067977, -0.6708203932]
[ 0.5,  0.5, -0.6708203932, -0.2236067977]
```

Compute the product of `U`, `S`, and the Hermitian transpose of `V` with the 10-digit accuracy. The result is the original matrix `A` with all its elements converted to floating-point numbers:

```
vpa(U*S*V',10)

ans =
[ 16.0,  2.0,  3.0, 13.0]
[  5.0, 11.0, 10.0,  8.0]
[  9.0,  7.0,  6.0, 12.0]
[  4.0, 14.0, 15.0,  1.0]
```

"Economy Size" Decomposition

Use the second input argument `0` to compute the "economy size" decomposition of this `2`-by-`3` matrix:

```
old = digits(10);
A = sym([1 1;2 2; 2 2]);
[U, S, V] = svd(A, 0)

U =
[ 0.3333333333, -0.6666666667]
[ 0.6666666667,  0.6666666667]
[ 0.6666666667, -0.3333333333]

S =
[ 4.242640687, 0]
[           0, 0]

V =
[ 0.7071067812,  0.7071067812]
[ 0.7071067812, -0.7071067812]
```

Now, use the second input argument `'econ'` to compute the "economy size" decomposition of matrix `B`. Here, the `3`-by-`2` matrix `B` is the transpose of `A`.

```
B = A';
[U, S, V] = svd(B, 'econ')
digits(old);
```

```
U =
[ 0.7071067812, -0.7071067812]
[ 0.7071067812,  0.7071067812]

S =
[ 4.242640687, 0]
[           0, 0]

V =
[ 0.3333333333,  0.6666666667]
[ 0.6666666667, -0.6666666667]
[ 0.6666666667,  0.3333333333]
```

See Also

`digits | eig | invnumeric::singularvalues | numeric::singularvectors | numeric::svd | svd | vpa`

Related Examples

- "Singular Value Decomposition" on page 2-71

Purpose

Create symbolic objects

Syntax

```
var = sym('var')
var = sym('var',set)
sym('var','clear')
Num = sym(Num)
Num = sym(Num,flag)
A = sym('A',dim)
A = sym(A,set)
sym(A,'clear')
f(arg1,...,argN) = sym('f(arg1,...,argN)')
```

Description

`var = sym('var')` creates the symbolic variable `var`.

`var = sym('var',set)` creates the symbolic variable `var` and states that `var` belongs to `set`.

`sym('var','clear')` clears assumptions previously set on the symbolic variable `var`.

`Num = sym(Num)` converts a number or a numeric matrix `Num` to symbolic form.

`Num = sym(Num,flag)` converts a number or a numeric matrix `Num` to symbolic form. The second argument specifies the technique for converting floating-point numbers.

`A = sym('A',dim)` creates a vector or a matrix of symbolic variables.

`A = sym(A,set)`, where `A` is an *existing* symbolic vector or matrix, sets an assumption that all elements of `A` belong to `set`. This syntax does not create `A`. To create a symbolic vector or a symbolic matrix `A`, use `A = sym('A',[m n])` or `A = sym('A',n)`.

`sym(A,'clear')`, where `A` is an *existing* symbolic vector or matrix, clears assumptions previously set on elements of `A`. This syntax does not create `A`. To create a symbolic vector or a symbolic matrix `A`, use `A = sym('A',[m n])` or `A = sym('A',n)`.

`f(arg1,...,argN) = sym('f(arg1,...,argN)')` creates the symbolic function `f` and specifies that `arg1,...,argN` are the input arguments of `f`. This syntax does not create symbolic variables `arg1,...,argN`. The arguments `arg1,...,argN` must be *existing* symbolic variables.

Tips

- For compatibility with previous versions, `sym('var','unreal')` is equivalent to `sym('var','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.
- `clear x` does *not* clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`. To remove assumptions, use one of these options:
  - `sym('x','clear')` removes assumptions from `x` without affecting any other symbolic variables.
  - `reset(symengine)` resets the symbolic engine and therefore removes assumptions on all variables. The variables themselves remain in the MATLAB workspace.
  - `clear all` clears all objects in the MATLAB workspace and resets the symbolic engine.
- If the input is a function handle, then the result is the symbolic form of the body of the function handle.

Input Arguments

var

String that represents the variable name. It must begin with a letter and can contain only alphanumeric characters.

set

Either `real` or `positive`.

Num

Number, vector, or matrix of numbers.

flag

One of these strings: `r`, `d`, `e`, or `f`.

- `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 might 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)`.
- `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 might 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`.
- `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)`.
- `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` .

  Default: `r`

A

String that represents the base for generated names of vector or matrix elements. It must be a valid variable name. (To verify if the name is a valid variable name, use `isvarname`.)

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 values of `k`, `i`, and `j` range from 1 to `m` or 1 to `n`. To specify another 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`.

dim

Integer or vector of two integers specifying dimensions of `A`. For example, if `dim` is a vector `[m n]`, then the syntax `A = sym('A',[m n])` creates an `m`-by-`n` matrix of symbolic variables. If `dim` is an integer `n`, then the syntax `A = sym('A',n)` creates a square `n`-by-`n` matrix of symbolic variables.

f

Name of a symbolic function. It must begin with a letter and contain only alphanumeric characters.

arg1,...,argN

Arguments of a symbolic function. Each argument must be an *existing* symbolic variable.

Output Arguments

var

Symbolic variable.

Num

Symbolic number or vector or matrix of symbolic numbers.

A

Vector or matrix of automatically generated symbolic variables.

f

Symbolic function.

Examples

Create the symbolic variables x and y:

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

Create the symbolic variables x and y assuming that x is real and y is positive:

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

Check the assumptions on x and y using assumptions:

```
assumptions

ans =
[ x in R_, 0 < y]
```

For further computations, clear the assumptions:

```
sym('x','clear');
sym('y','clear');
assumptions

ans =
[ empty sym ]
```

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 =
0.33333333333333333148296162562473909929395

e =
1/3 - eps/12
```

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
```

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
```

Create the symbolic function f whose input arguments are symbolic variables x and y:

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

f(x, y) =
f(x, y)
```

Alternatively, you can use the assignment operation to create the symbolic function `f`:

```
f(x, y) = x + y

f(x, y) =
x + y
```

Alternatives

- To create several symbolic variables in one function call, use `syms`. When using `syms`, do not enclose variables in quotes and do not use commas between variable names:

  ```
  syms var1 var2 var3
  ```

  `syms` also lets you create real variables or positive variables. It also lets you clear assumptions set on a variable.
- `assume` and `assumeAlso` provide more flexibility for setting assumptions on variable.
- When creating a symbolic function, use `sym` to create `arg1,...,argN` as symbolic variables. Then use the assignment operation to create the symbolic function `f`, for example:

  ```
  x = sym('x');
  y = sym('y');
  f(x, y) = x + y
  ```
- `syms f(x, y)` is equivalent to these commands:

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

See Also

`assume` | `assumeAlso` | `assumptions` | `clear` | `clear all` | `digits` | `double` | `eps` | `reset` | `symfun` | `syms` | `symvar`

Concepts

- "Create Symbolic Variables and Expressions" on page 1-8
- "Create Symbolic Functions" on page 1-10

- “Assumptions on Symbolic Objects” on page 1-35
- “Estimate Precision of Numeric to Symbolic Conversions” on page 1-22

sym2poly

Purpose

Symbolic-to-numeric polynomial conversion

Syntax

```
c = sym2poly(s)
```

Description

`c = sym2poly(s)` returns a row vector containing the numeric coefficients of a symbolic polynomial. The coefficients are ordered in descending powers of the polynomial's independent variable. In other words, the vector's first entry contains the coefficient of the polynomial's highest term; the second entry, the coefficient of the second highest term; and so on.

Examples

The 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`

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

`evalin` | `feval` | `read`

Purpose

Create symbolic functions

Syntax

```
f = symfun(formula,inputs)
```

Description

`f = symfun(formula,inputs)` creates the symbolic function `f` and symbolic variables `inputs` representing its input arguments. The symbolic expression `formula` defines the body of the function `f`.

Input Arguments

formula

Symbolic expression or vector or matrix of symbolic expressions. This argument represents the body of `f`. If it contains other symbolic variables besides `inputs`, those variables must already exist in the MATLAB workspace.

inputs

Array that contains input arguments of `f`. For each argument, `symfun` creates a symbolic variable. Argument names must begin with a letter and can contain only alphanumeric characters.

Output Arguments

f

Symbolic function. The name of a symbolic function must begin with a letter and contain only alphanumeric characters.

Examples

Create the symbolic variables `x` and `y`. Then use `symfun` to create the symbolic function `f(x, y) = x + y`:

```
syms x y
f = symfun(x + y, [x y])
```

```
f(x, y) =
x + y
```

Create the symbolic variables `x` and `y`. Then use `symfun` to create an arbitrary symbolic function `f(x, y)`. An arbitrary symbolic function does not have a mathematical expression assigned to it.

```
syms x y
f = symfun(sym('f(x, y)'), [x y])
```

```
f(x, y) =
f(x, y)
```

Alternatives

Use the assignment operation to simultaneously create a symbolic function and define its body. The arguments `x` and `y` must be symbolic variables in the MATLAB workspace.

```
syms x y
f(x, y) = x + y
```

Use `syms` to create an arbitrary symbolic function `f(x, y)`. The following command creates the symbolic function `f` and the symbolic variables `x` and `y`.

```
syms f(x, y)
```

Use `sym` to create an arbitrary symbolic function `f(x, y)`. The arguments `x` and `y` must be symbolic variables in the MATLAB workspace.

```
syms x y
f(x, y) = sym('f(x, y)')
```

See Also

`argnames` | `dsolve` | `formula` | `matlabFunction` | `odeToVectorField` | `sym` | `syms` | `symvar`

Concepts

- “Create Symbolic Functions” on page 1-10

Purpose

Product of series

Syntax

```
symprod(expr)
symprod(expr,v)
symprod(expr,a,b)
symprod(expr,v,a,b)
```

Description

`symprod(expr)` evaluates the product of a series, where expression `expr` defines the terms of a series, with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from 1 to `defaultVar`.

`symprod(expr,v)` evaluates the product of a series, where expression `expr` defines the terms of a series, with respect to the symbolic variable `v`. The value of the variable `v` changes from 1 to `v`.

`symprod(expr,a,b)` evaluates the product of a series, where expression `expr` defines the terms of a series, with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `a` to `b`.

`symprod(expr,v,a,b)` evaluates the product of a series, where expression `expr` defines the terms of a series, with respect to the symbolic variable `v`. The value of the variable `v` changes from `a` to `b`.

Tips

- `symprod` does not compute indefinite products.

Input Arguments

expr

Symbolic expression.

v

Symbolic variable representing the product index.

a

Symbolic number, variable, or expression representing the lower bound of the product index.

b

Symbolic number, variable, or expression representing the upper bound of the product index.

Definitions

Definite Product

The definite product of a series is defined as

$$\prod_{i=a}^{b} x_i = x_a \cdot x_{a+1} \cdot \ldots \cdot x_b$$

Indefinite Product

$$f(i) = \prod_{i} x_i$$

is called the indefinite product of x_i over i, if the following identity holds for all values of i:

$$\frac{f(i+1)}{f(i)} = x_i$$

Examples

Evaluate the product of a series for the symbolic expressions `k` and `k^2`:

```
syms k
symprod(k)
symprod((2*k - 1)/k^2)

ans =
factorial(k)

ans =
(1/2^(2*k)*2^(k + 1)*factorial(2*k))/(2*factorial(k)^3)
```

Evaluate the product of a series for these expressions specifying the limits:

```
syms k
symprod(1 - 1/k^2, k, 2, Inf)
symprod(k^2/(k^2 - 1), k, 2, Inf)

ans =
1/2

ans =
2
```

Evaluate the product of a series for this multivariable expression with respect to `k`:

```
syms k x
symprod(exp(k*x)/x, k, 1, 10000)

ans =
exp(50005000*x)/x^10000
```

See Also

`int` | `syms` | `symsum` | `symvar`

syms

Purpose

Shortcut for creating symbolic variables and functions

Syntax

```
syms var1 ... varN
syms var1 ... varN set
syms var1 ... varN clear
syms f(arg1,...,argN)
```

Description

`syms var1 ... varN` creates symbolic variables `var1 ... varN`.

`syms var1 ... varN set` creates symbolic variables `var1 ... varN` and states that these variables belong to `set`.

`syms var1 ... varN clear` removes assumptions previously set on symbolic variables `var1 ... varN`.

`syms f(arg1,...,argN)` creates the symbolic function `f` and symbolic variables `arg1,...,argN` representing the input arguments of `f`.

Tips

- For compatibility with previous versions, `syms var1 ... varN unreal` is equivalent to `syms var1 ... varN clear`.
- In functions and scripts, do not use `syms` 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 `sym`. For example:

  ```
  alpha = sym('alpha')
  ```

- `clear x` does *not* clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`. To remove assumptions, use one of these options:
  - `syms x clear` removes assumptions from `x` without affecting any other symbolic variables.
  - `reset(symengine)` resets the symbolic engine and therefore removes assumptions on all variables. The variables themselves remain in the MATLAB workspace.

- `clear all` removes all objects in the MATLAB workspace and resets the symbolic engine.

Input Arguments

var1 ... varN

Names of symbolic variables. Each name must begin with a letter and contain only alphanumeric characters.

set

Either `real` or `positive`.

f

Name of a symbolic function. It must begin with a letter and contain only alphanumeric characters.

arg1,...,argN

Arguments of a symbolic function. For each argument, `syms` creates a symbolic variable. Argument names must begin with a letter and contain only alphanumeric characters.

Examples

Create symbolic variables `x` and `y` using `syms`:

```
syms x y
```

Create symbolic variables `x` and `y`, and assume that they are real:

```
syms x y real
```

To see assumptions set on `x` and `y`, use `assumptions`:

```
assumptions(x)
assumptions(y)

ans =
x in R_
```

```
ans =
y in R_
```

Clear the assumptions that x and y are real:

```
syms x y clear
assumptions

ans =
[ empty sym ]
```

Create a symbolic function f that accepts two arguments, x and y:

```
syms f(x, y)
```

Specify the formula for this function:

```
f(x, y) = x + 2*y

f(x, y) =
x + 2*y
```

Compute the function value at the point x = 1 and y = 2:

```
f(1, 2)

ans =
5
```

Create symbolic function f and specify its formula by this symbolic matrix:

```
syms x
f(x) = [x x^2; x^3 x^4];
```

Compute the function value at the point x = 2:

```
f(2)

ans =
[ 2,  4]
[ 8, 16]
```

Now compute the value of this function for `x = [1 2; 3 4]`. The result is a cell array of symbolic matrices:

```
y = f([1 2; 3 4])

y =
    [2x2 sym]    [2x2 sym]
    [2x2 sym]    [2x2 sym]
```

To access the contents of each cell in a cell array, use braces:

```
y{1}

ans =
[ 1, 2]
[ 3, 4]

y{2}

ans =
[  1,  8]
[ 27, 64]

y{3}

ans =
[ 1,  4]
[ 9, 16]

y{4}

ans =
[  1,  16]
[ 81, 256]
```

Alternatives

- `syms` is a shortcut for `sym`. This shortcut lets you create several symbolic variables in one function call. Alternatively, you can use `sym` and create each variable separately:

  ```
  var1 = sym('var1');
  ...
  varN = sym('varN');
  ```

  `sym` also lets you create real variables or positive variables. It also lets you clear assumptions set on a variable.
- `assume` and `assumeAlso` provide more flexibility for setting assumptions on variable.
- When creating a symbolic function, use `syms` to create `arg1,...,argN` as symbolic variables. Then use the assignment operation to create the symbolic function `f`, for example:

  ```
  syms x y
  f(x, y) = x + y
  ```

See Also

`assume` | `assumeAlso` | `assumptions` | `clear all` | `reset` | `sym` | `symfun` | `symvar`

Concepts

- "Create Symbolic Variables and Expressions" on page 1-8
- "Create Symbolic Functions" on page 1-10
- "Assumptions on Symbolic Objects" on page 1-35

Purpose

Sum of series

Syntax

```
symsum(expr)
symsum(expr,v)
symsum(expr,a,b)
symsum(expr,v,a,b)
```

Description

`symsum(expr)` evaluates the sum of a series, where expression `expr` defines the terms of a series, with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from 0 to `defaultVar - 1`.

`symsum(expr,v)` evaluates the sum of a series, where expression `expr` defines the terms of a series, with respect to the symbolic variable `v`. The value of the variable `v` changes from 0 to `v - 1`.

`symsum(expr,a,b)` evaluates the sum of a series, where expression `expr` defines the terms of a series, with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `a` to `b`.

`symsum(expr,v,a,b)` evaluates the sum of a series, where expression `expr` defines the terms of a series, with respect to the symbolic variable `v`. The value of the variable `v` changes from `a` to `b`.

Tips

- `symsum` does not compute indefinite sums.

Input Arguments

expr

Symbolic expression.

v

Symbolic variable representing the summation index.

a

Symbolic number, variable, or expression representing the lower bound of the summation index.

b

Symbolic number, variable, or expression representing the upper bound of the summation index.

Definitions

Definite Sum

The definite sum of series is defined as

$$\sum_{i=a}^{b} x_i = x_a + x_{a+1} + \ldots + x_b$$

Indefinite Sum

$$f(i) = \sum_{i} x_i$$

is called the indefinite sum of x_i over i, if the following identity is true for all values of i:

$$f(i+1) - f(i) = x_i$$

Examples

Evaluate the sum of a series for the symbolic expressions `k` and `k^2`:

```
syms k
symsum(k)
symsum(1/k^2)
```

```
ans =
k^2/2 - k/2

ans =
-psi(1, k)
```

Evaluate the sum of a series for these expressions specifying the limits:

```
syms k
symsum(k^2, 0, 10)
symsum(1/k^2,1,Inf)

ans =
385

ans =
pi^2/6
```

Evaluate the sum of a series for this multivariable expression with respect to k:

```
syms k x
symsum(x^k/sym('k!'), k, 0, Inf)

ans =
exp(x)
```

See Also

int | symprod | syms | symvar

How To

- "Symbolic Summation" on page 2-21

symvar

Purpose

Find symbolic variables in symbolic expression, matrix, or function

Syntax

```
symvar(s)
symvar(s,n)
```

Description

`symvar(s)` returns a vector containing all the symbolic variables in `s` in alphabetical order with uppercase letters preceding lowercase letters.

`symvar(s,n)` returns a vector containing `n` symbolic variables in `s` alphabetically closest to `x`. If `s` is a symbolic function, `symvar(s,n)` returns the input arguments of `s` in front of other free variables in `s`.

Tips

- `symvar(s)` can return variables in a different order than `symvar(s,n)`.
- `symvar` does treat the constants `pi`, `i`, and `j` as variables.
- If there are no symbolic variables in `s`, `symvar` returns the empty vector.
- When performing differentiation, integration, substitution or solving equations, MATLAB uses the variable returned by `symvar(s,1)` as a default variable. For a symbolic expression or matrix, `symvar(s,1)` returns the variable closest to `x`. For a function, `symvar(s,1)` returns the first input argument of `s`.

Input Arguments

s

Symbolic expression, matrix, or function.

n

Integer.

Examples

Find all symbolic variables in the sum:

```
syms wa wb wx yx ya yb
symvar(wa + wb + wx + ya + yb + yx)
```

```
ans =
[ wa, wb, wx, ya, yb, yx]
```

Find all symbolic variables in this function:

```
syms x y a b
f(a, b) = a*x^2/(sin(3*y - b));
symvar(f)

ans =
[ a, b, x, y]
```

Now find the first three symbolic variables in `f`. For a symbolic function, `symvar` with two arguments returns the function inputs in front of other variables:

```
symvar(f, 3)

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

For a symbolic expression or matrix, `symvar` with two arguments returns variables sorted by their proximity to `x`:

```
symvar(a*x^2/(sin(3*y - b)), 3)

ans =
[ x, y, b]
```

Find the default symbolic variable of these expressions:

```
syms v z
g = v + z;
symvar(g, 1)

ans =
z

syms aaa aab
g = aaa + aab;
symvar(g, 1)

ans =
aaa

syms X1 x2 xa xb
g = X1 + x2 + xa + xb;
symvar(g, 1)

ans =
x2
```

Algorithms

When sorting the symbolic variables by their proximity to `x`, `symvar` uses this algorithm:

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.

See Also

`findsym` | `sym` | `symfun` | `syms`

Concepts

- "Find a Default Symbolic Variable" on page 1-15

Purpose

Taylor series expansion

Syntax

```
taylor(f)
taylor(f,Name,Value)
taylor(f,v)
taylor(f,v,Name,Value)
taylor(f,v,a)
taylor(f,v,a,Name,Value)
```

Description

`taylor(f)` computes the Taylor series expansion of `f` up to the fifth order. The expansion point is 0.

`taylor(f,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`taylor(f,v)` computes the Taylor series expansion of `f` with respect to `v`.

`taylor(f,v,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`taylor(f,v,a)` computes the Taylor series expansion of `f` with respect to `v` around the expansion point `a`.

`taylor(f,v,a,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Tips

- If you use both the third argument `a` and `ExpansionPoint` to specify the expansion point, the value specified via `ExpansionPoint` prevails.
- If `v` is a vector, then the expansion point `a` must be a scalar or a vector of the same length as `v`. If `v` is a vector and `a` is a scalar, then `a` is expanded into a vector of the same length as `v` with all elements equal to `a`.

Input Arguments

f

Symbolic expression.

v

Symbolic variable or vector of symbolic variables with respect to which you want to compute the Taylor series expansion.

> **Default:** Symbolic variable or vector of symbolic variables of `f` determined by `symvar`.

a

Real number (including infinities and symbolic numbers) specifying the expansion point. For multivariate Taylor series expansions, use a vector of numbers.

> **Default:** 0

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'ExpansionPoint'

Specify the expansion point `a`. The value `a` is a scalar or a vector.

> **Default:** If you specify the expansion point as a third argument `a` of `taylor`, then the value of that argument. Otherwise, 0.

'Order'

Specify the truncation order `n`, where `n` is a positive integer. `taylor` computes the Taylor polynomial approximation with the order `n-1`. The truncation order `n` is the exponent in the *O*-term: $O(v^{n})$.

> **Default:** 6

'OrderMode'

Specify whether you want to use absolute or relative order when computing the Taylor polynomial approximation. The value must be one of these strings: `Absolute` or `Relative`. *Absolute order* is the truncation order of the computed series. *Relative order* `n` means that the exponents of `v` in the computed series range from the leading order `m` to the highest exponent `m + n - 1`. Here `m + n` is the exponent of `v` in the *O*-term: $O(v^{m+n})$.

Default: `Absolute`

Definitions

Taylor Series Expansion

Taylor series expansion represents an analytic function $f(x)$ as an infinite sum of terms around the expansion point $x = a$:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots = \sum_{m=0}^{\infty} \frac{f^{(m)}(a)}{m!} \cdot (x-a)^m$$

Taylor series expansion requires a function to have derivatives up to an infinite order around the expansion point.

Maclaurin Series Expansion

Taylor series expansion around $x = 0$ is called Maclaurin series expansion:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots = \sum_{m=0}^{\infty} \frac{f^{(m)}(0)}{m!} x^m$$

Examples

Compute the Maclaurin series expansions of these functions:

```
syms x
taylor(exp(x))
taylor(sin(x))
taylor(cos(x))

ans =
x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1

ans =
x^5/120 - x^3/6 + x

ans =
x^4/24 - x^2/2 + 1
```

Compute the Taylor series expansions around $x = 1$ for these functions. The default expansion point is 0. To specify a different expansion point, use `ExpansionPoint`:

```
syms x
taylor(log(x), x, 'ExpansionPoint', 1)

ans =
x - (x - 1)^2/2 + (x - 1)^3/3 - (x - 1)^4/4
+ (x - 1)^5/5 - 1
```

Alternatively, specify the expansion point as the third argument of `taylor`:

```
taylor(acot(x), x, 1)

ans =
pi/4 - x/2 + (x - 1)^2/4 - (x - 1)^3/12 + (x - 1)^5/40 + 1/2
```

Compute the Maclaurin series expansion for this function. The default truncation order is 6. Taylor series approximation of this function does not have a fifth-degree term, so `taylor` approximates this function with the fourth-degree polynomial:

```
syms x
f = sin(x)/x;
t6 = taylor(f)
```

```
t6 =
x^4/120 - x^2/6 + 1
```

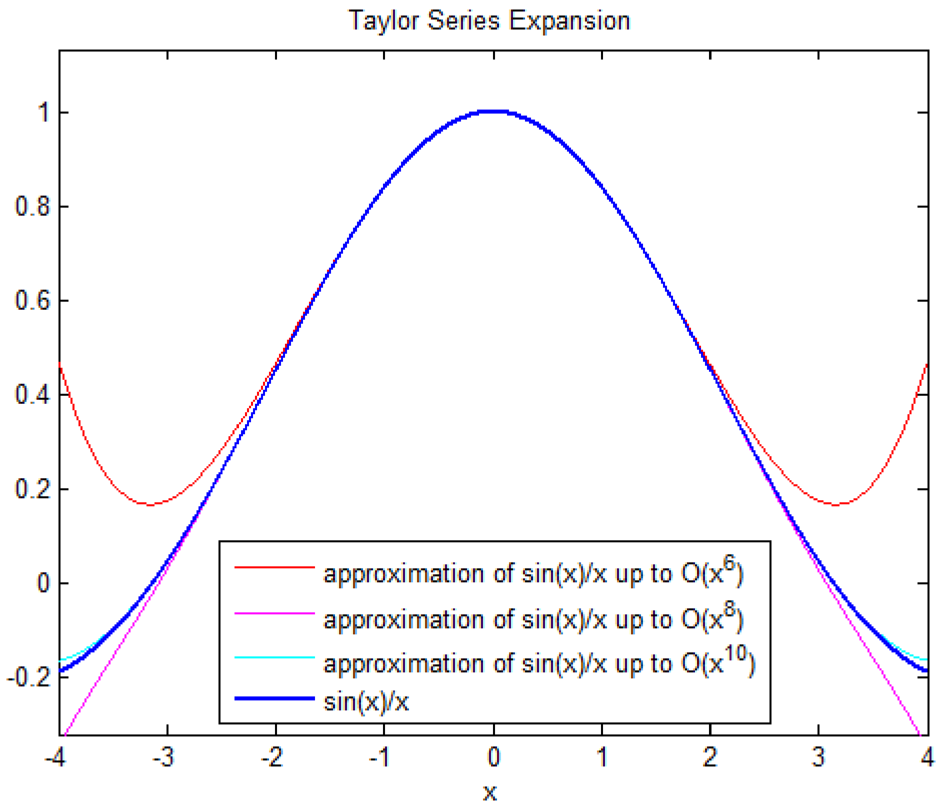
Use `Order` to control the truncation order. For example, approximate the function up to the orders 8 and 10:

```
t8 = taylor(f, 'Order', 8)
t10 = taylor(f, 'Order', 10)

t8 =
- x^6/5040 + x^4/120 - x^2/6 + 1

t10 =
x^8/362880 - x^6/5040 + x^4/120 - x^2/6 + 1
```

Plot the original function `f` and its approximations `t6`, `t8`, and `t10`. Note how the accuracy of the approximation depends on the truncation order.

```
plotT6 = ezplot(t6, [-4, 4]);
hold on
set(plotT6,'Color','red')

plotT8 = ezplot(t8, [-4, 4]);
set(plotT8,'Color','magenta')

plotT10 = ezplot(t10, [-4, 4]);
set(plotT10,'Color','cyan')

plotF = ezplot(f, [-4, 4]);
set(plotF,'Color','blue','LineWidth', 2)

legend('approximation of sin(x)/x up to O(x^6)',...
'approximation of sin(x)/x up to O(x^8)',...
'approximation of sin(x)/x up to O(x^1^0)',...
'sin(x)/x',...
'Location', 'South')

title('Taylor Series Expansion')
```

```
hold off
```



Compute the Taylor series expansion of this expression. By default, `taylor` uses an absolute order, which is the truncation order of the computed series.

```
taylor(1/(exp(x)) - exp(x) + 2*x, x, 'Order', 5)

ans =
-x^3/3
```

To compute the Taylor series expansion with a relative truncation order, use `OrderMode`. For some expressions, a relative truncation order provides more accurate approximations.

```
taylor(1/(exp(x)) - exp(x) + 2*x, x, 'Order', 5,
'OrderMode', 'Relative')

ans =
- x^7/2520 - x^5/60 - x^3/3
```

Compute the Maclaurin series expansion of this multivariate function. If you do not specify the vector of variables, `taylor` treats `f` as a function of one independent variable.

```
syms x y z
f = sin(x) + cos(y) + exp(z);
taylor(f)

ans =
x^5/120 - x^3/6 + x + cos(y) + exp(z)
```

Compute the multivariate Maclaurin expansion by specifying the vector of variables:

```
syms x y z
f = sin(x) + cos(y) + exp(z);
taylor(f, [x, y, z])

ans =
x^5/120 - x^3/6 + x + y^4/24 - y^2/2 + z^5/120 +
z^4/24 + z^3/6 + z^2/2 + z + 2
```

Compute the multivariate Taylor expansion by specifying both the vector of variables and the vector of values defining the expansion point:

```
syms x y
```

```
f = y*exp(x - 1) - x*log(y);
taylor(f, [x, y], [1, 1], 'Order', 3)

ans =
x + (x - 1)^2/2 + (y - 1)^2/2
```

If you specify the expansion point as a scalar `a`, `taylor` transforms that scalar into a vector of the same length as the vector of variables. All elements of the expansion vector equal `a`:

```
taylor(f, [x, y], 1, 'Order', 3)

ans =
x + (x - 1)^2/2 + (y - 1)^2/2
```

See Also

`symvar` | `taylortool`

How To

- “Taylor Series” on page 2-22

Purpose

Taylor series calculator

Syntax

```
taylortool
taylortool('f')
```

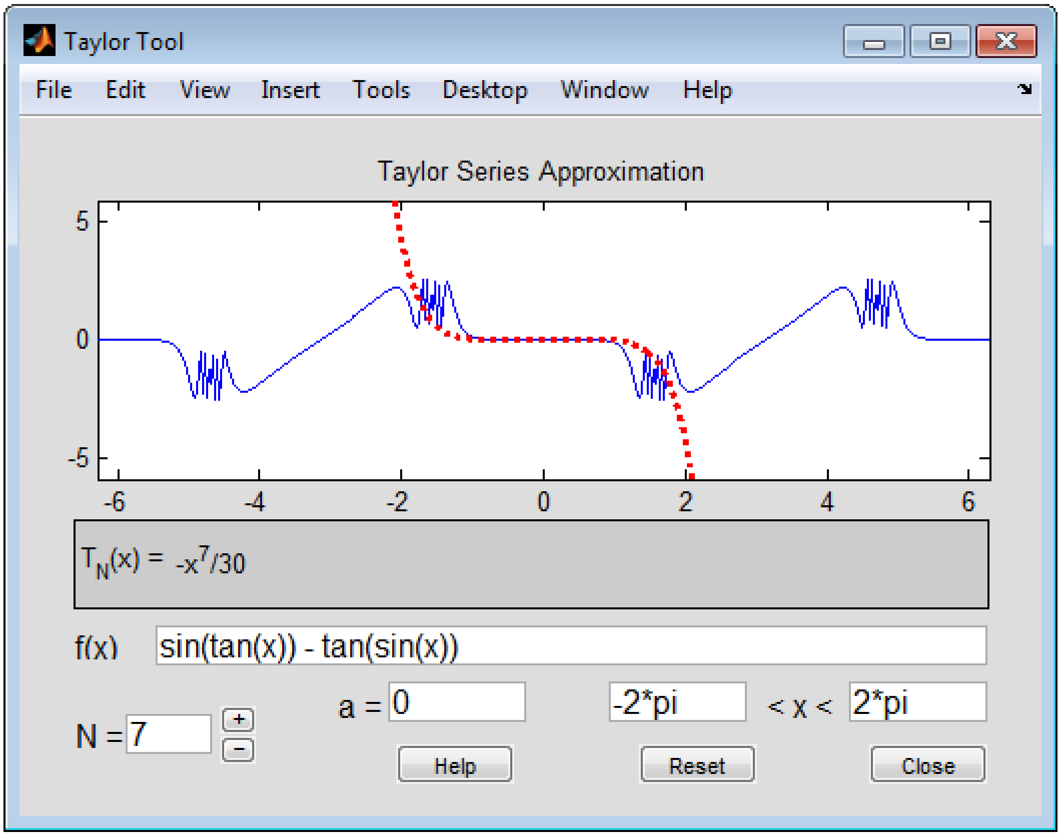
Description

`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
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`

How To

- "Taylor Series" on page 2-22

Purpose

Symbolic Toeplitz matrix

Syntax

```
toeplitz(c,r)
toeplitz(r)
```

Description

`toeplitz(c,r)` generates a nonsymmetric Toeplitz matrix having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, `toeplitz` issues a warning and uses the first element of the column.

`toeplitz(r)` generates a symmetric Toeplitz matrix if `r` is real. If `r` is complex, but its first element is real, then this syntax generates the Hermitian Toeplitz matrix formed from `r`. If the first element of `r` is not real, then the resulting matrix is Hermitian off the main diagonal, meaning that T_{ij} = conjugate(T_{ji}) for $i \neq j$.

Tips

- Calling `toeplitz` for numeric arguments that are not symbolic objects invokes the MATLAB `toeplitz` function.

Input Arguments

c

Vector specifying the first column of a Toeplitz matrix.

r

Vector specifying the first row of a Toeplitz matrix.

Definitions

Toeplitz Matrix

A Toeplitz matrix is a matrix that has constant values along each descending diagonal from left to right. For example, matrix T is a symmetric Toeplitz matrix:

$$T = \begin{pmatrix} t_0 & t_1 & t_2 & & & & t_k \\ t_{-1} & t_0 & t_1 & \cdots & & & \\ t_{-2} & t_{-1} & t_0 & & & & \\ & \vdots & & \ddots & & \vdots & \\ & & & & t_0 & t_1 & t_2 \\ & & & \cdots & t_{-1} & t_0 & t_1 \\ t_{-k} & & & & t_{-2} & t_{-1} & t_0 \end{pmatrix}$$

Examples

Generate the Toeplitz matrix from these vectors. Because these vectors are not symbolic objects, you get floating-point results.

```
c = [1 2 3 4 5 6];
r = [1 3/2 3 7/2 5];
toeplitz(c,r)

ans =
    1.0000    1.5000    3.0000    3.5000    5.0000
    2.0000    1.0000    1.5000    3.0000    3.5000
    3.0000    2.0000    1.0000    1.5000    3.0000
    4.0000    3.0000    2.0000    1.0000    1.5000
    5.0000    4.0000    3.0000    2.0000    1.0000
    6.0000    5.0000    4.0000    3.0000    2.0000
```

Now, convert these vectors to a symbolic object, and generate the Toeplitz matrix:

```
c = sym([1 2 3 4 5 6]);
r = sym([1 3/2 3 7/2 5]);
toeplitz(c,r)

ans =
[ 1, 3/2,   3, 7/2,   5]
[ 2,   1, 3/2,   3, 7/2]
[ 3,   2,   1, 3/2,   3]
[ 4,   3,   2,   1, 3/2]
[ 5,   4,   3,   2,   1]
[ 6,   5,   4,   3,   2]
```

Generate the Toeplitz matrix from this vector:

```
syms a b c d
T = toeplitz([a b c d])

T =
[       a,       b,       c,       d]
[ conj(b),       a,       b,       c]
[ conj(c), conj(b),       a,       b]
[ conj(d), conj(c), conj(b),       a]
```

If you specify that all elements are real, then the resulting Toeplitz matrix is symmetric:

```
syms a b c d real;
T = toeplitz([a b c d])

T =
[ a, b, c, d]
[ b, a, b, c]
[ c, b, a, b]
[ d, c, b, a]
```

For further computations, clear the assumptions:

```
syms a b c d clear
```

Generate the Toeplitz matrix from a vector containing complex numbers:

```
T = toeplitz(sym([1, 2, i]))

T =
[  1, 2, i]
[  2, 1, 2]
[ -i, 2, 1]
```

If the first element of the vector is real, then the resulting Toeplitz matrix is Hermitian:

```
logical(T == T')

ans =
     1     1     1
     1     1     1
     1     1     1
```

If the first element is not real, then the resulting Toeplitz matrix is Hermitian off the main diagonal:

```
T = toeplitz(sym([i, 2, 1]))

T =
[ i, 2, 1]
[ 2, i, 2]
[ 1, 2, i]

logical(T == T')

ans =
     0     1     1
     1     0     1
     1     1     0
```

Generate a Toeplitz matrix using these vectors to specify the first column and the first row. Because the first elements of these vectors are different, `toeplitz` issues a warning and uses the first element of the column:

```
syms a b c
toeplitz([a b c], [1 b/2 a/2])
```

```
Warning: First element of input column does not match
first element of input row.
Column wins diagonal conflict. [linalg::toeplitz]

ans =
[ a, b/2, a/2]
[ b,   a, b/2]
[ c,   b,   a]
```

See Also

`linalg::toeplitztoeplitz`

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

Triangular pulse function

Syntax

```
triangularPulse(a,b,c,x)
triangularPulse(a,c,x)
triangularPulse(x)
```

Description

`triangularPulse(a,b,c,x)` returns the triangular pulse function.

`triangularPulse(a,c,x)` is a shortcut for `triangularPulse(a, (a + c)/2, c, x)`.

`triangularPulse(x)` is a shortcut for `triangularPulse(-1, 0, 1, x)`.

Tips

- If `a`, `b`, and `c` are variables or expressions with variables, `triangularPulse` assumes that `a <= b <= c`. If `a`, `b`, and `c` are numerical values that do not satisfy this condition, `triangularPulse` throws an error.
- If `a = b = c`, `triangularPulse` returns 0.
- If `a = b` or `b = c`, the triangular function can be expressed in terms of the rectangular function.

Input Arguments

a

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression. This argument specifies the rising edge of the triangular pulse function.

Default: `-1`

b

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression. This argument specifies the peak of the triangular pulse function.

Default: If you specify `a` and `c`, then `(a + c)/2`. Otherwise, `0`.

c

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression. This argument specifies the falling edge of the triangular pulse function.

Default: `1`

x

Number (including infinities and symbolic numbers), symbolic variable, or symbolic expression.

Definitions

Triangular Pulse Function

If `a < x < b`, then the triangular pulse function equals `(x - a)/(b - a)`.

If `b < x < c`, then the triangular pulse function equals `(c - x)/(c - b)`.

If `x <= a` or `x >= c`, then the triangular pulse function equals 0.

The triangular pulse function is also called the triangle function, hat function, tent function, or sawtooth function.

Examples

Compute the triangular pulse function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[triangularPulse(-2, 0, 2, -3)
triangularPulse(-2, 0, 2, -1/2)
triangularPulse(-2, 0, 2, 0)
triangularPulse(-2, 0, 2, 3/2)
triangularPulse(-2, 0, 2, 3)]
```

```
ans =
         0
    0.7500
    1.0000
    0.2500
```

```
        0
```

Compute the triangular pulse function for the numbers converted to symbolic objects:

```
[triangularPulse(sym(-2), 0, 2, -3)
triangularPulse(-2, 0, 2, sym(-1/2))
triangularPulse(-2, sym(0), 2, 0)
triangularPulse(-2, 0, 2, sym(3/2))
triangularPulse(-2, 0, sym(2), 3)]

ans =
   0
 3/4
   1
 1/4
   0
```

Compute the triangular pulse function for `a < x < b`:

```
syms a b c x
assume(a < x < b)
triangularPulse(a, b, c, x)

ans =
(a - x)/(a - b)
```

For further computations, remove the assumption:

```
syms a b x clear
```

Compute the triangular pulse function for `b < x < c`:

```
assume(b < x < c)
triangularPulse(a, b, c, x)
```

```
ans =
-(c - x)/(b - c)
```

For further computations, remove the assumption:

```
syms b c x clear
```

Compute the triangular pulse function for `a = b`:

```
syms a b c x
assume(b < c)
triangularPulse(b, b, c, x)

ans =
-((c - x)*rectangularPulse(b, c, x))/(b - c)
```

Compute the triangular pulse function for `c = b`:

```
assume(a < b)
triangularPulse(a, b, b, x)

ans =
((a - x)*rectangularPulse(a, b, x))/(a - b)
```

For further computations, remove all assumptions on `a`, `b`, and `c`:

```
syms a b c clear
```

Use `triangularPulse` with one input argument as a shortcut for computing `triangularPulse(-1, 0, 1, x)`:

```
syms x
triangularPulse(x)

ans =
triangularPulse(-1, 0, 1, x)
```

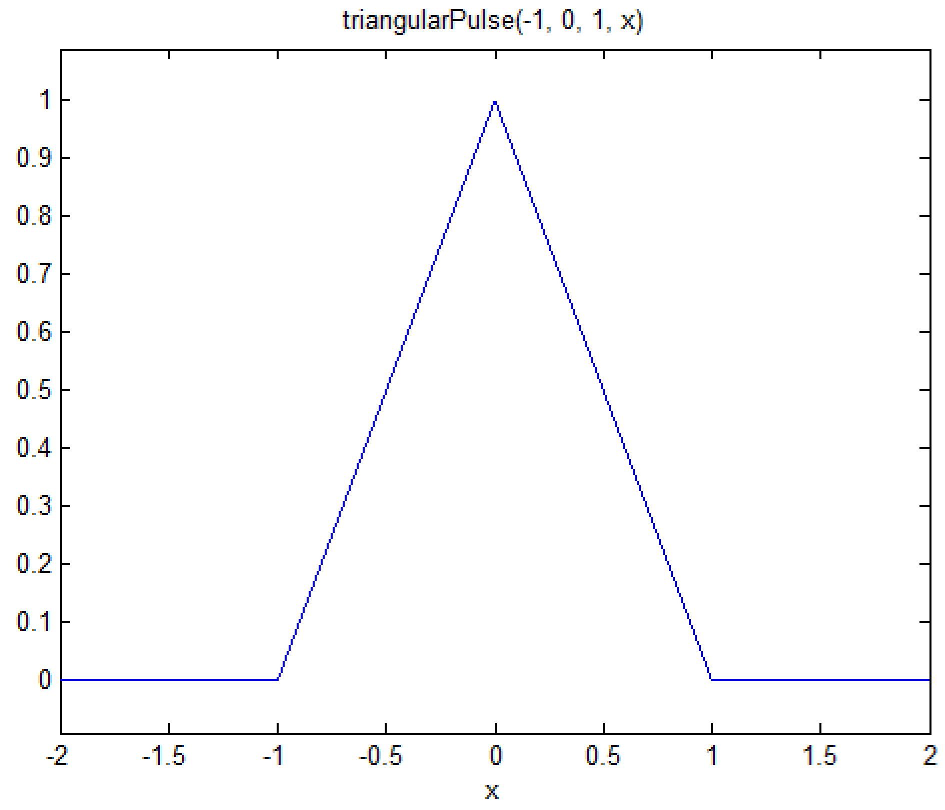
```
[triangularPulse(sym(-10))
triangularPulse(sym(-3/4))
triangularPulse(sym(0))
triangularPulse(sym(2/3))
triangularPulse(sym(1))]

ans =
   0
 1/4
   1
 1/3
   0
```

Use `triangularPulse` with three input arguments as a shortcut for computing `triangularPulse(a, (a + c)/2, c, x)`:

```
syms a c x
triangularPulse(a, c, x)

ans =
triangularPulse(a, a/2 + c/2, c, x)

[triangularPulse(sym(-10), 10, 3)
triangularPulse(sym(-1/2), -1/4, -2/3)
triangularPulse(sym(2), 4, 3)
triangularPulse(sym(2), 4, 6)
triangularPulse(sym(-1), 4, 0)]

ans =
 7/10
    0
    1
    0
  2/5
```

Plot the triangular pulse function:

```
syms x
ezplot(triangularPulse(x), [-2, 2])
```



Call `triangularPulse` with infinities as its rising and falling edges:

```
syms x
triangularPulse(-1, 0, inf, x)
triangularPulse(-inf, 0, 1, x)
triangularPulse(-inf, 0, inf, x)

ans =
heaviside(x) + (x + 1)*rectangularPulse(-1, 0, x)

ans =
heaviside(-x) - (x - 1)*rectangularPulse(0, 1, x)

ans =
1
```

See Also

dirac | heaviside | rectangularPulse

tril

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

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

Vector potential of vector field

Syntax

```
vectorPotential(V,X)
vectorPotential(V)
```

Description

`vectorPotential(V,X)` computes the vector potential of the vector field `V` with respect to the vector `X` in Cartesian coordinates. The vector field `V` and the vector `X` are both three-dimensional.

`vectorPotential(V)` returns the vector potential `V` with respect to a vector constructed from the first three symbolic variables found in `V` by `symvar`.

Tips

- The vector potential exists if and only if the divergence of a vector field `V` with respect to `X` equals 0. If `vectorPotential` cannot verify that `V` has a vector potential, it returns the vector with all three components equal to `NaN`.

Input Arguments

V

Three-dimensional vector of symbolic expressions or functions.

X

Three-dimensional vector with respect to which you compute the vector potential.

Definitions

Vector Potential of a Vector Field

The vector potential of a vector field `V` is a vector field `A`, such that:

$$V = \nabla \times A = curl(A)$$

Examples

Compute the vector potential of this row vector field with respect to the vector `[x, y, z]`:

```
syms x y z
vectorPotential([x^2*y, -1/2*y^2*x, -x*y*z], [x y z])
```

```
ans =
 -(x*y^2*z)/2
     -x^2*y*z
            0
```

Compute the vector potential of this column vector field with respect to the vector `[x, y, z]`:

```
syms x y z
f(x,y,z) = 2*y^3 - 4*x*y;
g(x,y,z) = 2*y^2 - 16*z^2+18;
h(x,y,z) = -32*x^2 - 16*x*y^2;
A = vectorPotential([f; g; h], [x y z])

A(x, y, z) =
 (2*z*(3*y^2 - 8*z^2 + 27))/3 + (16*x*y*(y^2 + 6*x))/3
                                    2*y*z*(- y^2 + 2*x)
                                                      0
```

To check whether the vector potential exists for a particular vector field, compute the divergence of that vector field:

```
syms x y z
V = [x^2 2*y z];
divergence(V, [x y z])

ans =
2*x + 3
```

If the divergence is not equal to 0, the vector potential does not exist. In this case, `vectorPotential` returns the vector with all three components equal to `NaN`:

```
vectorPotential(V, [x y z])

ans =
 NaN
 NaN
 NaN
```

See Also

`curl` | `diff` | `divergence` | `gradient` | `jacobian` | `hessian` | `laplacian` | `potential`

vpa

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`.

`R = vpa(A,d)` uses at least `d` significant (nonzero) digits, instead of the current setting of `digits`.

Tips

- The toolbox increases the internal precision of calculations by several digits (guard digits).
- When you apply `vpa` to a numeric expression, such as `1/3`, `2^(-5)`, or `sin(pi/4)`, it is evaluated to a double-precision number. Then, `vpa` is applied to that double-precision number. For more accurate results, convert numeric expressions to symbolic expressions. For example, to approximate `exp(1)` use `vpa(sym(exp(1)))`.
- If the value `d` is not an integer, `vpa` rounds it to the nearest integer.

Input Arguments

A

Symbolic object, string, or numeric expression.

d

Integer greater than 1 and smaller than $2^{29}+1$.

Output Arguments

R

Symbolic object representing a floating-point number

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, 0.333333333333333333333333333]

B =
[ 1.0,     0.5]
[ 0.5, 0.33333]
```

The vpa function lets you specify a 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`:

```
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 =
0.0000000000000000000086736173798840354720600815844403
```

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
```

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`

How To

- "Variable-Precision Arithmetic" on page 2-50

vpasolve

Purpose

Numeric solver

Syntax

```
S = vpasolve(eqn,var,init_guess)
Y = vpasolve(eqns,vars,init_guess)
[y1,...,yN] = vpasolve(eqns,vars,init_guess)
```

Description

`S = vpasolve(eqn,var,init_guess)` numerically solves the equation `eqn` for the variable `var` using the starting value or the search range `init_guess`.

`Y = vpasolve(eqns,vars,init_guess)` numerically solves the system of equations `eqns` for the variables `vars` using the starting values or the search range `init_guess`. This syntax returns a structure array that contains the solutions. The number of fields in the structure array corresponds to the number of independent variables in a system.

`[y1,...,yN] = vpasolve(eqns,vars,init_guess)` numerically solves the system of equations `eqns` for the variables `vars` using the starting values or the search range `init_guess`. This syntax assigns the solutions to the variables `y1,...,yN`.

Tips

- `vpasolve` returns all solutions only for polynomial equations. For non-polynomial equations, there is no general method of finding all solutions numerically. When you solve a non-polynomial equation or system numerically, and the solutions exist, `vpasolve` returns only one solution.
- When you solve a system of rational equations, the toolbox transforms it to a polynomial system by multiplying out the denominators. `vpasolve` returns all solutions of the resulting polynomial system, including those that are also roots of these denominators.
- `vpasolve` ignores assumptions set on variables. You can restrict the returned results to particular ranges by specifying appropriate search ranges using the argument `init_guess`.
- If `init_guess` specifies a search range `[a, b]`, and the values `a, b` are complex numbers, then `vpasolve` searches for the solutions in the rectangular search area in the complex plane. Here `a` specifies

the bottom-left corner of the rectangular search area, and `b` specifies the top-right corner of that area.

- If `vars` is a vector, then `init_guess` can be a scalar, a vector of the same length as `vars`, or a matrix with two columns and the number of rows equal to the number of `vars`. If `vars` is a vector and `init_guess` is a scalar, then `init_guess` is expanded into a vector of the same length as `vars` with all elements equal to `init_guess`.
- The output variables `y1,...,yN` do not specify the variables for which `vpasolve` solves equations or systems. If `y1,...,yN` are the variables that appear in `eqns`, that does not guarantee that `vpasolve(eqns)` will assign the solutions to `y1,...,yN` using the correct order. Thus, for the call `[a,b] = vpasolve(eqns)`, you might get the solutions for `a` assigned to `b` and vice versa.

  To ensure the order of the returned solutions, specify the variables `vars`. For example, the call `[b,a] = vpasolve(eqns,b,a)` assigns the solutions for `a` assigned to `a` and the solutions for `b` assigned to `b`.

Input Arguments

eqn

Symbolic equation defined by the relation operator == or symbolic expression. If `eqn` is a symbolic expression (without the right side), the solver assumes that the right side is 0, and solves the equation `eqn == 0`.

var

Variable for which you solve an equation.

Default: Variable determined by `symvar`

eqns

Symbolic equations or expressions that need to be solved as a system. These equations or expressions can be separated by commas or can be presented as a vector. If an equation is a symbolic expression (without the right side), the solver assumes that the right side of that equation is 0.

vars

Variables for which you solve an equation or a system of equations. These variables can be separated by commas or can be presented as a vector.

> **Default:** Variables determined by `symvar`

init_guess

Number, vector, or matrix with two columns that specifies the initial guess for the solution.

If `init_guess` is a number or, in case of multivariate equations, a vector of numbers, then the numeric solver uses it as a starting point.

If `init_guess` is a matrix with two columns, then the numeric solver uses `init_guess` as a search range. Also, if `init_guess` is a vector with two elements, and `eqn` is univariate, then the numeric solver uses `init_guess` as a search range.

> **Default:** `vpasolve` uses its own internal choices for starting points and search ranges.

Output Arguments

S

Symbolic array that contains solutions of an equation when you solve one equation. The size of a symbolic array corresponds to the number of the solutions.

Y

Structure array that contains solutions of a system when you solve a system of equations. The number of fields in the structure array corresponds to the number of independent variables in a system.

y1,...,yN

Variables to which the solver assigns the solutions of a system of equations. The number of output variables or symbolic arrays must equal the number of independent variables in a system. If you explicitly specify independent variables `vars`, then the solver uses the same order to return the solutions. If you do not specify `vars`, the toolbox sorts independent variables alphabetically, and then assigns the solutions for these variables to the output variables or symbolic arrays.

Examples

Solve this polynomial equation numerically. For polynomial equations, `vpasolve` returns all solutions.

```
syms x
vpasolve(4*x^4 + 3*x^3 + 2*x^2 + x + 5 == 0, x)

ans =
 - 0.88011377126068169817875190457835 - 0.76331583387715452512978468102263*i
   0.50511377126068169817875190457835 + 0.81598965068946312853227067890656*i
   0.50511377126068169817875190457835 - 0.81598965068946312853227067890656*i
 - 0.88011377126068169817875190457835 + 0.76331583387715452512978468102263*i
```

Solve this equation numerically. For non-polynomial equations, `vpasolve` returns the first solution that it finds.

```
syms x
vpasolve(sin(x^2) == 1/2, x)

ans =
-226.94447241941511682716953887638
```

When solving a system of equations, use one output argument to return the solutions in the form of a structure array:

```
syms x y
S = vpasolve([x^3 + 2*x == y, y^2 == x], [x, y])

S =
    x: [6x1 sym]
    y: [6x1 sym]
```

To display the solutions, access the elements of the structure array S:

```
S.x
```

```
ans =
                                                                          0
                                         0.2365742942773341617614871521768
 - 0.2812406533871196866619789549945 3 + 1.2348724236470142074859894531946*i
   0.1629535062484526057812353789061 3 + 1.6151544650555366917886585417926*i
   0.1629535062484526057812353789061 3 - 1.6151544650555366917886585417926*i
 - 0.2812406533871196866619789549945 3 - 1.2348724236470142074859894531946*i
```

```
S.y
```

```
ans =
                                                                            0
                                           0.48638903593454300001655725369801
   0.70187356885586188630668751791218 + 0.87969719792982402287026727381769*i
 - 0.94506808682313338631496614476119 - 0.85451751443904587692179191887616*i
 - 0.94506808682313338631496614476119 + 0.85451751443904587692179191887616*i
   0.70187356885586188630668751791218 - 0.87969719792982402287026727381769*i
```

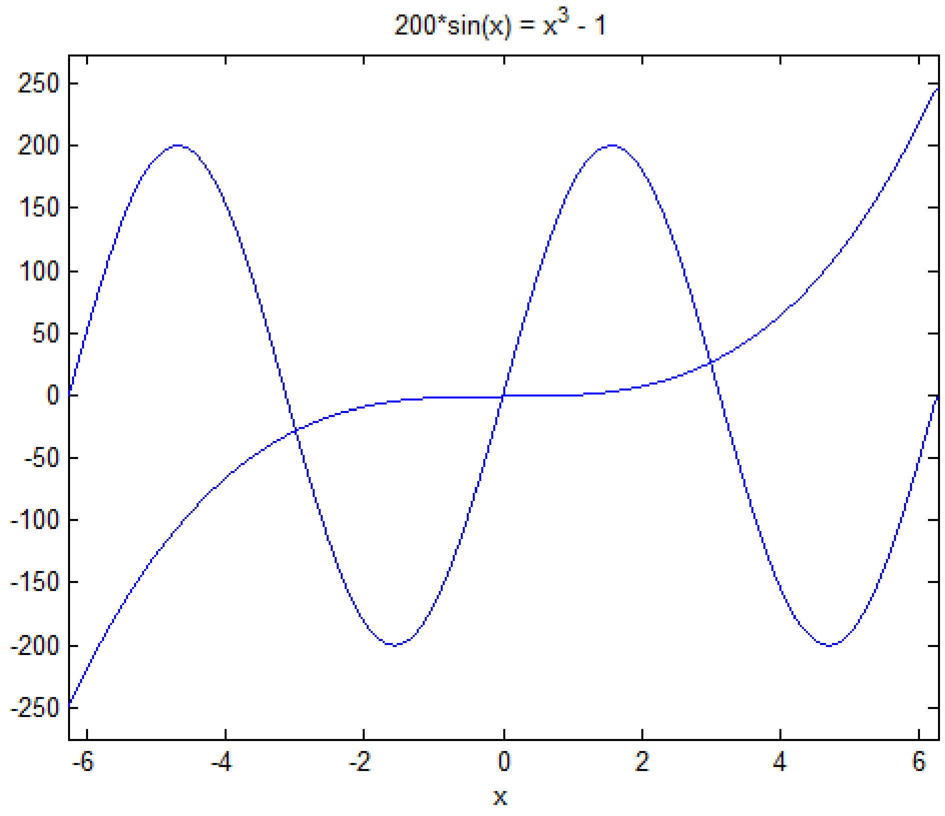
When solving a system of equations, use multiple output arguments to assign the solutions directly to output variables. To ensure the correct order of the returned solutions, specify the variables explicitly. The order in which you specify the variables defines the order in which the solver returns the solutions.

```
syms x y
[sol_x, sol_y] = vpasolve([x*sin(10*x) == y^3, y^2
== exp(-2*x/3)], [x, y])

sol_x =
88.90707209659114864849280774681
```

```
sol_y =
0.00000000000013470479710676694388973703681918
```

Plot the left and right sides of the equation `200*sin(x) = x^3 - 1`:

```
syms x
ezplot(200*sin(x))
hold on
ezplot(x^3 - 1)
title('200*sin(x) = x^3 - 1')
```



This equation has three solutions. If you do not specify the initial guess (zero-approximation), `vpasolve` returns the first solution that it finds:

```
vpasolve(200*sin(x) == x^3 - 1, x)

ans =
-0.0050000214585835715725440675982988
```

To find one of the other solutions, specify the initial point that is close to that solution:

```
vpasolve(200*sin(x) == x^3 - 1, x, -4)
```

```
ans =
-3.0009954677086430679926572924945

vpasolve(200*sin(x) == x^3 - 1, x, 3)

ans =
3.0098746383859522384063444361906
```

Solve this polynomial equation. The numeric solver returns all eight solutions of this equation:

```
syms x
vpasolve(x^8 - x^2 == 3, x)
```

```
ans =
                                             -1.2052497163799060695888397264341
                                              1.2052497163799060695888397264341
 - 0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
 - 0.77061431370803029127495426747428 - 0.85915207603993818859321142757164*i
                                            1.0789046020338265308047436284205*i
                                           -1.0789046020338265308047436284205*i
   0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
   0.77061431370803029127495426747428 - 0.85915207603993818859321142757164*i
```

Suppose you need only real solutions of this equation. You cannot use assumptions on variables because `vpasolve` ignores them:

```
assume(x, 'real')
vpasolve(x^8 - x^2 == 3, x)
```

```
ans =
                                             -1.2052497163799060695888397264341
                                              1.2052497163799060695888397264341
 - 0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
 - 0.77061431370803029127495426747428 - 0.85915207603993818859321142757164*i
                                            1.0789046020338265308047436284205*i
                                           -1.0789046020338265308047436284205*i
```

```
  0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
  0.77061431370803029127495426747428 - 0.85915207603993818859321142757164*i
```

Specify the search range to restrict the returned results to particular ranges. For example, to return only real solutions of this equation, specify the search interval as `[-inf inf]`:

```
vpasolve(x^8 - x^2 == 3, x, [-inf inf])

ans =
 -1.2052497163799060695888397264341
  1.2052497163799060695888397264341
```

Now return only nonnegative solutions:

```
vpasolve(x^8 - x^2 == 3, x, [0 inf])

ans =
1.2052497163799060695888397264341
```

The search range can contain complex numbers. In this case, `vpasolve` uses a rectangular search area in the complex plane:

```
vpasolve(x^8 - x^2 == 3, x, [-1, 1 + i])

ans =
 - 0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
   0.77061431370803029127495426747428 + 0.85915207603993818859321142757164*i
```

Alternatives

If possible, solve equations symbolically using `solve`, and then approximate the obtained symbolic results numerically using `vpa`. Using this approach, you get numeric approximations of all solutions found by the symbolic solver. Using the symbolic solver and postprocessing its results requires more time than using the numeric methods directly. This can significantly decrease performance.

See Also

`dsolve` | `equationsToMatrix` | `fzero` | `linsolve` | `solve` | `symvar` | `vpa`

Purpose

Whittaker M function

Syntax

```
whittakerM(a,b,z)
whittakerM(a,b,A)
```

Description

`whittakerM(a,b,z)` returns the value of the Whittaker M function.

`whittakerM(a,b,A)` returns the value of the Whittaker M function for each element of `A`.

Input Arguments

a

Symbolic number, variable, or expression.

b

Symbolic number, variable, or expression.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Whittaker M Function

The Whittaker functions $M_{a,b}$(z) and $W_{a,b}$(z) are linearly independent solutions of this differential equation:

$$\frac{d^2w}{dz^2}+\left(-\frac{1}{4}+\frac{a}{z}+\frac{1/4-b^2}{z^2}\right)w=0$$

The Whittaker M function is defined via the confluent hypergeometric functions:

$$M_{a,b}(z) = e^{-z/2} z^{b+1/2} M\left(b - a + \frac{1}{2}, 1 + 2b, z\right)$$

Examples

Solve this second-order differential equation. The solutions are given in terms of the Whittaker functions.

```
syms a b w(z)
dsolve(diff(w, 2) + (-1/4 + a/z + (1/4 - b^2)/z^2)*w == 0)

ans =
C2*whittakerM(-a,-b,-z) + C3*whittakerW(-a,-b,-z)
```

Verify that the Whittaker M function is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerM(a,b,z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(a,b,z)) == 0

ans =
     1
```

Verify that `whittakerM(-a,-b,-z)` also is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerM(-a,-b,-z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(-a,-b,-z)) == 0

ans =
     1
```

Compute the Whittaker M function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[whittakerM(1, 1, 1), whittakerM(-2, 1, 3/2 + 2*i),...
whittakerM(2, 2, 2), whittakerM(3, -0.3, 1/101)]

ans =
   0.7303            -9.2744 + 5.4705i
2.6328             0.3681
```

Compute the Whittaker M function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `whittakerM` returns unresolved symbolic calls.

```
[whittakerM(sym(1), 1, 1), whittakerM(-2,
sym(1), 3/2 + 2*i),...
whittakerM(2, 2, sym(2)), whittakerM(sym(3), -0.3, 1/101)]

ans =
[ whittakerM(1, 1, 1), whittakerM(-2, 1, 3/2 + 2*i),
whittakerM(2, 2, 2), whittakerM(3, -3/10, 1/101)]
```

For symbolic variables and expressions, `whittakerM` also returns unresolved symbolic calls:

```
syms a b x y
[whittakerM(a, b, x), whittakerM(1, x, x^2),...
whittakerM(2, x, y), whittakerM(3, x + y, x*y)]

ans =
[ whittakerM(a, b, x), whittakerM(1, x, x^2),...
whittakerM(2, x, y), whittakerM(3, x + y, x*y)]
```

The Whittaker M function has special values for some parameters:

```
whittakerM(sym(-3/2), 1, 1)

ans =
exp(1/2)
```

```
syms a b x
whittakerM(0, b, x)

ans =
4^b*x^(1/2)*gamma(b + 1)*besseli(b, x/2)

whittakerM(a + 1/2, a, x)

ans =
x^(a + 1/2)*exp(-x/2)

whittakerM(a, a - 5/2, x)

ans =
(2*x^(a - 2)*exp(-x/2)*(2*a^2 - 7*a + x^2/2 -...
x*(2*a - 3) + 6))/pochhammer(2*a - 4, 2)
```

Differentiate the expression involving the Whittaker M function:

```
syms a b z
diff(whittakerM(a,b,z), z)

ans =
(whittakerM(a + 1, b, z)*(a + b + 1/2))/z -...
(a/z - 1/2)*whittakerM(a, b, z)
```

Compute the Whittaker M function for the elements of matrix A:

```
syms x
A= [-1, x^2; 0, x];
whittakerM(-1/2, 0, A)

ans =
[ exp(-1/2)*i, exp(x^2/2)*(x^2)^(1/2)]
[           0,       x^(1/2)*exp(x/2)]
```

References

Slater, L. J. “Cofluent Hypergeometric Functions.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`hypergeom` | `whittakerW`

How To

- “Special Functions of Applied Mathematics” on page 2-142

whittakerW

Purpose

Whittaker W function

Syntax

```
whittakerW(a,b,z)
whittakerW(a,b,A)
```

Description

`whittakerW(a,b,z)` returns the value of the Whittaker W function.

`whittakerW(a,b,A)` returns the value of the Whittaker W function for each element of `A`.

Input Arguments

a

Symbolic number, variable, or expression.

b

Symbolic number, variable, or expression.

z

Symbolic number, variable, or expression.

A

Vector or matrix of symbolic numbers, variables, or expressions.

Definitions

Whittaker W Function

The Whittaker functions $M_{a,b}(z)$ and $W_{a,b}(z)$ are linearly independent solutions of this differential equation:

$$\frac{d^2w}{dz^2}+\left(-\frac{1}{4}+\frac{a}{z}+\frac{1/4-b^2}{z^2}\right)w=0$$

The Whittaker W function is defined via the confluent hypergeometric functions:

$$W_{a,b}(z) = e^{-z/2} z^{b+1/2} U\left(b - a + \frac{1}{2}, 1 + 2b, z\right)$$

Examples

Solve this second-order differential equation. The solutions are given in terms of the Whittaker functions.

```
syms a b w(z)
dsolve(diff(w, 2) + (-1/4 + a/z + (1/4 - b^2)/z^2)*w == 0)

ans =
C2*whittakerM(-a, -b, -z) + C3*whittakerW(-a, -b, -z)
```

Verify that the Whittaker W function is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerW(a, b, z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(a, b, z)) == 0

ans =
     1
```

Verify that `whittakerW(-a, -b, -z)` also is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerW(-a, -b, -z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(-a, -b, -z)) == 0

ans =
     1
```

Compute the Whittaker W function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[whittakerW(1, 1, 1), whittakerW(-2, 1, 3/2 + 2*i),...
whittakerW(2, 2, 2), whittakerW(3, -0.3, 1/101)]

ans =
   1.1953            -0.0156 - 0.0225i
4.8616           -0.1692
```

Compute the Whittaker W function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `whittakerW` returns unresolved symbolic calls.

```
[whittakerW(sym(1), 1, 1), whittakerW(-2,
sym(1), 3/2 + 2*i),...
whittakerW(2, 2, sym(2)), whittakerW(sym(3), -0.3, 1/101)]

ans =
[ whittakerW(1, 1, 1), whittakerW(-2, 1, 3/2 + 2*i),
whittakerW(2, 2, 2), whittakerW(3, -3/10, 1/101)]
```

For symbolic variables and expressions, `whittakerW` also returns unresolved symbolic calls:

```
syms a b x y
[whittakerW(a, b, x), whittakerW(1, x, x^2),...
whittakerW(2, x, y), whittakerW(3, x + y, x*y)]

ans =
[ whittakerW(a, b, x), whittakerW(1, x, x^2),
whittakerW(2, x, y), whittakerW(3, x + y, x*y)]
```

The Whittaker W function has special values for some parameters:

```
whittakerW(sym(-3/2), 1/2, 0)

ans =
4/(3*pi^(1/2))
```

```
syms a b x
whittakerW(0, b, x)

ans =
(x^(b + 1/2)*besselk(b, x/2))/(pi^(1/2)*x^b)

whittakerW(a, -a + 1/2, x)

ans =
x^(1 - a)*x^(2*a - 1)*exp(-x/2)

whittakerW(a - 1/2, a, x)

ans =
(x^(a + 1/2)*exp(-x/2)*exp(x)*igamma(2*a, x))/x^(2*a)
```

Differentiate the expression involving the Whittaker W function:

```
syms a b z
diff(whittakerW(a,b,z), z)

ans =
- (a/z - 1/2)*whittakerW(a, b, z) -...
whittakerW(a + 1, b, z)/z
```

Compute the Whittaker W function for the elements of matrix A:

```
syms x
A= [-1, x^2; 0, x];
whittakerW(-1/2, 0, A)

ans =
[ -exp(-1/2)*(pi*i + ei(1))*i,
exp(x^2)*exp(-x^2/2)*expint(x^2)*(x^2)^(1/2)]
[  0,
x^(1/2)*exp(-x/2)*exp(x)*expint(x)]
```

References

Slater, L. J. “Cofluent Hypergeometric Functions.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`hypergeom` | `whittakerM`

How To

- “Special Functions of Applied Mathematics” on page 2-142

Purpose

Wright omega function

Syntax

```
wrightOmega(x)
wrightOmega(A)
```

Description

`wrightOmega(x)` computes the Wright omega function of `x`.

`wrightOmega(A)` computes the Wright omega function of each element of `A`.

Input Arguments

x

Number, symbolic variable, or symbolic expression.

A

Vector or matrix of numbers, symbolic variables, or symbolic expressions.

Definitions

Wright omega Function

The Wright omega function is defined in terms of the Lambert W function:

$$\omega(x) = W_{\left\lceil \frac{\mathrm{Im}(x)-\pi}{2\pi} \right\rceil}\left(e^{x}\right)$$

The Wright omega function $\omega(x)$ is a solution of the equation $Y + \log(Y) = X$.

Examples

Compute the Wright omega function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
wrightOmega(1/2)

ans =
    0.7662

wrightOmega(pi)
```

```
ans =
    2.3061

wrightOmega(-1+i*pi)

ans =
    -1
```

Compute the Wright omega function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `wrightOmega` returns unresolved symbolic calls:

```
wrightOmega(sym(1/2))

ans =
wrightOmega(1/2)

wrightOmega(sym(pi))

ans =
wrightOmega(pi)
```

For some exact numbers, `wrightOmega` has special values:

```
wrightOmega(-1+i*sym(pi))

ans =
    -1
```

Compute the Wright omega function for `x` and `sin(x) + x*exp(x)`. For symbolic variables and expressions, `wrightOmega` returns unresolved symbolic calls:

```
syms x
wrightOmega(x)
wrightOmega(sin(x) + x*exp(x))
```

```
ans =
wrightOmega(x)

ans =
wrightOmega(sin(x) + x*exp(x))
```

Now compute the derivatives of these expressions:

```
diff(wrightOmega(x), x, 2)
diff(wrightOmega(sin(x) + x*exp(x)), x)

ans =
wrightOmega(x)/(wrightOmega(x) + 1)^2 -...
wrightOmega(x)^2/(wrightOmega(x) + 1)^3

ans =
(wrightOmega(sin(x) + x*exp(x))*(cos(x) +...
exp(x) + x*exp(x)))/(wrightOmega(sin(x) + x*exp(x)) + 1)
```

Compute the Wright omega function for elements of matrix `M` and vector `V`:

```
M =[0 pi; 1/3 -pi];
V = sym([0; -1+i*pi]);
wrightOmega(M)
wrightOmega(V)

ans =
    0.5671    2.3061
    0.6959    0.0415

ans =
 lambertw(0, 1)
             -1
```

References

Corless, R. M. and D. J. Jeffrey. “The Wright omega Function.” *Artificial Intelligence, Automated Reasoning, and Symbolic Computation* (J.

Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, eds.). Berlin: Springer-Verlag, 2002, pp. 76-89.

See Also

`lambertW` | `log`

How To

- "Special Functions of Applied Mathematics" on page 2-142

Purpose

Logical XOR for symbolic expressions

Syntax

```
xor(A,B)
```

Description

`xor(A,B)` represents the logical exclusive disjunction. `xor(A,B)` is true when either `A` or `B` are true. If both `A` and `B` are true or false, `xor(A,B)` is false.

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.
- `assume` and `assumeAlso` do not accept assumptions that contain `xor`.

Input Arguments

A

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

B

Symbolic equation, inequality, or logical expression that contains symbolic subexpressions.

Examples

Combine two symbolic inequalities into the logical expression using `xor`:

```
syms x
range = xor(x > -10, x < 10);
```

Replace variable `x` with these numeric values. If you replace `x` with 11, then inequality `x > -10` is valid and `x < 10` is invalid. If you replace `x` with 0, both inequalities are valid. Note that `subs` does not evaluate these inequalities to logical `1` or `0`.

```
x1 = subs(range, x, 11)
x2 = subs(range, x, 0)
```

```
x1 =
-10 < 11 xor 11 < 10

x2 =
-10 < 0 xor 0 < 10
```

To evaluate these inequalities to logical `1` or `0`, use `logical` or `isAlways`. If only one inequality is valid, the expression with `xor` evaluates to logical `1`. If both inequalities are valid, the expression with `xor` evaluates to logical `0`.

```
logical(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)

s1 =
TRUE

s2 =
FALSE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)

ans =
     1
```

```
ans =
     0
```

See Also

`all | and | any | isAlways | logical | not | or`

Purpose

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)

ans =
    2.6124
```

Compute the Riemann zeta function for the matrix:

```
zeta(1.2:0.1:2.1)

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])
```

```
ans =
[ zeta(x),      pi^2/6]
[ pi^4/90, zeta(x + y)]
```

Differentiate the Riemann zeta function:

```
diff(zeta(x), x, 3)
```

```
ans =
zeta(3, x)
```

Purpose

Z-transform

Syntax

`ztrans(f,trans_index,eval_point)`

Description

`ztrans(f,trans_index,eval_point)` computes the Z-transform of `f` with respect to the transformation index `trans_index` at the point `eval_point`.

Tips

- If you call `ztrans` with two arguments, it assumes that the second argument is the evaluation point `eval_point`.
- If `f` is a matrix, `ztrans` applies the Z-transform to all components of the matrix.
- To compute the inverse Z-transform, use `iztrans`.

Input Arguments

f

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

trans_index

Symbolic variable representing the transformation index. This variable is often called the "discrete time variable".

> **Default:** The variable `n`. If `f` does not contain `n`, then the default variable is determined by `symvar`.

eval_point

Symbolic variable or expression representing the evaluation point. This variable is often called the "complex frequency variable".

> **Default:** The variable `z`. If `z` is the transformation index of `f`, then the default evaluation point is the variable `w`.

Definitions

Z-Transform

The Z-transform of the expression $f = f(n)$ is defined as follows:

$$F(z) = \sum_{n=0}^{\infty} \frac{f(n)}{z^n}.$$

Examples

Compute the Z-transform of this expression with respect to the transformation index `k` at the evaluation point `x`:

```
syms k x
f = sin(k);
ztrans(f, k, x)

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

Compute the Z-transform of this expression calling the `ztrans` function with one argument. If you do not specify the transformation index, `ztrans` uses the variable `n`:

```
syms a n x
f = a^n;
ztrans(f, x)

ans =
-x/(a - x)
```

If you also do not specify the evaluation point, `ztrans` uses the variable `z`:

```
ztrans(f)

ans =
-z/(a - z)
```

Compute the following Z-transforms that involve the Heaviside function and the binomial coefficient:

```
syms n z
ztrans(heaviside(n - 3), n, z)

ans =
(1/(z - 1) + 1/2)/z^3

ztrans(nchoosek(n, 2)*heaviside(5 - n), n, z)

ans =
z/(z - 1)^3 + 5/z^5 + (6*z - z^6/(z - 1)^3
+ 3*z^2 + z^3)/z^5
```

If `ztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(n) z
F = ztrans(f, n, z)

F(z) =
ztrans(f(n), n, z)
```

`iztrans` returns the original expression:

```
iztrans(F, z, n)

ans(n) =
f(n)
```

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `laplace`

Concepts

- "Compute Z-Transforms and Inverse Z-Transforms" on page 2-108

Index