Symbolic Math Toolbox™

User's Guide

MATLAB®

R2017b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

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)
September 2013	Online only	Revised for Version 5.11 (Release 2013b)
March 2014	Online only	Revised for Version 6.0 (Release 2014a)
October 2014	Online only	Revised for Version 6.1 (Release 2014b)
March 2015	Online only	Revised for Version 6.2 (Release 2015a)
September 2015	Online only	Revised for Version 6.3 (Release 2015b)
March 2016	Online only	Revised for Version 7.0 (Release 2016a)
September 2016	Online only	Revised for Version 7.1 (Release 2016b)
March 2017	Online only	Revised for Version 7.2 (Release 2017a)
September 2017	Online only	Revised for Version 8.0 (Release 2017b)

Contents

1 Getting Started

2 Using Symbolic Math Toolbox Software

Getting Started

Symbolic Math Toolbox Product Description

Perform symbolic math computations

Symbolic Math Toolbox provides functions for solving, plotting, and manipulating symbolic math equations. You can create, run, and share symbolic math code using the MATLAB Live Editor. The toolbox provides functions in common mathematical areas such as calculus, linear algebra, algebraic and ordinary differential equations, equation simplification, and equation manipulation.

Symbolic Math Toolbox lets you analytically perform differentiation, integration, simplification, transforms, and equation solving. You can perform dimensional computations and conversions using SI and US unit systems. Your computations can be performed either analytically or using variable-precision arithmetic, with the results displayed in mathematical typeset.

You can share your symbolic work with other MATLAB users as live scripts or convert them to HTML or PDF for publication. You can generate MATLAB functions, Simulink® function blocks, and Simscape™ equations directly from symbolic expressions.

Key Features

- Symbolic integration, differentiation, transforms, and linear algebra
- Algebraic and ordinary differential equation (ODE) solvers
- Simplification and manipulation of symbolic expressions
- Unit systems for specifying, converting, and computing using SI, US, and custom unit systems
- Plotting of analytical functions in 2D and 3D
- Symbolic expression conversion to MATLAB, Simulink, Simscape, C, Fortran, and LaTeX code
- Variable-precision arithmetic

Create Symbolic Numbers, Variables, and Expressions

This page shows how to create symbolic numbers, variables, and expressions. To learn how to work with symbolic math, see “Perform Symbolic Computations” on page 1-12.

Create Symbolic Numbers

You can create symbolic numbers by using `sym`. Symbolic numbers are exact representations, unlike floating-point numbers.

Create a symbolic number by using `sym` and compare it to the same floating-point number.

```
sym(1/3)
1/3

ans =
1/3
ans =
    0.3333
```

The symbolic number is represented in exact rational form, while the floating-point number is a decimal approximation. The symbolic result is not indented, while the standard MATLAB result is indented.

Calculations on symbolic numbers are exact. Demonstrate this exactness by finding `sin(pi)` symbolically and numerically. The symbolic result is exact, while the numeric result is an approximation.

```
sin(sym(pi))
sin(pi)

ans =
0
ans =
   1.2246e-16
```

To learn more about symbolic representation of numbers, see “Numeric to Symbolic Conversion” on page 2-125.

Create Symbolic Variables

You can use two ways to create symbolic variables, `syms` and `sym`. The `syms` syntax is a shorthand for `sym`.

Create symbolic variables `x` and `y` using `syms` and `sym` respectively.

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

The first command creates a symbolic variable `x` in the MATLAB workspace with the value `x` assigned to the variable `x`. The second command creates a symbolic variable `y` with value `y`. Therefore, the commands are equivalent.

With `syms`, you can create multiple variables in one command. Create the variables `a`, `b`, and `c`.

```
syms a b c
```

If you want to create many variables, the `syms` syntax is inconvenient. Instead of using `syms`, use `sym` to create many numbered variables.

Create the variables `a1, ..., a20`.

```
A = sym('a', [1 20])
```

```
A =
[ a1, a2, a3, a4, a5, a6, a7, a8, a9, a10,...
 a11, a12, a13, a14, a15, a16, a17, a18, a19, a20]
```

The `syms` command is a convenient shorthand for the `sym` syntax. Use the `sym` syntax when you create many variables, when the variable value differs from the variable name, or when you create a symbolic number, such as `sym(5)`.

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 = (1 + sqrt(sym(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` = `a`x^2 + `bx` + `c`. First, create the symbolic variables `a`, `b`, `c`, and `x`:

```
syms a b c x
```

Then, assign the expression to `f`:

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

Tip To create a symbolic number, 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.

Reuse Names of Symbolic Objects

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, integer, and positive. These assumptions are stored separately from the symbolic object. For more information, see “Delete Symbolic Objects and Their Assumptions” on page 1-29.

See Also

More About

- “Create Symbolic Functions” on page 1-7
- “Create Symbolic Matrices” on page 1-9
- “Perform Symbolic Computations” on page 1-12
- “Use Assumptions on Symbolic Variables” on page 1-28

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`. 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
```

Note that the body of the function must be a symbolic number, variable, or expression. Assigning a number, such as `f(x,y) = 1`, causes an error.

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
```

See Also

More About

- “Create Symbolic Numbers, Variables, and Expressions” on page 1-3
- “Create Symbolic Matrices” on page 1-9
- “Perform Symbolic Computations” on page 1-12
- “Use Assumptions on Symbolic Variables” on page 1-28

Create Symbolic Matrices

In this section...

Use Existing Symbolic Variables

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 `isAlways` function:

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

The sums are equal:

```
ans =
  logical
   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 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 “Numeric to Symbolic Conversion” on page 2-125.

See Also

More About

- “Create Symbolic Numbers, Variables, and Expressions” on page 1-3
- “Create Symbolic Functions” on page 1-7
- “Perform Symbolic Computations” on page 1-12
- “Use Assumptions on Symbolic Variables” on page 1-28

Perform Symbolic Computations

In this section...

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 2-39.

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 2-4.

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 2-56.

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 2-4.

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, it returns an unresolved integral:

```
syms x
int(sin(sinh(x)))

ans =
int(sin(sinh(x)), x)
```

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 2-4.

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
```

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 = (1 + sqrt(sym(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 “Choose Function to Rearrange Expression” on page 2-94.

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 “Substitute Elements in Symbolic Matrices” on page 2-109.

Plot Symbolic Functions

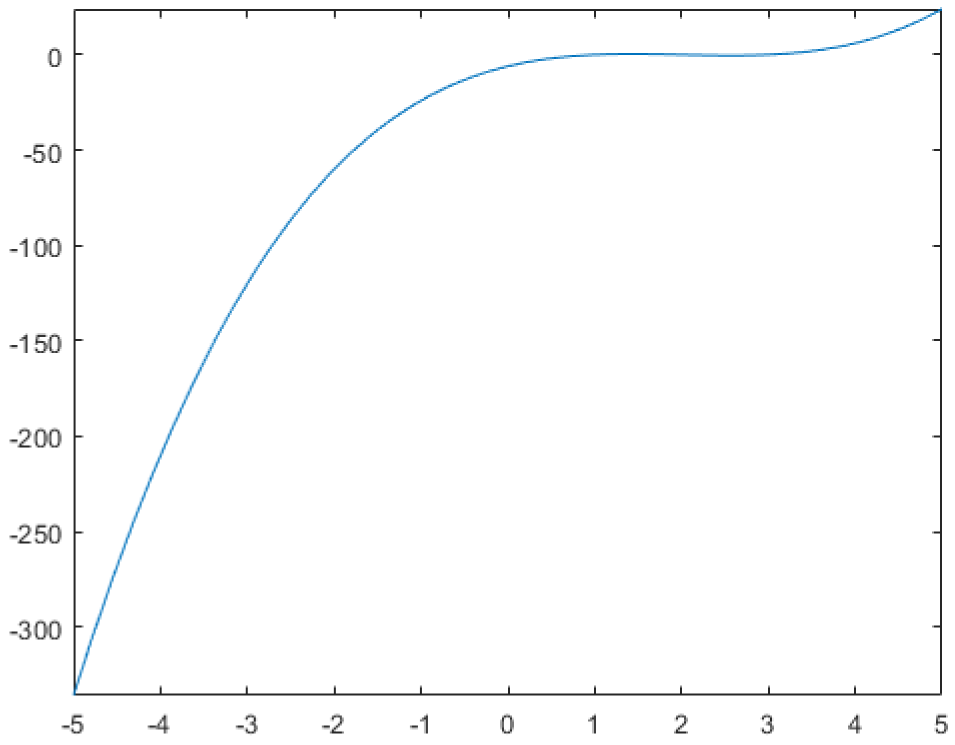
Symbolic Math Toolbox provides the plotting functions:

- `fplot` to create 2-D plots of symbolic expressions, equations, or functions in Cartesian coordinates.
- `fplot3` to create 3-D parametric plots.
- `ezpolar` to create plots in polar coordinates.
- `fsurf` to create surface plots.
- `fcontour` to create contour plots.
- `fmesh` to create mesh plots.

Explicit Function Plot

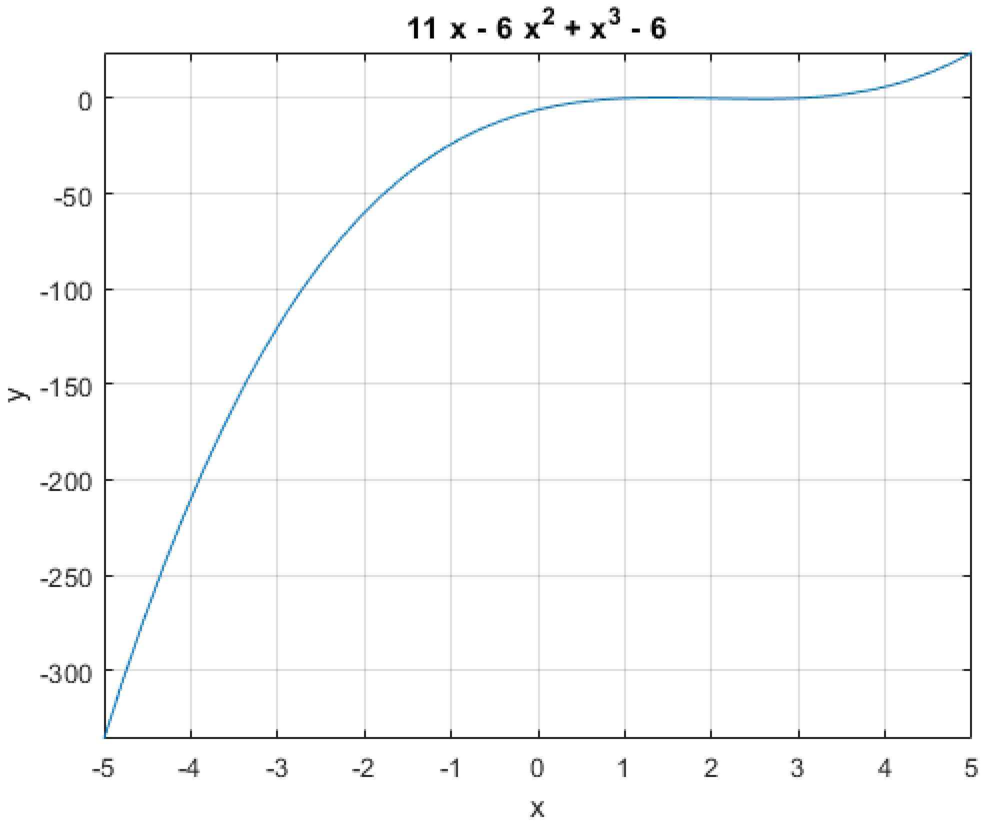
Create a 2-D line plot by using `fplot`. Plot the expression $x^3 - 6x^2 + 11x - 6$.

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



Add labels for the x- and y-axes. Generate the title by using `texlabel(f)`. Show the grid by using `grid on`. For details, see “Add Title, Axis Labels, and Legend to Graph” (MATLAB).

```
xlabel('x')
ylabel('y')
title(texlabel(f))
grid on
```

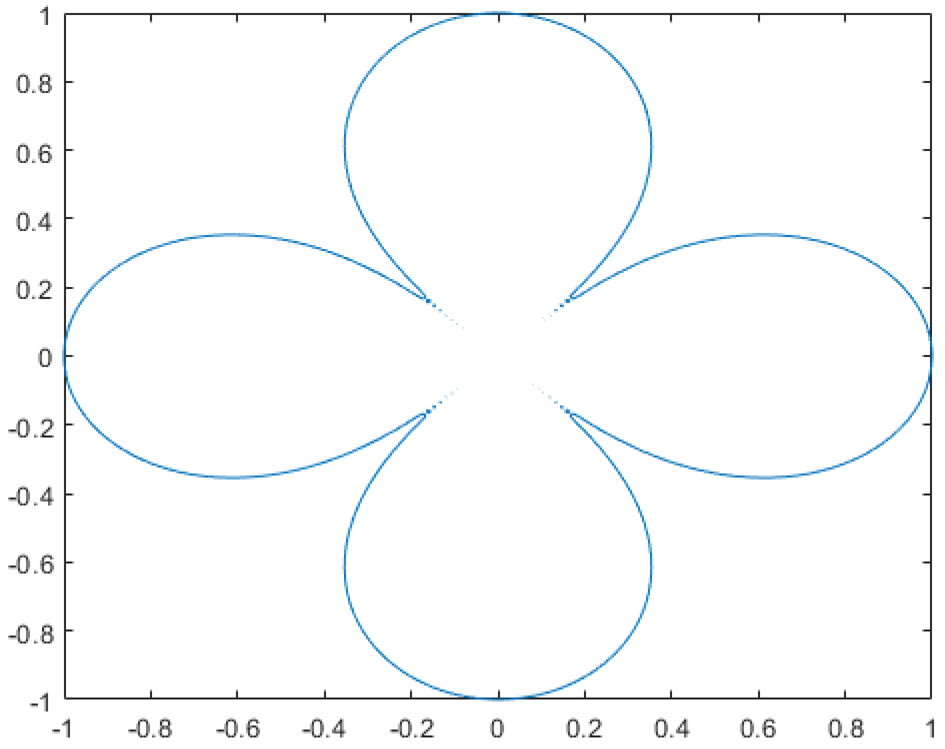


Implicit Function Plot

Plot equations and implicit functions using `fimplicit`.

Plot the equation $(x^2 + y^2)^4 = (x^2 - y^2)^2$ over $-1 < x < 1$.

```
syms x y
eqn = (x^2 + y^2)^4 == (x^2 - y^2)^2;
fimplicit(eqn, [-1 1])
```



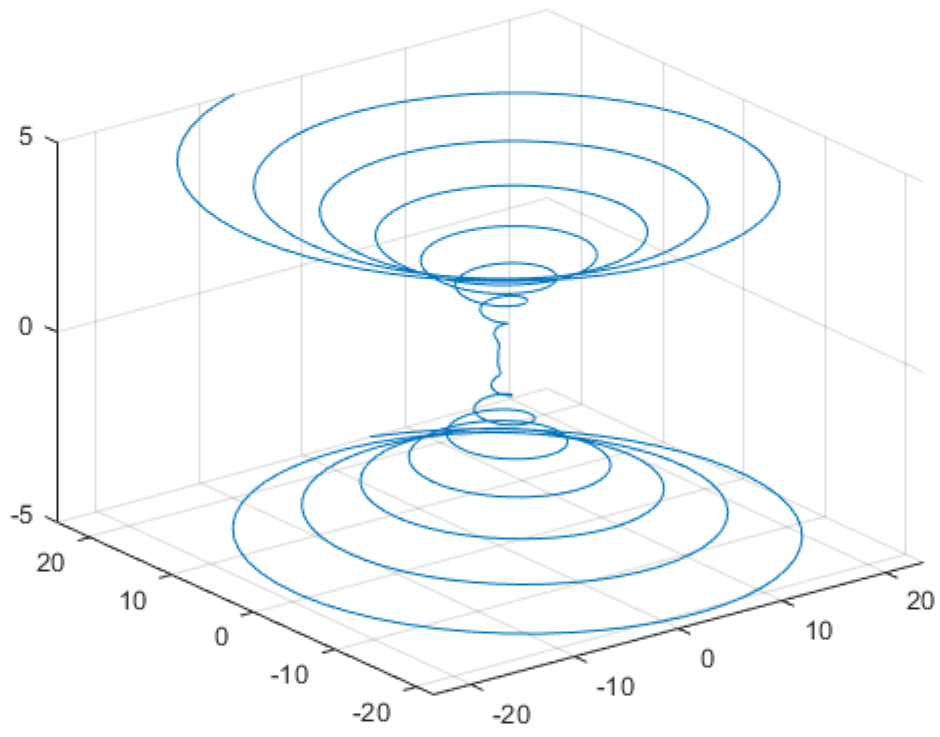
3-D Plot

Plot 3-D parametric lines by using `fplot3`.

Plot the parametric line

$$\begin{aligned} x &= t^2 \sin(10t) \\ y &= t^2 \cos(10t) \\ z &= t. \end{aligned}$$

```
syms t
fplot3(t^2*sin(10*t), t^2*cos(10*t), t)
```

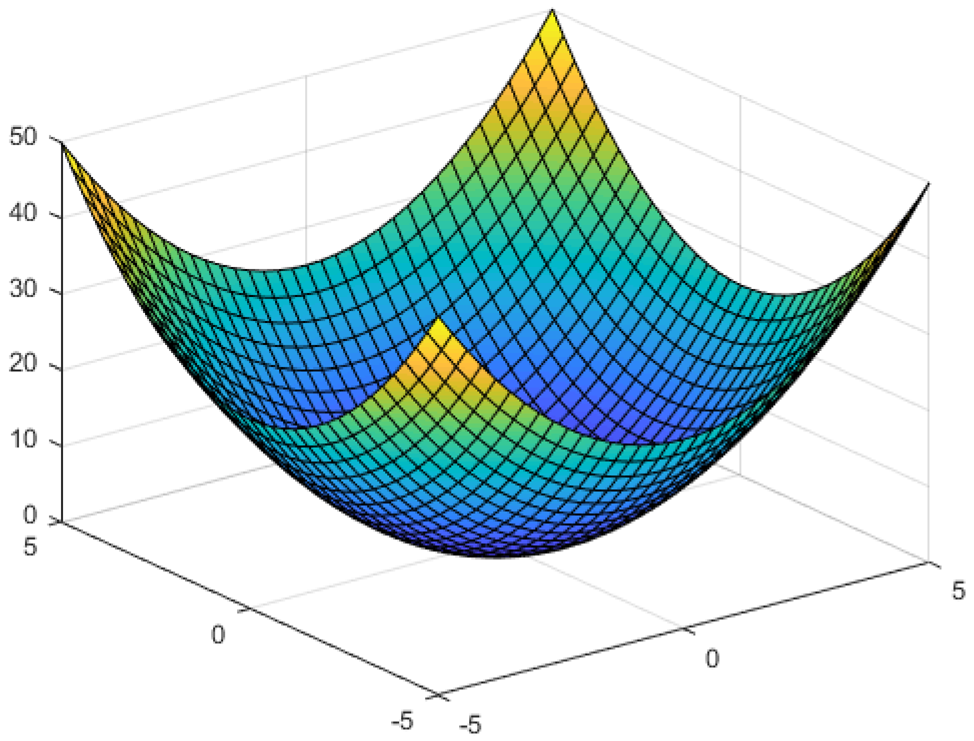


Create Surface Plot

Create a 3-D surface by using `fsurf`.

Plot the paraboloid $z = x^2 + y^2$.

```
syms x y
fsurf(x^2 + y^2)
```



See Also

More About

- “Create Symbolic Numbers, Variables, and Expressions” on page 1-3
- “Create Symbolic Functions” on page 1-7
- “Create Symbolic Matrices” on page 1-9
- “Use Assumptions on Symbolic Variables” on page 1-28

Use Assumptions on Symbolic Variables

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: 1-by-0
```

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, positive numbers, 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
```

The assumptions that you can assign to a symbolic object with sym or syms are real, rational, integer 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 3-68.

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)
```

```
ans =
Empty sym: 0-by-1
```

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

```
assume(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

   ```
   assume(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 3-67.

See Also

More About

- “Create Symbolic Numbers, Variables, and Expressions” on page 1-3
- “Create Symbolic Functions” on page 1-7
- “Create Symbolic Matrices” on page 1-9
- “Perform Symbolic Computations” on page 1-12

Using Symbolic Math Toolbox Software

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 =
[ b, t]
```

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

When you call `symvar` on a symbolic function, `symvar` returns the function inputs before other variables.

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

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

When called with two arguments for symbolic functions, `symvar` also follows this behavior.

```
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`.

Units of Measurement Tutorial

Use units of measurement with Symbolic Math Toolbox. This page shows how to define units, use units in equations (including differential equations), and verify the dimensions of expressions.

In this section...

Define and Convert Units

Load units by using `symunit`.

```
u = symunit;
```

Specify a unit by using `u.`*`unit`*. For example, specify a distance of `5` meters, a weight of `50` kilograms, and a speed of `10` kilometers per hour. In displayed output, units are placed in square brackets `[]`.

```
d = 5*u.m
w = 50*u.kg
s = 10*u.km/u.hr
```

```
d =
5*[m]
w =
50*[kg]
s =
10*([km]/[h])
```

Tip Use tab expansion to find names of units. Type `u.`, press **Tab**, and continue typing.

Units are treated like other symbolic expressions and can be used in any standard operation or function. Units are not automatically simplified, which provides flexibility.

Add `500` meters and `2` kilometers. The resulting distance is not automatically simplified.

```
d = 500*u.m + 2*u.km
```

```
d =
2*[km] + 500*[m]
```

Simplify `d` by using `simplify`. The `simplify` function automatically chooses the unit to simplify to.

```
d = simplify(d)
```

```
d =
(5/2)*[km]
```

Instead of automatically choosing a unit, rewrite `d` in terms of a specific unit by using `rewrite`. Rewrite `d` in terms of meters.

```
d = rewrite(d,u.m)
```

```
d =
2500*[m]
```

There are more unit conversion and unit system options. See “Unit Conversions and Unit Systems” on page 2-30.

Find the speed if the distance `d` is crossed in `50` seconds. The result has the correct units.

```
t = 50*u.s;
s = d/t
```

```
s =
50*([m]/[s])
```

Use Temperature Units in Absolute or Difference Forms

By default, temperatures are assumed to represent differences and not absolute measurements. For example, `5*u.Celsius` is assumed to represent a temperature difference of 5 degrees Celsius. This assumption allows arithmetical operations on temperature values.

To represent absolute temperatures, use kelvin, so that you do not have to distinguish an absolute temperature from a temperature difference.

Rewrite `23` degrees Celsius to kelvin, treating it first as a temperature difference and then as an absolute temperature.

```
u = symunit;
T = 23*u.Celsius;
diffK = rewrite(T,u.K)

diffK =
23*[K]

absK = rewrite(T,u.K,'Temperature','absolute')

absK =
(5923/20)*[K]
```

Verify Dimensions

In longer expressions, visually checking for units is difficult. You can check the dimensions of expressions automatically by verifying the dimensions of an equation.

First, define the kinematic equation $v^2 = {v_0}^2 + 2as$, where `v` represents velocity, `a` represents acceleration, and `s` represents distance. Assume `s` is in kilometers and all other units are in SI base units. To demonstrate dimension checking, the units of `a` are intentionally incorrect.

```
syms v v0 a s
u = symunit;
eqn = (v*u.m/u.s)^2 == (v0*u.m/u.s)^2 + 2*a*u.m/u.s*s*u.km

eqn =
v^2*([m]^2/[s]^2) == v0^2*([m]^2/[s]^2) + (2*a*s)*(([km]*[m])/[s])
```

Observe the units that appear in `eqn` by using `findUnits`. The returned units show that both kilometers and meters are used to represent distance.

```
findUnits(eqn)

ans =
[ [km], [m], [s]]
```

Check if the units have the same dimensions (such as length or time) by using `checkUnits` with the `'Compatible'` input. MATLAB assumes symbolic variables are dimensionless. `checkUnits` returns logical `0` (`false`), meaning the units are incompatible and not of the same physical dimensions.

```
checkUnits(eqn,'Compatible')

ans =
  logical
   0
```

Looking at `eqn`, the acceleration `a` has incorrect units. Correct the units and recheck for compatibility again. `eqn` now has compatible units.

```
eqn = (v*u.m/u.s)^2 == (v0*u.m/u.s)^2 + 2*a*u.m/u.s^2*s*u.km;
checkUnits(eqn,'Compatible')

ans =
  logical
   1
```

Now, to check that each dimension is consistently represented by the same unit, use `checkUnits` with the `'Consistent'` input. `checkUnits` returns logical `0` (`false`) because meters and kilometers are both used to represent distance in `eqn`.

```
checkUnits(eqn,'Consistent')

ans =
  logical
   0
```

Rewrite `eqn` to SI base units to make the units consistent. Run `checkUnits` again. `eqn` has both compatible and consistent units.

```
eqn = rewrite(eqn,'SI')

eqn =
 v^2*([m]^2/[s]^2) == v0^2*([m]^2/[s]^2) + (2000*a*s)*([m]^2/[s]^2)

checkUnits(eqn)

ans =
  struct with fields:

    Consistent: 1
    Compatible: 1
```

After you finish working with units and only need the dimensionless equation or expression, separate the units and the equation by using `separateUnits`.

```
[eqn,units] = separateUnits(eqn)
```

```
eqn =
v^2 == v0^2 + 2000*a*s
units =
1*([m]^2/[s]^2)
```

You can return the original equation with units by multiplying `eqn` with `units` and expanding the result.

```
expand(eqn*units)
```

```
ans =
v^2*([m]^2/[s]^2) == v0^2*([m]^2/[s]^2) + (2000*a*s)*([m]^2/[s]^2)
```

To calculate numeric values from your expression, substitute for symbolic variables using `subs`, and convert to numeric values using `double` or `vpa`.

Solve `eqn` for `v`. Then find the value of `v` where `v0 = 5`, `a = 2.5`, and `s = 10`. Convert the result to double.

```
v = solve(eqn,v);
v = v(2);        % choose the positive solution
vSol = subs(v,[v0 a s],[5 2.5 10]);
vSol = double(vSol)
```

```
vSol =
  223.6627
```

Use Units in Differential Equations

Use units in differential equations just as in standard equations. This section shows how to use units in differential equations by deriving the velocity relations $v = v_0 + at$ and

$v^2 = {v_0}^2 + 2as$ starting from the definition of acceleration $a = \frac{dv}{dt}$.

Represent the definition of acceleration symbolically using SI units. Given that the velocity `V` has units, `V` must be differentiated with respect to the correct units as `T = t*u.s` and not just `t`.

```
syms V(t) a
u = symunit;
T = t*u.s;          % time in seconds
A = a*u.m/u.s^2;    % acceleration in meters per second
eqn1 = A == diff(V,T)
```

```
eqn1(t) =
a*([m]/[s]^2) == diff(V(t), t)*(1/[s])
```

Because the velocity `V` is unknown and does not have units, `eqn1` has incompatible and inconsistent units.

```
checkUnits(eqn1)
```

```
ans =
  struct with fields:

    Consistent: 0
    Compatible: 0
```

Solve `eqn1` for `V` with the condition that the initial velocity is v_0. The result is the equation $v(t) = v_0 + at$.

```
syms v0
cond = V(0) == v0*u.m/u.s;
eqn2 = V == dsolve(eqn1,cond)
```

```
eqn2(t) =
V(t) == v0*([m]/[s]) + a*t*([m]/[s])
```

Check that the result has the correct dimensions by substituting `rhs(eqn2)` into `eqn1` and using `checkUnits`.

```
checkUnits(subs(eqn1,V,rhs(eqn2)))
```

```
ans =
  struct with fields:

    Consistent: 1
    Compatible: 1
```

Now, derive $v^2 = v_0^2 + 2as$. Because velocity is the rate of change of distance, substitute `V` with the derivative of distance `S`. Again, given that `S` has units, `S` must be differentiated with respect to the correct units as `T = t*u.s` and not just `t`.

```
syms S(t)
eqn2 = subs(eqn2,V,diff(S,T))
```

```
eqn2(t) =
diff(S(t), t)*(1/[s]) == v0*([m]/[s]) + a*t*([m]/[s])
```

Solve `eqn2` with the condition that the initial distance covered is `0`. Get the expected form of `S` by using `expand`.

```
cond2 = S(0) == 0;
eqn3 = S == dsolve(eqn2,cond2);
eqn3 = expand(eqn3)
```

```
eqn3(t) =
S(t) == t*v0*[m] + ((a*t^2)/2)*[m]
```

You can use this equation with the units in symbolic workflows. Alternatively, you can remove the units by returning the right side using `rhs`, separating units by using `separateUnits`, and using the resulting unitless expression.

```
[S units] = separateUnits(rhs(eqn3))
```

```
S(t) =
(a*t^2)/2 + v0*t

units(t) =
[m]
```

When you need to calculate numeric values from your expression, substitute for symbolic variables using `subs`, and convert to numeric values using `double` or `vpa`.

Find the distance traveled in `8` seconds where `v0 = 20` and `a = 1.3`. Convert the result to double.

```
S = subs(S,[v0 a],[20 1.3]);
dist = S(8);
dist = double(dist)
```

```
dist =
  201.6000
```

See Also

`checkUnits` | `findUnits` | `isUnit` | `newUnit` | `rewrite` | `separateUnits` | `symunit2str` | `unitConversionFactor`

More About

- "Unit Conversions and Unit Systems" on page 2-30
- "Units List" on page 2-13

External Websites

- The International System of Units (SI)

Units List

List of units of measurement available in Symbolic Math Toolbox. Every unit marked by *SI* accepts SI prefixes. For example, `m` accepts `nm`, `um`, `mm`, `cm`, `dm`, `km`.

For details on symbolic units, see "Units of Measurement Tutorial" on page 2-5.

Length

- `Ao` - angstrom
- `a_0` - Bohr radius
- `au` - astronomical unit
- `ch` - chain
- `ft` - foot
- `ft_US` - U.S. survey foot
- `ftm` - fathom
- `fur` - furlong
- `gg` - gauge
- `hand`
- `in` - inch
- `inm` - international nautical mile
- `land` - league
- `li` - link
- `line`
- `ly` - light-year
- `m` - meter (SI)
- `mi` - mile
- `mi_US` - U.S. survey mile
- `mil`
- `nmile` - British imperial nautical mile
- `pc` - parsec

- `pt` - point
- `rod`
- `span`
- `xu` - x unit
- `xu_Cu` - x unit (copper)
- `xu_Mo` - x unit (molybdenum)
- `yd` - yard

Mass

- `Mt` - metric megaton
- `ct` - carat
- `cwt` - U.S. customary short hundredweight
- `cwt_UK` - British imperial short hundredweight
- `dalton` - atomic mass constant
- `dr` - dram
- `g` - gram
- `gr` - grain
- `hyl`
- `kt` - metric kiloton
- `lbm` - pound mass
- `m_e` - electron mass
- `oz` - ounce
- `quarter`
- `slug`
- `stone`
- `t` - metric ton
- `tn` - U.S. customary short ton
- `ton_UK` - British imperial ton

Time

- `d` - day
- `fortnight` - 14 days
- `h` - hour
- `min` - minute
- `month_30` - 30-day month
- `s` - second (SI)
- `week` - 7-day week
- `year_360` - 360-day year
- `year_Julian` - Julian year
- `year_Tropical` - Tropical year
- `year_Gregorian` - Gregorian year

Absorbed Dose or Dose Equivalent

- `Gy` - gray (SI)
- `Rad` - absorbed radiation dose
- `Sv` - sievert (SI)
- `rem` - roentgen equivalent man

Acceleration

- `Gal` - gal
- `g_n` - earth gravitational acceleration

Activity

- `Bq` - becquerel (SI)
- `Ci` - curie

Amount of Substance

- `item` - number of items
- `mol` - mole (SI)
- `molecule` - number of molecules

Angular Momentum

- `Nms` - newton meter second
- `h_bar` - reduced Planck constant
- `h_c` - Planck constant

Area

- `a` - are
- `ac` - acre
- `barn`
- `circ_mil` - circular mil
- `circ_inch` - circular inch
- `ha` - metric hectare
- `ha_US` - U.S. survey hectare
- `ro` - rood
- `twp` - township

Capacitance

- `F` - farad (SI)
- `abF` - abfarad
- `statF` - statfarad

Catalytic Activity

- `kat` - katal (SI)

Conductance

- `S` - siemens (SI)
- `abS` - absiemens
- `statS` - statsiemens

Data Transfer Rate

- `Bd` - baud
- `bps` - bit per second

Digital Information

- `B` - byte
- `bit` - basic unit of information

Dose Equivalent

- `Sv` - sievert (SI)

Dynamic Viscosity

- `P` - poise
- `reyn` - reynolds

Electric Charge

- `C` - coulomb (SI)
- `Fr` - franklin
- `abC` - abcoulomb
- `e` - elementary charge
- `statC` - statcoulomb

Electric Current

- `A` - ampere (SI)
- `Bi` - biot
- `abA` - abampere
- `statA` - statampere

Electric Dipole Moment

- `debye`

Electric Potential

- `V` - volt (SI)
- `abV` - abvolt
- `statV` - statvolt

Electric Potential or Electromotive Force

- `V` - volt (SI)
- `abV` - abvolt
- `statV` - statvolt

Energy or Work or Heat

- `Btu_IT` - British thermal unit (International Table)
- `Btu_th` - British thermal unit (thermochemical)
- `E_h` - Hartree energy
- `J` - joule (SI)
- `Nm` - newton meter
- `Wh` - watt hour
- `Ws` - watt second
- `cal_4` - calorie (4 degree Celsius)

- `cal_20` - calorie (20 degree Celsius)
- `cal_15` - calorie (15 degree Celsius)
- `cal_IT` - calorie (International Table)
- `cal_th` - calorie (thermochemical)
- `cal_mean` - calorie (mean)
- `eV` - electronvolt
- `erg`
- `kcal_4` - kilocalorie (4 degree Celsius)
- `kcal_20` - kilocalorie (20 degree Celsius)
- `kcal_15` - kilocalorie (15 degree Celsius)
- `kcal_IT` - kilocalorie (International Table)
- `kcal_th` - kilocalorie (thermochemical)
- `kcal_mean` - kilocalorie (mean)
- `kpm` - kilopond meter
- `therm`

Energy Per Temperature

- `k_B` - Boltzmann constant

European Currency

- `Cent` - cent
- `EUR` - Euro

Field Strength

- `Oe` - oersted

Flow Rate

- `gpm` - U.S. customary gallon per minute

- `gpm_UK` - British imperial gallon per minute
- `lpm` - liter per minute

Force

- `N` - newton (SI)
- `dyn` - dyne
- `kgf` - kilogram force
- `kip`
- `kp` - kilopond
- `lbf` - pound force
- `ozf` - ounce force
- `p` - pond
- `pdl` - poundal
- `sn` - sthene
- `tonf` - short ton force

Former European Currency

- `ATS` - Austrian Schilling
- `BEF` - Belgian Franc
- `DM` - German Mark
- `ESP` - Spanish Peseta
- `FIM` - Finnish Markka
- `FRF` - French Franc
- `IEP` - Irish Pound
- `ITL` - Italian Lire
- `LUF` - Luxembourgian Franc
- `NLG` - Dutch Gulden
- `PTE` - Portugese Escudo

Frequency

- `Hz` - hertz (SI)

Frequency of Rotation

- `rpm` - revolution per minute
- `rps` - revolution per second

Fuel Consumption

- `l_100km` - liter per 100 km

Fuel Economy

- `mpg` - mile per gallon

Gravity

- `G_c` - Newtonian constant of gravitation

Heat

- `Btu_IT` - British thermal unit (International Table)
- `Btu_th` - British thermal unit (thermochemical)
- `cal_4` - calorie (4 degree Celsius)
- `cal_20` - calorie (20 degree Celsius)
- `cal_15` - calorie (15 degree Celsius)
- `cal_IT` - calorie (International Table)
- `cal_th` - calorie (thermochemical)
- `cal_mean` - calorie (mean)
- `kcal_4` - kilocalorie (4 degree Celsius)
- `kcal_20` - kilocalorie (20 degree Celsius)

- `kcal_15` - kilocalorie (15 degree Celsius)
- `kcal_IT` - kilocalorie (International Table)
- `kcal_th` - kilocalorie (thermochemical)
- `kcal_mean` - kilocalorie (mean)
- `therm`

Illuminance

- `lx` - lux (SI)
- `nx` - nox
- `ph` - phot

Inductance

- `H` - henry (SI)
- `abH` - abhenry
- `statH` - stathenry

Ionising Dosage

- `R` - roentgen

Kinematic Viscosity

- `St` - stokes
- `newt`

Luminance

- `asb` - apostilb
- `sb` - stilb

Luminous Flux

- `lm` - lumen (SI)

Luminous Intensity

- `cd` - candela (SI)
- `cp` - candlepower

Magnetic Flux

- `Mx` - maxwell
- `Wb` - weber (SI)
- `abWb` - abweber
- `statWb` - statweber

Magnetic Flux Density

- `G` - gauss
- `T` - tesla (SI)
- `abT` - abtesla
- `statT` - stattesla

Magnetic Force

- `Gb` - gilbert

Mass Per Length

- `den` - denier
- `tex` - filament tex

Particle Per Amount of Substance

- `N_A` - Avogadro constant

Plane Angle

- `arcsec` - arcsecond
- `arcmin` - arcminute
- `deg` - degree
- `rad` - radian (SI)
- `rev` - revolution

Power

- `HP_E` - electrical horsepower
- `HP_I` - mechanical horsepower
- `HP_UK` - British imperial horsepower
- `HP_DIN` - metric horsepower (DIN 66036)
- `PS_SAE` - net horsepower (SAE J1349)
- `PS_DIN` - horsepower (DIN 70020)
- `poncelet`

Power

- `HP_E` - electrical horsepower
- `HP_I` - mechanical horsepower
- `HP_UK` - British imperial horsepower
- `HP_DIN` - metric horsepower (DIN 66036)
- `PS_SAE` - net horsepower (SAE J1349)
- `PS_DIN` - horsepower (DIN 70020)
- `W` - watt (SI)
- `poncelet`

Pressure

- `Ba` - barye
- `Pa` - pascal (SI)
- `Torr` - torr
- `at` - technical atmosphere
- `atm` - standard atmosphere
- `bar`
- `cmHg` - centimeter of mercury (conventional)
- `cmH2O` - centimeter of water (conventional)
- `ftHg` - foot of mercury (conventional)
- `ftH2O` - foot of water (conventional)
- `inHg` - inch of mercury (conventional)
- `inH2O` - inch of water (conventional)
- `ksf` - kip per square foot
- `ksi` - kip per square inch
- `mH2O` - meter of water (conventional)
- `mHg` - meter of mercury (conventional)
- `mmHg` - millimeter of mercury (conventional)
- `mmH2O` - millimeter of water (conventional)
- `psf` - pound force per square foot
- `psi` - pound force per square inch
- `pz` - pieze

Pressure or Stress

- `Ba` - barye
- `Pa` - pascal (SI)
- `Torr` - torr
- `at` - technical atmosphere

- `atm` - standard atmosphere
- `bar`
- `cmHg` - centimeter of mercury (conventional)
- `cmH2O` - centimeter of water (conventional)
- `ftHg` - foot of mercury (conventional)
- `ftH2O` - foot of water (conventional)
- `inHg` - inch of mercury (conventional)
- `inH2O` - inch of water (conventional)
- `ksf` - kip per square foot
- `ksi` - kip per square inch
- `mH2O` - meter of water (conventional)
- `mHg` - meter of mercury (conventional)
- `mmHg` - millimeter of mercury (conventional)
- `mmH2O` - millimeter of water (conventional)
- `psf` - pound force per square foot
- `psi` - pound force per square inch
- `pz` - pieze

Radiation

- `lan` - langley

Radioactivity

- `Bq` - becquerel (SI)
- `Ci` - curie

Reciprocal Length

- `kayser`

Refractive Power of Lenses

- `dpt` - diopter

Resistance

- `Ohm` - ohm (SI)
- `abOhm` - abohm
- `statOhm` - statohm

Solid Angle

- `sr` - steradian (SI)

Substance Per Volume

- `molarity`

Temperature

- `Celsius` - degree Celsius (SI)
- `Fahrenheit` - degree Fahrenheit
- `K` - kelvin (SI)
- `Rankine` - degree Rankine
- `Reaumur` - degree Reaumur

Velocity

- `Kyne` - kyne
- `c_0` - speed of light in vacuum
- `fpm` - foot per minute
- `fps` - foot per second
- `kmh` - kilometer per hour

- `knot_UK` - British imperial knot
- `kts` - international knot
- `mach` - speed of sound
- `mph` - mile per hour

Volume

- `barrel`
- `bbl` - U.S. customary dry barrel
- `bu_UK` - British imperial bushel
- `chaldron`
- `dry_bu` - U.S. customary dry bushel
- `dry_pk` - U.S. customary dry peck
- `dry_pt` - U.S. customary dry pint
- `dry_qt` - U.S. customary dry quart
- `dry_gal` - U.S. customary dry gallon
- `fldr` - U.S. customary fluid dram
- `fldr_UK` - British imperial fluid drachm (dram)
- `floz` - U.S. customary fluid ounce
- `floz_UK` - British imperial fluid ounce
- `gal` - U.S. customary liquid gallon
- `gal_UK` - British imperial gallon
- `gill` - U.S. customary fluid gill
- `gill_UK` - British imperial gill
- `igal` - British imperial gallon
- `l` - liter
- `liq_pt` - U.S. customary liquid pint
- `liq_qt` - U.S. customary liquid quart
- `minim` - U.S. customary minim
- `minim_UK` - British imperial minim

- `pint` - U.S. customary liquid pint
- `pint_UK` - British imperial pint
- `pk_UK` - British imperial peck
- `pottle` - British imperial pottle
- `qt_UK` - British imperial quart
- `quart` - U.S. customary liquid quart

See Also

`checkUnits` | `isUnit` | `newUnit` | `rewrite` | `separateUnits` | `symunit` | `symunit2str` | `unitConversionFactor`

More About

- “Units of Measurement Tutorial” on page 2-5
- “Unit Conversions and Unit Systems” on page 2-30

See Also

Related Examples

- “Units of Measurement Tutorial” on page 2-5
- “Unit Conversions and Unit Systems” on page 2-30

More About

- “Units List” on page 2-13

External Websites

- The International System of Units (SI)

Unit Conversions and Unit Systems

Convert between units with Symbolic Math Toolbox. This page shows conversions between units and between systems of units, such as SI, CGS, or a user-defined unit system.

In this section...

Convert Units

Convert between units by using `rewrite`.

Convert 1.2 meters to centimeters.

```
u = symunit;
len = 1.2*u.m;
len = rewrite(len,u.cm)
```

```
len =
120*[cm]
```

Convert `len` to inches. The result is in exact symbolic form. Separate units and convert to double.

```
len = rewrite(len,u.in)
```

```
len =
(6000/127)*[in]
```

```
[len units] = separateUnits(len);
len = double(len)
```

```
len =
   47.2441
```

Calculate the force needed to accelerate a mass of 5 kg at 2 m/s^2.

```
m = 5*u.kg;
a = 2*u.m/u.s^2;
F = m*a
```

```
F =
10*(([kg]*[m])/[s]^2)
```

Convert the result to Newton.

```
F = rewrite(F,u.N)
```

```
F =
10*[N]
```

Tip Use tab expansion to find names of units. Type `u.`, press **Tab**, and continue typing.

Calculate the energy when force `F` is applied for 3 meters. Convert the result to Joule.

```
d = 3*u.m;
E = F*d
```

```
E =
30*[N]*[m]
```

```
E = rewrite(E,u.J)
```

```
E =
30*[J]
```

Convert `E` to kilowatt-hour.

```
E = rewrite(E,u.kWh)
```

```
E =
(1/120000)*[kWh]
```

Temperature Unit Conversion

Temperatures can represent either absolute temperatures or temperature differences. By default, temperatures are assumed to be differences. Convert temperatures assuming temperatures are absolute by specifying the `'Temperature'` input as `'absolute'`.

Rewrite 23 degrees Celsius to degrees Kelvin, first as a temperature difference and then as an absolute temperature.

```
u = symunit;
T = 23*u.Celsius;
relK = rewrite(T,u.K,'Temperature','difference')
```

```
relK =
23*[K]
```

```
absK = rewrite(T,u.K,'Temperature','absolute')
```

```
absK =
(5923/20)*[K]
```

Because the value 0 is dimensionless and 0 degrees cannot be represented, convert 0 degrees between temperature units by using cell input.

Convert 0 degrees Celsius to degrees Fahrenheit.

```
tC = {0,u.Celsius};
tF = rewrite(tC,u.Fahrenheit,'Temperature','Absolute')
```

```
tF =
32*[Fahrenheit]
```

Convert to SI, CGS, or US Unit Systems

Automatically convert to the correct units by converting to a unit system. Further, converting to the *derived* units of a unit system attempts to select convenient units. By default, the available unit systems are SI, CGS, and US. In addition, you can define custom unit systems.

Calculate the force due to a 5 kg mass accelerating at 2 m/s^2. The resulting units are hard to read. Convert them to convenient units by specifying the SI and Derived options. rewrite automatically chooses the correct units of Newton.

```
u = symunit;
m = 5*u.kg;
a = 2*u.m/u.s^2;
F = m*a
```

```
F =
10*(([kg]*[m])/[s]^2)
```

```
F = rewrite(F,'SI','Derived')

F =
10*[N]
```

Convert F to US units. By default, the converted units are base units. For convenience, also convert into derived units by specifying the Derived option. The derived units are easier to read.

```
F = rewrite(F,'US')

F =
(1250000000000/17281869297)*(([ft]*[lbm])/[s]^2)

F = rewrite(F,'US','Derived')

F =
(20000000000000/8896443230521)*[lbf]
```

Convert F to CGS derived units.

```
F = rewrite(F,'CGS','Derived')

F =
1000000*[dyn]
```

Convert a specification in SI to US derived units. Specify the temperatures as absolute.

```
loadCell = [    3*u.kg;       % capacity
               50*u.mm;       % length
               15*u.mm;       % width
               10*u.mm;       % height
              -10*u.Celsius;  % minimum temperature
               40*u.Celsius;  % maximum temperature
             ];
loadCell = rewrite(loadCell,'US','derived','Temperature','absolute')

loadCell =
 (300000000/45359237)*[lbm]
             (125/762)*[ft]
              (25/508)*[ft]
              (25/762)*[ft]
            14*[Fahrenheit]
           104*[Fahrenheit]
```

If rewrite does not choose your preferred unit, then adjust the result with further rewrite commands. Here, inches are more convenient than feet. Convert the result to inches.

```
loadCell = rewrite(loadCell,u.inch)
```

```
loadCell =
 (300000000/45359237)*[lbm]
            (250/127)*[in]
             (75/127)*[in]
             (50/127)*[in]
           14*[Fahrenheit]
          104*[Fahrenheit]
```

The exact symbolic values are hard to read. Separate the units and convert to double.

```
[loadCellDouble loadCellUnits] = separateUnits(loadCell);
loadCellDouble = double(loadCellDouble)
```

```
loadCellDouble =
    6.6139
    1.9685
    0.5906
    0.3937
   14.0000
  104.0000
```

Alternatively, approximate the result to high precision by using vpa. The vpa function also keeps the symbolic units because it returns symbolic output.

```
loadCell = vpa(loadCell)
```

```
loadCell =
 6.6138678655463274216892140403508*[lbm]
   1.968503937007874015748031496063*[in]
  0.59055118110236220472440944881889*[in]
  0.39370078740157480314960629921260*[in]
                       14.0*[Fahrenheit]
                      104.0*[Fahrenheit]
```

Define Custom Unit System from Existing System

Custom unit systems provide flexibility in converting units. You can easily define a custom unit system by modifying a default unit system. Alternatively, you can define the system directly. For definitions of unit system, base units, and derived units, see “Unit System Definition” on page 2-37.

In photonics, commonly used units are nanosecond (ns), electron volt (eV), and nanometer (nm). Define a unit system with these units by modifying the SI unit system. Get SI base and derived units by using `baseUnits` and `derivedUnits`. Modify the units by using `subs`.

```
u = symunit;
bunits = baseUnits('SI');
bunits = subs(bunits,[u.m u.s],[u.nm u.ns])
```

```
bunits =
[ [kg], [ns], [nm], [A], [cd], [mol], [K]]
```

```
dunits = derivedUnits('SI');
dunits = subs(dunits,u.J,u.eV)
```

```
dunits =
[ [F], [C], [S], [H], [V], [eV], [N], [lx], [lm], [Wb], [W], [Pa],...
 [Ohm], [T], [Gy], [Bq], [Sv], [Hz], [kat], [rad], [sr], [Celsius]]
```

Note Do not define variables called `baseUnits` and `derivedUnits` because the variables prevent access to the `baseUnits` and `derivedUnits` functions.

Define the new unit system by using `newUnitSystem`.

```
phSys = newUnitSystem('photonics',bunits,dunits)
```

```
phSys = 
    "photonics"
```

Calculate the energy of a photon of frequency 1 GHz and convert the result to derived units of the `phSys` system. The result is in electron volts.

```
f = 1*u.GHz;
E = u.h_c*f;
E = rewrite(E,phSys,'Derived')
```

```
E =
0.0000041356676623401643884479280999879*[eV]
```

The exact symbolic result is hard to read. Separate the units and convert to double.

```
[E Eunits] = separateUnits(E);
E = double(E)

E =
   4.1357e-06
```

After completing calculations, remove the unit system.

```
removeUnitSystem(phSys)
```

Define Custom Unit System Directly

Define a custom unit system for atomic units (au).

Define these base units:

Dimension	Unit	Implementation
Mass	Electron rest mass	`u.m_e`
Elementary charge	Electron charge	`u.e`
Length	Bohr radius (a_0)	`u.Bohr`
Time	$\hbar/E_{\mathrm{h}}$	Define by using `newUnit`.

```
u = symunit;
t_au = newUnit('t_au',u.hbar/u.E_h);
bunits = [u.m_e u.e u.Bohr u.t_au]

bunits =
[ [m_e], [e], [a_0], [t_au]]
```

Define these derived units:

Dimension	Unit	Implementation
Angular momentum	Reduced Planck's constant	`u.hbar`
Energy	Hartree	`u.E_h`
Electric dipole moment	ea_0	Define by using `newUnit`.
Magnetic dipole moment	2 Bohr Magneton = $e\hbar/2m_{\mathrm{e}}$	Define by using `newUnit`.
Electric potential	E_{h}/e	Define by using `newUnit`.

```
edm_au = newUnit('edm_au',u.e*u.bohr);
mdm_au  = newUnit('mdm_au', u.e*u.hbar/(2*u.me));
```

```
ep_au  = newUnit('ep_au', u.E_h/u.e);
dunits = [u.hbar u.E_h u.edm_au u.mdm_au u.ep_au]
```

```
dunits =
[ [h_bar], [E_h], [edm_au], [mdm_au], [ep_au]]
```

Define the unit system.

```
auSys = newUnitSystem('atomicUnits',bunits,dunits)
```

```
auSys = 
    "atomicUnits"
```

Rewrite the properties of a proton to atomic units.

```
proton = [  1.672624898e-27*u.kg;      % mass
            1.6021766208e-19*u.C;      % charge
            5.4e-24*u.e*u.cm;          % electric dipole moment
            1.4106067873e-26*u.J/u.T;  % magnetic dipole moment
         ];
proton = rewrite(proton,auSys,'Derived')
```

```
proton =
                   1836.1559670674356174696928918542*[m_e]
                     1.0000000000000000578208778346486*[e]
 0.000000000000001020452107747227250600843514806 1*[edm_au]
             0.0015210322058038370229109632800588*[mdm_au]
```

After completing calculations, remove the unit system and the added units.

```
removeUnitSystem(auSys)
removeUnit([u.t_au u.edm_au u.mdm_au u.ep_au])
```

Unit System Definition

A unit system is a collection of base units and derived units that follows these rules:

- Base units must be independent in terms of the dimensions mass, time, length, electric current, luminous intensity, amount of substance, and temperature. Therefore, a unit system has up to 7 base units. As long as the independence is satisfied, any unit can be a base unit, including units such as newton or watt.
- A unit system can have less than 7 base units. For example, mechanical systems need base units only for the dimensions length, mass, and time.

- Derived units in a unit system must have a representation in terms of the products of powers of the base units for that system. Unlike base units, derived units do not have to be independent.
- Derived units are optional and added for convenience of representation. For example, kg m/s^2 is abbreviated by Newton.
- An example of a unit system is the SI unit system, which has 7 base units: kilogram, second, meter, ampere, candela, mol, and kelvin. There are 22 derived units found by calling `derivedUnits('SI')`.

See Also

`baseUnits` | `derivedUnits` | `newUnitSystem` | `removeUnit` | `removeUnitSystem` | `rewrite` | `symunit`

More About

- “Units of Measurement Tutorial” on page 2-5
- “Units List” on page 2-13

External Websites

- The International System of Units (SI)

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`:

```
y = diff(g)
```

```
y =
exp(x)*cos(x) - exp(x)*sin(x)
```

To find the derivative of `g` for a given value of `x`, substitute `x` for the value using `subs` and return a numerical value using `vpa`. Find the derivative of `g` at `x = 2`.

```
vpa(subs(y,x,2))
```

```
ans =
-9.7937820180676088383807818261614
```

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 simplify an answer, in which case you can use the `simplify` command. For an example of such simplification, see “More Examples” on page 2-41.

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 2-4. 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*1i)*1i`

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)]
```

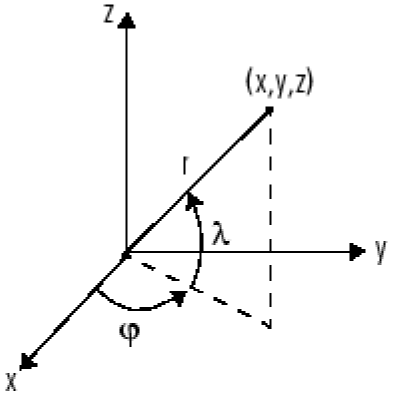
The command

```
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)`
$\frac{d^2f}{db^2}$	`diff(f, b, 2)`
$J = \frac{\partial(r,t)}{\partial(u,v)}$	`J = jacobian([r; t],[u; v])`

Functional Derivatives Tutorial

This example shows how to use functional derivatives in the Symbolic Math Toolbox™ using the example of the wave equation. The wave equation for a string fixed at its ends is solved using functional derivatives. A functional derivative is the derivative of a functional with respect to the function that the functional depends on. The Symbolic Math Toolbox™ implements functional derivatives using the `functionalDerivative` function.

Solving the wave equation is one application of functional derivatives. It describes the motion of waves, from the motion of a string to the propagation of an electromagnetic wave, and is an important equation in physics. You can apply the techniques illustrate in this example to applications in the calculus of variations from solving the Brachistochrone problem to finding minimal surfaces of soap bubbles.

Consider a string of length `L` suspended between the two points `x = 0` and `x = L`. The string has a characteristic density per unit length and a characteristic tension. Define the length, density, and tension as constants for later use. For simplicity, set these constants to `1`.

```
Length = 1;
Density = 1;
Tension = 1;
```

If the string is in motion, the string's kinetic and potential energies are a function of its displacement from rest `S(x,t)`, which varies with position `x` and time `t`. If `d` is the density per unit length, the kinetic energy is

$$T = \int_0^L \frac{d}{2}\left(\frac{d}{dt}S(x,t)\right)^2 dx.$$

The potential energy is

$$V = \int_0^L \frac{r}{2}\left(\frac{d}{dx}S(x,t)\right)^2 dx,$$

where *r* is the tension.

Enter these equations in MATLAB™. Since length must be positive, set this assumption. This assumption allows `simplify` to simplify the resulting equations into the expected form.

```
syms S(x,t) d r v L
assume(L>0)
T(x,t) = int(d/2*diff(S,t)^2,x,0,L);
V(x,t) = int(r/2*diff(S,x)^2,x,0,L);
```

The action `A` is `T-V`. The Principle of Least Action states that action is always minimized. Determine the condition for minimum action, by finding the functional derivative of `A` with respect to `S` using `functionalDerivative` and equate it to zero.

```
A = T-V;
eqn = functionalDerivative(A,S) == 0
```

```
eqn(x, t) =
```

$$L\,r\,\frac{\partial^2}{\partial x^2}\,S(x,t) - L\,d\,\frac{\partial^2}{\partial t^2}\,S(x,t) = 0$$

Simplify the equation using `simplify`. Convert the equation into its expected form by substituting for `r/d` with the square of the wave velocity `v`.

```
eqn = simplify(eqn)/r;
eqn = subs(eqn,r/d,v^2)
```

```
eqn(x, t) =
```

$$\frac{\frac{\partial^2}{\partial t^2}\,S(x,t)}{v^2} = \frac{\partial^2}{\partial x^2}\,S(x,t)$$

Solve the equation using the method of separation of variables. Set `S(x,t) = U(x)*V(t)` to separate the dependence on position `x` and time `t`. Separate both sides of the resulting equation using `children`.

```
syms U(x) V(t)
eqn2 = subs(eqn,S(x,t),U(x)*V(t));
eqn2 = eqn2/(U(x)*V(t))
```

```
eqn2(x, t) =
```

$$\frac{\frac{\partial^2}{\partial t^2}\ V(t)}{v^2\ V(t)} = \frac{\frac{\partial^2}{\partial x^2}\ U(x)}{U(x)}$$

```
tmp = children(eqn2);
```

Both sides of the equation depend on different variables, yet are equal. This is only possible if each side is a constant. Equate each side to an arbitrary constant `C` to get two differential equations.

```
syms C
eqn3 = tmp(1) == C
```

```
eqn3 =
```

$$\frac{\frac{\partial^2}{\partial t^2}\ V(t)}{v^2\ V(t)} = C$$

```
eqn4 = tmp(2) == C
```

```
eqn4 =
```

$$\frac{\frac{\partial^2}{\partial x^2}\ U(x)}{U(x)} = C$$

Solve the differential equations using `dsolve` with the condition that displacement is `0` at `x = 0` and `t = 0`. Simplify the equations to their expected form using `simplify` with the `Steps` option set to `50`.

```
V(t) = dsolve(eqn3,V(0)==0,t);
U(x) = dsolve(eqn4,U(0)==0,x);
V(t) = simplify(V(t),'Steps',50)
```

```
V(t) =
```

$$-2\,C_3\sinh(\sqrt{C}\ t\ v)$$

```
U(x) = simplify(U(x),'Steps',50)
```

```
U(x) =
```

$$-2\,C_6\sinh(\sqrt{C}\;x)$$

Obtain the constants in the equations.

```
p1 = setdiff(symvar(U(x)),sym([C,x]))
```

```
p1 =
```

$$C_6$$

```
p2 = setdiff(symvar(V(t)),sym([C,v,t]))
```

```
p2 =
```

$$C_3$$

The string is fixed at the positions `x = 0` and `x = L`. The condition `U(0) = 0` already exists. Apply the boundary condition that `U(L) = 0` and solve for `C`.

```
eqn_bc = U(L) == 0;
[solC,param,cond] = solve(eqn_bc,C,'ReturnConditions',true)
```

```
solC =
```

$$-\frac{k^2\,\pi^2}{L^2}$$

```
param =
```

$$k$$

```
cond =
```

$$C_6 \neq 0 \wedge 1 \leq k \wedge k \in \mathbb{Z}$$

```
assume(cond)
```

The solution `S(x,t)` is the product of `U(x)` and `V(t)`. Find the solution, and substitute the characteristic values of the string into the solution to obtain the final form of the solution.

```
S(x,t) = U(x)*V(t);
S = subs(S,C,solC);
S = subs(S,[L v],[Length sqrt(Tension/Density)]);
```

The parameters `p1` and `p2` determine the amplitude of the vibrations. Set `p1` and `p2` to `1` for simplicity.

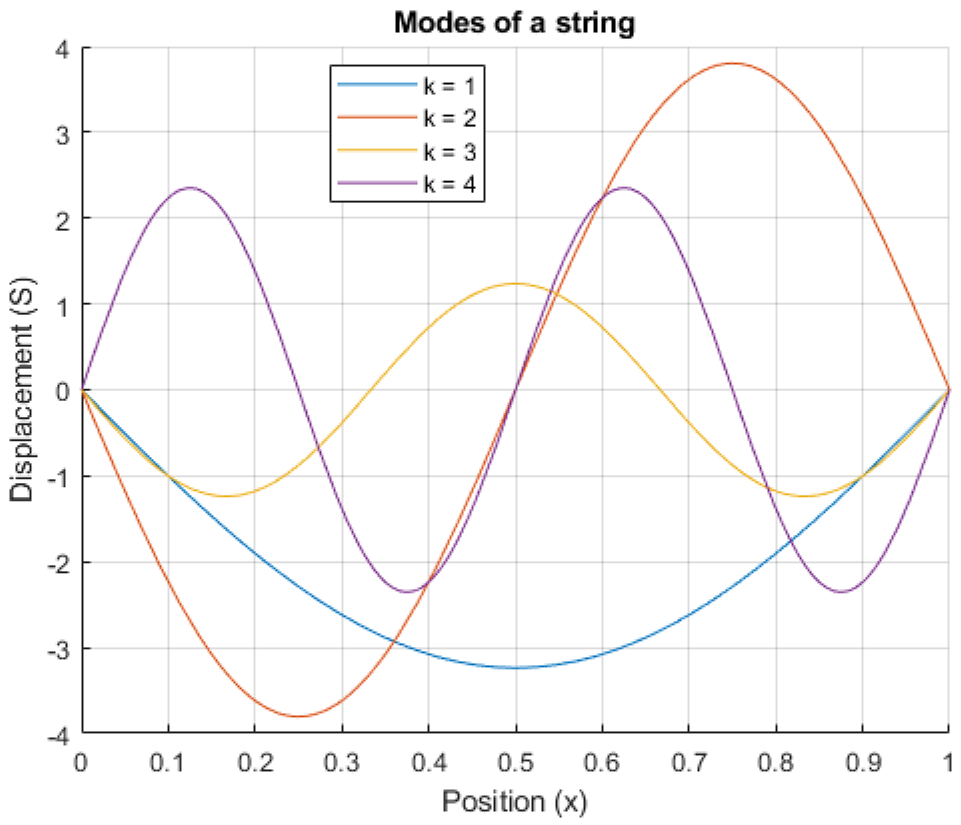
```
S = subs(S,[p1 p2],[1 1]);
S = simplify(S,'Steps',50)
```

```
S(x, t) =
```

$$-4\sin(\pi k t)\sin(\pi k x)$$

The string has different modes of vibration for different values of `k`. Plot the first four modes for an arbitrary value of time `t`. Use the `param` argument returned by `solve` to address parameter `k`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
Splot(x) = S(x,0.3);
figure(1)
hold on
grid on
ymin = double(coeffs(Splot));
for i = 1:4
    yplot = subs(Splot,param,i);
    fplot(yplot,[0 Length])
end
ylim([ymin -ymin])
legend('k = 1','k = 2','k = 3','k = 4','Location','best')
xlabel('Position (x)')
ylabel('Displacement (S)')
title('Modes of a string')
```



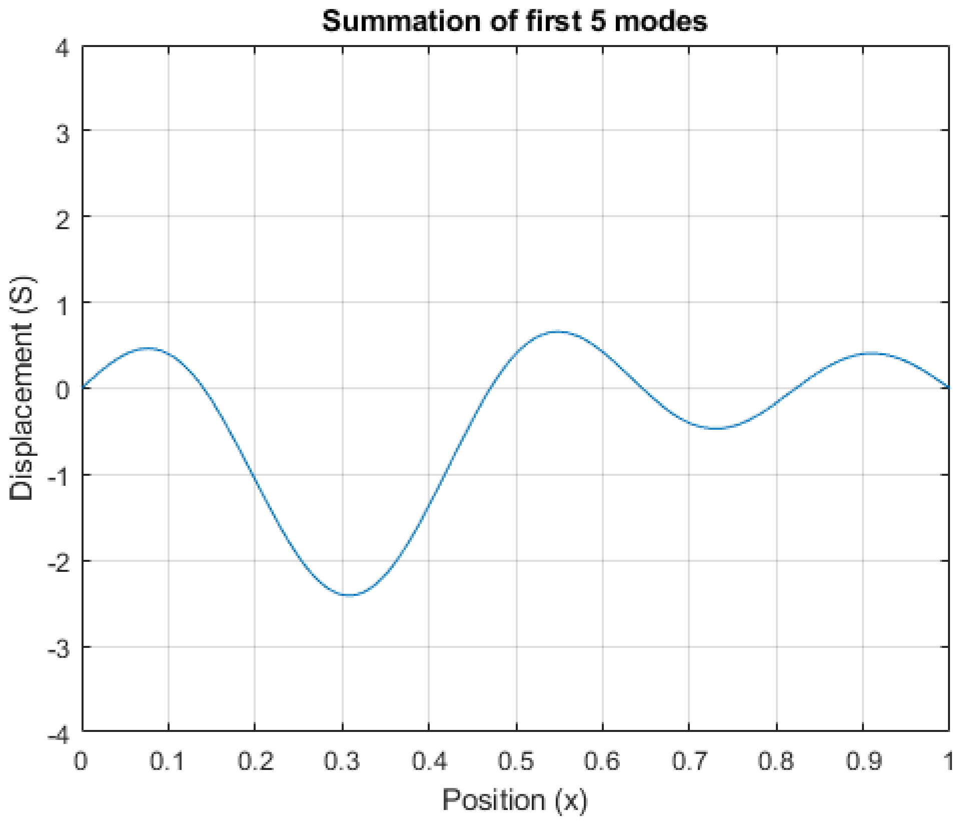
The wave equation is linear. This means that any linear combination of the allowed modes is a valid solution to the wave equation. Hence, the full solution to the wave equation with the given boundary conditions and initial values is a sum over allowed modes

$$F(x,t) = \sum_{k=n}^{m} A_k \sin(\pi k t) \sin(\pi k x),$$

where A_k denotes arbitrary constants.

Use `symsum` to sum the first five modes of the string. On a new figure, display the resulting waveform at the same instant of time as the previous waveforms for comparison.

```
figure(2)
S5(x) = 1/5*symsum(S,param,1,5);
fplot(subs(S5,t,0.3),[0 Length])
ylim([ymin -ymin])
grid on
xlabel('Position (x)')
ylabel('Displacement (S)')
title('Summation of first 5 modes')
```



The figure shows that summing modes allows you to model a qualitatively different waveform. Here, we specified the initial condition is $S(x, t = 0) = 0$ for all x.

You can calculate the values A_k in the equation $F(x,t) = \sum_{k=n}^{m} A_k \sin(\pi kt) \sin(\pi kx)$ by specifying a condition for initial velocity

$u_t(x, t = 0) = F_t(x, 0)$.

The appropriate summation of modes can represent any waveform, which is the same as using the Fourier series to represent the string's motion.

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

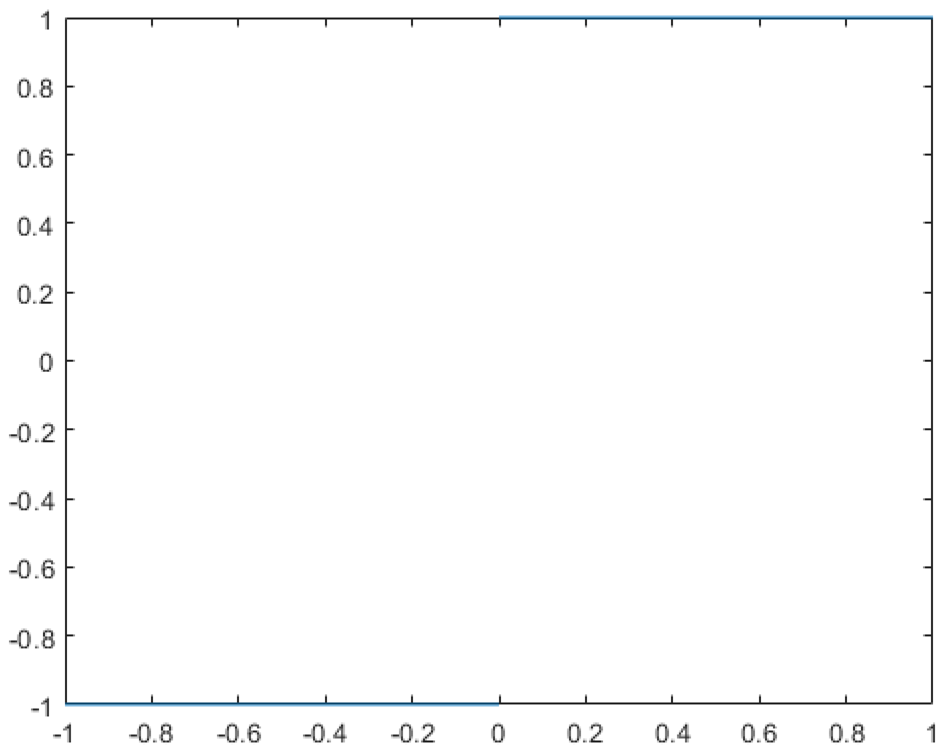
```
ans =
exp(x)
```

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. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(x/abs(x), [-1 1], 'ShowPoles', 'off')
```



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`

`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
```

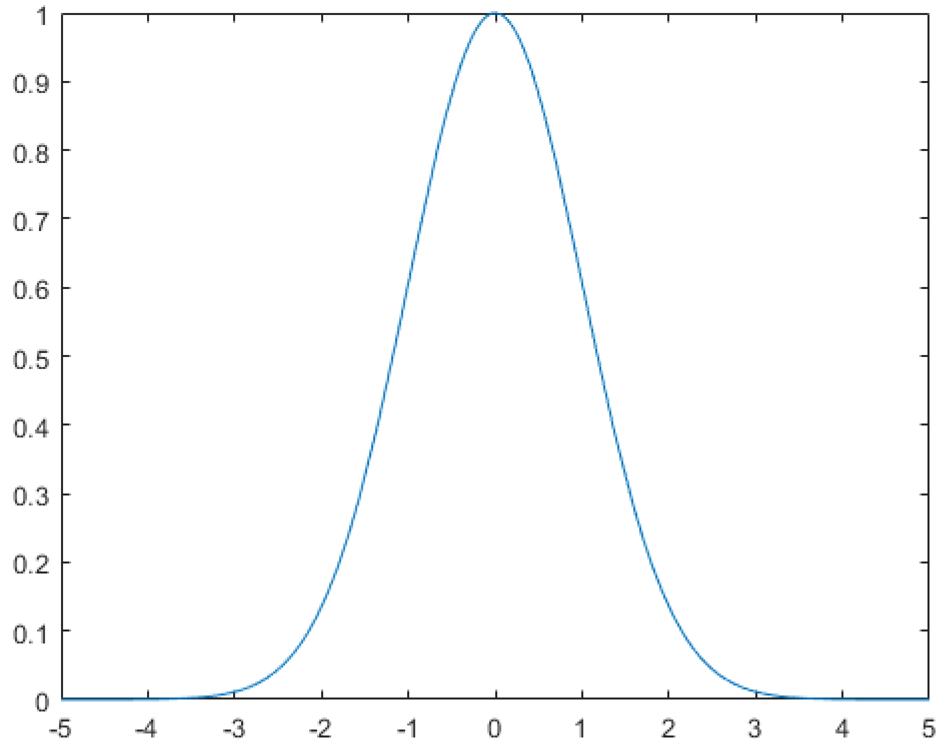
Integration with Real Parameters

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 $a = 1/2$. Prior to R2016a, use `ezplot` instead of `fplot`.

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

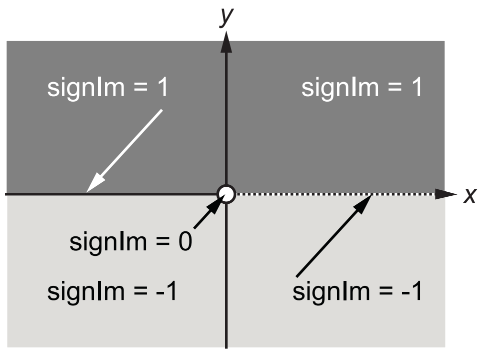
`syms` is used with the `clear` option to clear the all assumptions on `a`. For more information about symbolic variables and assumptions on them, see “Delete Symbolic Objects and Their Assumptions” on page 1-29.

The preceding commands produce the complex output

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

The function `signIm` is defined as:

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



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

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

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

double(g)

ans =
   1.5708 - 1.5708i
```

High-Precision Numerical Integration Using Variable-Precision Arithmetic

High-precision numerical integration is implemented in the `vpaintegral` function of the Symbolic Math Toolbox. `vpaintegral` uses variable-precision arithmetic in contrast to the MATLAB `integral` function, which uses double-precision arithmetic.

Integrate `besseli(5,25*u).*exp(-u*25)` by using both `integral` and `vpaintegral`. The `integral` function returns `NaN` and issues a warning while `vpaintegral` returns the correct result.

```
syms u
f = besseli(5,25*x).*exp(-x*25);
fun = @(u)besseli(5,25*u).*exp(-u*25);

usingIntegral = integral(fun, 0, 30)
usingVpaintegral = vpaintegral(f, 0, 30)
```

```
Warning: Infinite or Not-a-Number value encountered.
usingIntegral =
   NaN

usingVpaintegral =
0.688424
```

For more information, see `vpaintegral`.

Symbolic Summation

Symbolic Math Toolbox provides two functions for calculating sums:

- `sum` finds the sum of elements of symbolic vectors and matrices. Unlike the MATLAB `sum`, the symbolic `sum` function does not work on multidimensional arrays. For details, follow the MATLAB `sum` page.
- `symsum` finds the sum of a symbolic series.

In this section...
"Comparing symsum and sum" on page 2-64
"Computational Speed of symsum versus sum" on page 2-65
"Output Format Differences Between symsum and sum" on page 2-65

Comparing symsum and sum

You can find definite sums by using both `sum` and `symsum`. The `sum` function sums the input over a dimension, while the `symsum` function sums the input over an index.

Consider the definite sum $S = \sum_{k=1}^{10} \frac{1}{k^2}.$ First, find the terms of the definite sum by substituting the index values for `k` in the expression. Then, sum the resulting vector using `sum`.

```
syms k
f = 1/k^2;
V = subs(f, k, 1:10)
S_sum = sum(V)
```

```
V =
[ 1, 1/4, 1/9, 1/16, 1/25, 1/36, 1/49, 1/64, 1/81, 1/100]
S_sum =
1968329/1270080
```

Find the same sum by using `symsum` by specifying the index and the summation limits. `sum` and `symsum` return identical results.

```
S_symsum = symsum(f, k, 1, 10)
S_symsum =
1968329/1270080
```

Computational Speed of symsum versus sum

For summing definite series, `symsum` can be faster than `sum`. For summing an indefinite series, you can only use `symsum`.

You can demonstrate that `symsum` can be faster than `sum` by summing a large definite series such as $S = \sum_{k=1}^{100000} k^2.$

To compare runtimes on your computer, use the following commands.

```
syms k
tic
sum(sym(1:100000).^2);
toc
tic
symsum(k^2, k, 1, 100000);
toc
```

Output Format Differences Between symsum and sum

`symsum` can provide a more elegant representation of sums than `sum` provides. Demonstrate this difference by comparing the function outputs for the definite series $S = \sum_{k=1}^{10} x^k.$ To simplify the solution, assume `x > 1`.

```
syms x
assume(x > 1)
S_sum = sum(x.^(1:10))
S_symsum = symsum(x^k, k, 1, 10)

S_sum =
x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x
S_symsum =
x^11/(x - 1) - x/(x - 1)
```

Show that the outputs are equal by using `isAlways`. The `isAlways` function returns logical `1` (`true`), meaning that the outputs are equal.

```
isAlways(S_sum == S_symsum)
```

```
ans =
  logical
     1
```

For further computations, clear the assumptions.

```
assume(x, 'clear')
```

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

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
```

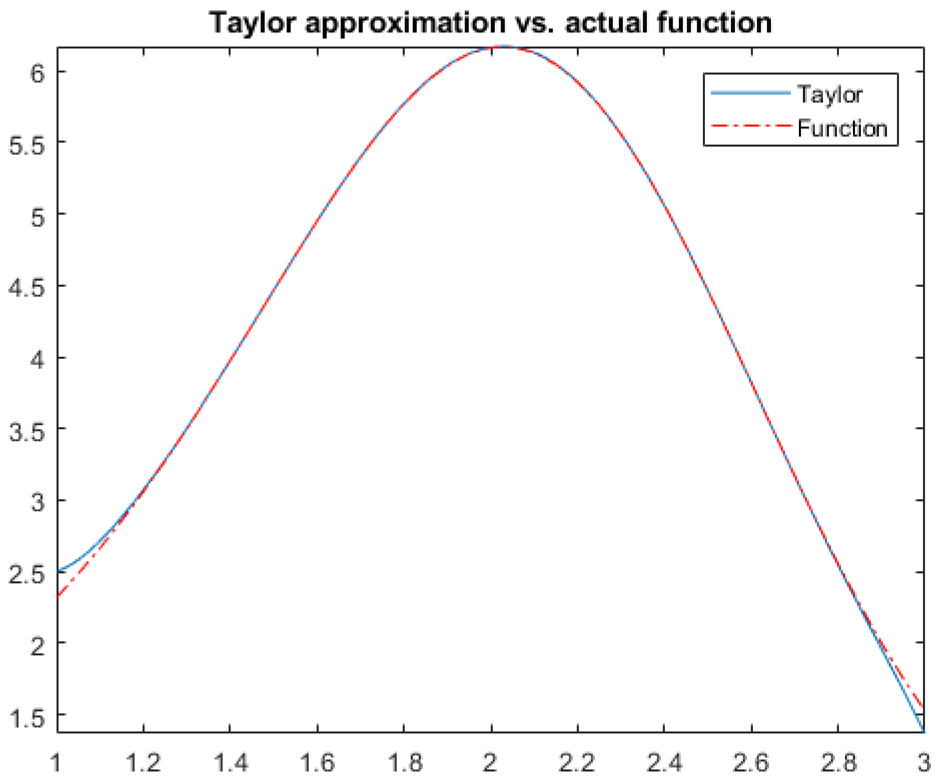
to find that `t` has about 100,000 characters in its printed form. In order to proceed with using `t`, first simplify its presentation:

```
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);
fplot(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.

Padé Approximant

The Padé approximant of order $[m, n]$ approximates the function $f(x)$ around $x = x_0$ as

$$\frac{a_0 + a_1(x - x_0) + \ldots + a_m(x - x_0)^m}{1 + b_1(x - x_0) + \ldots + b_n(x - x_0)^n}.$$

The Padé approximant is a rational function formed by a ratio of two power series. Because it is a rational function, it is more accurate than the Taylor series in approximating functions with poles. The Padé approximant is represented by the Symbolic Math Toolbox function `pade`.

When a pole or zero exists at the expansion point $x = x_0$, the accuracy of the Padé approximant decreases. To increase accuracy, an alternative form of the Padé approximant can be used which is

$$\frac{(x - x_0)^p \left(a_0 + a_1(x - x_0) + \ldots + a_m(x - x_0)^m\right)}{1 + b_1(x - x_0) + \ldots + b_n(x - x_0)^n}.$$

The `pade` function returns the alternative form of the Padé approximant when you set the `OrderMode` input argument to `Relative`.

The Padé approximant is used in control system theory to model time delays in the response of the system. Time delays arise in systems such as chemical and transport processes where there is a delay between the input and the system response. When these inputs are modeled, they are called dead-time inputs. This example shows how to use the Symbolic Math Toolbox to model the response of a first-order system to dead-time inputs using Padé approximants.

The behavior of a first-order system is described by this differential equation

$$\tau \frac{dy(t)}{dt} + y(t) = ax(t).$$

Enter the differential equation in MATLAB.

```
syms tau a x(t) y(t) xS(s) yS(s) H(s) tmp
F = tau*diff(y)+y == a*x;
```

Find the Laplace transform of `F` using `laplace`.

```
F = laplace(F,t,s)
```

```
F =
```

$$\mathrm{laplace}(y(t),t,s) - \tau\ (y(0) - s\,\mathrm{laplace}(y(t),t,s)) = a\,\mathrm{laplace}(x(t),t,s)$$

Assume the response of the system at `t = 0` is `0`. Use `subs` to substitute for `y(0) = 0`.

```
F = subs(F,y(0),0)
```

```
F =
```

$$\mathrm{laplace}(y(t),t,s) + s\,\tau\,\mathrm{laplace}(y(t),t,s) = a\,\mathrm{laplace}(x(t),t,s)$$

To collect common terms, use `simplify`.

```
F = simplify(F)
```

```
F =
```

$$(s\,\tau + 1)\ \mathrm{laplace}(y(t),t,s) = a\,\mathrm{laplace}(x(t),t,s)$$

For readability, replace the Laplace transforms of `x(t)` and `y(t)` with `xS(s)` and `yS(s)`.

```
F = subs(F,[laplace(x(t),t,s) laplace(y(t),t,s)],[xS(s) yS(s)])
```

```
F =
```

$$\mathrm{yS}(s)\ (s\,\tau + 1) = a\,\mathrm{xS}(s)$$

The Laplace transform of the transfer function is `yS(s)/xS(s)`. Divide both sides of the equation by `xS(s)` and use `subs` to replace `yS(s)/xS(s)` with `H(s)`.

```
F = F/xS(s);
F = subs(F,yS(s)/xS(s),H(s))
```

```
F =
```

$$H(s)\ (s\,\tau + 1) = a$$

Solve the equation for `H(s)`. Substitute for `H(s)` with a dummy variable, solve for the dummy variable using `solve`, and assign the solution back to `H(s)`.

```
F = subs(F,H(s),tmp);
H(s) = solve(F,tmp)
```

```
H(s) =
```

$$\frac{a}{s\,\tau+1}$$

The input to the first-order system is a time-delayed step input. To represent a step input, use `heaviside`. Delay the input by three time units. Find the Laplace transform using `laplace`.

```
step = heaviside(t - 3);
step = laplace(step)
```

```
step =
```

$$\frac{\mathrm{e}^{-3\,s}}{s}$$

Find the response of the system, which is the product of the transfer function and the input.

```
y = H(s)*step
```

```
y =
```

$$\frac{a\,\mathrm{e}^{-3\,s}}{s\,(s\,\tau+1)}$$

To allow plotting of the response, set parameters `a` and `tau` to their values. For `a` and `tau`, choose values `1` and `3`, respectively.

```
y = subs(y,[a tau],[1 3]);
y = ilaplace(y,s);
```

Find the Padé approximant of order `[2 2]` of the step input using the `Order` input argument to `pade`.

```
stepPade22 = pade(step,'Order',[2 2])
```

```
stepPade22 =
```

$$\frac{3\,s^2-4\,s+2}{2\,s\,(s+1)}$$

Find the response to the input by multiplying the transfer function and the Padé approximant of the input.

```
yPade22 = H(s)*stepPade22
```

```
yPade22 =
```

$$\frac{a\,(3\,s^2-4\,s+2)}{2\,s\,(s\,\tau+1)\,(s+1)}$$

Find the inverse Laplace transform of `yPade22` using `ilaplace`.

```
yPade22 = ilaplace(yPade22,s)
```

```
yPade22 =
```

$$a+\frac{9\,a\,e^{-s}}{2\,\tau-2}-\frac{a\,e^{-\frac{s}{\tau}}\,(2\,\tau^2+4\,\tau+3)}{\tau\,(2\,\tau-2)}$$

To plot the response, set parameters `a` and `tau` to their values of `1` and `3`, respectively.

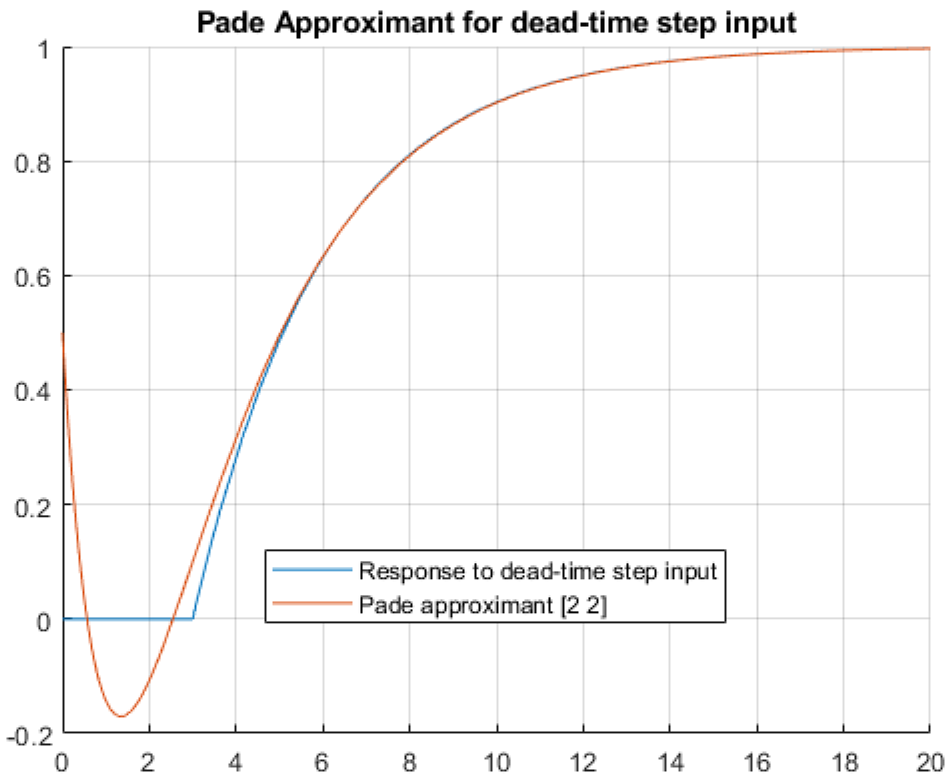
```
yPade22 = subs(yPade22,[a tau],[1 3])
```

```
yPade22 =
```

$$\frac{9\,e^{-s}}{4}-\frac{11\,e^{-\frac{s}{3}}}{4}+1$$

Plot the response of the system `y` and the response calculated from the Padé approximant `yPade22`.

```
hold on
grid on
fplot([y yPade22],[0 20])
title('Pade Approximant for dead-time step input')
legend('Response to dead-time step input',...
       'Pade approximant [2 2]',...
       'Location', 'Best')
```



The `[2 2]` Padé approximant does not represent the response well because a pole exists at the expansion point of `0`. To increase the accuracy of `pade` when there is a pole or zero at the expansion point, set the `OrderMode` input argument to `Relative` and repeat the steps. For details, see `pade`.

```
stepPade22Rel = pade(step,'Order',[2 2],'OrderMode','Relative')
```

```
stepPade22Rel =
```

$$\frac{3\,s^2 - 6\,s + 4}{s\,(3\,s^2 + 6\,s + 4)}$$

```
yPade22Rel = H(s)*stepPade22Rel
```

```
yPade22Rel =
```

$$\frac{a\,(3\,s^2-6\,s+4)}{s\,(s\,\tau+1)\,(3\,s^2+6\,s+4)}$$

```
yPade22Rel = ilaplace(yPade22Rel)
```

```
yPade22Rel =
```

$$a-\frac{a\,\mathrm{e}^{-\frac{t}{\tau}}\,(4\,\tau^2+6\,\tau+3)}{\sigma_1}+\frac{12\,a\,\tau\,\mathrm{e}^{-t}\,\left(\cos\left(\frac{\sqrt{3}\,t}{3}\right)-\sqrt{3}\,\sin\left(\frac{\sqrt{3}\,t}{3}\right)\,\left(\frac{36\,a-72\,a\,\tau}{36\,a\,\tau}+1\right)\right)}{\sigma_1}$$

where

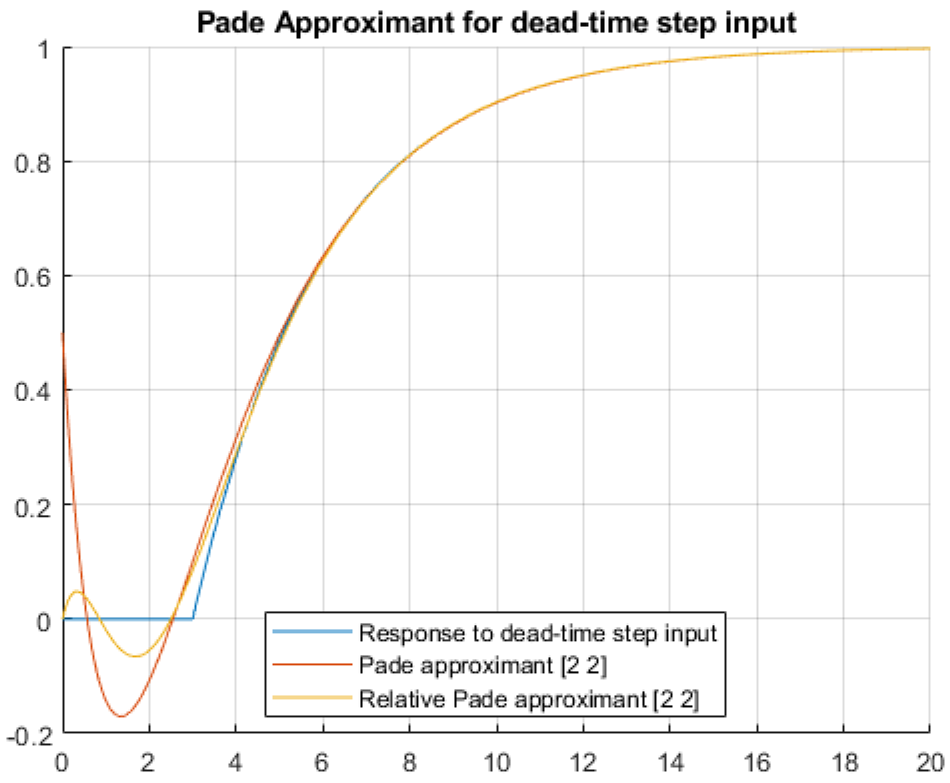
$$\sigma_1=4\,\tau^2-6\,\tau+3$$

```
yPade22Rel = subs(yPade22Rel,[a tau],[1 3])
```

```
yPade22Rel =
```

$$\frac{12\,\mathrm{e}^{-t}\,\left(\cos\left(\frac{\sqrt{3}\,t}{3}\right)+\frac{2\,\sqrt{3}\,\sin\left(\frac{\sqrt{3}\,t}{3}\right)}{3}\right)}{7}-\frac{19\,\mathrm{e}^{-\frac{t}{3}}}{7}+1$$

```
fplot(yPade22Rel,[0 20],'DisplayName','Relative Pade approximant [2 2]')
```



The accuracy of the Padé approximant can also be increased by increasing its order. Increase the order to `[4 5]` and repeat the steps. The `[n-1 n]` Padé approximant is better at approximating the response at `t = 0` than the `[n n]` Padé approximant.

```
stepPade45 = pade(step,'Order',[4 5])
```

```
stepPade45 =
```

$$\frac{27\,s^4 - 180\,s^3 + 540\,s^2 - 840\,s + 560}{s\,\left(27\,s^4 + 180\,s^3 + 540\,s^2 + 840\,s + 560\right)}$$

```
yPade45 = H(s)*stepPade45
```

```
yPade45 =
```

$$\frac{a\ (27\ s^4 - 180\ s^3 + 540\ s^2 - 840\ s + 560)}{s\ (s\ \tau + 1)\ (27\ s^4 + 180\ s^3 + 540\ s^2 + 840\ s + 560)}$$

```
yPade45 = subs(yPade45,[a tau],[1 3])
```

```
yPade45 =
```

$$\frac{27\ s^4 - 180\ s^3 + 540\ s^2 - 840\ s + 560}{s\ (3\ s + 1)\ (27\ s^4 + 180\ s^3 + 540\ s^2 + 840\ s + 560)}$$

```
yPade45 = ilaplace(yPade45)
```

```
yPade45 =
```

$$\frac{294120\ \left(\sum_{k=1}^{4} \frac{e^{\sigma_2 t}\ \sigma_2^2}{\sigma_1}\right)}{1001} - \frac{2721\ e^{-\frac{t}{3}}}{1001} + \frac{46440\ \left(\sum_{k=1}^{4} \frac{e^{\sigma_2 t}\ \sigma_2^3}{\sigma_1}\right)}{1001} + \frac{172560\ \left(\sum_{k=1}^{4} \frac{e^{\sigma_2 t}}{\sigma_1}\right)}{143} + \frac{101520\ \left(\sum_{k=1}^{4} \frac{\sigma_2\ e^{\sigma_2 t}}{12\ (90\ \sigma_2 + 45\ \sigma_2^2 + 9\ \sigma_2^3 + 70)}\right)}{143} + 1$$

where

$$\sigma_1 = 12\ (45\ \sigma_2^2 + 9\ \sigma_2^3 + 90\ \sigma_2 + 70)$$

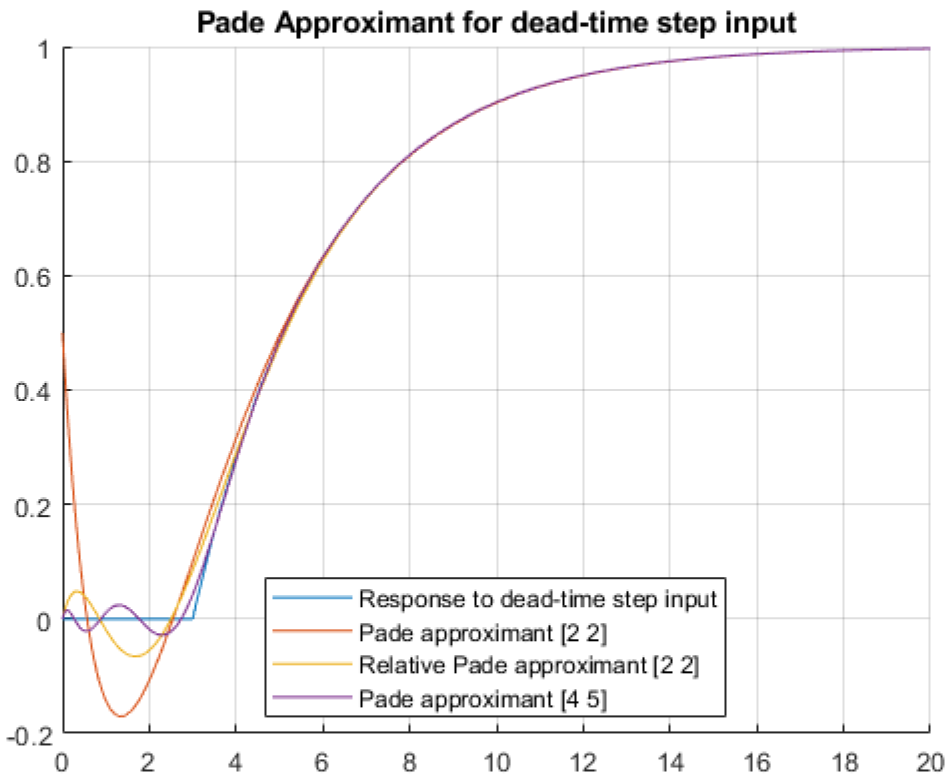
$$\sigma_2 = \mathrm{root}\left(s_5^4 + \frac{20\ s_5^3}{3} + 20\ s_5^2 + \frac{280\ s_5}{9} + \frac{560}{27}, s_5, k\right)$$

```
yPade45 = vpa(yPade45)
```

```
yPade45 =
```

$$3.2418384981662546679005910164486\ e^{-1.9308070685469147789295959950184\ t} \cos(0.57815608595633583454598214328008\ t) - 2.7182817182817182817182817182817\ e^{-0.33333333333333333333333333333333\ t} - 1.5235567798845363861823092981669\ e^{-1.4025262647864185544037373831494\ t} \cos(1.7716120279045018112388813990878\ t) + 1$$

```
fplot(yPade45,[0 20],'DisplayName','Pade approximant [4 5]')
```



The following points have been shown:

- Padé approximants can model dead-time step inputs.
- The accuracy of the Padé approximant increases with the increase in the order of the approximant.
- When a pole or zero exists at the expansion point, the Padé approximant is inaccurate about the expansion point. To increase the accuracy of the approximant, set the `OrderMode` option to `Relative`. You can also use increase the order of the denominator relative to the numerator.

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

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

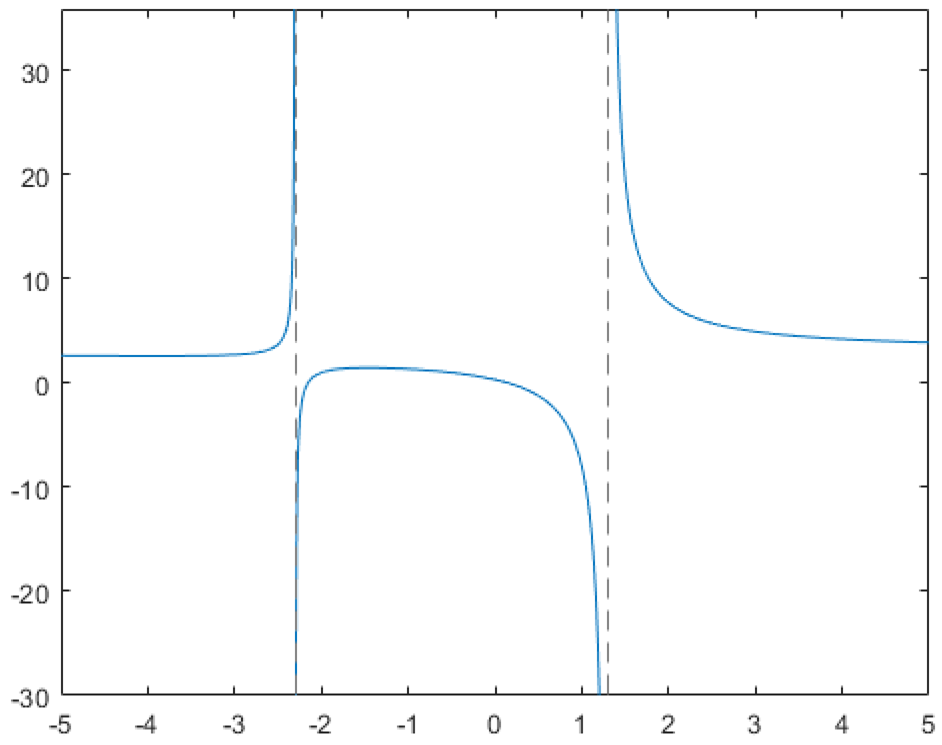
To create the function, enter the following commands:

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

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

Plot the function `f` by using `fplot`. The `fplot` function automatically shows horizontal asymptotes. Prior to R2016a, use `ezplot` instead of `fplot`.

```
fplot(f)
```



Find Asymptotes

To mathematically find the horizontal asymptote 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 result means the line $y = 3$ is a horizontal asymptote to `f`.

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 the 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

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

Find Maximum and Minimum

You can see from the graph that f has a local maximum 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

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

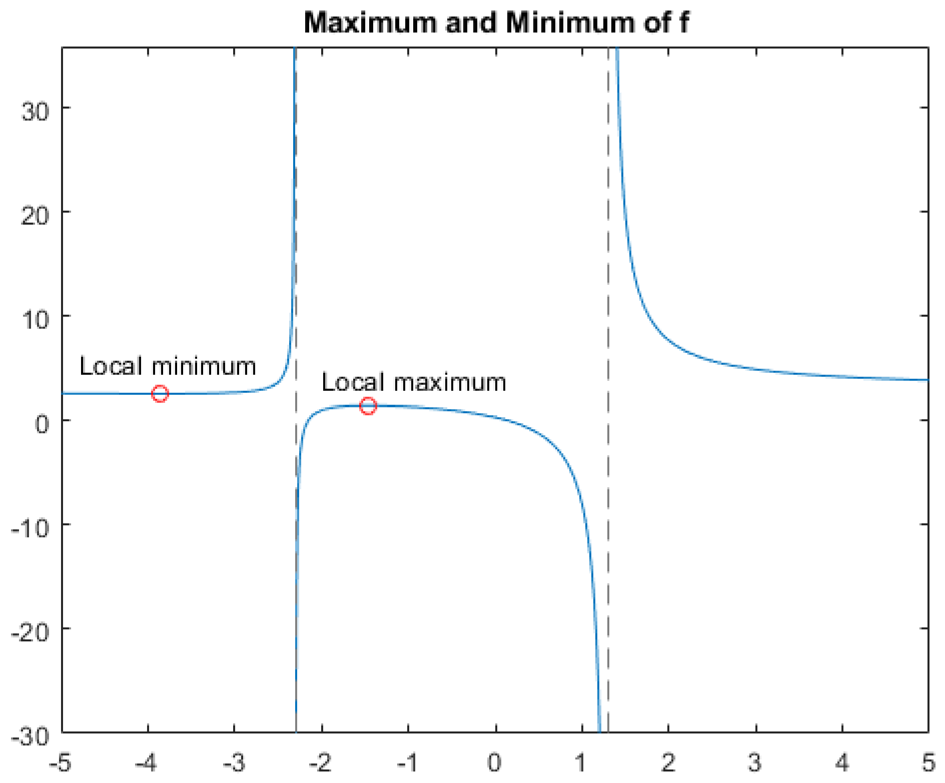
and a local maximum at

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

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:

```
fplot(f)
hold on
plot(double(crit_pts), double(subs(f,crit_pts)),'ro')
title('Maximum and Minimum of f')
text(-4.8,5.5,'Local minimum')
text(-2,4,'Local maximum')
hold off
```



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,'MaxDegree',3);
double(inflec_pt)
```

This returns

```
ans =
  -5.2635 + 0.0000i
  -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);
```

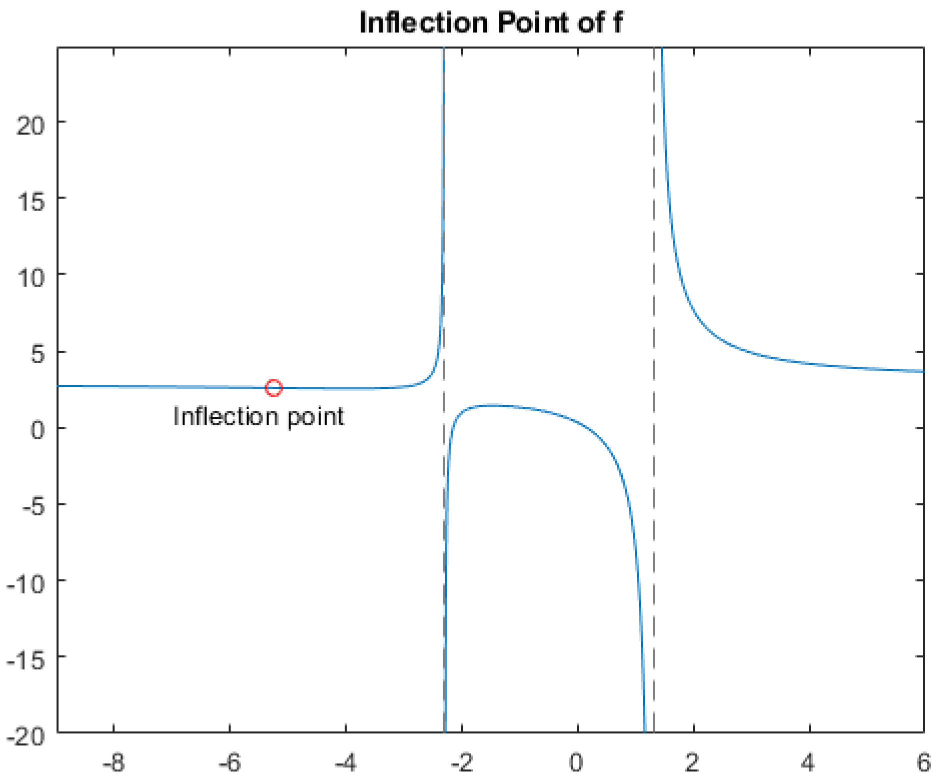
To see the symbolic expression for the inflection point, enter

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

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

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

```
fplot(f, [-9 6])
hold on
plot(double(inflec_pt), double(subs(f,inflec_pt)),'ro')
title('Inflection Point of f')
text(-7,1,'Inflection point')
hold off
```


Inflection Point of f
Inflection point
20
15
10
5
0
-5
-10
-15
-20
-8
-6
-4
-2
0
2
4
6

Simplify Symbolic Expressions

Simplification of a 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 turns out to be complicated or even unsuitable for another problem. For example, the following two mathematical expressions present the same polynomial in different forms:

$(x + 1)(x - 2)(x + 3)(x - 4)$,

$x^4 - 2x^3 - 13x^2 + 14x + 24$.

The first form clearly shows the roots of this polynomial. This form is simpler for working with the roots. The second form serves best when you want to see the coefficients of the polynomial. For example, this form is convenient when you differentiate or integrate polynomials.

If the problem you want to solve requires a particular form of an expression, the best approach is to choose the appropriate simplification function. See "Choose Function to Rearrange Expression" on page 2-94.

Besides specific simplifiers, Symbolic Math Toolbox offers a general simplifier, `simplify`.

If you do not need a particular form of expressions (expanded, factored, or expressed in particular terms), use `simplify` to shorten mathematical expressions. For example, use this simplifier to find a shorter form for a final result of your computations.

`simplify` works on various types of symbolic expressions, such as polynomials, expressions with trigonometric, logarithmic, and special functions. For example, simplify these polynomials.

```
syms x y
simplify((1 - x^2)/(1 - x))
simplify((x - 1)*(x + 1)*(x^2 + x + 1)*(x^2 + 1)*(x^2 - x + 1)*(x^4 - x^2 + 1))
```

```
ans =
x + 1

ans =
x^12 - 1
```

Simplify expressions involving trigonometric functions.

```
simplify(cos(x)^(-2) - tan(x)^2)
simplify(cos(x)^2 - sin(x)^2)

ans =
1

ans =
cos(2*x)
```

Simplify expressions involving exponents and logarithms. In the third expression, use `log(sym(3))` instead of `log(3)`. If you use `log(3)`, then MATLAB calculates `log(3)` with the double precision, and then converts the result to a symbolic number.

```
simplify(exp(x)*exp(y))
simplify(exp(x) - exp(x/2)^2)
simplify(log(x) + log(sym(3)) - log(3*x) + (exp(x) - 1)/(exp(x/2) + 1))

ans =
exp(x + y)

ans =
0

ans =
exp(x/2) - 1
```

Simplify expressions involving special functions.

```
simplify(gamma(x + 1) - x*gamma(x))
simplify(besselj(2, x) + besselj(0, x))

ans =
0

ans =
(2*besselj(1, x))/x
```

You also can simplify symbolic functions by using `simplify`.

```
syms f(x,y)
f(x,y) = exp(x)*exp(y)
f = simplify(f)

f(x, y) =
exp(x)*exp(y)
```

```
f(x, y) =
exp(x + y)
```

Simplify Using Options

By default, `simplify` uses strict simplification rules and ensures that simplified expressions are always mathematically equivalent to initial expressions. For example, it does not combine logarithms.

```
syms x
simplify(log(x^2) + log(x))
```

```
ans =
log(x^2) + log(x)
```

You can apply additional simplification rules which are not correct for all values of parameters and all cases, but using which `simplify` can return shorter results. For this approach, use `IgnoreAnalyticConstraints`. For example, simplifying the same expression with `IgnoreAnalyticConstraints`, you get the result with combined logarithms.

```
simplify(log(x^2) + log(x),'IgnoreAnalyticConstraints',true)
```

```
ans =
3*log(x)
```

`IgnoreAnalyticConstraints` provides a shortcut allowing you to simplify expressions under commonly used assumptions about values of the variables. Alternatively, you can set appropriate assumptions on variables explicitly. For example, combining logarithms is not valid for complex values in general. If you assume that `x` is a real value, `simplify` combines logarithms without `IgnoreAnalyticConstraints`.

```
assume(x,'real')
simplify(log(x^2) + log(x))
```

```
ans =
log(x^3)
```

For further computations, clear the assumption on `x`.

```
syms x clear
```

Another approach that can improve simplification of an expression or function is the syntax `simplify(f,'Steps',n)`, where `n` is a positive integer that controls how many steps `simplify` takes. Specifying more simplification steps can help you simplify the expression better, but it takes more time. By default, `n = 1`. For example, create and simplify this expression. The result is shorter than the original expression, but it can be simplified further.

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

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

Specify the number of simplification steps for the same expression. First, use 25 steps.

```
simplify(y,'Steps',25)
```

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

Use 50 steps to simplify the expression even further.

```
simplify(y,'Steps',50)
```

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

Suppose, you already simplified an expression or function, but want to simplify it further. The more efficient approach is to simplify the result instead of simplifying the original expression.

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

```
y =
(sin(4*x)*(exp(x) - 1))/(2*cos(4*x)*(exp(x) + 1))
```

```
y = simplify(y,'Steps',25)
```

```
y =
(tan(4*x)*(exp(x) - 1))/(2*(exp(x) + 1))
```

```
y = simplify(y,'Steps',50)

y =
(tan(4*x)*tanh(x/2))/2
```

Simplify Using Assumptions

Some expressions cannot be simplified in general, but become much shorter under particular assumptions. For example, simplifying this trigonometric expression without additional assumptions returns the original expression.

```
syms n
simplify(sin(2*n*pi))

ans =
sin(2*pi*n)
```

However, if you assume that variable `n` represents an integer, the same trigonometric expression simplifies to 0.

```
assume(n,'integer')
simplify(sin(2*n*pi))

ans =
0
```

For further computations, clear the assumption.

```
syms n clear
```

Simplify Fractions

You can use the general simplification function, `simplify`, to simplify fractions. However, Symbolic Math Toolbox offers a more efficient function specifically for this task: `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. For example, simplify these expressions.

```
syms x y
simplifyFraction((x^3 - 1)/(x - 1))

ans =
x^2 + x + 1
```

```
simplifyFraction((x^3 - x^2*y - x*y^2 + y^3)/(x^3 + y^3))
```

```
ans =
(x^2 - 2*x*y + y^2)/(x^2 - x*y + y^2)
```

By default, `simplifyFraction` does not expand expressions in the numerator and denominator of the returned result. To expand the numerator and denominator in the resulting expression, use the `Expand` option. For comparison, first simplify this fraction without `Expand`.

```
simplifyFraction((1 - exp(x)^4)/(1 + exp(x))^4)
```

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

Now, simplify the same expressions with `Expand`.

```
simplifyFraction((1 - exp(x)^4)/(1 + exp(x))^4,'Expand',true)
```

```
ans =
(exp(2*x) - exp(3*x) - exp(x) + 1)/(3*exp(2*x) + exp(3*x) + 3*exp(x) + 1)
```

Abbreviate Common Terms in Long Expressions

Often, long expressions contain several instances of the same subexpression. Such expressions look shorter if you replace the subexpression with an abbreviation. For example, solve this equation.

```
syms x
s = solve(sqrt(x) + 1/x == 1, x)
```

```
s =
 (1/(18*(25/54 - (23^(1/2)*108^(1/2))/108)^(1/3)) -...
 (3^(1/2)*(1/(9*(25/54 - (23^(1/2)*108^(1/2))/108)^(1/3)) -...
 (25/54 - (23^(1/2)*108^(1/2))/108)^(1/3))*1i)/2 +...
 (25/54 - (23^(1/2)*108^(1/2))/108)^(1/3)/2 + 1/3)^2
 ...
 ((3^(1/2)*(1/(9*(25/54 - (23^(1/2)*108^(1/2))/108)^(1/3)) -...
 (25/54 - (23^(1/2)*108^(1/2))/108)^(1/3))*1i)/2 + 1/(18*(25/54 -...
 (23^(1/2)*108^(1/2))/108)^(1/3)) +...
 (25/54 - (23^(1/2)*108^(1/2))/108)^(1/3)/2 + 1/3)^2
```

The returned result is a long expression that might be difficult to parse. To represent it in a more familiar typeset form, use `pretty`. When displaying results, the `pretty` function can use abbreviations to shorten long expressions.

```
pretty(s)
```

```
/ /   1           #2   1 \2 \
| | ----- - #1 + -- + - |   |
| \ 18 #2         2   3 /   |
|                           |
| /        1      #2   1 \2 |
| | #1 + ----- + -- + - |   |
\ \      18 #2    2   3 /   /

where

                   /   1       \
          sqrt(3) | ---- - #2 | 1i
                   \ 9 #2      /
   #1 == ------------------------
                       2

          / 25   sqrt(23) sqrt(108) \1/3
   #2 == | -- - ----------------- |
         \ 54           108       /
```

`pretty` uses an internal algorithm to choose which subexpressions to abbreviate. It also can use nested abbreviations. For example, the term `#1` contains the subexpression abbreviated as `#2`. This function does not provide any options to enable, disable, or control abbreviations.

`subexpr` is another function that you can use to shorten long expressions. This function abbreviates only one common subexpression and, unlike `pretty`, it does not support nested abbreviations. It also does not let you choose which subexpressions to replace.

Use the second input argument of `subexpr` to specify the variable name that replaces the common subexpression. For example, replace the common subexpression in `s` by variable `t`.

```
[s1,t] = subexpr(s,'t')
```

```
s1 =
 (1/(18*t^(1/3)) - (3^(1/2)*(1/(9*t^(1/3)) -...
 t^(1/3))*1i)/2 + t^(1/3)/2 + 1/3)^2
 ...
 ((3^(1/2)*(1/(9*t^(1/3)) - t^(1/3))*1i)/2 +...
 1/(18*t^(1/3)) + t^(1/3)/2 + 1/3)^2

t =
25/54 - (23^(1/2)*108^(1/2))/108
```

For the syntax with one input argument, `subexpr` uses variable `sigma` to abbreviate the common subexpression. Output arguments do not affect the choice of abbreviation variable.

```
[s2,sigma] = subexpr(s)
```

```
s2 =
 (1/(18*sigma^(1/3)) - (3^(1/2)*(1/(9*sigma^(1/3)) -...
 sigma^(1/3))*1i)/2 + sigma^(1/3)/2 + 1/3)^2
 ...
 ((3^(1/2)*(1/(9*sigma^(1/3)) - sigma^(1/3))*1i)/2 +...
 1/(18*sigma^(1/3)) + sigma^(1/3)/2 + 1/3)^2

sigma =
25/54 - (23^(1/2)*108^(1/2))/108
```

Choose Function to Rearrange Expression

Type of Transformation	Function
"Combine Terms of Same Algebraic Structures" on page 2-94	`combine`
"Expand Expressions" on page 2-96	`expand`
"Factor Expressions" on page 2-97	`factor`
"Extract Subexpressions from Expression" on page 2-99	`children`
"Collect Terms with Same Powers" on page 2-100	`collect`
"Rewrite Expressions in Terms of Other Functions" on page 2-101	`rewrite`
"Compute Partial Fraction Decompositions of Expressions" on page 2-102	`partfrac`
"Compute Normal Forms of Rational Expressions" on page 2-103	`simplifyFraction`
"Represent Polynomials Using Horner Nested Forms" on page 2-103	`horner`

Combine Terms of Same Algebraic Structures

Symbolic Math Toolbox provides the `combine` function for combining subexpressions of an original expression. The `combine` function uses mathematical identities for the functions you specify. For example, combine the trigonometric expression.

```
syms x y
combine(2*sin(x)*cos(x),'sincos')
```

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

If you do not specify a target function, `combine` uses the identities for powers wherever these identities are valid:

- $a^b\,a^c = a^{b+c}$
- $a^c\,b^c = (a\,b)^c$

- $(a^b)^c = a^{bc}$

For example, by default the function combines the following square roots.

```
combine(sqrt(2)*sqrt(x))

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

The function does not combine these square roots because the identity is not valid for negative values of variables.

```
combine(sqrt(x)*sqrt(y))

ans =
x^(1/2)*y^(1/2)
```

To combine these square roots, use the `IgnoreAnalyticConstraints` option.

```
combine(sqrt(x)*sqrt(y),'IgnoreanalyticConstraints',true)

ans =
(x*y)^(1/2)
```

`IgnoreAnalyticConstraints` provides a shortcut allowing you to combine expressions under commonly used assumptions about values of the variables. Alternatively, you can set appropriate assumptions on variables explicitly. For example, assume that `x` and `y` are positive values.

```
assume([x,y],'positive')
combine(sqrt(x)*sqrt(y))

ans =
(x*y)^(1/2)
```

For further computations, clear the assumptions on `x` and `y`.

```
syms x y clear
```

As target functions, `combine` accepts `atan`, `exp`, `gamma`, `int`, `log`, `sincos`, and `sinhcosh`.

Expand Expressions

For elementary expressions, use the `expand` function to transform the original expression by multiplying sums of products. This function provides an easy way to expand polynomials.

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

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

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

ans =
x^3 - 6*x^2 + 11*x - 6
```

The function also expands exponential and logarithmic expressions. For example, expand this expression containing exponentials.

```
expand(exp(x + y)*(x + exp(x - y)))

ans =
exp(2*x) + x*exp(x)*exp(y)
```

Expand this logarithm. Expanding logarithms is not valid for generic complex values, but it is valid for positive values.

```
syms a b c positive
expand(log(a*b*c))

ans =
log(a) + log(b) + log(c)
```

For further computations, clear the assumptions.

```
syms a b c clear
```

Alternatively, use the `IgnoreAnalyticConstraints` option when expanding logarithms.

```
expand(log(a*b*c),'IgnoreAnalyticConstraints',true)

ans =
log(a) + log(b) + log(c)
```

`expand` also works on trigonometric expressions. For example, expand this expression.

```
expand(cos(x + y))
```

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

`expand` uses mathematical identities between the functions.

```
expand(sin(5*x))
```

```
ans =
sin(x) - 12*cos(x)^2*sin(x) + 16*cos(x)^4*sin(x)
```

```
expand(cos(3*acos(x)))
```

```
ans =
4*x^3 - 3*x
```

`expand` works recursively for all subexpressions.

```
expand((sin(3*x) + 1)*(cos(2*x) - 1))
```

```
ans =
2*sin(x) + 2*cos(x)^2 - 10*cos(x)^2*sin(x) + 8*cos(x)^4*sin(x) - 2
```

To prevent the expansion of all trigonometric, logarithmic, and exponential subexpressions, use the option `ArithmeticOnly`.

```
expand(exp(x + y)*(x + exp(x - y)),'ArithmeticOnly',true)
```

```
ans =
exp(x - y)*exp(x + y) + x*exp(x + y)
```

```
expand((sin(3*x) + 1)*(cos(2*x) - 1),'ArithmeticOnly',true)
```

```
ans =
cos(2*x) - sin(3*x) + cos(2*x)*sin(3*x) - 1
```

Factor Expressions

To return all irreducible factors of an expression, use the `factor` function. For example, find all irreducible polynomial factors of this polynomial expression. The result shows that this polynomial has three roots: `x = 1`, `x = 2`, and `x = 3`.

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

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

If a polynomial expression is irreducible, `factor` returns the original expression.

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

ans =
x^3 - 6*x^2 + 11*x - 5
```

Find irreducible polynomial factors of this expression. By default, `factor` uses factorization over rational numbers keeping rational numbers in their exact symbolic form. The resulting factors for this expression do not show polynomial roots.

```
factor(x^6 + 1)

ans =
[ x^2 + 1, x^4 - x^2 + 1]
```

Using other factorization modes lets you factor this expression further. For example, factor the same expression over complex numbers.

```
factor(x^6 + 1,'FactorMode','complex')

ans =
[ x + 0.86602540378443864676372317075294 + 0.5i,...
  x + 0.86602540378443864676372317075294 - 0.5i,...
  x + 1.0i,...
  x - 1.0i,...
  x - 0.86602540378443864676372317075294 + 0.5i,...
  x - 0.86602540378443864676372317075294 - 0.5i]
```

`factor` also works on expressions other than polynomials and rational expressions. For example, you can factor the following expression that contains logarithm, sine, and cosine functions. Internally, `factor` converts such expressions into polynomials and rational expressions by substituting subexpressions with variables. After computing irreducible factors, the function restores original subexpressions.

```
factor((log(x)^2 - 1)/(cos(x)^2 - sin(x)^2))

ans =
[ log(x) - 1, log(x) + 1, 1/(cos(x) - sin(x)), 1/(cos(x) + sin(x))]
```

Use `factor` to factor symbolic integers and symbolic rational numbers.

```
factor(sym(902834092))
factor(1/sym(210))

ans =
[ 2, 2, 47, 379, 12671]

ans =
[ 1/2, 1/3, 1/5, 1/7]
```

`factor` also can factor numbers larger than `flintmax` that the MATLAB `factor` cannot. To represent a large number accurately, place the number in quotation marks.

```
factor(sym('41758540882408627201'))

ans =
[ 479001599, 87178291199]
```

Extract Subexpressions from Expression

The `children` function returns the subexpressions of an expression.

Define an expression `f` with several subexpressions.

```
syms x y
f = exp(3*x)*y^3 + exp(2*x)*y^2 + exp(x)*y;
```

Extract the subexpressions of `f` by using `chlidren`.

```
expr = children(f)

expr =
[ y^2*exp(2*x), y^3*exp(3*x), y*exp(x)]
```

You can extract lower-level subexpressions by calling `children` repeatedly on the results.

Extract the subexpressions of `expr(1)` by calling `children` repeatedly. When the input to `children` is a vector, the output is a cell array.

```
expr1 = children(expr(1))
expr2 = children(expr1)

expr1 =
[ y^2, exp(2*x)]
```

```
expr2 =
  1×2 cell array
    {1×2 sym}    {1×1 sym}
```

Access the contents of the cell array `expr2` using braces.

```
expr2{1}
expr2{2}
```

```
ans =
[ y, 2]
ans =
2*x
```

Collect Terms with Same Powers

If a mathematical expression contains terms with the same powers of a specified variable or expression, the `collect` function reorganizes the expression by rouping such terms. When calling `collect`, specify the variables that the function must consider as unknowns. The `collect` function regards the original expression as a polynomial in the specified unknowns, and groups the coefficients with equal powers. Group the terms of an expression with the equal powers of `x`.

```
syms x y z
expr = x*y^4 + x*z + 2*x^3 + x^2*y*z +...
        3*x^3*y^4*z^2 + y*z^2 + 5*x*y*z;
collect(expr, x)
```

```
ans =
(3*y^4*z^2 + 2)*x^3 + y*z*x^2 + (y^4 + 5*z*y + z)*x + y*z^2
```

Group the terms of the same expression with the equal powers of `y`.

```
collect(expr, y)
```

```
ans =
(3*x^3*z^2 + x)*y^4 + (x^2*z + 5*x*z + z^2)*y + 2*x^3 + z*x
```

Group the terms of the same expression with the equal powers of `z`.

```
collect(expr, z)
```

```
ans =
(3*x^3*y^4 + y)*z^2 + (x + 5*x*y + x^2*y)*z + 2*x^3 + x*y^4
```

If you do not specify variables that `collect` must consider as unknowns, the function uses `symvar` to determine the default variable.

```
collect(expr)

ans =
(3*y^4*z^2 + 2)*x^3 + y*z*x^2 + (y^4 + 5*z*y + z)*x + y*z^2
```

Collect terms of an expression with respect to several unknowns by specifying those unknowns as a vector.

```
collect(expr, [y,z])

ans =
3*x^3*y^4*z^2 + x*y^4 + y*z^2 + (x^2 + 5*x)*y*z + x*z + 2*x^3
```

Rewrite Expressions in Terms of Other Functions

To present an expression in terms of a particular function, use `rewrite`. This function uses mathematical identities between functions. For example, rewrite an expression containing trigonometric functions in terms of a particular trigonometric function.

```
syms x
rewrite(sin(x),'tan')

ans =
(2*tan(x/2))/(tan(x/2)^2 + 1)

rewrite(cos(x),'tan')

ans =
-(tan(x/2)^2 - 1)/(tan(x/2)^2 + 1)

rewrite(sin(2*x) + cos(3*x)^2,'tan')

ans =
(tan((3*x)/2)^2 - 1)^2/(tan((3*x)/2)^2 + 1)^2 +...
(2*tan(x))/(tan(x)^2 + 1)
```

Use `rewrite` to express these trigonometric functions in terms of the exponential function.

```
rewrite(sin(x),'exp')
```

```
ans =
(exp(-x*1i)*1i)/2 - (exp(x*1i)*1i)/2
```

```
rewrite(cos(x),'exp')
```

```
ans =
exp(-x*1i)/2 + exp(x*1i)/2
```

Use `rewrite` to express these hyperbolic functions in terms of the exponential function.

```
rewrite(sinh(x),'exp')
```

```
ans =
exp(x)/2 - exp(-x)/2
```

```
rewrite(cosh(x),'exp')
```

```
ans =
exp(-x)/2 + exp(x)/2
```

`rewrite` also expresses inverse hyperbolic functions in terms of logarithms.

```
rewrite(asinh(x),'log')
```

```
ans =
log(x + (x^2 + 1)^(1/2))
```

```
rewrite(acosh(x),'log')
```

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

Compute Partial Fraction Decompositions of Expressions

The `partfrac` function returns a rational expression in the form of a sum of a polynomial and rational terms. In each rational term, the degree of the numerator is smaller than the degree of the denominator. For some expressions, `partfrac` returns visibly simpler forms.

```
syms x
n = x^6 + 15*x^5 + 94*x^4 + 316*x^3 + 599*x^2 + 602*x + 247;
d = x^6 + 14*x^5 + 80*x^4 + 238*x^3 + 387*x^2 + 324*x + 108;
partfrac(n/d, x)
```

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

The denominators in rational terms represent the factored common denominator of the original expression.

```
factor(d)

ans =
[ x + 1, x + 2, x + 2, x + 3, x + 3, x + 3]
```

Compute Normal Forms of Rational Expressions

The `simplifyFraction` function represents the original rational expression as a single rational term with expanded numerator and denominator. The greatest common divisor of the numerator and denominator of the returned expression is 1. This function is more efficient for simplifying fractions than the `simplify` function.

```
syms x y
simplifyFraction((x^3 + 3*y^2)/(x^2 - y^2) + 3)

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

`simplifyFraction` cancels common factors that appear in numerator and denominator.

```
simplifyFraction(x^2/(x + y) - y^2/(x + y))

ans =
x - y
```

`simplifyFraction` also handles expressions other than polynomials and rational functions. Internally, it converts such expressions into polynomials or rational functions by substituting subexpressions with identifiers. After normalizing the expression with temporary variables, `simplifyFraction` restores the original subexpressions.

```
simplifyFraction((exp(2*x) - exp(2*y))/(exp(x) - exp(y)))

ans =
exp(x) + exp(y)
```

Represent Polynomials Using Horner Nested Forms

The Horner, or nested, form of a polynomial expression is efficient for numerical evaluation because it often involves fewer arithmetical operations than other mathematically equivalent forms of the same polynomial. Typically, this form of an expression is numerically stable. To represent a polynomial expression in a nested form, use the `horner` function.

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

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

If polynomial coefficients are floating-point numbers, the resulting Horner form represents them as rational numbers.

```
horner(1.1 + 2.2*x + 3.3*x^2)
```

```
ans =
x*((33*x)/10 + 11/5) + 11/10
```

To convert the coefficients in the result to floating-point numbers, use `vpa`.

```
vpa(ans)
```

```
ans =
x*(3.3*x + 2.2) + 1.1
```

Extract Numerators and Denominators of Rational Expressions

To extract the numerator and denominator of a rational symbolic expression, use the `numden` function. The first output argument of `numden` is a numerator, the second output argument is a denominator. Use `numden` to find numerators and denominators of symbolic rational numbers.

```
[n,d] = numden(1/sym(3))

n =
1

d =
3
```

Use `numden` to find numerators and denominators of a symbolic expressions.

```
syms x y
[n,d] = numden((x^2 - y^2)/(x^2 + y^2))

n =
x^2 - y^2

d =
x^2 + y^2
```

Use `numden` to find numerators and denominators of symbolic functions. If the input is a symbolic function, `numden` returns the numerator and denominator as symbolic functions.

```
syms f(x) g(x)
f(x) = sin(x)/x^2;
g(x) = cos(x)/x;
[n,d] = numden(f)

n(x) =
sin(x)

d(x) =
x^2

[n,d] = numden(f/g)

n(x) =
sin(x)
```

```
d(x) =
x*cos(x)
```

numden converts the input to its one-term rational form, such that the greatest common divisor of the numerator and denominator is 1. Then it returns the numerator and denominator of that form of the expression.

```
[n,d] = numden(x/y + y/x)

n =
x^2 + y^2

d =
x*y
```

numden works on vectors and matrices. If an input is a vector or matrix, numden returns two vectors or two matrices of the same size as the input. The first vector or matrix contains numerators of each element. The second vector or matrix contains denominators of each element. For example, find numerators and denominators of each element of the 3-by-3 Hilbert matrix.

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

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

[n,d] = numden(H)

n =
[ 1, 1, 1]
[ 1, 1, 1]
[ 1, 1, 1]

d =
[ 1, 2, 3]
[ 2, 3, 4]
[ 3, 4, 5]
```

Substitute Variables in Symbolic Expressions

Solve the following trigonometric equation using the ReturnConditions option of the solver to obtain the complete solution. The solver returns the solution, parameters used in the solution, and conditions on those parameters.

```
syms x
eqn = sin(2*x) + cos(x) == 0;
[solx, params, conds] = solve(eqn, x, 'ReturnConditions', true)

solx =
       pi/2 + pi*k
     2*pi*k - pi/6
 (7*pi)/6 + 2*pi*k

params =
k

conds =
 in(k, 'integer')
 in(k, 'integer')
 in(k, 'integer')
```

Replace the parameter k with a new symbolic variable a. First, create symbolic variables k and a. (The solver does not create variable k in the MATLAB workspace.)

```
syms k a
```

Now, use the subs function to replace k by a in the solution vector solx, parameters params, and conditions conds.

```
solx = subs(solx, k, a)
params = subs(params, k, a)
conds = subs(conds, k, a)

solx =
       pi/2 + pi*a
     2*pi*a - pi/6
 (7*pi)/6 + 2*pi*a
params =
a
conds =
 in(a, 'integer')
 in(a, 'integer')
 in(a, 'integer')
```

Suppose, you know that the value of the parameter `a` is `2`. Substitute `a` with `2` in the solution vector `solx`.

```
subs(solx, a, 2)

ans =
  (5*pi)/2
 (23*pi)/6
 (31*pi)/6
```

Alternatively, substitute `params` with `2`. This approach returns the same result.

```
subs(solx, params, 2)

ans =
  (5*pi)/2
 (23*pi)/6
 (31*pi)/6
```

Substitute parameter `a` with a floating-point number. The toolbox converts numbers to floating-point values, but it keeps intact the symbolic expressions, such as `sym(pi)`, `exp(sym(1))`, and so on.

```
subs(solx, params, vpa(2))

ans =
                               2.5*pi
 3.8333333333333333333333333333333*pi
 5.1666666666666666666666666666667*pi
```

Approximate the result of substitution with floating-point values by using `vpa` on the result returned by `subs`.

```
vpa(subs(solx, params, 2))

ans =
 7.8539816339744830961566084581988
 12.042771838760874080773466302571
 16.231562043547265065390324146944
```

Substitute Elements in Symbolic Matrices

Create a 3-by-3 circulant matrix using the backward shift.

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

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

Replace variable `b` in this matrix by the expression `a + 1`. The `subs` function replaces all `b` elements in matrix `M` with the expression `a + 1`.

```
M = subs(M, b, a + 1)
```

```
M =
[     a, a + 1,     c]
[ a + 1,     c,     a]
[     c,     a, a + 1]
```

You also can specify the value to replace by indexing into matrix. That is, to replace all elements whose value is `c`, you can specify the value to replace as `c`, `M(1,3)` or `M(3,1)`.

Replace all elements whose value is `M(1,3) = c` with the expression `a + 2`.

```
M = subs(M, M(1,3), a + 2)
```

```
M =
[     a, a + 1, a + 2]
[ a + 1, a + 2,     a]
[ a + 2,     a, a + 1]
```

Tip To replace a particular element of a matrix with a new value while keeping all other elements unchanged, use the assignment operation. For example, `M(1,1) = 2` replaces only the first element of the matrix `M` with the value `2`.

Find eigenvalues and eigenvectors of the matrix.

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

```
V =
[ 1,   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]

E =
[ 3*a + 3,       0,        0]
[       0, 3^(1/2),        0]
[       0,       0, -3^(1/2)]
```

Replace the symbolic parameter `a` with the value `1`.

```
subs(E, a, 1)
```

```
ans =
[ 6,       0,        0]
[ 0, 3^(1/2),        0]
[ 0,       0, -3^(1/2)]
```

Substitute Scalars with Matrices

Create the following expression representing the sine function.

```
syms w t
f = sin(w*t);
```

Suppose, your task involves creating a matrix whose elements are sine functions with angular velocities represented by a Toeplitz matrix. First, create a 4-by-4 Toeplitz matrix.

```
W = toeplitz(sym([3 2 1 0]))
```

```
W =
[ 3, 2, 1, 0]
[ 2, 3, 2, 1]
[ 1, 2, 3, 2]
[ 0, 1, 2, 3]
```

Next, replace the variable `w` in the expression `f` with the Toeplitz matrix `W`. When you replace a scalar in a symbolic expression with a matrix, `subs` expands the expression into a matrix. In this example, `subs` expands `f = sin(w*t)` into a 4-by-4 matrix whose elements are `sin(w*t)`. Then it replaces `w` in that matrix with the corresponding elements of the Toeplitz matrix `W`.

```
F = subs(f, w, W)
```

```
F =
[ sin(3*t), sin(2*t),   sin(t),        0]
[ sin(2*t), sin(3*t), sin(2*t),   sin(t)]
[   sin(t), sin(2*t), sin(3*t), sin(2*t)]
[        0,   sin(t), sin(2*t), sin(3*t)]
```

Find the sum of these sine waves at `t = π`, `t = π/2`, `t = π/3`, `t = π/4`, `t = π/5`, and `t = π/6`. First, find the sum of all elements of matrix `F`. Here, the first call to `sum` returns a row vector containing sums of elements in each column. The second call to `sum` returns the sum of elements of that row vector.

```
S = sum(sum(F))
```

```
S =
6*sin(2*t) + 4*sin(3*t) + 4*sin(t)
```

Now, use `subs` to evaluate `S` for particular values of the variable `t`.

```
subs(S, t, sym(pi)./[1:6])
```

```
[ 0,...
  0,...
  5*3^(1/2), 4*2^(1/2) + 6,...
  2^(1/2)*(5 - 5^(1/2))^(1/2) + (5*2^(1/2)*(5^(1/2) + 5)^(1/2))/2,...
  3*3^(1/2) + 6]
```

You also can use `subs` to replace a scalar element of a matrix with another matrix. In this case, `subs` expands the matrix to accommodate new elements. For example, replace zero elements of the matrix `F` with a column vector `[1;2]`. The original 4-by-4 matrix `F` expands to an 8-by-4 matrix. The `subs` function duplicates each row of the original matrix, not only the rows containing zero elements.

```
F = subs(F, 0, [1;2])
```

```
F =
[ sin(3*t), sin(2*t),   sin(t),        1]
[ sin(3*t), sin(2*t),   sin(t),        2]
[ sin(2*t), sin(3*t), sin(2*t),   sin(t)]
[ sin(2*t), sin(3*t), sin(2*t),   sin(t)]
[   sin(t), sin(2*t), sin(3*t), sin(2*t)]
[   sin(t), sin(2*t), sin(3*t), sin(2*t)]
[        1,   sin(t), sin(2*t), sin(3*t)]
[        2,   sin(t), sin(2*t), sin(3*t)]
```

Evaluate Symbolic Expressions Using subs

When a value is assigned to a symbolic variable, expressions containing the variable are not automatically evaluated. Instead, evaluate expressions by using `subs`.

Define the expression `y = x^2`.

```
syms x
y = x^2;
```

Assign `2` to `x`. The value of `y` is still `x^2` instead of `4`.

```
x = 2;
y

y =
x^2
```

If you change the value of `x` again, the value of `y` stays `x^2`. Instead, evaluate `y` with the new value of `x` by using `subs`.

```
subs(y)

ans =
4
```

The evaluated result is `4`. However, `y` has not changed. Change the value of `y` by assigning the result to `y`.

```
y = subs(y)

y =
4
```

Show that `y` is independent of `x` after this assignment.

```
x = 5;
subs(y)

ans =
4
```

Choose Symbolic or Numeric Arithmetic

Symbolic Math Toolbox operates on numbers by using either symbolic or numeric arithmetic. Numeric arithmetic is either variable precision or double precision. The following information compares symbolic, variable-precision, and double-precision arithmetic.

	Symbolic	Variable Precision	Double Precision

Example: Find sin(π)	`a = sym(pi)`
	`sin(a)`
	
	`a =`
	`pi`
	`ans =`
	`0`
		`b = vpa(pi)`
		`sin(b)`
		
		`b =`
		`3.1415926535897932384626433832795`
		`ans =`
		`-3.2101083013100396069547145883568e-40`
			`pi`
			`sin(pi)`
			
			`ans =`
			`3.1416`
			`ans =`
			`1.2246e-16`

Functions Used	`sym`	`vpa` `digits`	`double`
Round-Off Errors	No, finds exact results	Yes, magnitude depends on precision used	Yes, has 16 digits of precision
Speed	Slowest	Faster, depends on precision used	Faster
Memory Usage	Greatest	Adjustable, depends on precision used	Least

Symbolic Arithmetic

By default, Symbolic Math Toolbox uses exact numbers, such as `1/3`, `sqrt(2)`, or `pi`, to perform exact symbolic computations on page 1-12.

Variable-Precision Arithmetic

Variable-precision arithmetic using `vpa` is the recommended approach for numeric calculations in Symbolic Math Toolbox. For greater precision, increase the number of significant digits on page 2-116. For faster computations and decreased memory usage, decrease the number of significant digits on page 2-123.

Double-Precision Arithmetic

Double-precision, floating-point arithmetic uses the same precision as most numeric computations in MATLAB. This arithmetic is recommended when you do not have Symbolic Math Toolbox or are using functions that do not accept symbolic input. Otherwise, exact symbolic numbers and variable-precision arithmetic are recommended. To approximate a value with double precision, use the `double` function.

Increase Precision of Numeric Calculations

By default, MATLAB uses 16 digits of precision. For higher precision, use the `vpa` function in Symbolic Math Toolbox. `vpa` provides variable precision which can be increased without limit.

When you choose variable-precision arithmetic, by default, `vpa` uses 32 significant decimal digits of precision. For details, see “Choose Symbolic or Numeric Arithmetic” on page 2-114. You can set a higher precision by using the `digits` function.

Approximate a sum using the default precision of 32 digits. If at least one input is wrapped with `vpa`, all other inputs are converted to variable precision automatically.

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

```
ans =
0.83333333333333333333333333333333
```

You must wrap all inner inputs with `vpa`, such as `exp(vpa(200))`. Otherwise, the inputs are automatically converted to double by MATLAB.

Increase the precision to `50` digits by using `digits` and save the old value of `digits` in `digitsOld`. Repeat the sum.

```
digitsOld = digits(50);
sum50 = vpa(1/3) + 1/2
```

```
sum50 =
0.83333333333333333333333333333333333333333333333333
```

Restore the old value of digits for further calculations.

```
digits(digitsOld)
```

Note `vpa` output is symbolic. To use symbolic output with a MATLAB function that does not accept symbolic values, convert symbolic values to double precision by using `double`.

Check the current `digits` setting by calling `digits`.

```
digits
```

```
Digits = 32
```

Change the precision for a single `vpa` call by specifying the precision as the second input to `vpa`. This call does not affect `digits`. For example, approximate `pi` with `100` digits.

```
vpa(pi,100)

ans =
3.141592653589793238462643383279502884197169399375105820974944
592307816406286208998628034825342117068

digits    % digits remains 32

Digits = 32
```

Variable precision can be increased arbitrarily. Find `pi` to `500` digits.

```
digitsOld = digits(500);
vpa(pi)
digits(digitsOld)

ans =
3.1415926535897932384626433832795028841971693993751058209749
4459230781640628620899862803482534211706798214808651328230664
7093844609550582231725359408128481117450284102701938521105559
6446229489549303819644288109756659334461284756482337867831652
7120190914564856692346034861045432664821339360726024914127372
4587006606315588174881520920962829254091715364367892590360011
3305305488204665213841469519415116094330572703657595919530921
8611738193261179310511854807446237996274956735188575272489122
79381830119491
```

`digits` and `vpa` control the number of *significant* decimal digits. For example, approximating `1/111` with four-digit accuracy returns six digits after the decimal point because the first two digits are zeros.

```
vpa(1/111,4)

ans =
0.009009
```

Note If you want to increase performance by *decreasing* precision, see “Increase Speed by Reducing Precision” on page 2-123.

Recognize and Avoid Round-Off Errors

When approximating a value numerically, remember that floating-point results can be sensitive to the precision used. Also, floating-point results are prone to round-off errors. The following approaches can help you recognize and avoid incorrect results.

In this section...

Use Symbolic Computations When Possible

Performing computations symbolically on page 2-114 is recommended because exact symbolic computations are not prone to round-off errors. For example, standard mathematical constants have their own symbolic representations in Symbolic Math Toolbox:

```
pi
sym(pi)

ans =
    3.1416

ans =
pi
```

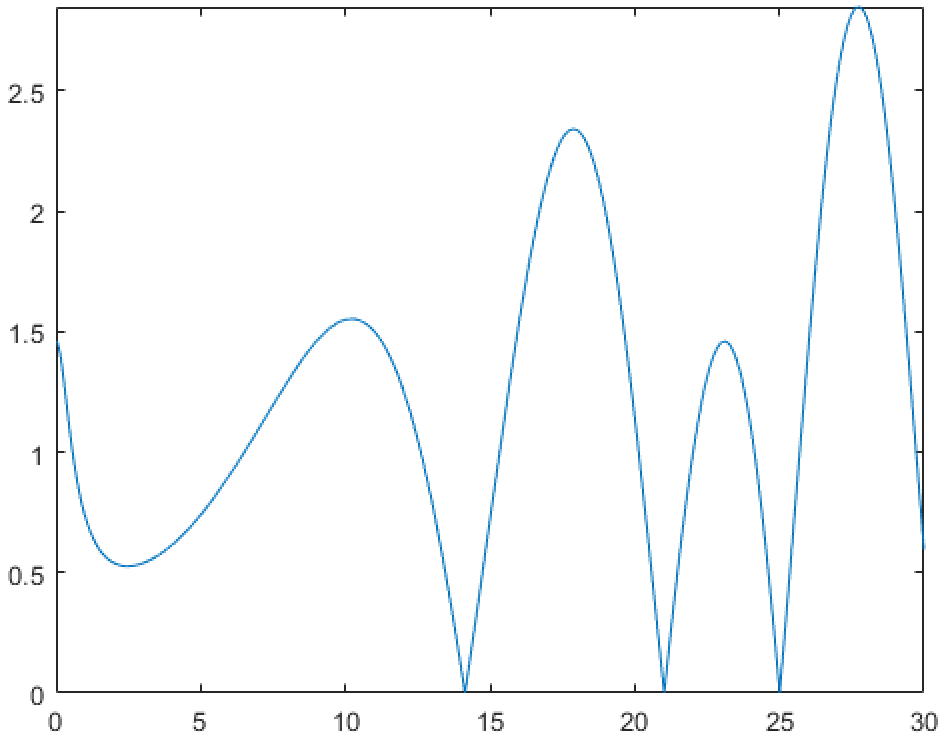
Avoid unnecessary use of numeric approximations. A floating-point number approximates a constant; it is not the constant itself. Using this approximation, you can get incorrect results. For example, the `heaviside` special function returns different results for the sine of `sym(pi)` and the sine of the numeric approximation of `pi`:

```
heaviside(sin(sym(pi)))
heaviside(sin(pi))

ans =
1/2

ans =
     1
```

Perform Calculations with Increased Precision

The Riemann hypothesis states that all nontrivial zeros of the Riemann Zeta function $\zeta(z)$ have the same real part $\Re(z) = 1/2$. To locate possible zeros of the Zeta function, plot its absolute value $|\zeta(1/2 + iy)|$. The following plot shows the first three nontrivial roots of the Zeta function $|\zeta(1/2 + iy)|$. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms y
fplot(abs(zeta(1/2 + i*y)), [0 30])
```



Use the numeric solver `vpasolve` to approximate the first three zeros of this Zeta function:

```
vpasolve(zeta(1/2 + i*y), y, 15)
vpasolve(zeta(1/2 + i*y), y, 20)
vpasolve(zeta(1/2 + i*y), y, 25)

ans =
14.134725141734693790457251983562

ans =
21.022039638771554992628479593897

ans =
25.010857580145688763213790992563
```

Now, consider the same function, but slightly increase the real part,

$\zeta\left(\frac{1000000001}{2000000000}+iy\right)$. According to the Riemann hypothesis, this function does not have a zero for any real value y. If you use `vpasolve` with the 10 significant decimal digits, the solver finds the following (nonexisting) zero of the Zeta function:

```
old = digits;
digits(10)
vpasolve(zeta(1000000001/2000000000 + i*y), y, 15)

ans =
14.13472514
```

Increasing the number of digits shows that the result is incorrect. The Zeta function

$\zeta\left(\frac{1000000001}{2000000000}+iy\right)$ does not have a zero for any real value $14 < y < 15$:

```
digits(15)
vpasolve(zeta(1000000001/2000000000 + i*y), y, 15)
digits(old)

ans =
14.1347251417347 + 0.000000000499989207306345i
```

For further computations, restore the default number of digits:

```
digits(old)
```

Compare Symbolic and Numeric Results

Bessel functions with half-integer indices return exact symbolic expressions. Approximating these expressions by floating-point numbers can produce very unstable results. For example, the exact symbolic expression for the following Bessel function is:

```
B = besselj(53/2, sym(pi))
```

```
B =
(351*2^(1/2)*(119409675/pi^4 - 20300/pi^2 - 315241542000/pi^6...
 + 445475704038750/pi^8 - 366812794263762000/pi^10 +...
 182947881139051297500/pi^12 - 55720697512636766610000/pi^14...
 + 10174148683695239020903125/pi^16 - 1060253389142977540073062500/pi^18...
 + 57306695683177936040949028125/pi^20 - 1331871030107060331702688875000/pi^22...
 + 8490677816932509614604641578125/pi^24 + 1))/pi^2
```

Use `vpa` to approximate this expression with the 10-digit accuracy:

```
vpa(B, 10)
```

```
ans =
-2854.225191
```

Now, call the Bessel function with the floating-point parameter. Significant difference between these two approximations indicates that one or both results are incorrect:

```
besselj(53/2, pi)
```

```
ans =
   6.9001e-23
```

Increase the numeric working precision to obtain a more accurate approximation for `B`:

```
vpa(B, 50)
```

```
ans =
0.000000000000000000000069001456069172842068862232841396473796597233761161
```

Plot the Function or Expression

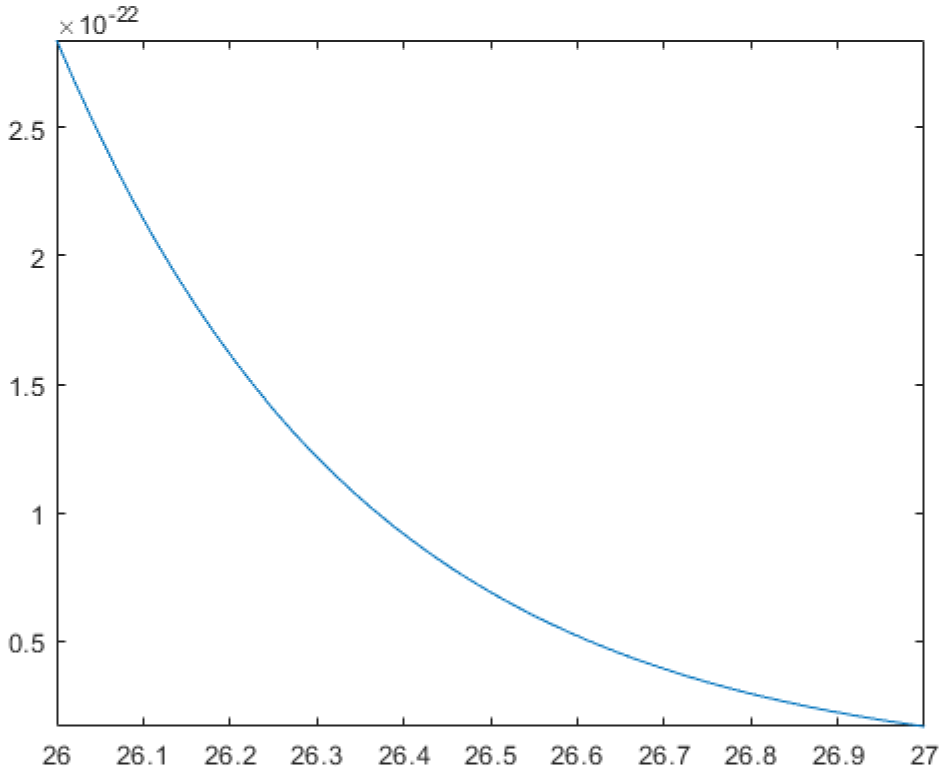
Plotting the results can help you recognize incorrect approximations. For example, the numeric approximation of the following Bessel function returns:

```
B = besselj(53/2, sym(pi));
vpa(B, 10)
```

```
ans =
-2854.225191
```

Plot this Bessel function for the values of x around 53/2. The function plot shows that the approximation is incorrect:

```
syms x
fplot(besselj(x, sym(pi)), [26 27])
```



Increase Speed by Reducing Precision

Increase MATLAB's speed by reducing the precision of your calculations. Reduce precision by using variable-precision arithmetic provided by the `vpa` and `digits` functions in Symbolic Math Toolbox. When you reduce precision, you are gaining performance by reducing accuracy. For details, see "Choose Symbolic or Numeric Arithmetic" on page 2-114.

For example, finding the Riemann zeta function of the large matrix `C` takes a long time. First, initialize `C`.

```
[X,Y] = meshgrid((0:0.0025:.75),(5:-0.05:0));
C = X + Y*i;
```

Then, find the time taken to calculate `zeta(C)`.

```
tic
zeta(C);
toc
```

```
Elapsed time is 340.204407 seconds.
```

Now, repeat this operation with a lower precision by using `vpa`. First, change the precision used by `vpa` to a lower precision of `10` digits by using `digits`. Then, use `vpa` to reduce the precision of `C` and find `zeta(C)` again. The operation is significantly faster.

```
digits(10)
vpaC = vpa(C);
tic
zeta(vpaC);
toc
```

```
Elapsed time is 113.792543 seconds.
```

Note `vpa` output is symbolic. To use symbolic output with a MATLAB function that does not accept symbolic values, convert symbolic values to double precision by using `double`.

For larger matrices, the difference in computation time can be even more significant. For example, consider the `1001`-by-`301` matrix `C`.

```
    [X,Y] = meshgrid((0:0.0025:.75),(5:-0.005:0));
C = X + Y*i;
```

Running `zeta(vpa(C))` with 10-digit precision takes 15 minutes, while running `zeta(C)` takes three times as long.

```
digits(10)
vpaC = vpa(C);
tic
zeta(vpaC);
toc
```

```
Elapsed time is 886.035806 seconds.
```

```
tic
zeta(C);
toc
```

```
Elapsed time is 2441.991572 seconds.
```

Note If you want to *increase* precision, see “Increase Precision of Numeric Calculations” on page 2-116.

Numeric to Symbolic Conversion

This topic shows how Symbolic Math Toolbox converts numbers into symbolic form. For an overview of symbolic and numeric arithmetic, see “Choose Symbolic or Numeric Arithmetic” on page 2-114.

To convert numeric input to symbolic form, use the `sym` command. By default, `sym` returns a rational approximation of a numeric expression.

```
t = 0.1;
sym(t)
```

```
ans =
1/10
```

`sym` determines that the double-precision value `0.1` approximates the exact symbolic value `1/10`. In general, `sym` tries to correct the round-off error in floating-point inputs to return the exact symbolic form. Specifically, `sym` corrects round-off error in numeric inputs that match the forms p/q, $p\pi/q$, $(p/q)^{1/2}$, 2^q, and 10^q, where p and q are modest-sized integers.

For these forms, demonstrate that `sym` converts floating-point inputs to the exact symbolic form. First, numerically approximate 1/7, pi, and $1/\sqrt{2}$.

```
N1 = 1/7
N2 = pi
N3 = 1/sqrt(2)
```

```
N1 =
    0.1429
N2 =
    3.1416
N3 =
    0.7071
```

Convert the numeric approximations to exact symbolic form. `sym` corrects the round-off error.

```
S1 = sym(N1)
S2 = sym(N2)
S3 = sym(N3)
```

```
S1 =
1/7
```

```
S2 =
pi
S3 =
2^(1/2)/2
```

To return the error between the input and the estimated exact form, use the syntax `sym(num,'e')`. See “Conversion to Rational Symbolic Form with Error Term” on page 2-127.

You can force `sym` to accept the input as is by placing the input in quotes. Demonstrate this behavior on the previous input `0.142857142857143`. The `sym` function does not convert the input to `1/7`.

```
sym('0.142857142857143')

ans =
0.142857142857143
```

When you convert large numbers, use quotes to exactly represent them. Demonstrate this behavior by comparing `sym(133333333333333333333)` with `sym('133333333333333333333')`.

```
sym(1333333333333333333)
sym('1333333333333333333')

ans =
1333333333333333248
ans =
1333333333333333333
```

You can specify the technique used by `sym` to convert floating-point numbers using the optional second argument, which can be `'f'`, `'r'`, `'e'`, or `'d'`. The default flag is `'r'`, for rational form on page 2-127.

Conversion to Rational Symbolic Form

Convert input to exact rational form by calling `sym` with the `'r'` flag. This is the default behavior when you call `sym` without flags.

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

```
ans =
1/10
```

Conversion by Using Floating-Point Expansion

If you call `sym` with the flag `'f'`, `sym` converts double-precision, floating-point numbers to their numeric value by using `N*2^e`, where `N` and `e` are the exponent and mantissa respectively.

Convert `t` by using a floating-point expansion.

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

```
ans =
3602879701896397/36028797018963968
```

Conversion to Rational Symbolic Form with Error Term

If you call `sym` with the flag `'e'`, `sym` returns the rational form of `t` plus the error between the estimated, exact value for `t` and its floating-point representation. This error is expressed in terms of `eps` (the floating-point relative precision).

Convert `t` to symbolic form. Return the error between its estimated symbolic form and its floating-point value.

```
sym(t, 'e')
```

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

The error term `eps/40` is the difference between `sym('0.1')` and `sym(0.1)`.

Conversion to Decimal Form

If you call `sym` with the flag `'d'`, `sym` returns the decimal expansion of the input. The `digits` function specifies the number of significant digits used. The default value of `digits` is 32.

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

```
ans =
0.10000000000000000555111512312578
```

Change the number of significant digits by using `digits`.

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

```
ans =
0.1
```

For further calculations, restore the old value of `digits`.

```
digits(digitsOld)
```

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: 1-by-0
```

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(3))
```

```
g =
[ 1/(2160*x), 1, (2160*x^2 - 3072*x + 253)/x]
ans =
 32/45 - 12589^(1/2)/180
 12589^(1/2)/180 + 32/45
```

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

```
R = sym(rosser)
```

generates

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

The commands

```
p = charpoly(R, x);
pretty(factor(p))
```

produce

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

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

You can rewrite `g` in terms of exponents:

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

```
g =
 exp(-t*1i)
  exp(t*1i)
```

Jordan Canonical Form

The Jordan canonical form (Jordan normal 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 where `J = inv(V)*A*V`. The columns of `V` are the generalized eigenvectors of `A`.

The Jordan form is extremely sensitive to changes. Almost any change in `A` causes its Jordan form to be diagonal. This implies that `A` must be known exactly (i.e., without round-off error, etc.) and makes it very difficult to compute the Jordan form reliably with floating-point arithmetic. Thus, computing the Jordan form with floating-point values is unreliable and not recommended.

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]
```

Show that `J` and `inv(V)*A*V` are equal by using `isequal`. The `isequal` function returns logical `1` (`true`) meaning that the inputs are equal.

```
isequal(J, inv(V)*A*V)

ans =
  logical
   1
```

From `J`, we see that `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,...
                                        0.36054891952096214791189887728353]
[ -0.82525689650849463222502853672224,  0.37514965283965451993171338605042,...
                                        0.42215375485651489522488031917364]
[  0.18801243961043281839917114171742, -0.52236064041897439447429784257224,...
                                        0.83173955292075192178421874331406]

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

V =
[  0.18801243961043281839917114171742,  0.52236064041897439447429784257224,...
                                        0.83173955292075192178421874331406]
[ -0.82525689650849463222502853672224, -0.37514965283965451993171338605042,...
                                        0.42215375485651489522488031917364]
[  0.53254331027335338470683368360204, -0.76576895948802052989304092179952,...
                                        0.36054891952096214791189887728353]
```

Solve Algebraic Equation

Symbolic Math Toolbox offers both symbolic and numeric equation solvers. This topic shows you how to solve an equation symbolically using the symbolic solver `solve`. To compare symbolic and numeric solvers, see “Select Numeric or Symbolic Solver” on page 2-151.

In this section...

Solve an Equation

If `eqn` is an equation, `solve(eqn, x)` solves `eqn` for the symbolic variable `x`.

Use the == operator to specify the familiar quadratic equation and solve it using `solve`.

```
syms a b c x
eqn = a*x^2 + b*x + c == 0;
solx = solve(eqn, x)
```

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

`solx` is a symbolic vector containing the two solutions of the quadratic equation. If the input `eqn` is an expression and not an equation, `solve` solves the equation `eqn == 0`.

To solve for a variable other than `x`, specify that variable instead. For example, solve `eqn` for `b`.

```
solb = solve(eqn, b)
```

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

If you do not specify a variable, solve uses symvar to select the variable to solve for. For example, solve(eqn) solves eqn for x.

Return the Full Solution to an Equation

solve does not automatically return all solutions of an equation. Solve the equation cos(x) == -sin(x). The solve function returns one of many solutions.

```
syms x
solx = solve(cos(x) == -sin(x), x)

solx =
-pi/4
```

To return all solutions along with the parameters in the solution and the conditions on the solution, set the ReturnConditions option to true. Solve the same equation for the full solution. Provide three output variables: for the solution to x, for the parameters in the solution, and for the conditions on the solution.

```
syms x
[solx, param, cond] = solve(cos(x) == -sin(x), x, 'ReturnConditions', true)

solx =
pi*k - pi/4
param =
k
cond =
in(k, 'integer')
```

solx contains the solution for x, which is pi*k - pi/4. The param variable specifies the parameter in the solution, which is k. The cond variable specifies the condition in(k, 'integer') on the solution, which means k must be an integer. Thus, solve returns a periodic solution starting at pi/4 which repeats at intervals of pi*k, where k is an integer.

Work with the Full Solution, Parameters, and Conditions Returned by solve

You can use the solutions, parameters, and conditions returned by solve to find solutions within an interval or under additional conditions.

To find values of x in the interval -2*pi<x<2*pi, solve solx for k within that interval under the condition cond. Assume the condition cond using assume.

```
assume(cond)
solk = solve(-2*pi<solx, solx<2*pi, param)
```

```
solk =
 -1
  0
  1
  2
```

To find values of x corresponding to these values of k, use subs to substitute for k in solx.

```
xvalues = subs(solx, solk)
```

```
xvalues =
 -(5*pi)/4
     -pi/4
  (3*pi)/4
  (7*pi)/4
```

To convert these symbolic values into numeric values for use in numeric calculations, use vpa.

```
xvalues = vpa(xvalues)
```

```
xvalues =
  -3.9269908169872415480783042290994
 -0.78539816339744830961566084581988
   2.3561944901923449288469825374596
   5.4977871437821381673096259207391
```

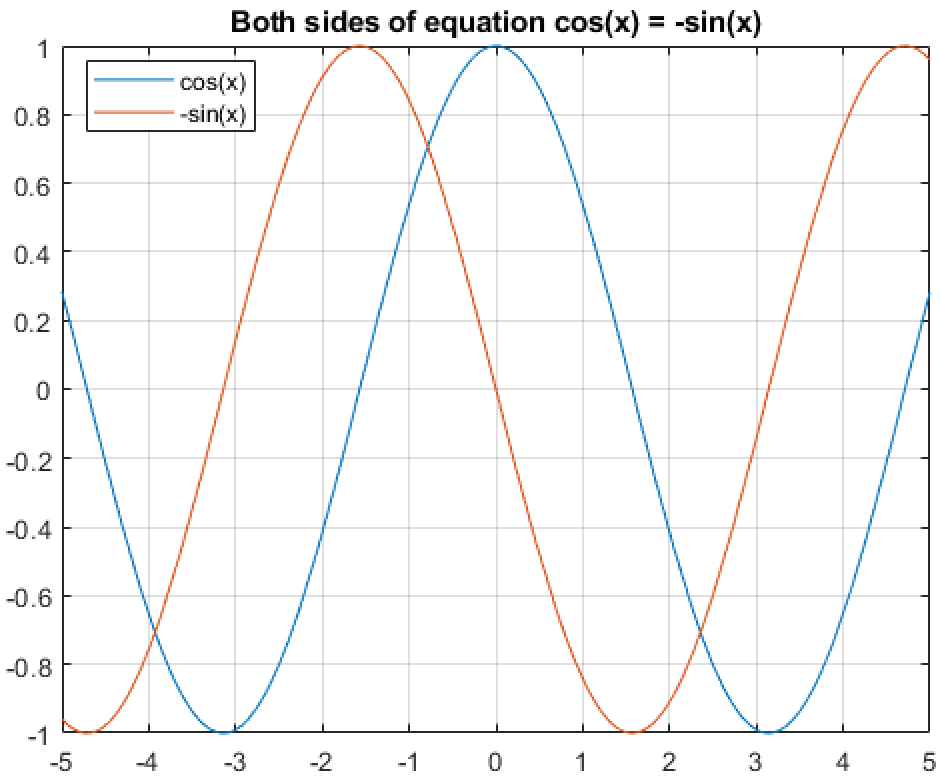
Visualize and Plot Solutions Returned by solve

The previous sections used solve to solve the equation cos(x) == -sin(x). The solution to this equation can be visualized using plotting functions such as fplot and scatter.

Plot both sides of equation cos(x) == -sin(x).

```
fplot(cos(x))
hold on
```

```
grid on
fplot(-sin(x))
title('Both sides of equation cos(x) = -sin(x)')
legend('cos(x)','-sin(x)','Location','best','AutoUpdate','off')
```



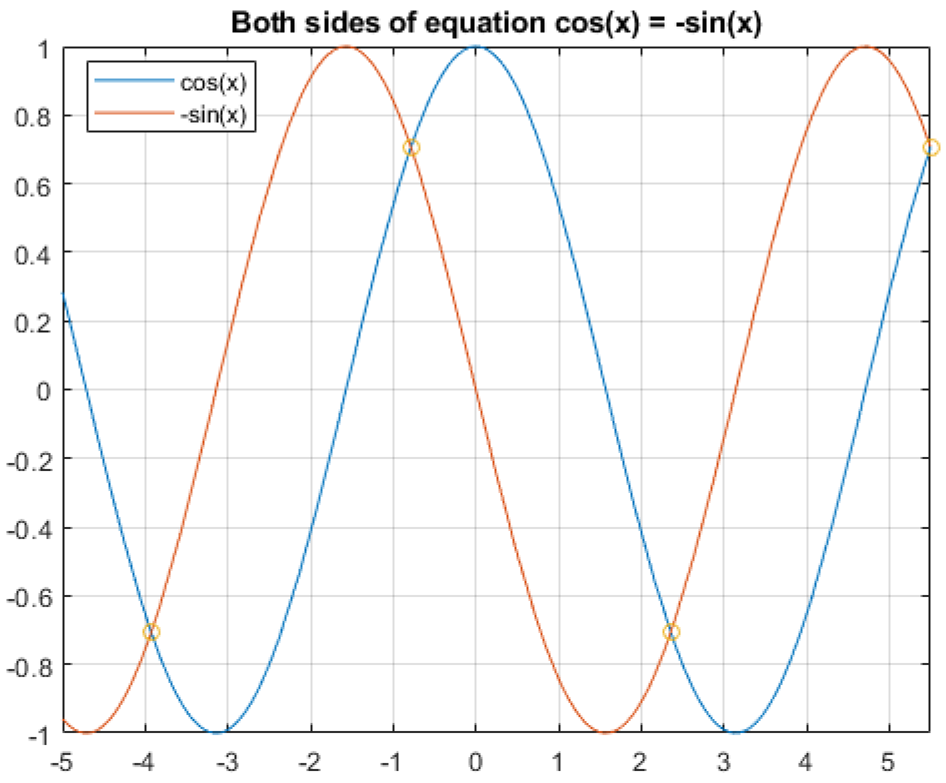
Calculate the values of the functions at the values of `x`, and superimpose the solutions as points using `scatter`.

```
yvalues = cos(xvalues)
```

```
yvalues =
```

$$\begin{pmatrix} -0.70710678118654752440084436210485 \\ 0.70710678118654752440084436210485 \\ -0.70710678118654752440084436210485 \\ 0.70710678118654752440084436210485 \end{pmatrix}$$

```
scatter(xvalues, yvalues)
```



As expected, the solutions appear at the intersection of the two plots.

Simplify Complicated Results and Improve Performance

If results look complicated, `solve` is stuck, or if you want to improve performance, see, "Troubleshoot Equation Solutions from solve Function" on page 2-164.

Select Numeric or Symbolic Solver

You can solve equations to obtain a symbolic or numeric answer. For example, a solution to $\cos(x) = -1$ is `pi` in symbolic form and `3.14159` in numeric form. The symbolic solution is exact, while the numeric solution approximates the exact symbolic solution. Symbolic Math Toolbox offers both symbolic and numeric equation solvers. This table can help you choose either the symbolic solver (`solve`) or the numeric solver (`vpasolve`). A possible strategy is to try the symbolic solver first, and use the numeric solver if the symbolic solver is stuck.

Solve Equations Symbolically Using solve	Solve Equations Numerically Using vpasolve
Returns exact solutions. Solutions can then be approximated using `vpa`.	Returns approximate solutions. Precision can be controlled arbitrarily using `digits`.
Returns a general form of the solution.	For polynomial equations, returns all numeric solutions that exist. For nonpolynomial equations, returns the first numeric solution found.
General form allows insight into the solution.	Numeric solutions provide less insight.
Runs slower.	Runs faster.
Search ranges can be specified using inequalities.	Search ranges and starting points can be specified.
`solve` solves equations and inequalities that contain parameters.	`vpasolve` does not solve inequalities, nor does it solve equations that contain parameters.
`solve` can return parameterized solutions.	`vpasolve` does not return parameterized solutions.

`vpasolve` uses variable-precision arithmetic. You can control precision arbitrarily using `digits`. For examples, see “Increase Precision of Numeric Calculations” on page 2-116.

See Also

`solve` | `vpasolve`

Related Examples

- "Solve Algebraic Equation" on page 2-145
- "Solve Equations Numerically" on page 2-172
- "Solve System of Linear Equations" on page 2-169

Solve System of Algebraic Equations

This topic shows you how to solve a system of equations symbolically using Symbolic Math Toolbox. This toolbox offers both numeric and symbolic equation solvers. For a comparison of numeric and symbolic solvers, see "Select Numeric or Symbolic Solver" on page 2-151.

In this section...

Handle the Output of solve

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 a
```

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

```
[solx,soly] = solve(x^2*y^2 == 0, x-y/2 == a)
```

The call returns the following.

```
solx =
 0
 a
```

```
soly =
 -2*a
    0
```

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

```
[solx,soly] = solve(x^2*y^2 == 1, x-y/2 == a)
```

Four distinct solutions are produced.

```
solx =
 a/2 - (a^2 - 2)^(1/2)/2
 a/2 - (a^2 + 2)^(1/2)/2
 a/2 + (a^2 - 2)^(1/2)/2
 a/2 + (a^2 + 2)^(1/2)/2
soly =
 - a - (a^2 - 2)^(1/2)
 - a - (a^2 + 2)^(1/2)
   (a^2 - 2)^(1/2) - a
   (a^2 + 2)^(1/2) - a
```

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. For instance, if you have a 10-by-10 system of equations, typing the following is both awkward and time consuming.

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

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 =
  struct with fields:

    a: [2×1 sym]
    u: [2×1 sym]
    v: [2×1 sym]
```

The solutions for a reside in the “a-field” of S.

```
S.a
```

```
ans =
 -1
  3
```

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

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

```
s2 =
[  3,  5, -4]
```

The following statement creates the solution matrix M whose rows comprise the distinct solutions of the system.

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

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

Clear solx and soly for further use.

```
clear solx soly
```

Solve a Linear System of Equations

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

[A,b] = equationsToMatrix(eqns,x,y);
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.

Return the Full Solution of a System of Equations

`solve` does not automatically return all solutions of an equation. To return all solutions along with the parameters in the solution and the conditions on the solution, set the `ReturnConditions` option to `true`.

Consider the following system of equations:

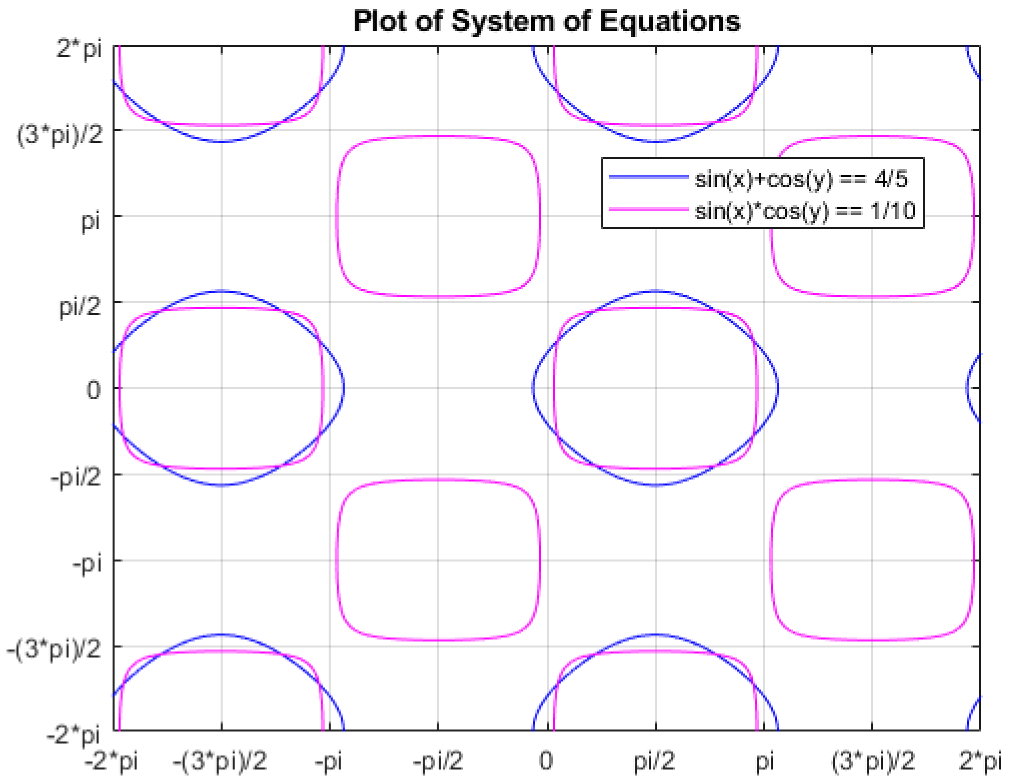
$$\sin(x) + \cos(y) = \frac{4}{5}$$

$$\sin(x)\cos(y) = \frac{1}{10}$$

Visualize the system of equations using `fimplicit`. To set the *x*-axis and *y*-axis values in terms of `pi`, get the axes handles using `axes` in `a`. Create the symbolic array `S` of the values `-2*pi` to `2*pi` at intervals of `pi/2`. To set the ticks to `S`, use the `XTick` and `YTick` properties of `a`. To set the labels for the x-and y-axes, convert `S` to character vectors. Use `arrayfun` to apply `char` to every element of `S` to return `T`. Set the `XTickLabel` and `YTickLabel` properties of `a` to `T`.

```
syms x y
eqn1 = sin(x)+cos(y) == 4/5;
eqn2 = sin(x)*cos(y) == 1/10;
a = axes;
fimplicit(eqn1,[-2*pi 2*pi],'b');
hold on
grid on
fimplicit(eqn2,[-2*pi 2*pi],'m');
L = sym(-2*pi:pi/2:2*pi);
a.XTick = double(L);
a.YTick = double(L);
```

```
M = arrayfun(@char, L, 'UniformOutput', false);
a.XTickLabel = M;
a.YTickLabel = M;
title('Plot of System of Equations')
legend('sin(x)+cos(y) == 4/5','sin(x)*cos(y) == 1/10',...
    'Location','best','AutoUpdate','off')
```



The solutions lie at the intersection of the two plots. This shows the system has repeated, periodic solutions. To solve this system of equations for the full solution set, use `solve` and set the `ReturnConditions` option to `true`.

```
S = solve(eqn1, eqn2, 'ReturnConditions', true)
```

```
S = 
  struct with fields:

             x: [2×1 sym]
             y: [2×1 sym]
    parameters: [1×2 sym]
    conditions: [2×1 sym]
```

`solve` returns a structure `S` with the fields `S.x` for the solution to `x`, `S.y` for the solution to `y`, `S.parameters` for the parameters in the solution, and `S.conditions` for the conditions on the solution. Elements of the same index in `S.x`, `S.y`, and `S.conditions` form a solution. Thus, `S.x(1)`, `S.y(1)`, and `S.conditions(1)` form one solution to the system of equations. The parameters in `S.parameters` can appear in all solutions.

Index into `S` to return the solutions, parameters, and conditions.

```
S.x
S.y
S.parameters
S.conditions
```

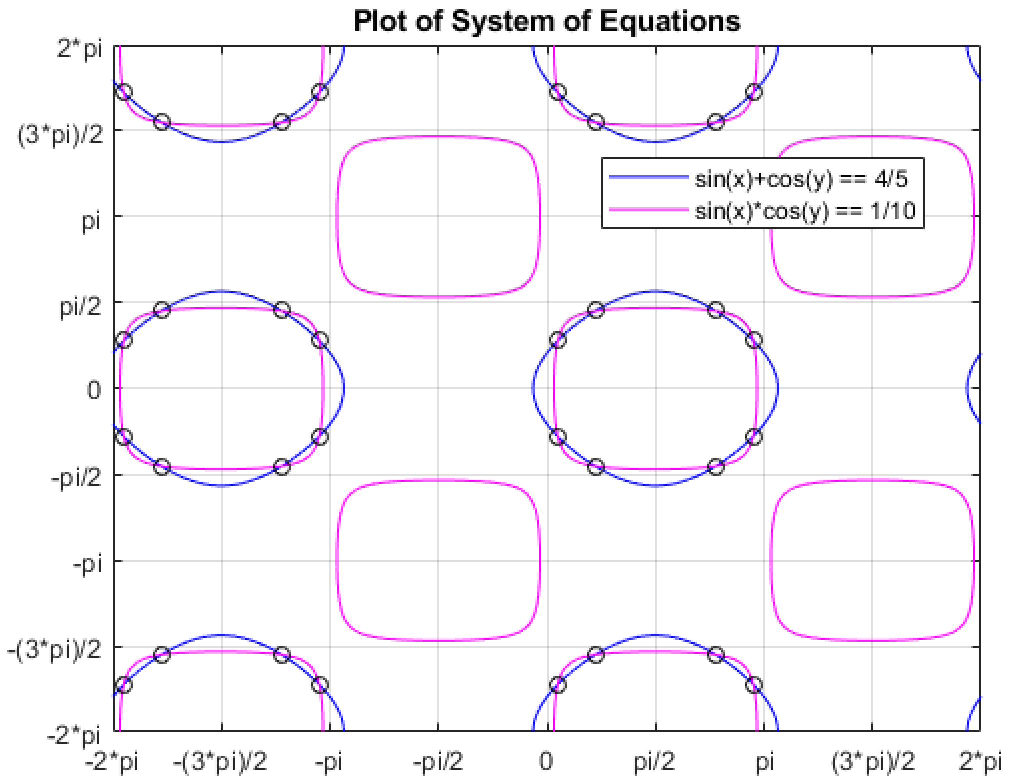
```
ans =
 z1
 z1
ans =
 z
 z
ans =
[ z, z1]
ans =
 (in((z - acos(6^(1/2)/10 + 2/5))/(2*pi), 'integer') |...
 in((z + acos(6^(1/2)/10 + 2/5))/(2*pi), 'integer')) &...
 (in(-(pi - z1 + asin(6^(1/2)/10 - 2/5))/(2*pi), 'integer') |...
 in((z1 + asin(6^(1/2)/10 - 2/5))/(2*pi), 'integer'))
  (in((z1 - asin(6^(1/2)/10 + 2/5))/(2*pi), 'integer') |...
 in((z1 - pi + asin(6^(1/2)/10 + 2/5))/(2*pi), 'integer')) &...
 (in((z - acos(2/5 - 6^(1/2)/10))/(2*pi), 'integer') |...
 in((z + acos(2/5 - 6^(1/2)/10))/(2*pi), 'integer'))
```

Solve a System of Equations Under Conditions

To solve the system of equations under conditions, specify the conditions in the input to `solve`.

Solve the system of equations considered above for x and y in the interval -2*pi to 2*pi. Overlay the solutions on the plot using scatter.

```
Srange = solve(eqn1, eqn2, -2*pi<x, x<2*pi, -2*pi<y, y<2*pi, 'ReturnConditions', true);
scatter(Srange.x, Srange.y,'k')
```



Work with Solutions, Parameters, and Conditions Returned by solve

You can use the solutions, parameters, and conditions returned by solve to find solutions within an interval or under additional conditions. This section has the same goal as the previous section, to solve the system of equations within a search range, but with a different approach. Instead of placing conditions directly, it shows how to work with the parameters and conditions returned by solve.

For the full solution S of the system of equations, find values of x and y in the interval -2*pi to 2*pi by solving the solutions S.x and S.y for the parameters S.parameters within that interval under the condition S.conditions.

Before solving for x and y in the interval, assume the conditions in S.conditions using assume so that the solutions returned satisfy the condition. Assume the conditions for the first solution.

```
assume(S.conditions(1))
```

Find the parameters in S.x and S.y.

```
paramx = intersect(symvar(S.x), S.parameters)
paramy = intersect(symvar(S.y), S.parameters)
```

```
paramx =
z1
paramy =
z
```

Solve the first solution of x for the parameter paramx.

```
solparamx(1,:) = solve(S.x(1) > -2*pi, S.x(1) < 2*pi, paramx)
```

```
solparamx =
[ pi + asin(6^(1/2)/10 - 2/5), asin(6^(1/2)/10 - 2/5) - pi,
 -asin(6^(1/2)/10 - 2/5), - 2*pi - asin(6^(1/2)/10 - 2/5)]
```

Similarly, solve the first solution of y for paramy.

```
solparamy(1,:) = solve(S.y(1) > -2*pi, S.y(1) < 2*pi, paramy)
```

```
solparamy =
[ acos(6^(1/2)/10 + 2/5), acos(6^(1/2)/10 + 2/5) - 2*pi,
 -acos(6^(1/2)/10 + 2/5), 2*pi - acos(6^(1/2)/10 + 2/5)]
```

Clear the assumptions set by S.conditions(1) using assume. Call asumptions to check that the assumptions are cleared.

```
assume(S.parameters,'clear')
assumptions
```

```
ans =
Empty sym: 1-by-0
```

Assume the conditions for the second solution.

```
assume(S.conditions(2))
```

Solve the second solution to x and y for the parameters paramx and paramy.

```
solparamx(2,:) = solve(S.x(2) > -2*pi, S.x(2) < 2*pi, paramx)
solparamy(2,:) = solve(S.y(2) > -2*pi, S.y(2) < 2*pi, paramy)
```

```
solparamx =
[ pi + asin(6^(1/2)/10 - 2/5), asin(6^(1/2)/10 - 2/5) - pi,
  -asin(6^(1/2)/10 - 2/5), - 2*pi - asin(6^(1/2)/10 - 2/5)]
[ asin(6^(1/2)/10 + 2/5), pi - asin(6^(1/2)/10 + 2/5),
  asin(6^(1/2)/10 + 2/5) - 2*pi, - pi - asin(6^(1/2)/10 + 2/5)]
solparamy =
[ acos(6^(1/2)/10 + 2/5), acos(6^(1/2)/10 + 2/5) - 2*pi,
  -acos(6^(1/2)/10 + 2/5), 2*pi - acos(6^(1/2)/10 + 2/5)]
[ acos(2/5 - 6^(1/2)/10), acos(2/5 - 6^(1/2)/10) - 2*pi,
  -acos(2/5 - 6^(1/2)/10), 2*pi - acos(2/5 - 6^(1/2)/10)]
```

The first rows of paramx and paramy form the first solution to the system of equations, and the second rows form the second solution.

To find the values of x and y for these values of paramx and paramy, use subs to substitute for paramx and paramy in S.x and S.y.

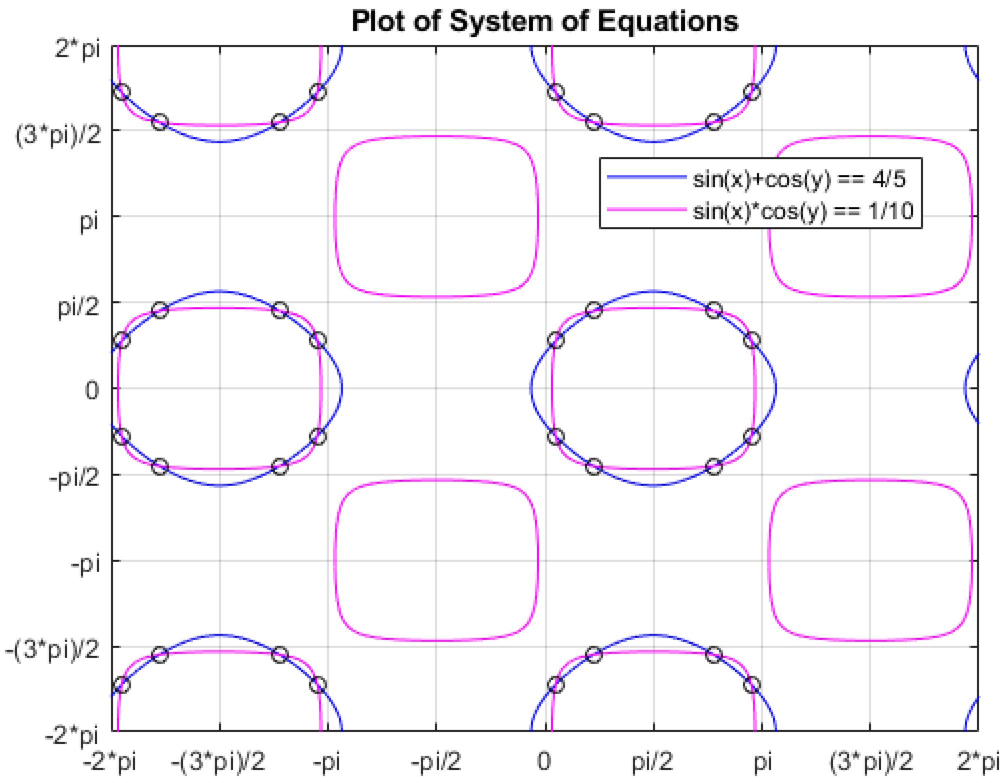
```
solx(1,:) = subs(S.x(1), paramx, solparamx(1,:));
solx(2,:) = subs(S.x(2), paramx, solparamx(2,:))
soly(1,:) = subs(S.y(1), paramy, solparamy(1,:));
soly(2,:) = subs(S.y(2), paramy, solparamy(2,:))
```

```
solx =
[ pi + asin(6^(1/2)/10 - 2/5), asin(6^(1/2)/10 - 2/5) - pi,
  -asin(6^(1/2)/10 - 2/5), - 2*pi - asin(6^(1/2)/10 - 2/5)]
[ asin(6^(1/2)/10 + 2/5), pi - asin(6^(1/2)/10 + 2/5),
  asin(6^(1/2)/10 + 2/5) - 2*pi,   - pi - asin(6^(1/2)/10 + 2/5)]
soly =
[ acos(6^(1/2)/10 + 2/5), acos(6^(1/2)/10 + 2/5) - 2*pi,
 -acos(6^(1/2)/10 + 2/5), 2*pi - acos(6^(1/2)/10 + 2/5)]
[ acos(2/5 - 6^(1/2)/10), acos(2/5 - 6^(1/2)/10) - 2*pi,
 -acos(2/5 - 6^(1/2)/10), 2*pi - acos(2/5 - 6^(1/2)/10)]
```

Note that solx and soly are the two sets of solutions to x and to y. The full sets of solutions to the system of equations are the two sets of points formed by all possible combinations of the values in solx and soly.

Plot these two sets of points using `scatter`. Overlay them on the plot of the equations. As expected, the solutions appear at the intersection of the plots of the two equations.

```
for i = 1:length(solx(1,:))
    for j = 1:length(soly(1,:))
        scatter(solx(1,i), soly(1,j), 'k')
        scatter(solx(2,i), soly(2,j), 'k')
    end
end
```



Convert Symbolic Results to Numeric Values

Symbolic calculations provide exact accuracy, while numeric calculations are approximations. Despite this loss of accuracy, you might need to convert symbolic results to numeric approximations for use in numeric calculations. For a high-accuracy conversion, use variable-precision arithmetic provided by the `vpa` function. For standard accuracy and better performance, convert to double precision using `double`.

Use `vpa` to convert the symbolic solutions `solx` and `soly` to numeric form.

```
vpa(solx)
vpa(soly)
```

```
ans =
[ 2.9859135500977407388300518406219,...
 -3.2972717570818457380952349259371,...
  0.15567910349205249963259154265761,...
 -6.1275062036875339772926952239014]
...
[ 0.70095651347102524787213653614929,...
  2.4406361401187679905905068471302,...
 -5.5822287937085612290531502304097,...
 -3.8425491670608184863347799194288]

ans =
[ 0.86983981332387137135918515549046,...
 -5.4133454938557151055661016110685,...
 -0.86983981332387137135918515549046,...
  5.4133454938557151055661016110685]
...
[ 1.4151172233028441195987301489821,...
 -4.8680680838767423573265566175769,...
 -1.4151172233028441195987301489821,...
  4.8680680838767423573265566175769]
```

Simplify Complicated Results and Improve Performance

If results look complicated, `solve` is stuck, or if you want to improve performance, see, “Troubleshoot Equation Solutions from solve Function” on page 2-164.

Troubleshoot Equation Solutions from solve Function

If `solve` returns solutions that look complicated, or if `solve` cannot handle an input, there are many options. These options simplify the solution space for `solve`. These options also help `solve` when the input is complicated, and might allow `solve` to return a solution where it was previously stuck.

In this section...
"Return Only Real Solutions" on page 2-164
"Apply Simplification Rules" on page 2-164
"Use Assumptions to Narrow Results" on page 2-165
"Simplify Solutions" on page 2-167
"Tips" on page 2-167

Return Only Real Solutions

Solve the equation `x^5 - 1 == 0`. This equation has five solutions.

```
syms x
solve(x^5 - 1 == 0, x)
```

```
ans =
                                                        1
 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 - 5^(1/2)/4 - 1/4
   (2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 - 5^(1/2)/4 - 1/4
   5^(1/2)/4 - (2^(1/2)*(5^(1/2) + 5)^(1/2)*1i)/4 - 1/4
   5^(1/2)/4 + (2^(1/2)*(5^(1/2) + 5)^(1/2)*1i)/4 - 1/4
```

If you only need real solutions, specify the `Real` option as `true`. The `solve` function returns the one real solution.

```
solve(x^5 - 1, x, 'Real', true)
```

```
ans =
1
```

Apply Simplification Rules

Solve the following equation. The `solve` function returns a complicated solution.

```
syms x
solve(x^(5/2) + 1/x^(5/2) == 1, x)
```

```
ans =
                                                       1/(1/2 - (3^(1/2)*1i)/2)^(2/5)
                                                       1/((3^(1/2)*1i)/2 + 1/2)^(2/5)
 -(5^(1/2)/4 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 + 1/4)/(1/2 - (3^(1/2)*1i)/2)^(2/5)
 -((2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 + 5^(1/2)/4 + 1/4)/(1/2 - (3^(1/2)*1i)/2)^(2/5)
 -(5^(1/2)/4 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 + 1/4)/(1/2 + (3^(1/2)*1i)/2)^(2/5)
 -((2^(1/2)*(5 - 5^(1/2))^(1/2)*1i)/4 + 5^(1/2)/4 + 1/4)/(1/2 + (3^(1/2)*1i)/2)^(2/5)
```

To apply simplification rules when solving equations, specify the `IgnoreAnalyticConstraints` option as `true`. The applied simplification rules are not generally correct mathematically but might produce useful solutions, especially in physics and engineering. With this option, the solver does not guarantee the correctness and completeness of the result.

```
solve(x^(5/2) + 1/x^(5/2) == 1, x, 'IgnoreAnalyticConstraints', true)
```

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

This solution is simpler and more usable.

Use Assumptions to Narrow Results

For solutions to specific cases, set assumptions to return appropriate solutions. Solve the following equation. The `solve` function returns seven solutions.

```
syms x
solve(x^7 + 2*x^6 - 59*x^5 - 106*x^4 + 478*x^3 + 284*x^2 - 1400*x + 800, x)
```

```
ans =
                  1
      - 5^(1/2) - 1
 - 17^(1/2)/2 - 1/2
   17^(1/2)/2 - 1/2
         -5*2^(1/2)
          5*2^(1/2)
        5^(1/2) - 1
```

Assume `x` is a positive number and solve the equation again. The `solve` function only returns the four positive solutions.

```
assume(x > 0)
solve(x^7 + 2*x^6 - 59*x^5 - 106*x^4 + 478*x^3 + 284*x^2 - 1400*x + 800, x)
```

```
ans =
                1
 17^(1/2)/2 - 1/2
        5*2^(1/2)
      5^(1/2) - 1
```

Place the additional assumption that `x` is an integer using `in(x,'integer')`. Place additional assumptions on variables using `assumeAlso`.

```
assumeAlso(in(x,'integer'))
solve(x^7 + 2*x^6 - 59*x^5 - 106*x^4 + 478*x^3 + 284*x^2 - 1400*x + 800, x)
```

```
ans =
1
```

`solve` returns the only positive, integer solution to `x`.

Clear the assumptions on `x` for further computations.

```
syms x clear
```

Alternatively, to make several assumptions, use the `&` operator. Make the following assumptions, and solve the following equations.

```
syms a b c f g h y
assume(f == c & a == h & a~= 0)
S = solve([a*x + b*y == c, h*x - g*y == f], [x, y], 'ReturnConditions', true);
S.x
S.y
S.conditions
```

```
ans =
f/h
ans =
0
ans =
b + g ~= 0
```

Under the specified assumptions, the solution is `x = f/h` and `y = 0` under the condition `b + g ~= 0`.

Clear the assumptions on the variables for further computations.

```
syms a c f h clear
```

Simplify Solutions

The `solve` function does not call simplification functions for the final results. To simplify the solutions, call `simplify`.

Solve the following equation. Convert the numbers to symbolic numbers using `sym` to return a symbolic result.

```
syms x
S = solve((sin(x) - 2*cos(x))/(sin(x) + 2*cos(x)) == 1/2, x)
```

```
S =
 -log(-(- 140/37 + 48i/37)^(1/2)/2)*1i
  -log((- 140/37 + 48i/37)^(1/2)/2)*1i
```

Call `simplify` to simplify solution `S`.

```
simplify(S)
```

```
ans =
        -log(37^(1/2)*(- 1/37 - 6i/37))*1i
 log(2)*1i - (log(- 140/37 + 48i/37)*1i)/2
```

Call `simplify` with more steps to simplify the result even further.

```
simplify(S, 'Steps', 50)
```

```
ans =
 atan(6) - pi
      atan(6)
```

Tips

- To represent a number exactly, use `sym` to convert the number to a floating-point object. For example, use `sym(13)/5` instead of `13/5`. This represents `13/5` exactly instead of converting `13/5` to a floating-point number. For a large number, place the number in quotes. Compare `sym(13)/5`, `sym(133333333333333333333)/5`, and `sym('133333333333333333333')/5`.

  ```
  sym(13)/5
  sym(133333333333333333333)/5
  sym('133333333333333333333')/5
  ```

```
ans =
13/5
ans =
133333333333333327872/5
ans =
133333333333333333333/5
```

Placing the number in quotes and using `sym` provides the highest accuracy.

- If possible, simplify the system of equations manually before using `solve`. Try to reduce the number of equations, parameters, and variables.

Solve System of Linear Equations

This section shows you how to solve a system of linear equations using the Symbolic Math Toolbox.

In this section...

Solve System of Linear Equations Using linsolve

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,

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

If you do not have the system of linear equations in the form `AX = B`, use `equationsToMatrix` to convert the equations into this form. Consider the following system.

$$\begin{aligned}
&2x + y + z = 2\\
&-x + y - z = 3\\
&x + 2y + 3z = -10
\end{aligned}$$

Declare the system of equations.

```
syms x y z
eqn1 = 2*x + y + z == 2;
eqn2 = -x + y - z == 3;
eqn3 = x + 2*y + 3*z == -10;
```

Use `equationsToMatrix` to convert the equations into the form `AX = B`. The second input to `equationsToMatrix` specifies the independent variables in the equations.

```
[A,B] = equationsToMatrix([eqn1, eqn2, eqn3], [x, y, z])
```

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

B =
   2
   3
 -10
```

Use `linsolve` to solve `AX = B` for the vector of unknowns `X`.

```
X = linsolve(A,B)
```

```
X =
  3
  1
 -5
```

From `X`, $x = 3$, $y = 1$ and $z = -5$.

Solve System of Linear Equations Using solve

Use `solve` instead of `linsolve` if you have the equations in the form of expressions and not a matrix of coefficients. Consider the same system of linear equations.

$$2x + y + z = 2$$

$$-x + y - z = 3$$

$$x + 2y + 3z = -10$$

Declare the system of equations.

```
syms x y z
eqn1 = 2*x + y + z == 2;
eqn2 = -x + y - z == 3;
eqn3 = x + 2*y + 3*z == -10;
```

Solve the system of equations using `solve`. The inputs to `solve` are a vector of equations, and a vector of variables to solve the equations for.

```
sol = solve([eqn1, eqn2, eqn3], [x, y, z]);
xSol = sol.x
ySol = sol.y
zSol = sol.z
```

```
xSol =
3
ySol =
1
zSol =
-5
```

`solve` returns the solutions in a structure array. To access the solutions, index into the array.

See Also

More About

- "Solve Algebraic Equation" on page 2-145
- "Solve System of Algebraic Equations" on page 2-153

Solve Equations Numerically

The Symbolic Math Toolbox offers both numeric and symbolic equation solvers. For a comparison of numeric and symbolic solvers, please see “Select Numeric or Symbolic Solver” on page 2-151. An equation or a system of equations can have multiple solutions. To find these solutions numerically, use the function `vpasolve`. For polynomial equations, `vpasolve` returns all solutions. For nonpolynomial equations, `vpasolve` returns the first solution it finds. This shows you how to use `vpasolve` to find solutions to both polynomial and nonpolynomial equations, and how to obtain these solutions to arbitrary precision.

In this section...

Find All Roots of a Polynomial Function

Use `vpasolve` to find all the solutions to function $f(x) = 6x^7 - 2x^6 + 3x^3 - 8$.

```
syms f(x)
f(x) = 6*x^7-2*x^6+3*x^3-8;
sol = vpasolve(f)
```

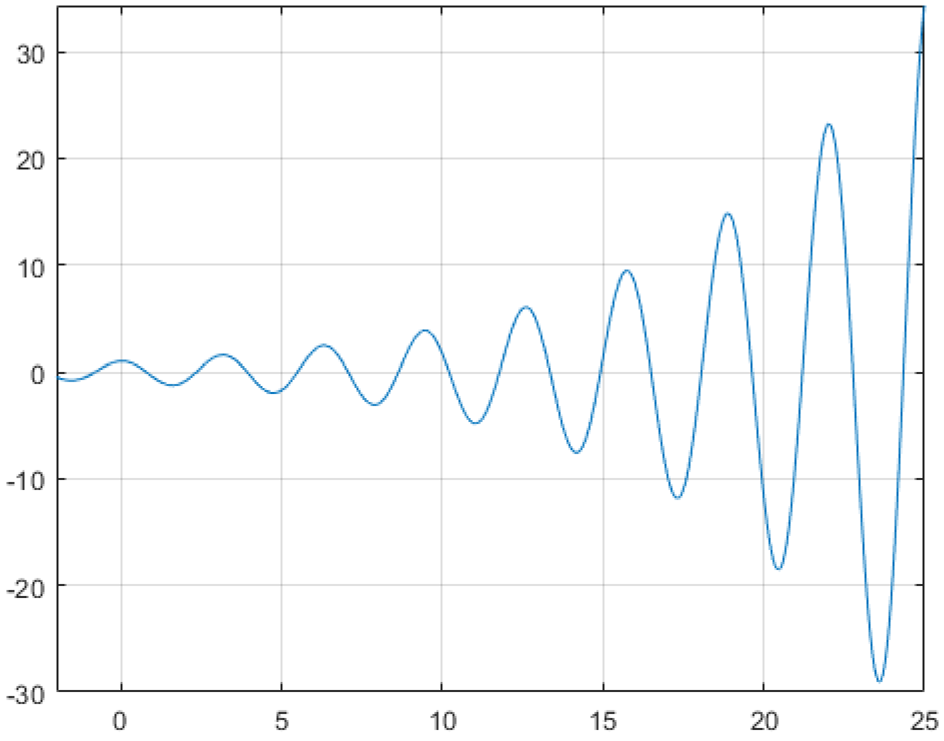
```
sol =
                                            1.0240240759053702941448316563337
 - 0.88080620051762149639205672298326 + 0.50434058840127584376331806592405i
 - 0.88080620051762149639205672298326 - 0.50434058840127584376331806592405i
 - 0.22974795226118163963098570610724 + 0.96774615576744031073999010695171i
 - 0.22974795226118163963098570610724 - 0.96774615576744031073999010695171i
   0.7652087814927846556172932675903 + 0.83187331431049713218367239317121i
   0.7652087814927846556172932675903 - 0.83187331431049713218367239317121i
```

`vpasolve` returns seven roots of the function, as expected, because the function is a polynomial of degree seven.

Find Zeros of a Nonpolynomial Function Using Search Ranges and Starting Points

Consider the function $f(x) = e^{(x/7)}\cos(2x)$. A plot of the function reveals periodic zeros, with increasing slopes at the zero points as x increases.

```
syms x
h = fplot(exp(x/7)*cos(2*x),[-2 25]);
grid on
```



Use vpasolve to find a zero of the function f. Note that vpasolve returns only one solution of a nonpolynomial equation, even if multiple solutions exist. On repeated calls, vpasolve returns the same result, even if multiple zeros exist.

```
f = exp(-x/20)*cos(2*x);
for i = 1:3
    vpasolve(f,x)
end
```

```
ans =
19.634954084936207740391521145497
ans =
19.634954084936207740391521145497
ans =
19.634954084936207740391521145497
```

To find multiple solutions, set the option random to true. This makes vpasolve choose starting points randomly. For information on the algorithm that chooses random starting points, see “Algorithms” on page 4-1718 on the vpasolve page.

```
for i = 1:3
        vpasolve(f,x,'random',true)
end
```

```
ans =
-226.98006922186256147892598444194
ans =
98.174770424681038701957605727484
ans =
58.904862254808623221174563436491
```

To find a zero close to $x = 10$ and to $x = 1000$, set the starting point to 10, and then to 1000.

```
vpasolve(f,x,10)
vpasolve(f,x,1000)
```

```
ans =
10.210176124166828025003590995658
ans =
999.81186200495169814073625672 87
```

To find a zero in the range $15 \le x \le 25$, set the search range to [15 25].

```
vpasolve(f,x,[15 25])
```

```
ans =
21.205750411731104359622842837137
```

To find multiple zeros in the range `[15 25]`, you cannot call `vpasolve` repeatedly as it returns the same result on each call, as previously shown. Instead, set `random` to `true` in conjunction with the search range.

```
for i = 1:3
vpasolve(f,x,[15 25],'random',true)
end

ans =
21.205750411731104359622842837137
ans =
16.493361431346414501928877762217
ans =
16.493361431346414501928877762217
```

If you specify the `random` option while also specifying a starting point, `vpasolve` warns you that the two options are incompatible.

```
vpasolve(f,x,15,'random',true)

Warning: 'Random' has no effect because
 all variables have a starting value.
> In sym/vpasolve (line 168)
ans =
14.922565104551517882697556070578
```

Create the function `findzeros` below to systematically find all zeros for `f` in a given search range, within the error tolerance. It starts with the input search range and calls `vpasolve` to find a zero. Then, it splits the search range into two around the zero's value, and recursively calls itself with the new search ranges as inputs to find more zeros. The first input is the function, the second input is the range, and the optional third input allows you to specify the error between a zero and the higher and lower bounds generated from it.

The function is explained section by section here.

Declare the function with the two inputs and one output.

```
function sol = findzeros(f,range,err)
```

If you do not specify the optional argument for error tolerance, `findzeros` sets `err` to `0.001`.

```
if nargin < 2
    err = 1e-3;
end
```

Find a zero in the search range using `vpasolve`.

```
sol = vpasolve(f,range);
```

If `vpasolve` does not find a zero, exit.

```
if(isempty(sol))
    return
```

If `vpasolve` finds a zero, split the search range into two search ranges above and below the zero.

```
else
    lowLimit = sol-err;
    highLimit = sol+err;
```

Call `findzeros` with the lower search range. If `findzeros` returns zeros, copy the values into the solution array and sort them.

```
    temp = findzeros(f,[range(1) lowLimit],1);
    if ~isempty(temp)
        sol = sort([sol temp]);
    end
```

Call `findzeros` with the higher search range. If `findzeros` returns zeros, copy the values into the solution array and sort them.

```
    temp = findzeros(f,[highLimit range(2)],1);
    if ~isempty(temp)
        sol = sort([sol temp]);
    end
    return
end
end
```

The entire function `findzeros` is as follows.

```
function sol = findzeros(f,range,err)
if nargin < 3
    err = 1e-3;
end
sol = vpasolve(f,range);
if(isempty(sol))
    return
else
    lowLimit = sol-err;
    highLimit = sol+err;
    temp = findzeros(f,[range(1) lowLimit],1);
    if ~isempty(temp)
        sol = sort([sol temp]);
    end
    temp = findzeros(f,[highLimit range(2)],1);
    if ~isempty(temp)
        sol = sort([sol temp]);
    end
    return
end
end
```

Call `findzeros` with search range `[10 20]` to find all zeros in that range for `f(x) = exp(-x/20)*cos(2*x)`, within the default error tolerance.

```
syms f(x)
f(x) = exp(-x/20)*cos(2*x);
findzeros(f,[10 20])
```

```
ans =
[ 10.210176124166828025003590995658, 11.780972450961724644234912687298,...
 13.351768777756621263466234378938, 14.922565104551517882697556070578,...
 16.493361431346414501928877762217, 18.064157758141311121160199453857,...
 19.634954084936207740391521145497]
```

Obtain Solutions to Arbitrary Precision

Use `digits` to set the precision of the solutions. By default, `vpasolve` returns solutions to a precision of 32 significant figures. Use `digits` to increase the precision to 64 significant figures. When modifying `digits`, ensure that you save its current value so that you can restore it.

```
f = exp(x/7)*cos(2*x);
vpasolve(f)
```

```
digitsOld = digits;
digits(64)
vpasolve(f)
digits(digitsOld)

ans =
-7.068583470577034786540947612378 9
ans =
-7.068583470577034786540947612378881489443631148593988097193625333
```

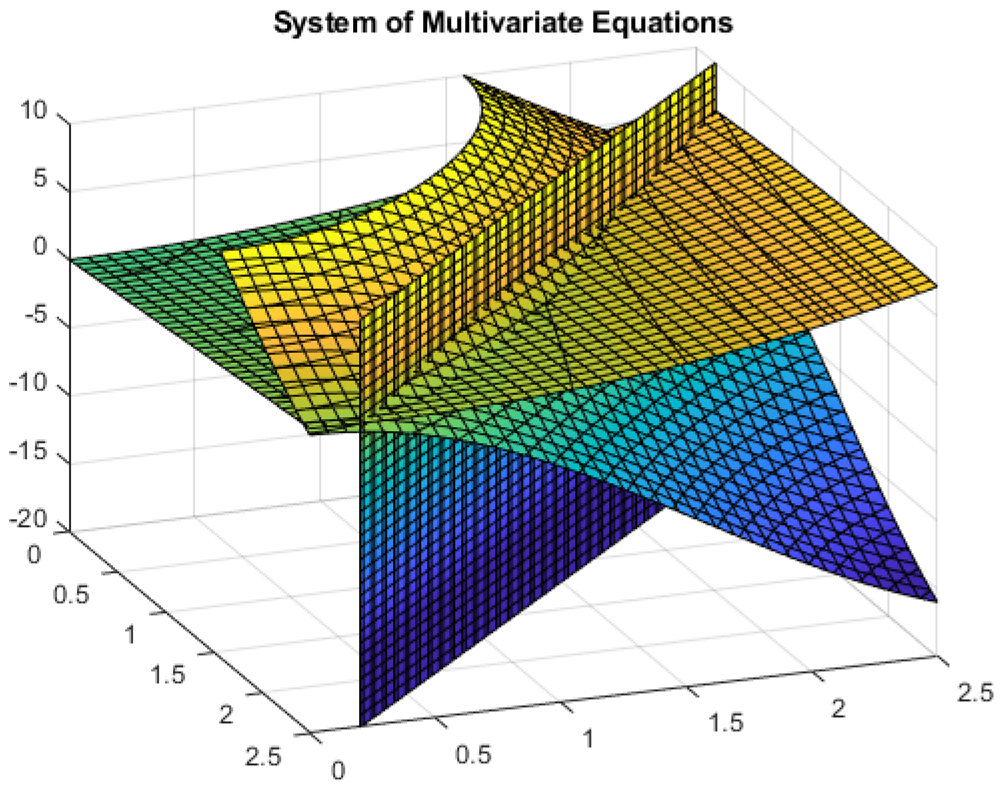
Solve Multivariate Equations Using Search Ranges

Consider the following system of equations.

$$z = 10(\cos(x) + \cos(y))$$
$$z = x + y^2 - 0.1x^2 y$$
$$x + y - 2.7 = 0$$

A plot of the equations for $0 \le x \le 2.5$ and $0 \le x \le 2.5$ shows that the three surfaces intersect in two points. To better visualize the plot, use `view`. To scale the colormap values, use `caxis`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y z
eqn1 = z == 10*(cos(x) + cos(y));
eqn2 = z == x+y^2-0.1*x^2*y;
eqn3 = x+y-2.7 == 0;
equations = [eqn1 eqn2 eqn3];
fimplicit3(equations)
axis([0 2.5 0 2.5 -20 10])
title('System of Multivariate Equations')
view(69, 28)
caxis([-15 10])
```



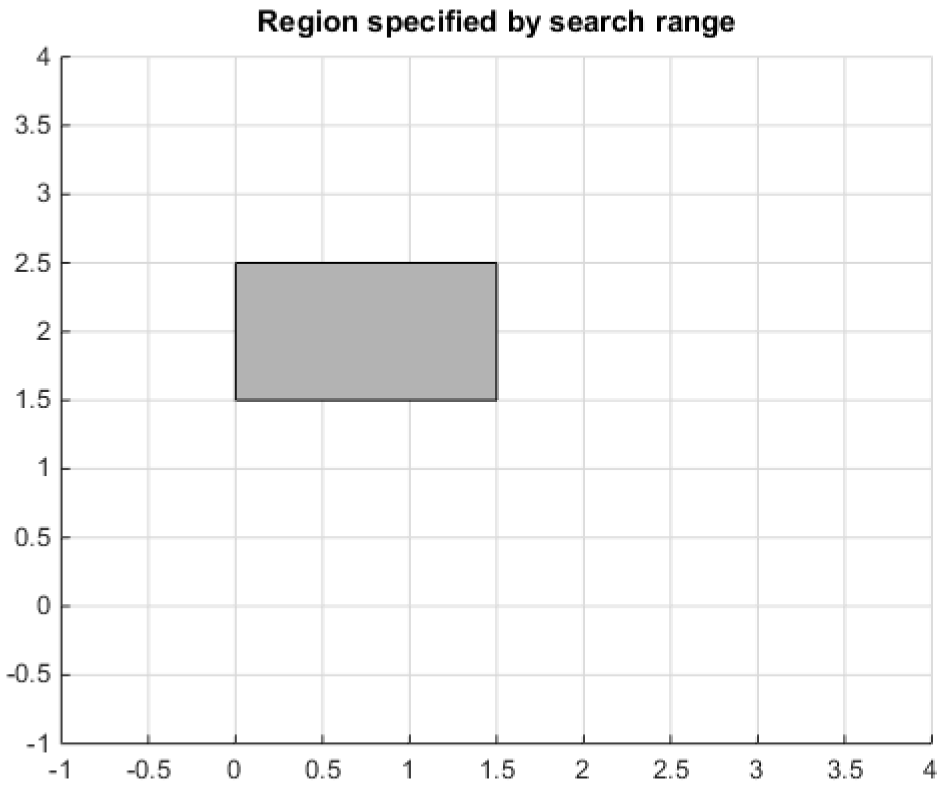
Use `vpasolve` to find a point where the surfaces intersect. The function `vpasolve` returns a structure. To access the solution, index into the structure.

```
sol = vpasolve(equations);
[sol.x sol.y sol.z]
```

```
ans =
```

$(2.3697477224547979209101337160174 \quad 0.33025227754520207908986628398261 \quad 2.2933543768232277431243854708612)$

To search a region of the solution space, specify search ranges for the variables. If you specify the ranges $0 \le x \le 1.5$ and $1.5 \le y \le 2.5$, then `vpasolve` function searches the bounded area shown in the picture.



Use `vpasolve` to find a solution for this search range $0 \le x \le 1.5$ and $1.5 \le y \le 2.5$. To omit a search range for `z`, set the search range to `[NaN NaN]`.

```
vars = [x y z];
range = [0 1.5; 1.5 2.5; NaN NaN];
sol = vpasolve(equations, vars, range);
[sol.x sol.y sol.z]
```

```
ans =
```

$$\left(0.91062661725633361176950031551069 \quad 1.7893733827436663882304996844893 \quad 3.9641015721356254724107884666807\right)$$

To find multiple solutions, you can set the `random` option to `true`. This makes `vpasolve` use random starting points on successive runs. The `random` option can be used in conjunction with search ranges to make `vpasolve` use random starting points within a search range. Because `random` selects starting points randomly, the same solution might be found on successive calls. Call `vpasolve` repeatedly to ensure you find both solutions.

```
clear sol
range = [0 3; 0 3; NaN NaN];
for i = 1:5
    temp = vpasolve(equations, vars, range, 'random', true);
    sol(i,1) = temp.x;
    sol(i,2) = temp.y;
    sol(i,3) = temp.z;
end
sol
```

```
sol =
```

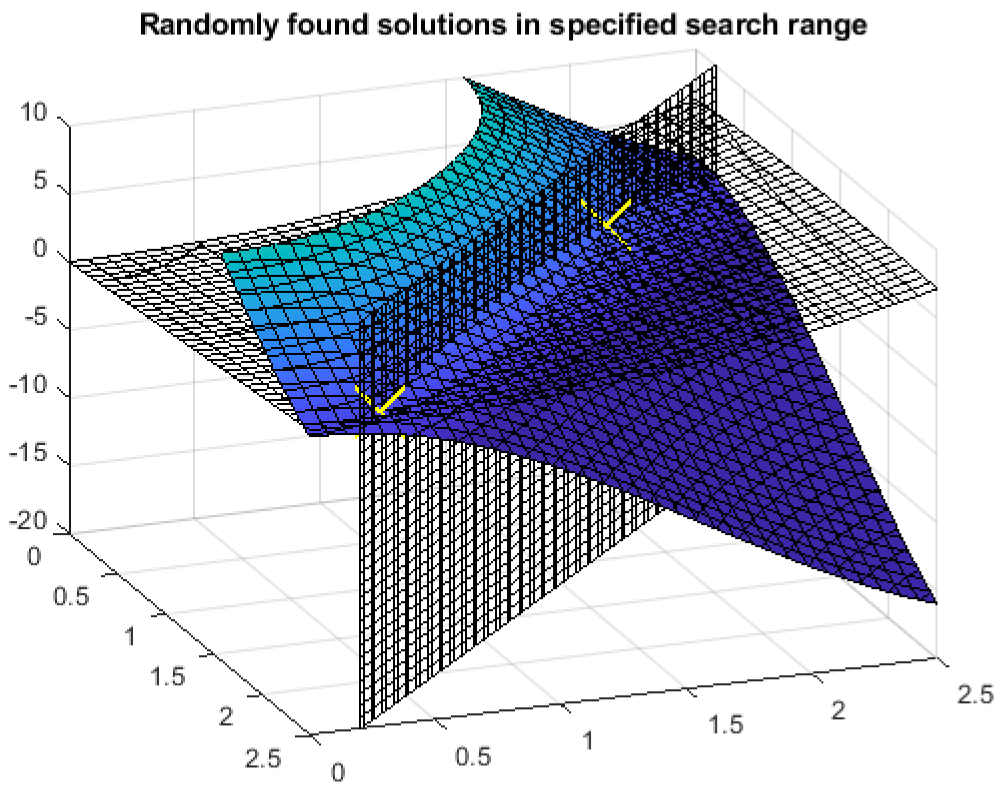
$$\begin{pmatrix} 0.91062661725633361176950031551069 & 1.7893733827436663882304996844893 & 3.9641015721356254724107884666807 \\ 2.3697477224547979209101337160174 & 0.33025227754520207908986628398261 & 2.2933543768232277431243854708612 \\ 0.91062661725633361176950031551069 & 1.7893733827436663882304996844893 & 3.9641015721356254724107884666807 \\ 0.91062661725633361176950031551069 & 1.7893733827436663882304996844893 & 3.9641015721356254724107884666807 \\ 0.91062661725633361176950031551069 & 1.7893733827436663882304996844893 & 3.9641015721356254724107884666807 \end{pmatrix}$$

Plot the equations. Superimpose the solutions as a scatter plot of points with yellow `X` markers using `scatter3`. To better visualize the plot, make two of the surfaces transparent using `alpha`. Scale the colormap to the plot values using `caxis`, and change the perspective using `view`.

`vpasolve` finds solutions at the intersection of the surfaces formed by the equations as shown.

```
clf
ax = axes;
h = fimplicit3(equations);
h(2).FaceAlpha = 0;
h(3).FaceAlpha = 0;
axis([0 2.5 0 2.5 -20 10])
hold on
scatter3(sol(:,1),sol(:,2),sol(:,3),600,'yellow','X','LineWidth',2)
title('Randomly found solutions in specified search range')
cz = ax.Children;
caxis([0 20])
```

```
view(69,28)
hold off
```



Solve Differential Equation

Solve a differential equations by using the `dsolve` function, with or without initial conditions. This page shows how to solve single differential equations. To solve a system of differential equations, see "Solve a System of Differential Equations" on page 2-187.

In this section...

First-Order Linear ODE with Initial Condition

Solve this differential equation.

$$\frac{dy}{dt} = ty.$$

First, represent y by using `syms` to create the symbolic function `y(t)`.

```
syms y(t)
```

Define the equation using == and represent differentiation using the `diff` function.

```
ode = diff(y,t) == t*y
```

```
ode(t) =
diff(y(t), t) == t*y(t)
```

Solve the equation using `dsolve`.

```
ySol(t) = dsolve(ode)
```

```
ySol(t) =
C1*exp(t^2/2)
```

The constant `C1` appears because no condition was specified. Solve the equation with the initial condition `y(0) == 2`. The `dsolve` function finds a value of `C1` that satisfies the condition.

```
cond = y(0) == 2;
ySol(t) = dsolve(ode,cond)
```

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

Nonlinear Differential Equation with Initial Condition

Solve this nonlinear differential equation with an initial condition. The equation has multiple solutions.

$$\left(\frac{dy}{dt}+y\right)^2=1,$$
$$y(0)=0.$$

```
syms y(t)
ode = (diff(y,t)+y)^2 == 1;
cond = y(0) == 0;
ySol(t) = dsolve(ode,cond)
```

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

Second-Order ODE with Initial Conditions

Solve this second-order differential equation with two initial conditions.

$$\frac{d^2y}{dx^2}=\cos(2x)-y,$$
$$y(0)=1,$$
$$y'(0)=0.$$

Define the equation and conditions. The second initial condition involves the first derivative of `y`. Represent the derivative by creating the symbolic function `Dy = diff(y)` and then define the condition using `Dy(0)==0`.

```
syms y(x)
Dy = diff(y);

ode = diff(y,x,2) == cos(2*x)-y;
cond1 = y(0) == 1;
cond2 = Dy(0) == 0;
```

Solve `ode` for `y`. Simplify the solution using the `simplify` function.

```
conds = [cond1 cond2];
ySol(x) = dsolve(ode,conds);
ySol = simplify(ySol)
```

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

Third-Order ODE with Initial Conditions

Solve this third-order differential equation with three initial conditions.

$$\frac{d^3u}{dx^3} = u,$$
$$u(0) = 1,$$
$$u'(0) = -1,$$
$$u''(0) = \pi.$$

Because the initial conditions contain the first- and second-order derivatives, create two symbolic functions, `Du = diff(u,x)` and `D2u = diff(u,x,2)`, to specify the initial conditions.

```
syms u(x)
Du = diff(u,x);
D2u = diff(u,x,2);
```

Create the equation and initial conditions, and solve it.

```
ode = diff(u,x,3) == u;
cond1 = u(0) == 1;
cond2 = Du(0) == -1;
cond3 = D2u(0) == pi;
conds = [cond1 cond2 cond3];

uSol(x) = dsolve(ode,conds)
uSol(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 Commands

$\frac{dy}{dt}+4y(t)=e^{-t},$
$y(0)=1.$
	`syms y(t)`
	`ode = diff(y)+4*y == exp(-t);`
	`cond = y(0) == 1;`
	`ySol(t) = dsolve(ode,cond)`
	
	`ySol(t) =`
	`exp(-t)/3 + (2*exp(-4*t))/3`

$2x^2\frac{d^2y}{dx^2}+3x\frac{dy}{dx}-y=0.$	`syms y(x)`
	`ode = 2*x^2*diff(y,x,2)+3*x*diff(y,x)-y == 0;`
	`ySol(x) = dsolve(ode)`
	
	`ySol(x) =`
	`C2/(3*x) + C3*x^(1/2)`

The Airy equation.
$\frac{d^2y}{dx^2}=xy(x).$
	`syms y(x)`
	`ode = diff(y,x,2) == x*y;`
	`ySol(x) = dsolve(ode)`
	
	`ySol(x) =`
	`C1*airy(0,x) + C2*airy(2,x)`

See Also

"Solve a System of Differential Equations" on page 2-187

Solve a System of Differential Equations

Solve a system of several ordinary differential equations in several variables by using the `dsolve` function, with or without initial conditions. To solve a single differential equation, see "Solve Differential Equation" on page 2-183.

In this section...

Solve System of Differential Equations

Solve this system of linear first-order differential equations.

$$\frac{du}{dt} = 3u + 4v,$$
$$\frac{dv}{dt} = -4u + 3v.$$

First, represent u and v by using `syms` to create the symbolic functions `u(t)` and `v(t)`.

```
syms u(t) v(t)
```

Define the equations using == and represent differentiation using the `diff` function.

```
ode1 = diff(u) == 3*u + 4*v;
ode2 = diff(v) == -4*u + 3*v;
odes = [ode1; ode2]
```

```
odes(t) =
 diff(u(t), t) == 3*u(t) + 4*v(t)
 diff(v(t), t) == 3*v(t) - 4*u(t)
```

Solve the system using the `dsolve` function which returns the solutions as elements of a structure.

```
S = dsolve(odes)
```

```
S =
  struct with fields:
    v: [1×1 sym]
    u: [1×1 sym]
```

To access `u(t)` and `v(t)`, index into the structure `S`.

```
uSol(t) = S.u
vSol(t) = S.v
```

```
uSol(t) =
C2*cos(4*t)*exp(3*t) + C1*sin(4*t)*exp(3*t)
vSol(t) =
C1*cos(4*t)*exp(3*t) - C2*sin(4*t)*exp(3*t)
```

Alternatively, store `u(t)` and `v(t)` directly by providing multiple output arguments.

```
[uSol(t), vSol(t)] = dsolve(odes)
```

```
uSol(t) =
C2*cos(4*t)*exp(3*t) + C1*sin(4*t)*exp(3*t)
vSol(t) =
C1*cos(4*t)*exp(3*t) - C2*sin(4*t)*exp(3*t)
```

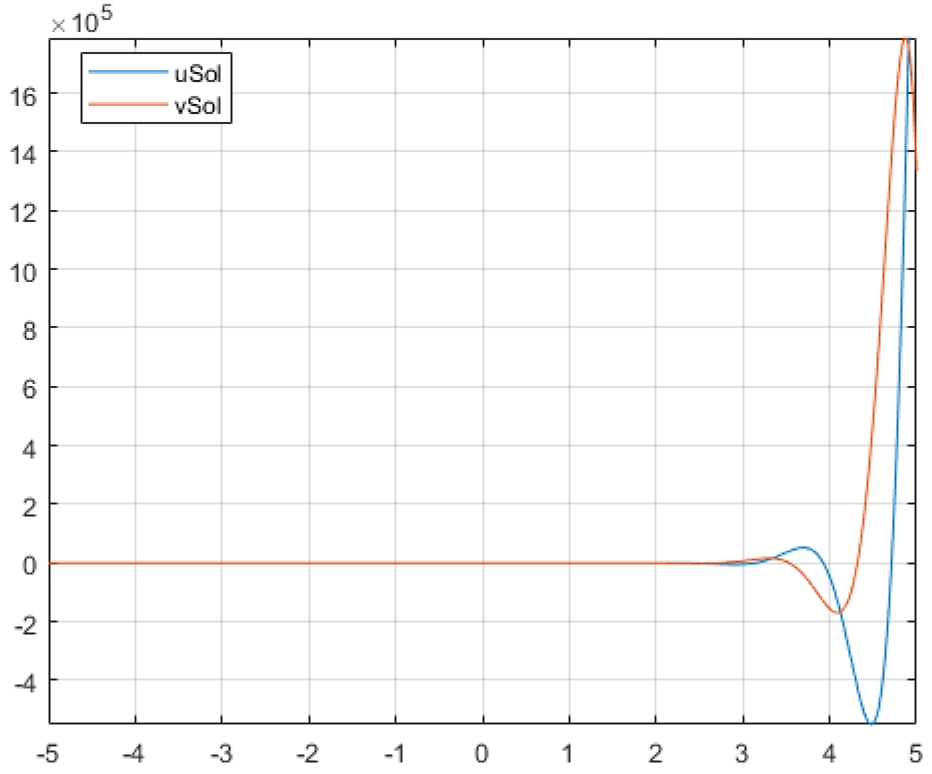
The constants `C1` and `C2` appear because no conditions are specified. Solve the system with the initial conditions `u(0) == 0` and `v(0) == 0`. The `dsolve` function finds values for the constants that satisfy these conditions.

```
cond1 = u(0) == 0;
cond2 = v(0) == 1;
conds = [cond1; cond2];
[uSol(t), vSol(t)] = dsolve(odes,conds)
```

```
uSol(t) =
sin(4*t)*exp(3*t)
vSol(t) =
cos(4*t)*exp(3*t)
```

Visualize the solution using `fplot`. Before R2016a, use `ezplot` instead.

```
fplot(uSol)
hold on
fplot(vSol)
grid on
legend('uSol','vSol','Location','best')
```



Solve Differential Equations in Matrix Form

Solve differential equations in matrix form by using `dsolve`.

Consider this system of differential equations.

$$\frac{dx}{dt} = x + 2y + 1,$$

$$\frac{dy}{dt} = -x + y + t.$$

The matrix form of the system is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ t \end{bmatrix}.$$

Let

$$Y = \begin{bmatrix} x \\ y \end{bmatrix}, A = \begin{bmatrix} 1 & 2 \\ -1 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 \\ t \end{bmatrix}.$$

The system is now $Y' = AY + B.$

Define these matrices and the matrix equation.

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

```
odes(t) =
  diff(x(t), t) == x(t) + 2*y(t) + 1
   diff(y(t), t) == t - x(t) + y(t)
```

Solve the matrix equation using `dsolve`. Simplify the solution by using the `simplify` function.

```
[xSol(t), ySol(t)] = dsolve(odes);
xSol(t) = simplify(xSol(t))
ySol(t) = simplify(ySol(t))
```

```
xSol(t) =
(2*t)/3 + 2^(1/2)*C2*exp(t)*cos(2^(1/2)*t) + 2^(1/2)*C1*exp(t)*sin(2^(1/2)*t) + 1/9
ySol(t) =
C1*exp(t)*cos(2^(1/2)*t) - t/3 - C2*exp(t)*sin(2^(1/2)*t) - 2/9
```

The constants `C1` and `C2` appear because no conditions are specified.

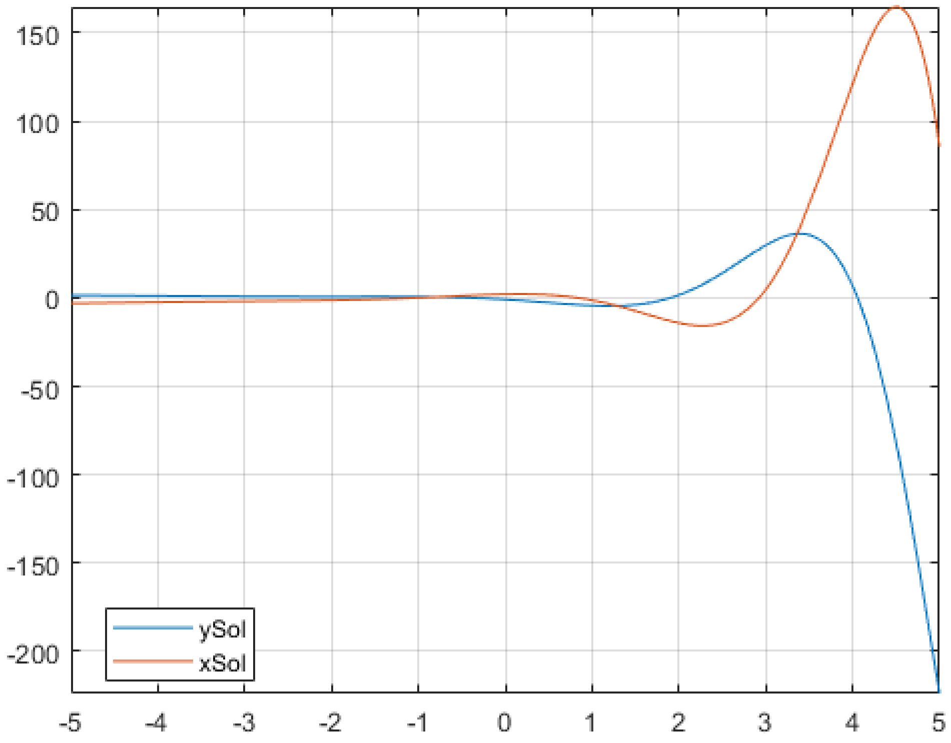
Solve the system with the initial conditions $u(0) = 2$ and $v(0) = –1$. When specifying equations in matrix form, you must specify initial conditions in matrix form too. `dsolve` finds values for the constants that satisfy these conditions.

```
C = Y(0) == [2; -1];
[xSol(t), ySol(t)] = dsolve(odes,C)
```

```
xSol(t) =
(2*t)/3 + (17*exp(t)*cos(2^(1/2)*t))/9 - (7*2^(1/2)*exp(t)*sin(2^(1/2)*t))/9 + 1/9
ySol(t) =
- t/3 - (7*exp(t)*cos(2^(1/2)*t))/9 - (17*2^(1/2)*exp(t)*sin(2^(1/2)*t))/18 - 2/9
```

Visualize the solution using `fplot`. Before R2016a, use `ezplot` instead.

```
clf
fplot(ySol)
hold on
fplot(xSol)
grid on
legend('ySol','xSol','Location','best')
```



See Also

"Solve Differential Equation" on page 2-183

Solve Differential Algebraic Equations (DAEs)

You can solve differential algebraic equations (DAEs) by using MATLAB and Symbolic Math Toolbox.

Differential algebraic equations (involving functions, or state variables,

$x(t) = [x_1(t),\ldots,x_n(t)]$ have the form

$F\left(t, x(t), \dot{x}(t)\right) = 0$

where t is the independent variable. The number of equations $F = [F_1,\ldots,F_n]$ must match the number of state variables $x(t) = [x_1(t),\ldots,x_n(t)]$.

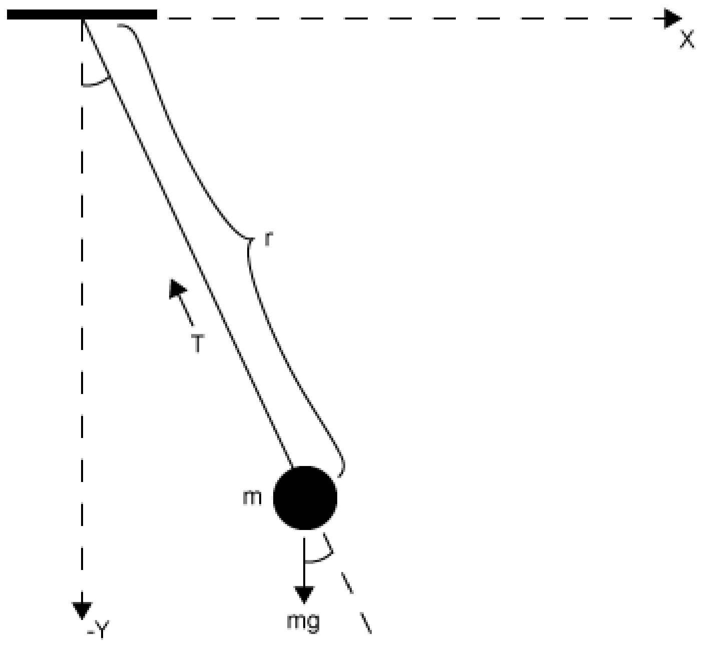
Because most DAE systems are not suitable for direct input to MATLAB solvers, such as `ode15i`, first convert them to a suitable form by using Symbolic Math Toolbox functionality. This functionality reduces the differential index (*number of differentiations needed to reduce the system to ODEs*) of the DAEs to 1 or 0, and then converts the DAE system to numeric function handles suitable for MATLAB solvers. Now, use MATLAB solvers, such as `ode15i`, `ode15s`, or `ode23t`, to solve the DAEs.

Solve your DAE system by completing these steps.

- “Step 1. Specify Equations and Variables” on page 2-193
- “Step 2. Reduce Differential Order” on page 2-195
- “Step 3. Check and Reduce Differential Index” on page 2-197
- “Step 4. Convert DAE Systems to MATLAB Function Handles” on page 2-199
- “Step 5. Find Initial Conditions For Solvers” on page 2-199
- “Step 6. Solve DAEs Using ode15i” on page 2-201

Step 1. Specify Equations and Variables

This shows the DAE workflow by solving the DAEs for a pendulum.



The state variables are:

- Horizontal position of pendulum $x(t)$
- Vertical position of pendulum $y(t)$
- Force preventing pendulum from flying away $T(t)$

The variables are:

- Pendulum mass m
- Pendulum length r
- Gravitational constant g

The DAE system of equations is

$$m\frac{d^2x}{dt^2} = T(t)\frac{x(t)}{r}$$
$$m\frac{d^2y}{dt^2} = T(t)\frac{y(t)}{r} - mg$$
$$x(t)^2 + y(t)^2 = r^2$$

Specify independent variables and state variables by using `syms`.

```
syms x(t) y(t) T(t) m r g
```

Specify equations by using the == operator.

```
eqn1 = m*diff(x(t), 2) == T(t)/r*x(t);
eqn2 = m*diff(y(t), 2) == T(t)/r*y(t) - m*g;
eqn3 = x(t)^2 + y(t)^2 == r^2;
eqns = [eqn1 eqn2 eqn3];
```

Place the state variables in a column vector. Store the number of original variables for reference.

```
vars = [x(t); y(t); T(t)];
origVars = length(vars);
```

Step 2. Reduce Differential Order

2.1 (Optional) Check Incidence of Variables

This step is *optional*. You can check where variables occur in the DAE system by viewing the incidence matrix. This step finds any variables that do not occur in your input and can be removed from the `vars` vector.

Display the incidence matrix by using `incidenceMatrix`. The output of `incidenceMatrix` has a row for each equation and a column for each variable. Because the system has three equations and three state variables, `incidenceMatrix` returns a `3`-by-`3` matrix. The matrix has `1`s and `0`s, where `1`s represent the occurrence of a state variable. For example, the `1` in position `(2,3)` means the second equation contains the third state variable `T(t)`.

```
M = incidenceMatrix(eqns, vars)
```

```
M =
     1     0     1
     0     1     1
     1     1     0
```

If a column of the incidence matrix is all `0`s, then that state variable does not occur in the DAE system and should be removed.

2.2 Reduce Differential Order

The *differential order* of a DAE system is the highest differential order of its equations. To solve DAEs using MATLAB, the differential order must be reduced to `1`. Here, the first and second equations have second-order derivatives of `x(t)` and `y(t)`. Thus, the differential order is `2`.

Reduce the system to a first-order system by using `reduceDifferentialOrder`. The `reduceDifferentialOrder` function substitutes derivatives with new variables, such as `Dxt(t)` and `Dyt(t)`. The right side of the expressions in `eqns` is `0`.

```
[eqns, vars] = reduceDifferentialOrder(eqns, vars)
```

```
eqns =
       m*diff(Dxt(t), t) - (T(t)*x(t))/r
 g*m + m*diff(Dyt(t), t) - (T(t)*y(t))/r
                   x(t)^2 + y(t)^2 - r^2
                  Dxt(t) - diff(x(t), t)
                  Dyt(t) - diff(y(t), t)


vars =
   x(t)
   y(t)
   T(t)
 Dxt(t)
 Dyt(t)
```

Optionally, to show how the derivatives correspond to the new variables, call `reduceDifferentialOrder` with three output arguments. To find the locations of the new variables in the system, use `incidenceMatrix`.

Step 3. Check and Reduce Differential Index

3.1 Check Differential Index of System

Check the differential index of the DAE system by using `isLowIndexDAE`. If the index is `0` or `1`, then `isLowIndexDAE` returns logical `1` (`true`) and you can skip step 3.2 and go to “Step 4. Convert DAE Systems to MATLAB Function Handles” on page 2-199. Here, `isLowIndexDAE` returns logical `0` (`false`), which means the differential index is greater than `1` and must be reduced.

```
isLowIndexDAE(eqns,vars)
```

```
ans =
  logical
     0
```

3.2 Reduce Differential Index with reduceDAEIndex

To reduce the differential index, the `reduceDAEIndex` function adds new equations that are derived from the input equations, and then replaces higher-order derivatives with new variables. If `reduceDAEIndex` fails and issues a warning, then use the alternative function `reduceDAEToODE` as described in the workflow “Solve Semilinear DAE System” on page 2-205.

Reduce the differential index of the DAEs described by `eqns` and `vars`.

```
[DAEs,DAEvars] = reduceDAEIndex(eqns,vars)
```

```
DAEs =
                                                      m*Dxtt(t) - (T(t)*x(t))/r
                                                g*m + m*Dytt(t) - (T(t)*y(t))/r
                                                         x(t)^2 + y(t)^2 - r^2
                                                              Dxt(t) - Dxt1(t)
                                                              Dyt(t) - Dyt1(t)
                                               2*Dxt1(t)*x(t) + 2*Dyt1(t)*y(t)
 2*Dxt1t(t)*x(t) + 2*Dxt1(t)^2 + 2*Dyt1(t)^2 + 2*y(t)*diff(Dyt1(t), t)
                                                            Dxtt(t) - Dxt1t(t)
                                                     Dytt(t) - diff(Dyt1(t), t)
                                                        Dyt1(t) - diff(y(t), t)


DAEvars =
     x(t)
     y(t)
     T(t)
   Dxt(t)
   Dyt(t)
  Dytt(t)
  Dxtt(t)
  Dxt1(t)
  Dyt1(t)
 Dxt1t(t)
```

Note If `reduceDAEIndex` fails and issues a warning, use the alternative workflow described in “Solve Semilinear DAE System” on page 2-205.

Often, `reduceDAEIndex` introduces redundant equations and variables that can be eliminated. Eliminate redundant equations and variables using `reduceRedundancies`.

```
[DAEs,DAEvars] = reduceRedundancies(DAEs,DAEvars)
```

```
DAEs =
                               -(T(t)*x(t) - m*r*Dxtt(t))/r
                        (g*m*r - T(t)*y(t) + m*r*Dytt(t))/r
                                      x(t)^2 + y(t)^2 - r^2
                                 2*Dxt(t)*x(t) + 2*Dyt(t)*y(t)
 2*Dxtt(t)*x(t) + 2*Dytt(t)*y(t) + 2*Dxt(t)^2 + 2*Dyt(t)^2
                                 Dytt(t) - diff(Dyt(t), t)
                                    Dyt(t) - diff(y(t), t)
DAEvars =
    x(t)
    y(t)
    T(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Check the differential index of the new system. Now, `isLowIndexDAE` returns logical `1` (`true`), which means that the differential index of the system is `0` or `1`.

```
isLowIndexDAE(DAEs,DAEvars)
```

```
ans =
  logical
     1
```

Step 4. Convert DAE Systems to MATLAB Function Handles

This step creates function handles for the MATLAB ODE solver `ode15i`, which is a general purpose solver. To use specialized mass matrix solvers such as `ode15s` and `ode23t`, see "Solve DAEs Using Mass Matrix Solvers" on page 2-213. Also see "Choose an ODE Solver" (MATLAB).

`reduceDAEIndex` outputs a vector of equations in `DAEs` and a vector of variables in `DAEvars`. To use `ode15i`, you need a function handle that describes the DAE system.

First, the equations in `DAEs` can contain symbolic parameters that are not specified in the vector of variables `DAEvars`. Find these parameters by using `setdiff` on the output of `symvar` from `DAEs` and `DAEvars`.

```
pDAEs = symvar(DAEs);
pDAEvars = symvar(DAEvars);
extraParams = setdiff(pDAEs, pDAEvars)

extraParams =
[ g, m, r]
```

The extra parameters that you need to specify are the mass `m`, radius `r`, and gravitational constant `g`.

Create the function handle by using `daeFunction`. Specify the extra symbolic parameters as additional input arguments of `daeFunction`.

```
f = daeFunction(DAEs, DAEvars, g, m, r);
```

The rest of the workflow is purely numerical. Set the parameter values and create the function handle for `ode15i`.

```
g = 9.81;
m = 1;
r = 1;
F = @(t, Y, YP) f(t, Y, YP, g, m, r);
```

Step 5. Find Initial Conditions For Solvers

The `ode15i` solver requires initial values for all variables in the function handle. Find initial values that satisfy the equations by using the MATLAB `decic` function. `decic` accepts guesses (which might not satisfy the equations) for the initial conditions and tries to find satisfactory initial conditions using those guesses. `decic` can fail, in which case you must manually supply consistent initial values for your problem.

First, check the variables in `DAEvars`.

```
DAEvars
```

```
DAEvars =
    x(t)
    y(t)
    T(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Here, `Dxt(t)` is the first derivative of `x(t)`, `Dytt(t)` is the second derivative of `y(t)`, and so on. There are 7 variables in a `7`-by-`1` vector. Therefore, guesses for initial values of variables and their derivatives must also be `7`-by-`1` vectors.

Assume the initial angular displacement of the pendulum is 30° or `pi/6`, and the origin of the coordinates is at the suspension point of the pendulum. Given that we used a radius `r` of `1`, the initial horizontal position `x(t)` is `r*sin(pi/6)`. The initial vertical position `y(t)` is `-r*cos(pi/6)`. Specify these initial values of the variables in the vector `y0est`.

Arbitrarily set the initial values of the remaining variables and their derivatives to `0`. These are not good guesses. However, they suffice for this problem. In your problem, if `decic` errors, then provide better guesses and refer to `decic`.

```
y0est = [r*sin(pi/6); -r*cos(pi/6); 0; 0; 0; 0; 0];
yp0est = zeros(7,1);
```

Create an option set that specifies numerical tolerances for the numerical search.

```
opt = odeset('RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

Find consistent initial values for the variables and their derivatives by using `decic`.

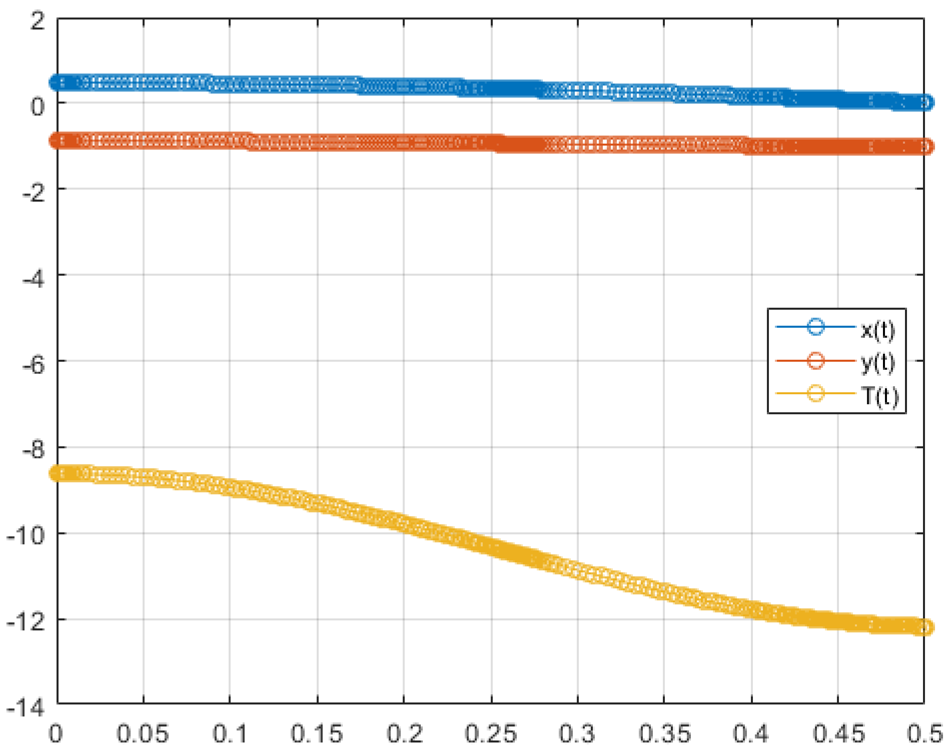
```
[y0, yp0] = decic(F, 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4771
   -0.8788
   -8.6214
         0
    0.0000
   -2.2333
   -4.1135

yp0 =
         0
    0.0000
         0
         0
   -2.2333
         0
         0
```

Step 6. Solve DAEs Using ode15i

Solve the system integrating over the time span $0 \leq t \leq 0.5$. Add the grid lines and the legend to the plot.

```
[tSol,ySol] = ode15i(F, [0, 0.5], y0, yp0, opt);
plot(tSol,ySol(:,1:origVars),'-o')

for k = 1:origVars
  S{k} = char(DAEvars(k));
end

legend(S, 'Location', 'Best')
grid on
```



Solve the system for different parameter values by setting the new value and regenerating the function handle and initial conditions.

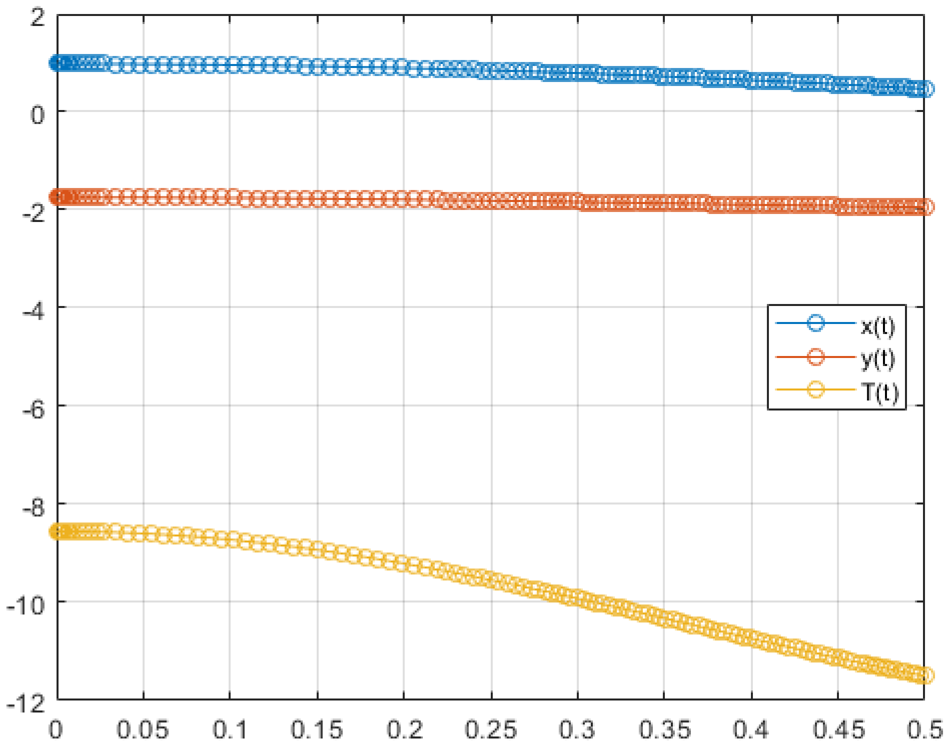
Set `r` to `2` and regenerate the function handle and initial conditions.

```
r = 2;
F = @(t, Y, YP) f(t, Y, YP, g, m, r);

y0est = [r*cos(pi/6); r*sin(pi/6); 0; 0; 0; 0; 0];
[y0, yp0] = decic(F, 0, y0est, [], yp0est, [], opt);
```

Solve the system for the new parameter value.

```
[tSol,y] = ode15i(F, [0, 0.5], y0, yp0, opt);
plot(tSol,y(:,1:origVars),'-o')
```

```
for k = 1:origVars
  S{k} = char(DAEvars(k));
end
legend(S, 'Location', 'Best')
grid on
```



See Also

Related Examples

- "Solve Semilinear DAE System" on page 2-205

- “Solve DAEs Using Mass Matrix Solvers” on page 2-213

Solve Semilinear DAE System

This workflow is an alternative workflow to solving semilinear DAEs, used only if `reduceDAEIndex` failed in the standard workflow with the warning below. For the standard workflow, see “Solve Differential Algebraic Equations (DAEs)” on page 2-193.

```
Warning: The index of the reduced DAEs is larger...
than 1. [daetools::reduceDAEIndex]
```

To solve your DAE system complete these steps.

- “Step 1. Reduce Differential Index with reduceDAEToODE” on page 2-205
- “Step 2. ODEs to Function Handles for ode15s and ode23t” on page 2-206
- “Step 3. Initial Conditions for ode15s and ode23t” on page 2-207
- “Step 4. Solve an ODE System with ode15s or ode23t” on page 2-209

Step 1. Reduce Differential Index with reduceDAEToODE

Complete steps 1 and 2 in “Solve Differential Algebraic Equations (DAEs)” on page 2-193 before beginning this step. Then, in step 3 when `reduceDAEIndex` fails, reduce the differential index using `reduceDAEToODE`. The advantage of `reduceDAEToODE` is that it reliably reduces semilinear DAEs to ODEs (DAEs of index `0`). However, this function is slower and works only on semilinear DAE systems. `reduceDAEToODE` can fail if the system is not semilinear.

To reduce the differential index of the DAEs described by `eqns` and `vars`, use `reduceDAEToODE`. To reduce the index, `reduceDAEToODE` adds new variables and equations to the system. `reduceDAEToODE` also returns constraints, which are conditions that help find initial values to ensure that the resulting ODEs are equal to the initial DAEs.

```
[ODEs,constraints] = reduceDAEToODE(eqns,vars)
```

```
ODEs =
                                 Dxt(t) - diff(x(t), t)
                                 Dyt(t) - diff(y(t), t)
                     m*diff(Dxt(t), t) - (T(t)*x(t))/r
              m*diff(Dyt(t), t) - (T(t)*y(t) - g*m*r)/r
            -(4*T(t)*y(t) - 2*g*m*r)*diff(y(t), t) -...
               diff(T(t), t)*(2*x(t)^2 + 2*y(t)^2) -...
```

```
                      4*T(t)*x(t)*diff(x(t), t) -...
                   4*m*r*Dxt(t)*diff(Dxt(t), t) -...
                        4*m*r*Dyt(t)*diff(Dyt(t), t)

constraints =
 2*g*m*r*y(t) - 2*T(t)*y(t)^2 - 2*m*r*Dxt(t)^2 -...
                      2*m*r*Dyt(t)^2 - 2*T(t)*x(t)^2
                               r^2 - y(t)^2 - x(t)^2
                        2*Dxt(t)*x(t) + 2*Dyt(t)*y(t)
```

Step 2. ODEs to Function Handles for ode15s and ode23t

From the output of `reduceDAEToODE`, you have a vector of equations in `ODEs` and a vector of variables in `vars`. To use `ode15s` or `ode23t`, you need two function handles: one representing the mass matrix of the ODE system, and the other representing the vector containing the right sides of the mass matrix equations. These function handles are the equivalent mass matrix representation of the ODE system where *M*(*t*,*y*(*t*))*y*'(*t*) = *f*(*t*,*y*(*t*)).

Find these function handles by using `massMatrixForm` to get the mass matrix `massM` (*M* in the equation) and right sides `f`.

```
[massM,f] = massMatrixForm(ODEs,vars)
```

```
massM =
[          -1,                     0,                     0,            0,            0]
[           0,                    -1,                     0,            0,            0]
[           0,                     0,                     0,            m,            0]
[           0,                     0,                     0,            0,            m]
[ -4*T(t)*x(t), 2*g*m*r - 4*T(t)*y(t), - 2*x(t)^2 - 2*y(t)^2, -4*m*r*Dxt(t), -4*m*r*Dyt(t)]

f =
              -Dxt(t)
              -Dyt(t)
         (T(t)*x(t))/r
 (T(t)*y(t) - g*m*r)/r
                     0
```

The equations in `ODEs` can contain symbolic parameters that are not specified in the vector of variables `vars`. Find these parameters by using `setdiff` on the output of `symvar` from `ODEs` and `vars`.

```
pODEs = symvar(ODEs);
pvars = symvar(vars);
extraParams = setdiff(pODEs, pvars)
extraParams =
[ g, m, r]
```

The extra parameters that you need to specify are the mass `m`, radius `r`, and gravitational constant `g`.

Convert `massM` and `f` to function handles using `odeFunction`. Specify the extra symbolic parameters as additional inputs to `odeFunction`.

```
massM = odeFunction(massM, vars, m, r, g);
f = odeFunction(f, vars, m, r, g);
```

The rest of the workflow is purely numerical. Set the parameter values and substitute the parameter values in `DAEs` and `constraints`.

```
m = 1;
r = 1;
g = 9.81;
ODEsNumeric = subs(ODEs);
constraintsNumeric = subs(constraints);
```

Create the function handle suitable for input to `ode15s` or `ode23s`.

```
M = @(t, Y) massM(t, Y, m, r, g);
F = @(t, Y) f(t, Y, m, r, g);
```

Step 3. Initial Conditions for ode15s and ode23t

The solvers require initial values for all variables in the function handle. Find initial values that satisfy the equations by using the MATLAB `decic` function. The `decic` accepts guesses (which might not satisfy the equations) for the initial conditions and tries to find satisfactory initial conditions using those guesses. `decic` can fail, in which case you must manually supply consistent initial values for your problem.

First, check the variables in `vars`.

```
vars
```

```
vars =
   x(t)
   y(t)
   T(t)
 Dxt(t)
 Dyt(t)
```

Here, Dxt(t) is the first derivative of x(t), and so on. There are 5 variables in a 5-by-1 vector. Therefore, guesses for initial values of variables and their derivatives must also be 5-by-1 vectors.

Assume the initial angular displacement of the pendulum is 30° or pi/6, and the origin of the coordinates is at the suspension point of the pendulum. Given that we used a radius r of 1, the initial horizontal position x(t) is r*sin(pi/6). The initial vertical position y(t) is -r*cos(pi/6). Specify these initial values of the variables in the vector y0est.

Arbitrarily set the initial values of the remaining variables and their derivatives to 0. These are not good guesses. However, they suffice for this problem. In your problem, if decic errors, then provide better guesses and refer to the decic page.

```
y0est = [r*sin(pi/6); -r*cos(pi/6); 0; 0; 0];
yp0est = zeros(5,1);
```

Create an option set that contains the mass matrix M of the system and specifies numerical tolerances for the numerical search.

```
opt = odeset('Mass', M, 'RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

Find initial values consistent with the system of ODEs and with the algebraic constraints by using decic. The parameter [1,0,0,0,1] in this function call fixes the first and the last element in y0est, so that decic does not change them during the numerical search. Here, this fixing is necessary to ensure decic finds satisfactory initial conditions.

```
[y0, yp0] = decic(ODEsNumeric, vars, constraintsNumeric, 0,...
                y0est, [1,0,0,0,1], yp0est, opt)
```

```
y0 =
    0.5000
   -0.8660
   -8.4957
         0
         0

yp0 =
         0
         0
         0
   -4.2479
   -2.4525
```

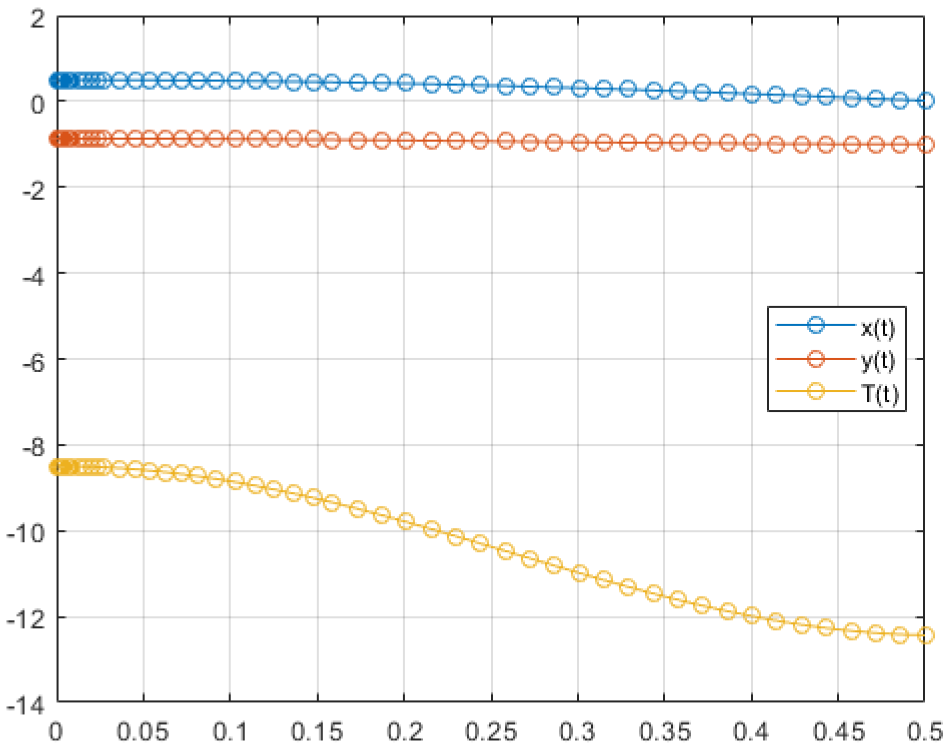
Now create an option set that contains the mass matrix `M` of the system and the vector `yp0` of consistent initial values for the derivatives. You will use this option set when solving the system.

```
opt = odeset(opt, 'InitialSlope', yp0);
```

Step 4. Solve an ODE System with ode15s or ode23t

Solve the system integrating over the time span `0 ≤ t ≤ 0.5`. Add the grid lines and the legend to the plot. Use `ode23s` by replacing `ode15s` with `ode23s`.

```
[tSol,ySol] = ode15s(F, [0, 0.5], y0, opt);
plot(tSol,ySol(:,1:origVars),'-o')

for k = 1:origVars
  S{k} = char(vars(k));
end

legend(S, 'Location', 'Best')
grid on
```



Solve the system for different parameter values by setting the new value and regenerating the function handle and initial conditions.

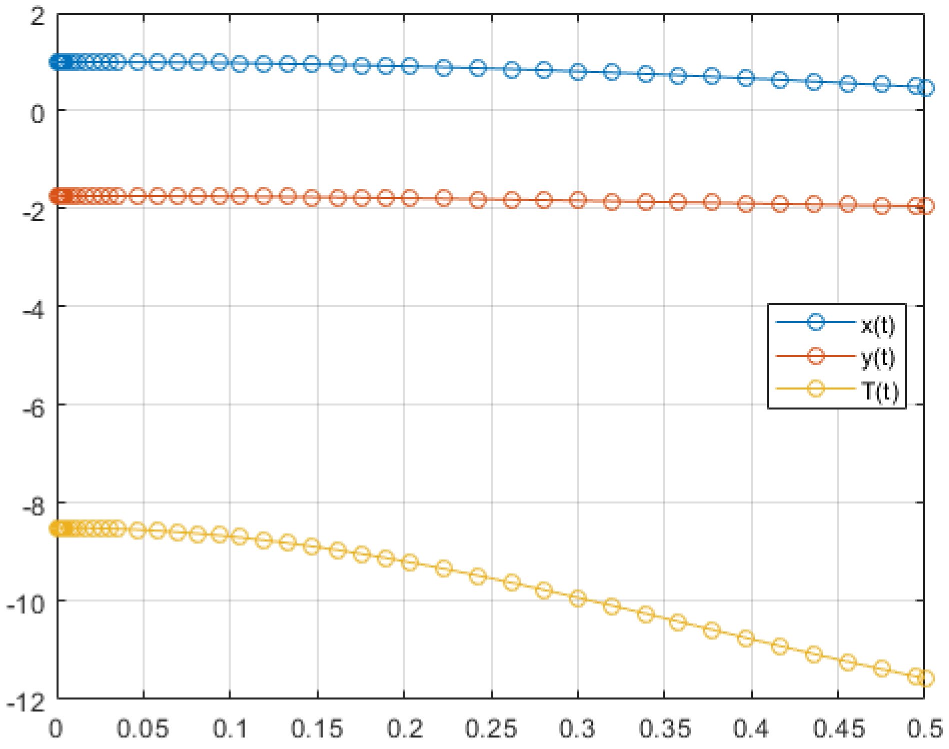
Set `r` to `2` and repeat the steps.

```
r = 2;

ODEsNumeric = subs(ODEs);
constraintsNumeric = subs(constraints);
M = @(t, Y) massM(t, Y, m, r, g);
F = @(t, Y) f(t, Y, m, r, g);

y0est = [r*cos(pi/6); -r*sin(pi/6); 0; 0; 0];
opt = odeset('Mass', M, 'RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

```
[y0, yp0] = decic(ODEsNumeric, vars, constraintsNumeric, 0,...
         y0est, [1,0,0,0,1], yp0est, opt);

opt = odeset(opt, 'InitialSlope', yp0);
```

Solve the system for the new parameter value.

```
[tSol,ySol] = ode15s(F, [0, 0.5], y0, opt);
plot(tSol,ySol(:,1:origVars),'-o')

for k = 1:origVars
  S{k} = char(vars(k));
end

legend(S, 'Location', 'Best')
grid on
```



See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Related Examples

- "Solve Differential Algebraic Equations (DAEs)" on page 2-193
- "Solve DAEs Using Mass Matrix Solvers" on page 2-213

Solve DAEs Using Mass Matrix Solvers

Solve differential algebraic equations by using one of the mass matrix solvers available in MATLAB. To use this workflow, first complete steps 1, 2, and 3 from “Solve Differential Algebraic Equations (DAEs)” on page 2-193. Then, use a mass matrix solver instead of `ode15i`.

This example demonstrates the use of `ode15s` or `ode23t`. For details on the other solvers, see “Choose an ODE Solver” (MATLAB) and adapt the workflow on this page.

In this section...

Step 1. Convert DAEs to Function Handles

From the output of `reduceDAEIndex`, you have a vector of equations `DAEs` and a vector of variables `DAEvars`. To use `ode15s` or `ode23t`, you need two function handles: one representing the mass matrix of a DAE system, and the other representing the right sides of the mass matrix equations. These function handles form the equivalent mass matrix representation of the ODE system where $M(t,y(t))y'(t) = f(t,y(t))$.

Find these function handles by using `massMatrixForm` to get the mass matrix `M` and the right sides `F`.

```
[M,f] = massMatrixForm(DAEs,DAEvars)

M =
[ 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,  0, 0, 0]
[ 0,  0, 0, 0,  0, 0, 0]
[ 0,  0, 0, 0, -1, 0, 0]
[ 0, -1, 0, 0,  0, 0, 0]

f =
                                 (T(t)*x(t) - m*r*Dxtt(t))/r
                         -(g*m*r - T(t)*y(t) + m*r*Dytt(t))/r
```

```
                                         r^2 - y(t)^2 - x(t)^2
                              - 2*Dxt(t)*x(t) - 2*Dyt(t)*y(t)
 - 2*Dxtt(t)*x(t) - 2*Dytt(t)*y(t) - 2*Dxt(t)^2 - 2*Dyt(t)^2
                                                      -Dytt(t)
                                                       -Dyt(t)
```

The equations in DAEs can contain symbolic parameters that are not specified in the vector of variables DAEvars. Find these parameters by using setdiff on the output of symvar from DAEs and DAEvars.

```
pDAEs = symvar(DAEs);
pDAEvars = symvar(DAEvars);
extraParams = setdiff(pDAEs, pDAEvars)

extraParams =
[ g, m, r]
```

The mass matrix M does not have these extra parameters. Therefore, convert M directly to a function handle by using odeFunction.

```
M = odeFunction(M, DAEvars);
```

Convert f to a function handle. Specify the extra parameters as additional inputs to odeFunction.

```
f = odeFunction(f, DAEvars, g, m, r);
```

The rest of the workflow is purely numerical. Set parameter values and create the function handle.

```
g = 9.81;
m = 1;
r = 1;
F = @(t, Y) f(t, Y, g, m, r);
```

Step 2. Find Initial Conditions

The solvers require initial values for all variables in the function handle. Find initial values that satisfy the equations by using the MATLAB decic function. The decic accepts guesses (which might not satisfy the equations) for the initial conditions, and tries to find satisfactory initial conditions using those guesses. decic can fail, in which case you must manually supply consistent initial values for your problem.

First, check the variables in DAEvars.

```
DAEvars
```

```
DAEvars =
    x(t)
    y(t)
    T(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Here, Dxt(t) is the first derivative of x(t), Dytt(t) is the second derivative of y(t), and so on. There are 7 variables in a 7-by-1 vector. Thus, guesses for initial values of variables and their derivatives must also be 7-by-1 vectors.

Assume the initial angular displacement of the pendulum is 30° or pi/6, and the origin of the coordinates is at the suspension point of the pendulum. Given that we used a radius r of 1, the initial horizontal position x(t) is r*sin(pi/6). The initial vertical position y(t) is -r*cos(pi/6). Specify these initial values of the variables in the vector y0est.

Arbitrarily set the initial values of the remaining variables and their derivatives to 0. These are not good guesses. However, they suffice for our problem. In your problem, if decic errors, then provide better guesses and refer to the decic page.

```
y0est = [r*sin(pi/6); -r*cos(pi/6); 0; 0; 0; 0; 0];
yp0est = zeros(7,1);
```

Create an option set that contains the mass matrix M and initial guesses yp0est, and specifies numerical tolerances for the numerical search.

```
opt = odeset('Mass', M, 'InitialSlope', yp0est,...
                'RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

Find consistent initial values for the variables and their derivatives by using the MATLAB decic function. The first argument of decic must be a function handle describing the DAE as f(t,y,yp) = f(t,y,y') = 0. In terms of M and F, th means f(t,y,yp) = M(t,y)*yp - F(t,y).

```
implicitDAE = @(t,y,yp) M(t,y)*yp - F(t,y);
[y0, yp0] = decic(implicitDAE, 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4771
   -0.8788
   -8.6214
         0
    0.0000
   -2.2333
   -4.1135

yp0 =
         0
    0.0000
         0
         0
   -2.2333
         0
         0
```

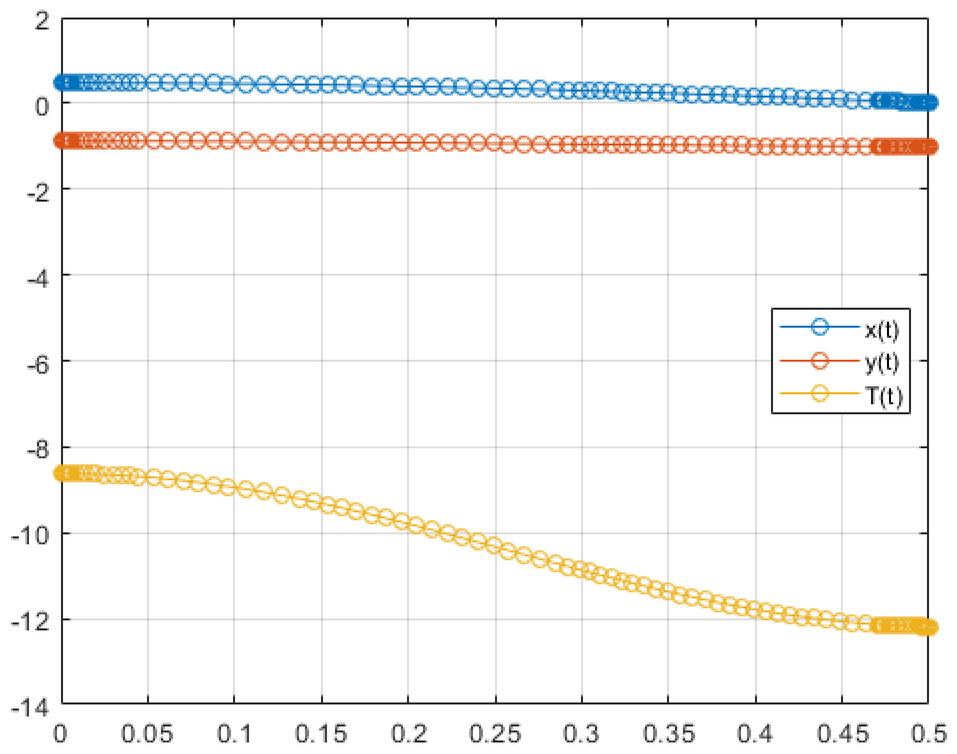
Now create an option set that contains the mass matrix `M` of the system and the vector `yp0` of consistent initial values for the derivatives. You will use this option set when solving the system.

```
opt = odeset(opt, 'InitialSlope', yp0);
```

Step 3. Solve DAE System

Solve the system integrating over the time span `0` ≤ `t` ≤ `0.5`. Add the grid lines and the legend to the plot. The code uses `ode15s`. Instead, you can use `ode23s` by replacing `ode15s` with `ode23s`.

```
[tSol,ySol] = ode15s(F, [0, 0.5], y0, opt);
plot(tSol,ySol(:,1:origVars),'-o')

for k = 1:origVars
  S{k} = char(DAEvars(k));
end

legend(S, 'Location', 'Best')
grid on
```



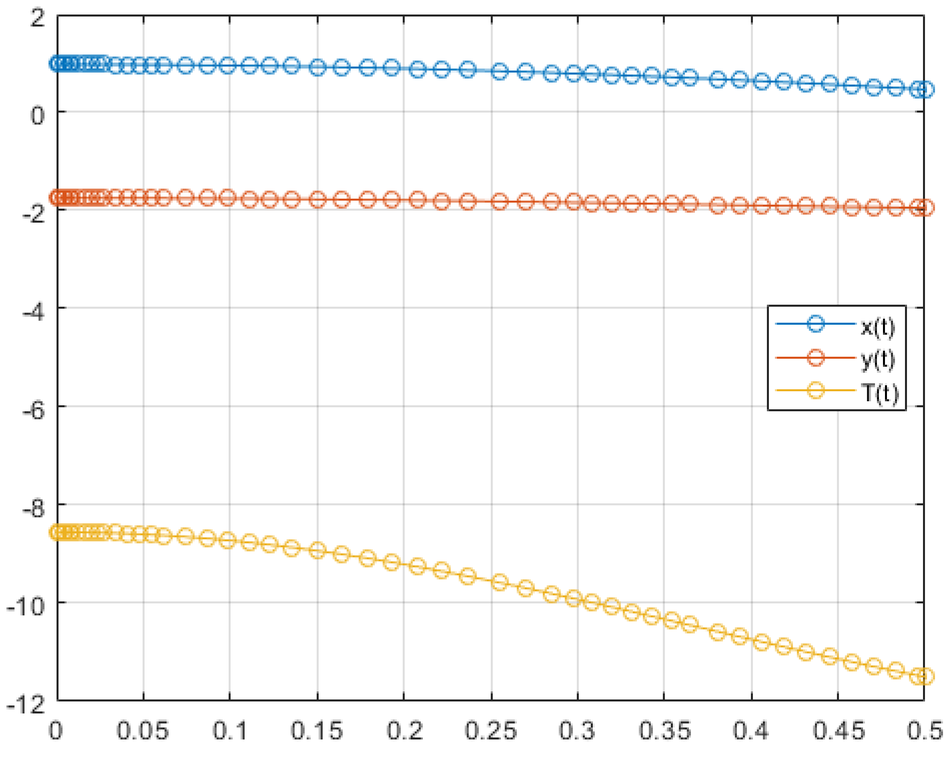
Solve the system for different parameter values by setting the new value and regenerating the function handle and initial conditions.

Set `r` to `2` and regenerate the function handle and initial conditions.

```
r = 2;

F = @(t, Y) f(t, Y, g, m, r);
y0est = [r*sin(pi/6); -r*cos(pi/6); 0; 0; 0; 0; 0];
implicitDAE = @(t,y,yp) M(t,y)*yp - F(t,y);
[y0, yp0] = decic(implicitDAE, 0, y0est, [], yp0est, [], opt);

opt = odeset(opt, 'InitialSlope', yp0);
```

Solve the system for the new parameter value.

```
[tSol,ySol] = ode15s(F, [0, 0.5], y0, opt);
plot(tSol,ySol(:,1:origVars),'-o')

for k = 1:origVars
  S{k} = char(DAEvars(k));
end

legend(S, 'Location', 'Best')
grid on
```



See Also

Related Examples

- “Solve Differential Algebraic Equations (DAEs)” on page 2-193
- “Solve Semilinear DAE System” on page 2-205

Fourier and Inverse Fourier Transforms

This page shows the workflow for Fourier and inverse Fourier transforms in Symbolic Math Toolbox. For simple examples, see `fourier` and `ifourier`. Here, the workflow for Fourier transforms is demonstrated by calculating the deflection of a beam due to a force. The associated differential equation is solved by the Fourier transform.

Fourier Transform Definition

The Fourier transform of $f(x)$ with respect to x at w is

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

The inverse Fourier transform is

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

Concept: Using Symbolic Workflows

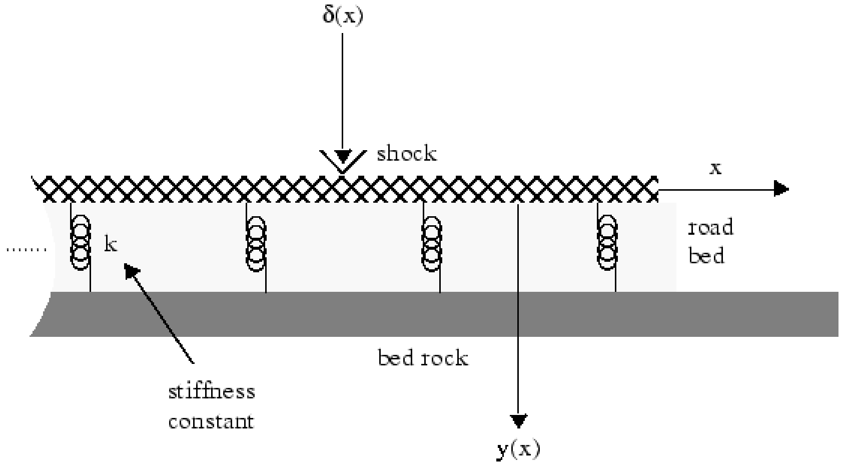
Symbolic workflows keep calculations in the natural symbolic form instead of numeric form. This approach helps you understand the properties of your solution and use exact symbolic values. You only substitute symbolic variables for numeric values when you require a numeric result or you cannot continue symbolically. For details, see “Choose Symbolic or Numeric Arithmetic” on page 2-114. Typically, the steps are:

1. Declare equations.
2. Solve equations.
3. Substitute values.
4. Plot results.
5. Analyze results.

Calculate Beam Deflection Using Fourier Transform

Define Equations

Fourier transform can be used to solve ordinary and partial differential equations. For example, you can model the deflection of an infinitely long beam resting on an elastic foundation under a point force. A corresponding real-world example is railway tracks on a foundation. The railway tracks are the infinitely long beam while the foundation is elastic.



Let

- E be the elasticity of the beam (or railway track).
- I be the second moment of area of the cross-section of the beam.
- k be the spring stiffness of the foundation.

The differential equation is

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

Define the function `y(x)` and the variables. Assume `E`, `I`, and `k` are positive.

```
syms Y(x) w E I k f
assume([E I k] > 0)
```

Assign units to the variables by using `symunit`.

```
u = symunit;
Eu = E*u.Pa;     % Pascal
Iu = I*u.m^4;    % meter^4
ku = k*u.N/u.m^2;  % Newton/meter^2
X = x*u.m;
F = f*u.N/u.m;
```

Define the differential equation.

```
eqn = diff(Y,X,4) + ku/(Eu*Iu)*Y == F/(Eu*Iu)
```

```
eqn(x) =
diff(Y(x), x, x, x, x)*(1/[m]^4) + ((k*Y(x))/(E*I))*([N]/([Pa]*[m]^6)) == ...
            (f/(E*I))*([N]/([Pa]*[m]^5))
```

Represent the force `f` by the Dirac delta function δ(*x*).

```
eqn = subs(eqn,f,dirac(x))
```

```
eqn(x) =
diff(Y(x), x, x, x, x)*(1/[m]^4) + ((k*Y(x))/(E*I))*([N]/([Pa]*[m]^6)) ==...
           (dirac(x)/(E*I))*([N]/([Pa]*[m]^5))
```

Solve Equations

Calculate the Fourier transform of `eqn` by using `fourier` on both sides of `eqn`. The Fourier transform converts differentiation into exponents of `w`.

```
eqnFT = fourier(lhs(eqn)) == fourier(rhs(eqn))
```

```
eqnFT =
w^4*fourier(Y(x), x, w)*(1/[m]^4) + ((k*fourier(Y(x), x, w))/(E*I))*([N]/([Pa]*[m]^6))
```

Isolate `fourier(Y(x),x,w)` in the equation.

```
eqnFT = isolate(eqnFT, fourier(Y(x),x,w))
```

```
eqnFT =
fourier(Y(x), x, w) == (1/(E*I*w^4*[Pa]*[m]^2 + k*[N]))*[N]*[m]
```

Calculate `Y(x)` by calculating the inverse Fourier transform of the right side. Simplify the result.

```
YSol = ifourier(rhs(eqnFT));
YSol = simplify(YSol)
YSol =
((exp(-(2^(1/2)*k^(1/4)*abs(x))/(2*E^(1/4)*I^(1/4)))*sin((2*2^(1/2)*k^(1/4)*abs(x) +...
 pi*E^(1/4)*I^(1/4))/(4*E^(1/4)*I^(1/4))))/(2*E^(1/4)*I^(1/4)*k^(3/4)))*[m]
```

Check that `YSol` has the correct dimensions by substituting `YSol` into `eqn` and using the `checkUnits` function. `checkUnits` returns logical `1` (`true`), meaning `eqn` now has compatible units of the same physical dimensions.

```
checkUnits(subs(eqn,Y,YSol))
```

```
ans =
  struct with fields:

    Consistent: 1
    Compatible: 1
```

Separate the expression from the units by using `separateUnits`.

```
YSol = separateUnits(YSol)
```

```
YSol =
(exp(-(2^(1/2)*k^(1/4)*abs(x))/(2*E^(1/4)*I^(1/4)))*sin((2*2^(1/2)*k^(1/4)*abs(x) + pi*
```

Substitute Values

Use the values $E = 10^6$ Pa, $I = 10^{-3}$ m^4, and k = 10^6 N/m^2. Substitute these values into `YSol` and convert to floating point by using `vpa` with 16 digits of accuracy.

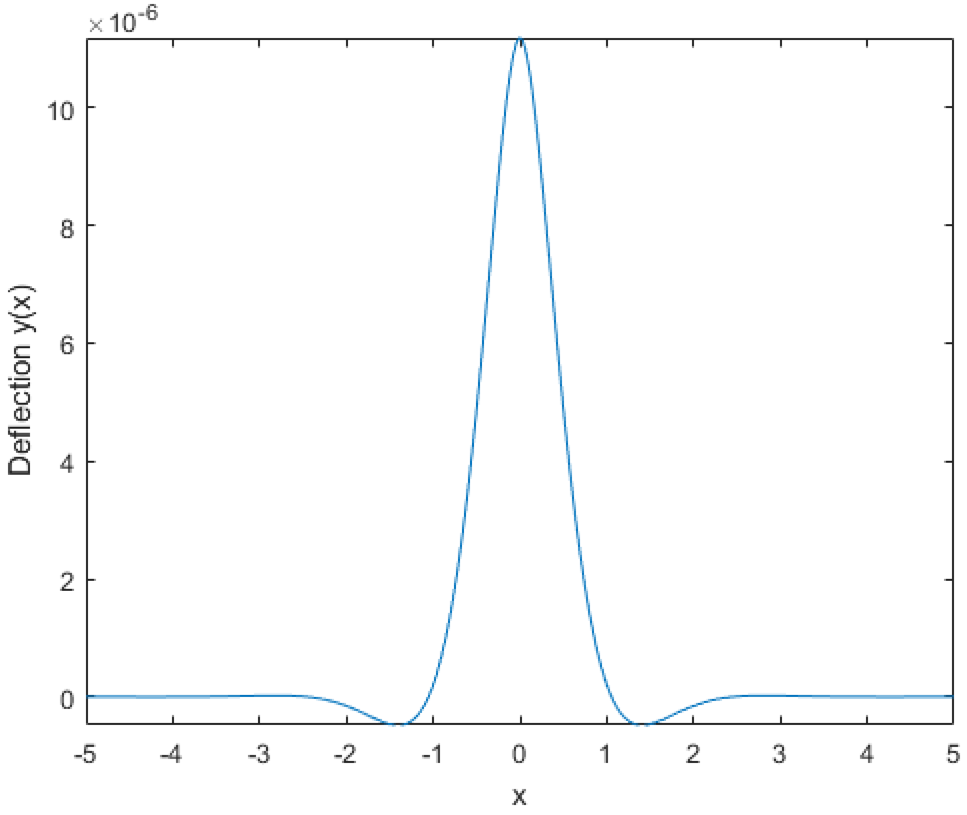
```
values = [1e6 1e-3 1e5];
YSol = subs(YSol,[E I k],values);
YSol = vpa(YSol,16)
```

```
YSol =
0.0000158113883008419*exp(-2.23606797749979*abs(x))*sin(2.23606797749979*abs(x) + 0.785
```

Plot Results

Plot the result by using `fplot`.

```
fplot(YSol)
xlabel('x')
ylabel('Deflection y(x)')
```



Analyze Results

The plot shows that the deflection of a beam due to a point force is highly localized. The deflection is greatest at the point of impact and then decreases quickly. The symbolic result enables you to analyze the properties of the result, which is not possible with numeric results.

Notice that `YSol` is a product of terms. The term with `sin` shows that the response is vibrating oscillatory behavior. The term with `exp` shows that the oscillatory behavior is quickly damped by the exponential decay as the distance from the point of impact increases.

Solve Differential Equations Using Laplace Transform

Solve differential equations by using Laplace transforms in Symbolic Math Toolbox with this workflow. For simple examples on the Laplace transform, see `laplace` and `ilaplace`.

Definition: Laplace Transform

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

$$F(s) = \int_0^\infty f(t)e^{-ts}\,dt.$$

Concept: Using Symbolic Workflows

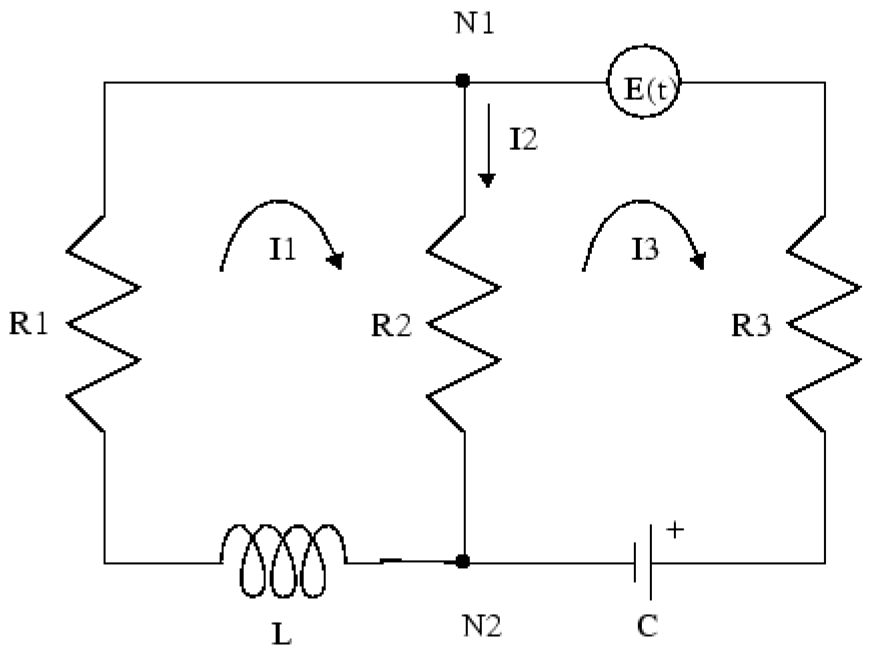
Symbolic workflows keep calculations in the natural symbolic form instead of numeric form. This approach helps you understand the properties of your solution and use exact symbolic values. You only substitute symbolic variables for numeric values when you require a numeric result or you cannot continue symbolically. For details, see “Choose Symbolic or Numeric Arithmetic” on page 2-114. Typically, the steps are:

1. Declare equations.
2. Solve equations.
3. Substitute values.
4. Plot results.
5. Analyze results.

Workflow: Solve RLC Circuit Using Laplace Transform

Declare Equations

You can use the Laplace transform to solve differential equations with initial conditions. For example, you can solve resistance-inductor-capacitor (RLC) circuits, such as this circuit.



- Resistances in ohm: R_1, R_2, R_3
- Currents in ampere: I_1, I_2, I_3
- Inductance in henry: L
- Capacitance in farad: C
- Electromotive force in volts: $E(t)$
- Charge in coulomb: $Q(t)$

Apply Kirchhoff's voltage and current laws to get the differential equations for the RLC circuit.

$$\frac{dI_1}{dt} + \frac{R_2}{L}\frac{dQ}{dt} = \frac{R_2 - R_1}{L} I_1.$$

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

Declare the variables. Because the physical quantities have positive values, set the corresponding assumptions on the variables. Let $E(t)$ be an alternating voltage of 1 V.

```
syms L C I1(t) Q(t) s
R = sym('R%d',[1 3]);
assume([t L C R] > 0)
E(t) = 1*sin(t);        % Voltage = 1 V
```

Declare the differential equations.

```
dI1 = diff(I1,t);
dQ = diff(Q,t);
eqn1 = dI1 + (R(2)/L)*dQ == (R(2)-R(1))/L*I1
eqn2 = dQ == (1/(R(2)+R(3))*(E-Q/C)) + R(2)/(R(2)+R(3))*I1
```

```
eqn1(t) =
diff(I1(t), t) + (R2*diff(Q(t), t))/L == -(I1(t)*(R1 - R2))/L
eqn2(t) =
diff(Q(t), t) == (sin(t) - Q(t)/C)/(R2 + R3) + (R2*I1(t))/(R2 + R3)
```

Assume that the initial current and charge, I_0 and Q_0, are both `0`. Declare these initial conditions.

```
cond1 = I1(0) == 0
cond2 = Q(0) == 0
```

```
cond1 =
I1(0) == 0
cond2 =
Q(0) == 0
```

Solve Equations

Compute the Laplace transform of `eqn1` and `eqn2`.

```
eqn1LT = laplace(eqn1,t,s)
eqn2LT = laplace(eqn2,t,s)
```

```
eqn1LT =
s*laplace(I1(t), t, s) - I1(0) - (R2*(Q(0) - s*laplace(Q(t), t, s)))/L == ...
-((R1 - R2)*laplace(I1(t), t, s))/L
eqn2LT =
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))
```

The function `solve` solves only for symbolic variables. Therefore, to use `solve`, first substitute `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)` with the variables `I1_LT` and `Q_LT`.

```
syms I1_LT Q_LT
eqn1LT = subs(eqn1LT,[laplace(I1,t,s) laplace(Q,t,s)],[I1_LT Q_LT])
```

```
eqn1LT =
I1_LT*s - I1(0) - (R2*(Q(0) - Q_LT*s))/L == -(I1_LT*(R1 - R2))/L
```

```
eqn2LT = subs(eqn2LT,[laplace(I1,t,s) laplace(Q,t,s)],[I1_LT Q_LT])
```

```
eqn2LT =
Q_LT*s - Q(0) == (I1_LT*R2)/(R2 + R3) - (Q_LT - C/(s^2 + 1))/(C*(R2 + R3))
```

Solve the equations for `I1_LT` and `Q_LT`.

```
eqns = [eqn1LT eqn2LT];
vars = [I1_LT Q_LT];
[I1_LT, Q_LT] = solve(eqns,vars)
```

```
I1_LT =
(R2*Q(0) + L*I1(0) - C*R2*s + L*s^2*I1(0) + R2*s^2*Q(0) + C*L*R2*s^3*I1(0) + ...
C*L*R3*s^3*I1(0) + C*L*R2*s*I1(0) + C*L*R3*s*I1(0))/((s^2 + 1)*(R1 - R2 + L*s + ...
C*L*R2*s^2 + C*L*R3*s^2 + C*R1*R2*s + C*R1*R3*s - C*R2*R3*s))
Q_LT =
(C*(R1 - R2 + L*s + L*R2*I1(0) + R1*R2*Q(0) + R1*R3*Q(0) - R2*R3*Q(0) + ...
L*R2*s^2*I1(0) + L*R2*s^3*Q(0) + L*R3*s^3*Q(0) + R1*R2*s^2*Q(0) + R1*R3*s^2*Q(0) - ...
R2*R3*s^2*Q(0) + L*R2*s*Q(0) + ...
L*R3*s*Q(0)))/((s^2 + 1)*(R1 - R2 + L*s + C*L*R2*s^2 + C*L*R3*s^2 + ...
                C*R1*R2*s + C*R1*R3*s - C*R2*R3*s))
```

Calculate I_1 and Q by computing the inverse Laplace transform of `I1_LT` and `Q_LT`. Simplify the result. Suppress the output because it is long.

```
I1sol = ilaplace(I1_LT,s,t);
Qsol = ilaplace(Q_LT,s,t);
I1sol = simplify(I1sol);
Qsol = simplify(Qsol);
```

Substitute Values

Before plotting the result, substitute symbolic variables by the numeric values of the circuit elements. Let $R1$ = 4 Ω , $R2$ = 2 Ω, $R3$ = 3 Ω, C = 1/4 F, L = 1.6 H, $I_1(0)$ = 15 A , and $Q(0)$ = 2 C.

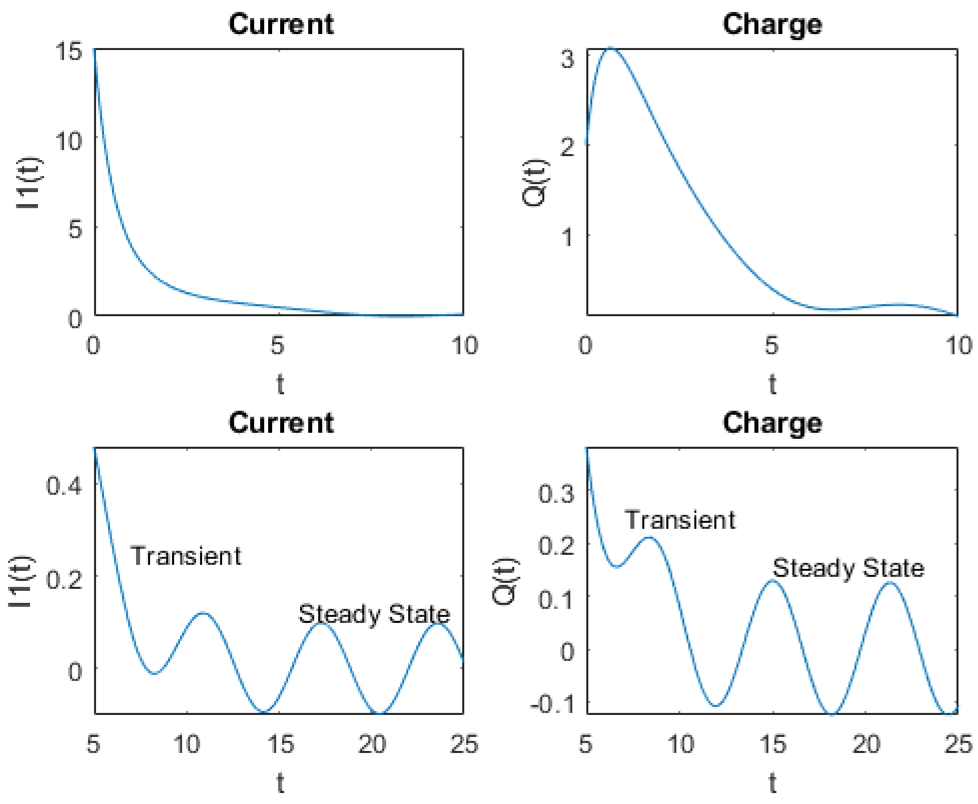
```
vars = [R L C I1(0) Q(0)];
values = [4 2 3 1.6 1/4 15 2];
I1sol = subs(I1sol,vars,values)
Qsol = subs(Qsol,vars,values)
```

```
I1sol =
15*exp(-(51*t)/40)*(cosh((1001^(1/2)*t)/40) - (293*1001^(1/2)*sinh((1001^(1/2)*t)/40))/
Qsol =
(4*sin(t))/51 - (5*cos(t))/51 + (107*exp(-(51*t)/40)*(cosh((1001^(1/2)*t)/40) + (2039*1
```

Plot Results

Plot the current `I1sol` and charge `Qsol`. Show both the transient and steady state behavior by using two different time intervals: 0 ≤ `t` ≤ 10 and 5 ≤ `t` ≤ 25. Before R2016a, use `ezplot` instead of `fplot`.

```
subplot(2,2,1)
fplot(I1sol,[0 10])
title('Current')
ylabel('I1(t)')
xlabel('t')

subplot(2,2,2)
fplot(Qsol,[0 10])
title('Charge')
ylabel('Q(t)')
xlabel('t')

subplot(2,2,3)
fplot(I1sol,[5 25])
title('Current')
ylabel('I1(t)')
xlabel('t')
text(7,0.25,'Transient')
text(16,0.125,'Steady State')

subplot(2,2,4)
fplot(Qsol,[5 25])
title('Charge')
ylabel('Q(t)')
xlabel('t')
text(7,0.25,'Transient')
text(15,0.16,'Steady State')
```



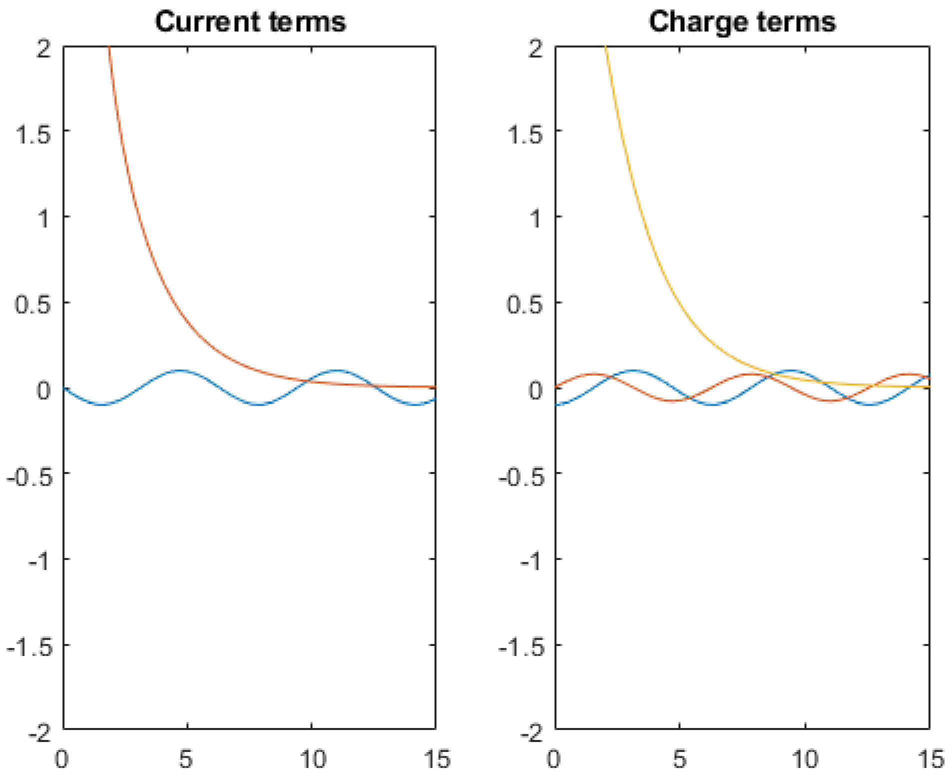
Analyze Results

Initially, the current and charge decrease exponentially. However, over the long term, they are oscillatory. These behaviors are called "transient" and "steady state", respectively. With the symbolic result, you can analyze the result's properties, which is not possible with numeric results.

Visually inspect `I1sol` and `Qsol`. They are a sum of terms. Find the terms by using `children`. Then, find the contributions of the terms by plotting them over `[0 15]`. The plots show the transient and steady state terms.

```
I1terms = children(I1sol);
Qterms = children(Qsol);
```

```
subplot(1,2,1)
fplot(I1terms,[0 15])
ylim([-2 2])
title('Current terms')

subplot(1,2,2)
fplot(Qterms,[0 15])
ylim([-2 2])
title('Charge terms')
```



The plots show that `I1sol` has a transient and steady state term, while `Qsol` has a transient and two steady state terms. From visual inspection, notice `I1sol` and `Qsol` have a term containing the `exp` function. Assume that this term causes the transient exponential decay. Separate the transient and steady state terms in `I1sol` and `Qsol` by checking terms for `exp` using `has`.

```
I1transient = I1terms(has(I1terms,'exp'))
I1steadystate = I1terms(~has(I1terms,'exp'))
```

```
I1transient =
15*exp(-(51*t)/40)*(cosh((1001^(1/2)*t)/40) - (293*1001^(1/2)*sinh((1001^(1/2)*t)/40))/21879)
I1steadystate =
-(5*sin(t))/51
```

Similarly, separate `Qsol` into transient and steady state terms. This result demonstrates how symbolic calculations help you analyze your problem.

```
Qtransient = Qterms(has(Qterms,'exp'))
Qsteadystate = Qterms(~has(Qterms,'exp'))
```

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

Solve Difference Equations Using Z-Transform

Solve difference equations by using Z-transforms in Symbolic Math Toolbox with this workflow. For simple examples on the Z-transform, see `ztrans` and `iztrans`.

Definition: Z-transform

The Z-transform of a function $f(n)$ is defined as

$$F(z) = \sum_{n=0}^{\infty} \frac{f(n)}{z^n}.$$

Concept: Using Symbolic Workflows

Symbolic workflows keep calculations in the natural symbolic form instead of numeric form. This approach helps you understand the properties of your solution and use exact symbolic values. You only substitute symbolic variables for numeric values when you require a numeric result or you cannot continue symbolically. For details, see “Choose Symbolic or Numeric Arithmetic” on page 2-114. Typically, the steps are:

1. Declare equations.
2. Solve equations.
3. Substitute values.
4. Plot results.
5. Analyze results.

Workflow: Solve "Rabbit Growth" Problem Using Z-Transform

Declare Equations

You can use the Z-transform to solve difference equations, such as the well-known "Rabbit Growth" problem. If a pair of rabbits matures in one year, and then produces another pair of rabbits every year, the rabbit population $p(n)$ at year n is described by this difference equation.

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

Declare the equation as an expression assuming the right side is `0`. Because `n` represents years, assume that `n` is a positive integer. This assumption simplifies the results.

```
syms p(n) z
assume(n>=0 & in(n,'integer'))
f = p(n+2) - p(n+1) - p(n)
```

```
f =
p(n + 2) - p(n + 1) - p(n)
```

Solve Equations

Find the Z-transform of the equation.

```
fZT = ztrans(f,n,z)
```

```
fZT =
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
```

The function `solve` solves only for symbolic variables. Therefore, to use `solve`, first substitute `ztrans(p(n),n,z)` with the variables `pZT`.

```
syms pZT
fZT = subs(fZT,ztrans(p(n),n,z),pZT)
```

```
fZT =
z*p(0) - pZT - z*p(1) - pZT*z - z^2*p(0) + pZT*z^2
```

Solve for `pZT`.

```
pZT = solve(fZT,pZT)
```

```
pZT =
-(z*p(1) - z*p(0) + z^2*p(0))/(- z^2 + z + 1)
```

Calculate *p*(*n*) by computing the inverse Z-transform of `pZT`. Simplify the result.

```
pSol = iztrans(pZT,z,n);
pSol = simplify(pSol)
```

```
pSol =
2*(-1)^(n/2)*cos(n*(pi/2 + asinh(1/2)*1i))*p(1) + ...
                (2^(2 - n)*5^(1/2)*(5^(1/2) + 1)^(n - 1)*(p(0)/2 - p(1)))/...
                 5 - (2*2^(1 - n)*5^(1/2)*(1 - 5^(1/2))^(n - 1)*(p(0)/2 - p(1)))/5
```

Substitute Values

To plot the result, first substitute the values of the initial conditions. Let `p(0)` and `p(1)` be `1` and `2`, respectively.

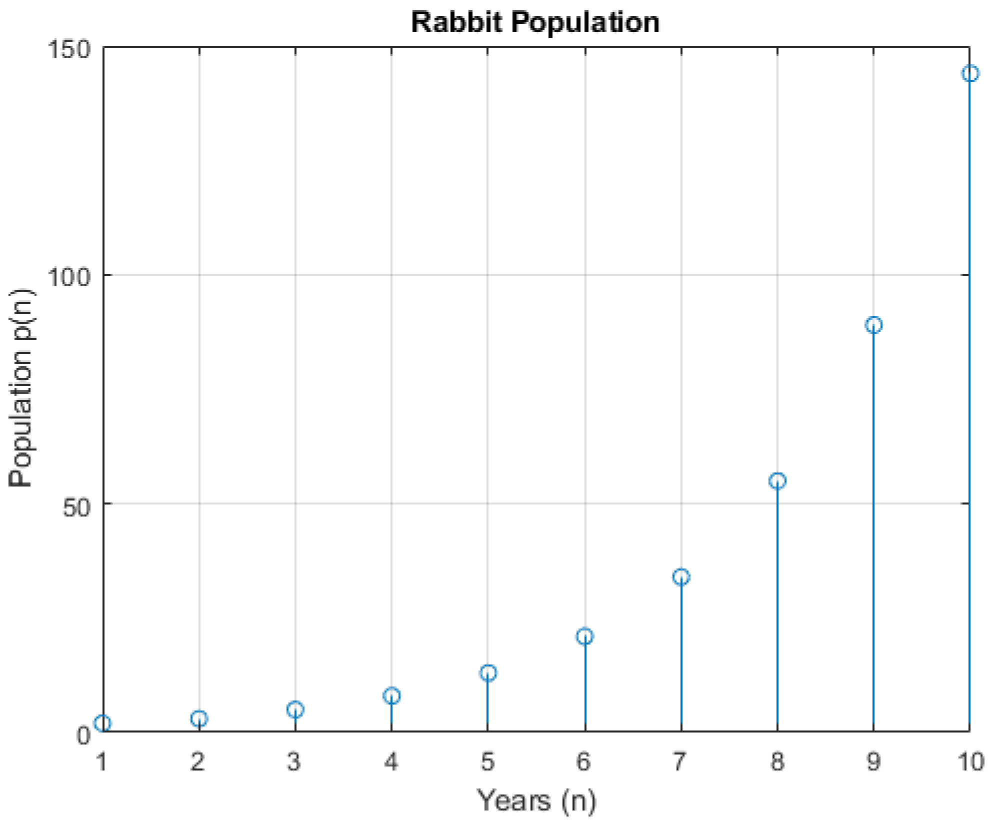
```
pSol = subs(pSol,[p(0) p(1)],[1 2])
```

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

Plot Results

Show the growth in rabbit population over time by plotting `pSol`.

```
nValues = 1:10;
pSolValues = subs(pSol,n,nValues);
pSolValues = double(pSolValues);
pSolValues = real(pSolValues);
stem(nValues,pSolValues)
title('Rabbit Population')
xlabel('Years (n)')
ylabel('Population p(n)')
grid on
```



Analyze Results

The plot shows that the solution appears to increase exponentially. However, because the solution `pSol` contains many terms, finding the terms that produce this behavior requires analysis.

Because all the functions in `pSol` can be expressed in terms of `exp`, rewrite `pSol` to `exp`. Simplify the result by using `simplify` with `80` additional simplification steps. Now, you can analyze `pSol`.

```
pSol = rewrite(pSol,'exp');
pSol = simplify(pSol,'Steps',80)
pSol =
(2*2^n)/(- 5^(1/2) - 1)^n - (3*5^(1/2)*(1/2 - 5^(1/2)/2)^n)/10 + (3*5^(1/2)*(5^(1/2)/2
```

Visually inspect `pSol`. Notice that `pSol` is a sum of terms. Each term is a ratio that can increase or decrease as `n` increases. For each term, you can confirm this hypothesis in several ways:

- Check if the limit at `n = Inf` goes to `0` or `Inf` by using `limit`.
- Plot the term for increasing `n` and check behavior.
- Calculate the value at a large value of `n`.

For simplicity, use the third approach. Calculate the terms at `n = 100`, and then verify the approach. First, find the individual terms by using `children`, substitute for `n`, and convert to double.

```
pSolTerms = children(pSol);
pSolTermsDbl = subs(pSolTerms,n,100);
pSolTermsDbl = double(pSolTermsDbl)
```

```
pSolTermsDbl =
   1.0e+20 *
    0.0000   -0.0000    5.3134   -0.0000    3.9604
```

The result shows that some terms are `0` while other terms have a large magnitude. Hypothesize that the large-magnitude terms produce the exponential behavior. Approximate `pSol` with these terms.

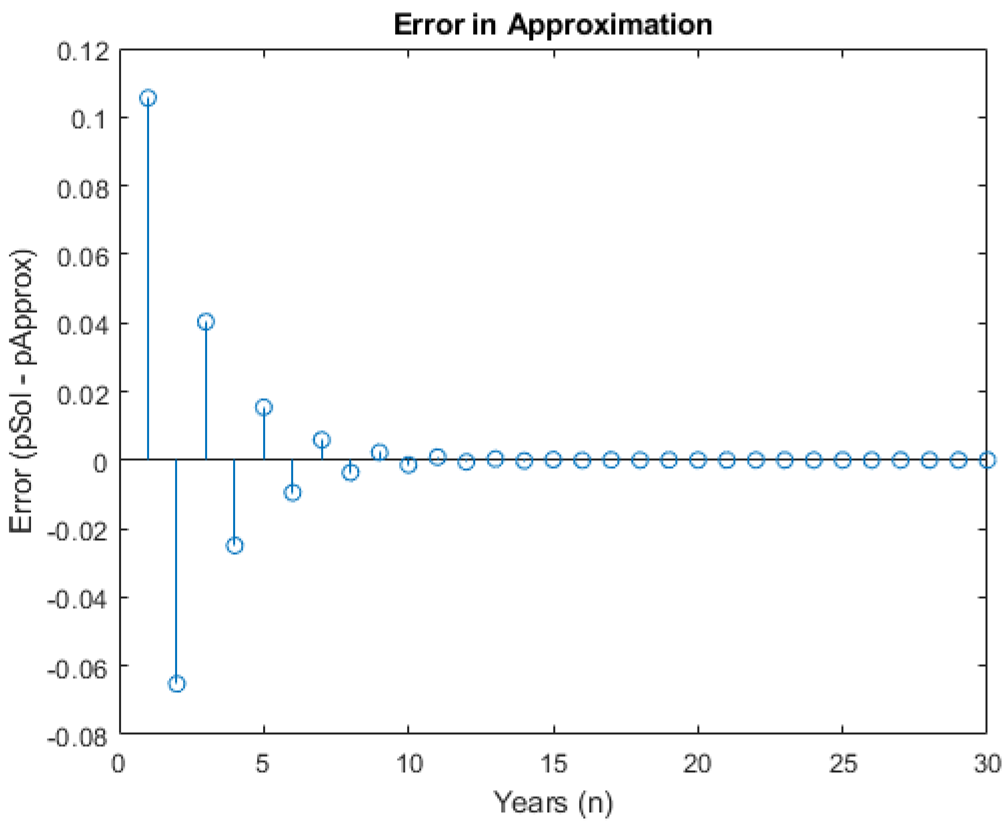
```
idx = abs(pSolTermsDbl)>1; % use arbitrary cutoff
pApprox = pSolTerms(idx);
pApprox = sum(pApprox)
```

```
pApprox =
(3*5^(1/2)*(5^(1/2)/2 + 1/2)^n)/10 + (5^(1/2)/2 + 1/2)^n/2
```

Verify the hypothesis by plotting the approximation error between `pSol` and `pApprox`. As expected, the error goes to `0` as `n` increases. This result demonstrates how symbolic calculations help you analyze your problem.

```
Perror = pSol - pApprox;
nValues = 1:30;
Perror = subs(Perror,n,nValues);
stem(nValues,Perror)
xlabel('Years (n)')
```

```
ylabel('Error (pSol - pApprox)')
title('Error in Approximation')
```



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...

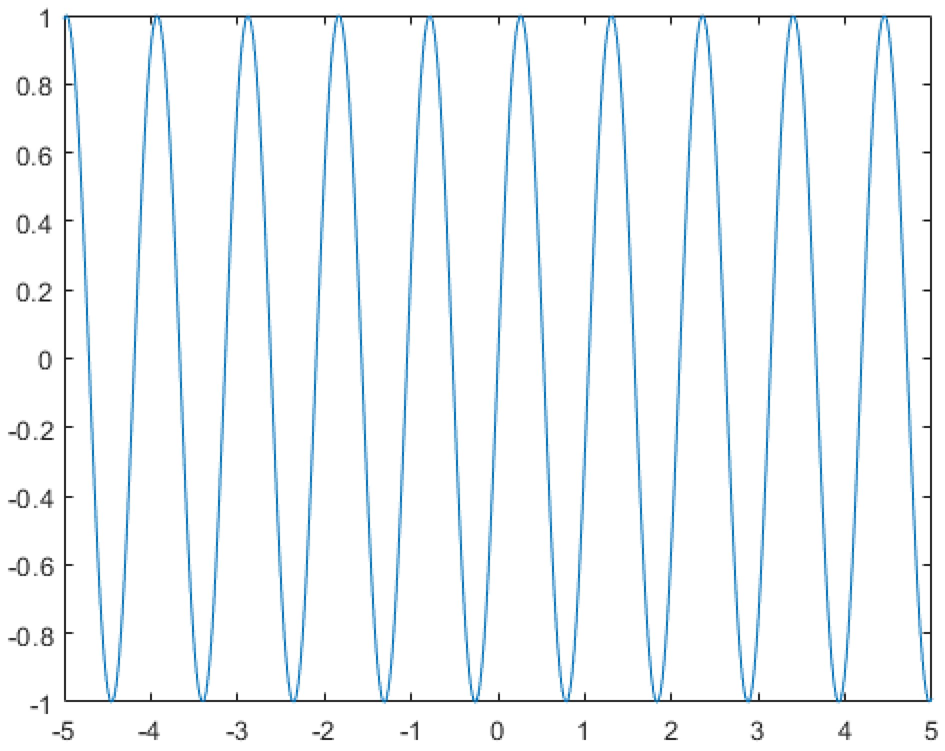
Plot with Symbolic Plotting Functions

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:

- `fplot` to create 2-D plots of symbolic expressions, equations, or functions in Cartesian coordinates.
- `fplot3` to create 3-D parametric plots.
- `ezpolar` to create plots in polar coordinates.
- `fsurf` to create surface plots.
- `fcontour` to create contour plots.
- `fmesh` to create mesh plots.

Plot the symbolic expression $\sin(6x)$ by using `fplot`. By default, `fplot` uses the range $-5 < x < 5$. Prior to R2016a, use `ezplot` instead of `fplot`.

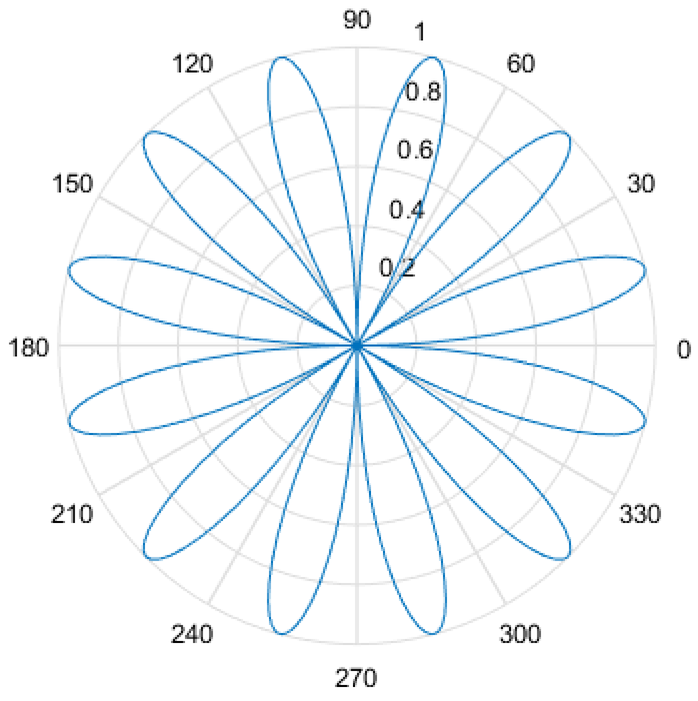
```
syms x
fplot(sin(6*x))
```



Plot a symbolic expression or function in polar coordinates r (radius) and θ (polar angle) by using `ezpolar`. By default, `ezpolar` plots a symbolic expression or function over the interval $0 < \theta < 2\pi$.

Plot the symbolic expression $\sin(6t)$ in polar coordinates.

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



r = sin(6 t)

Plot Functions Numerically

As an alternative to plotting expressions symbolically, you can substitute symbolic variables with numeric values by using `subs`. Then, you can use these numeric values with plotting functions in MATLAB™.

In the following expressions `u` and `v`, substitute the symbolic variables `x` and `y` with the numeric values defined by `meshgrid`.

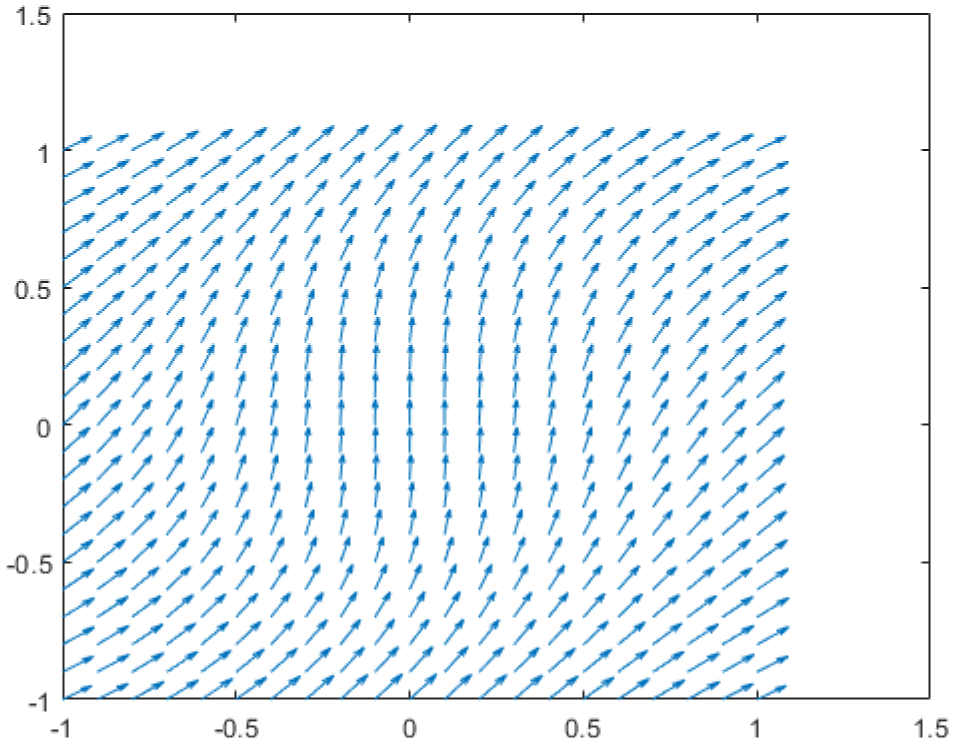
```
syms x y
u = sin(x^2 + y^2);
v = cos(x*y);
[X, Y] = meshgrid(-1:.1:1,-1:.1:1);
```

```
U = subs(u, [x y], {X,Y});
V = subs(v, [x y], {X,Y});
```

Now, you can plot `U` and `V` by using standard MATLAB plotting functions.

Create a plot of the vector field defined by the functions `U(X,Y)` and `V(X,Y)` by using the MATLAB `quiver` function.

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



Plot Multiple Symbolic Functions in One Graph

Plot several functions on one graph by adding the functions sequentially. After plotting the first function, add successive functions by using the `hold on` command. The `hold` `on` command keeps the existing plots. Without the `hold on` command, each new plot replaces any existing plot. After the `hold on` command, each new plot appears on top of existing plots. Switch back to the default behavior of replacing plots by using the `hold off` command.

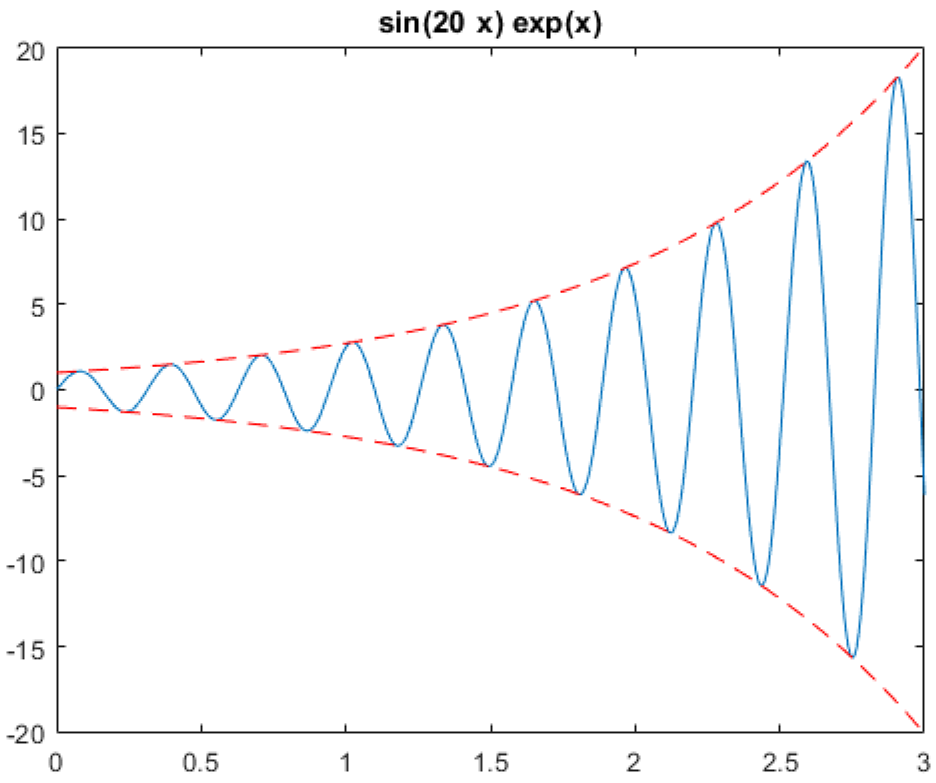
Plot $f = e^x \sin(20x)$ using `fplot`. Show the bounds of f by superimposing plots of e^x and e^{-x} as dashed red lines. Set the title by using the `DisplayName` property of the object returned by `fplot`.

```
syms x y
f = exp(x)*sin(20*x)
```

```
f =
```

$\sin(20\,x)\,e^x$

```
obj = fplot(f,[0 3]);
hold on
fplot(exp(x), [0 3], '--r')
fplot(-exp(x), [0 3], '--r')
title(obj.DisplayName)
hold off
```

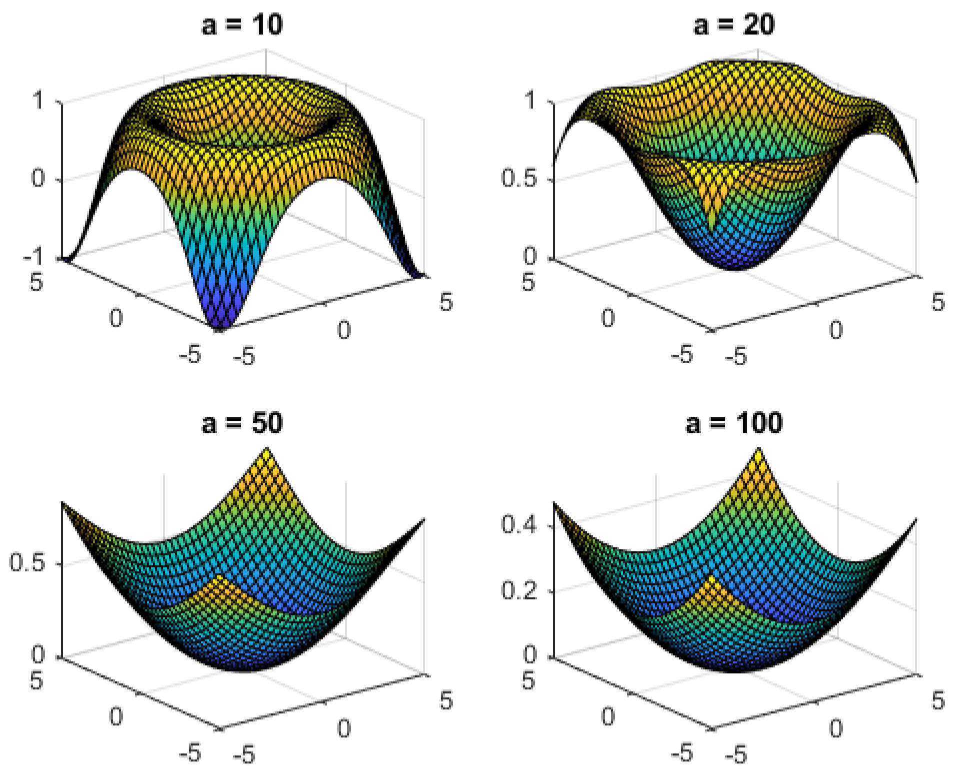


Plot Multiple Symbolic Functions in One Figure

Display several functions side-by-side in one figure by dividing the figure window into several subplots using `subplot`. The command `subplot(m,n,p)` divides the figure into a `m` by `n` matrix of subplots and selects the subplot `p`. Display multiple plots in separate subplots by selecting the subplot and using plotting commands. Plotting into multiple subplots is useful for side-by-side comparisons of plots.

Compare plots of $sin((x^2+y^2)/a)$ for $a = 10, 20, 50, 100$ by using `subplot` to create side-by-side subplots.

```
syms x y a
f = sin((x^2 + y^2)/a);

subplot(2, 2, 1)
fsurf(subs(f, a, 10))
title('a = 10')

subplot(2, 2, 2)
fsurf(subs(f, a, 20))
title('a = 20')

subplot(2, 2, 3)
fsurf(subs(f, a, 50))
title('a = 50')

subplot(2, 2, 4)
fsurf(subs(f, a, 100))
title('a = 100')
```

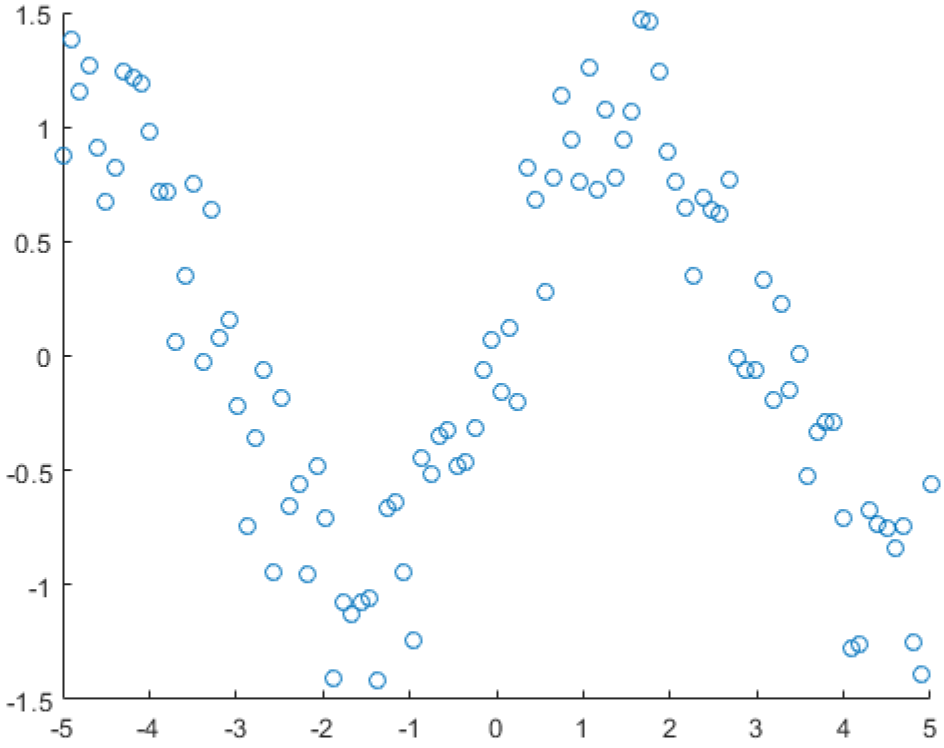


Combine Symbolic Function Plots and Numeric Data Plots

Plot numeric and symbolic data on the same graph by using MATLAB and Symbolic Math Toolbox functions together.

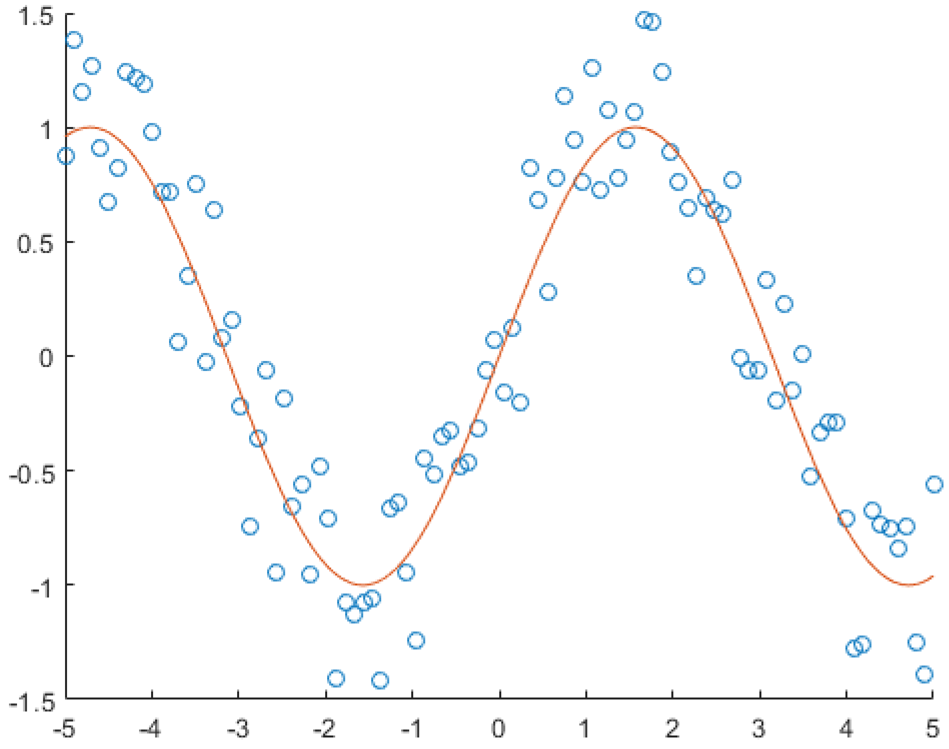
For numeric values of x between $[-5, 5]$, return a noisy sine curve by finding $y = \sin(x)$ and adding random values to y. View the noisy sine curve by using `scatter` to plot the points $(x1, y1), (x2, y2), \cdots$.

```
x = linspace(-5,5);
y = sin(x) + (-1).^randi(10, 1, 100).*rand(1, 100)./2;
scatter(x, y)
```



Show the underlying structure in the points by superimposing a plot of the sine function. First, use `hold on` to retain the scatter plot. Then, use `fplot` to plot the sine function.

```
hold on
syms t
fplot(sin(t))
hold off
```



Combine Numeric and Symbolic Plots in 3-D

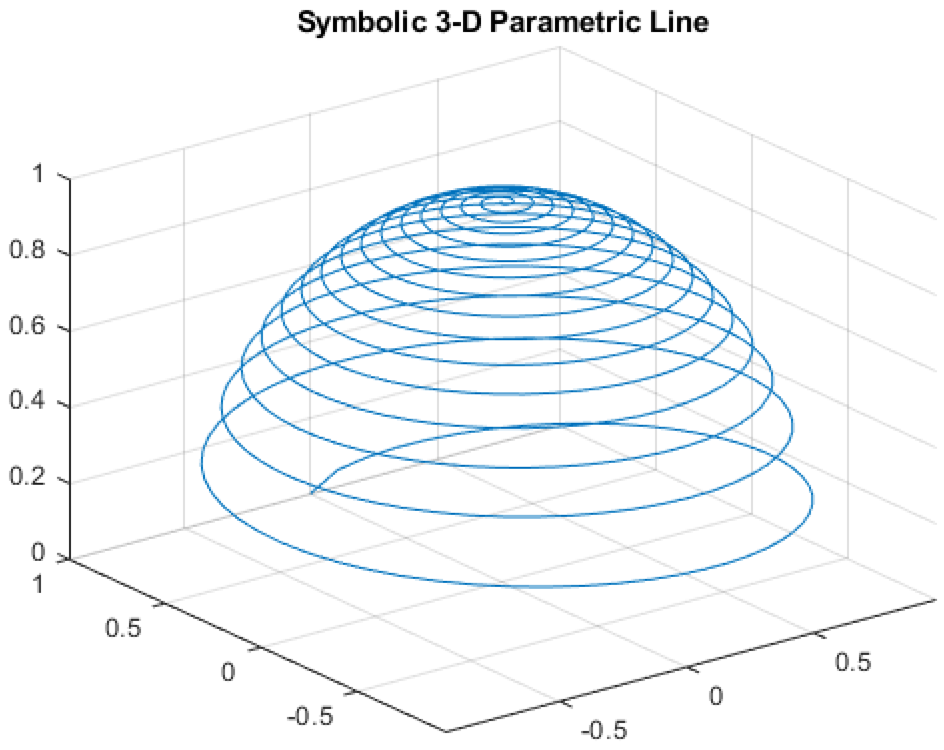
Combine symbolic and numeric plots in 3-D by using MATLAB and Symbolic Math Toolbox plotting functions. Symbolic Math Toolbox provides these 3-D plotting functions:

- `fplot3` creates 3-D parameterized line plots.
- `fsurf` creates 3-D surface plots.
- `fmesh` creates 3-D mesh plots.

Create a spiral plot by using `fplot3` to plot the parametric line

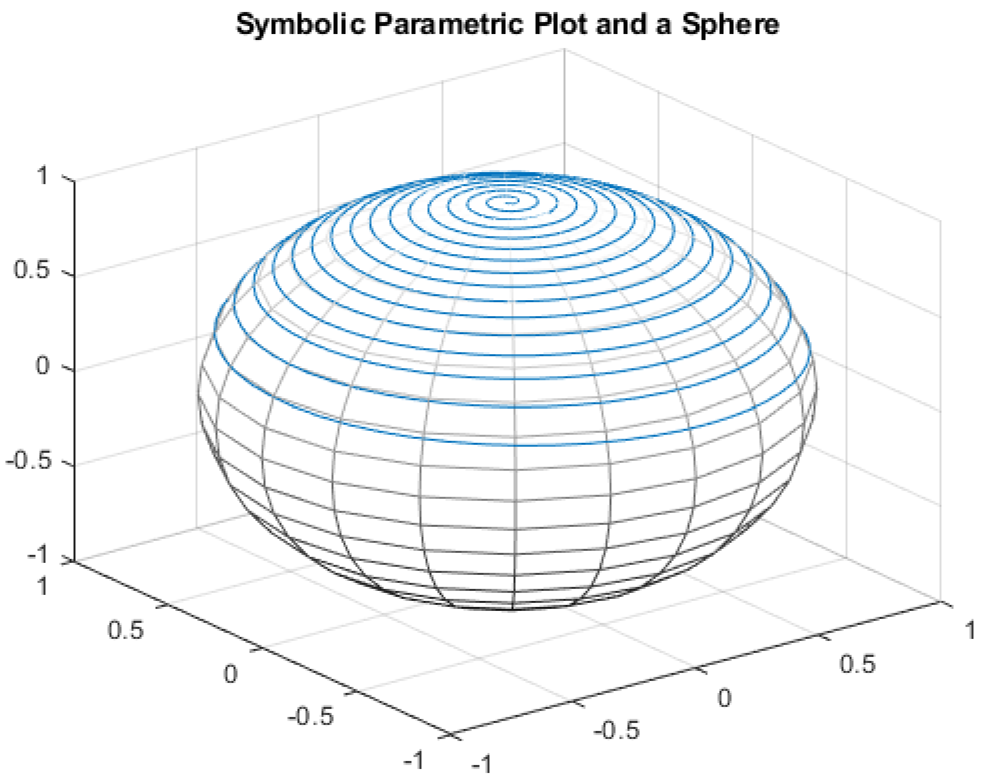
$$x = (1 - t)\sin(100t)$$
$$y = (1 - t)\cos(100t)$$
$$z = \sqrt{1 - x^2 - y^2}.$$

```
syms t
x = (1-t)*sin(100*t);
y = (1-t)*cos(100*t);
z = sqrt(1 - x^2 - y^2);
fplot3(x, y, z, [0 1])
title('Symbolic 3-D Parametric Line')
```



Superimpose a plot of a sphere with radius 1 and center at (0, 0, 0). Find points on the sphere numerically by using `sphere`. Plot the sphere by using `mesh`. The resulting plot shows the symbolic parametric line wrapped around the top hemisphere.

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



Generate C or Fortran Code from Symbolic Expressions

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.

Generate MATLAB Functions from Symbolic Expressions

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, see “Create MATLAB Functions from MuPAD Expressions” on page 3-71.

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 =
  function_handle with value:
    @(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
```

Tip Some symbolic expressions cannot be represented using MATLAB functions. `matlabFunction` cannot convert these symbolic expressions, but issues a warning. Since these expressions might result in undefined function calls, always check conversion results and verify the results by executing the resulting function.

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 2-254 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 character vectors 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 =
  function_handle with value:
    @(y,x,z)tanh(sqrt(x.^2+y.^2.*3.0+z.^2.*5.0))

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

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

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

ht3 =
  function_handle with value:
    @(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 character vector 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)
...
```

Generate MATLAB Function Blocks from Symbolic Expressions

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, see “Create MATLAB Function Blocks from MuPAD Expressions” on page 3-75.

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);
```

Tip Some symbolic expressions cannot be represented using MATLAB functions. `matlabFunctionBlock` cannot convert these symbolic expressions, but issues a warning. Since these expressions might result in undefined function calls, always check conversion results and verify results by running the simulation containing the resulting block.

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'})
```

Generate Simscape Equations from Symbolic Expressions

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 "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, see "Create Simscape Equations from MuPAD Expressions" on page 3-77.

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 =
    'if (x ~= 0.0)
         s == -atan(1.0/x^2*(u-1.0))+atan(1.0/x^2*(u+1.0));
       else
         s == NaN;
       end'
```

Clear the assumption that *x* is real:

```
syms x clear
```

Limitations

The equation section of a Simscape component file supports a limited number of functions. For details and the list of supported functions, see Simscape `equations`. If a symbolic expression contains functions that are not supported by Simscape, then `simscapeEquation` cannot represent the symbolic expression as a Simscape equation, but instead issues a warning. Always verify the conversion result. The following types of expressions are prone to invalid conversion:

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

MuPAD in Symbolic Math Toolbox

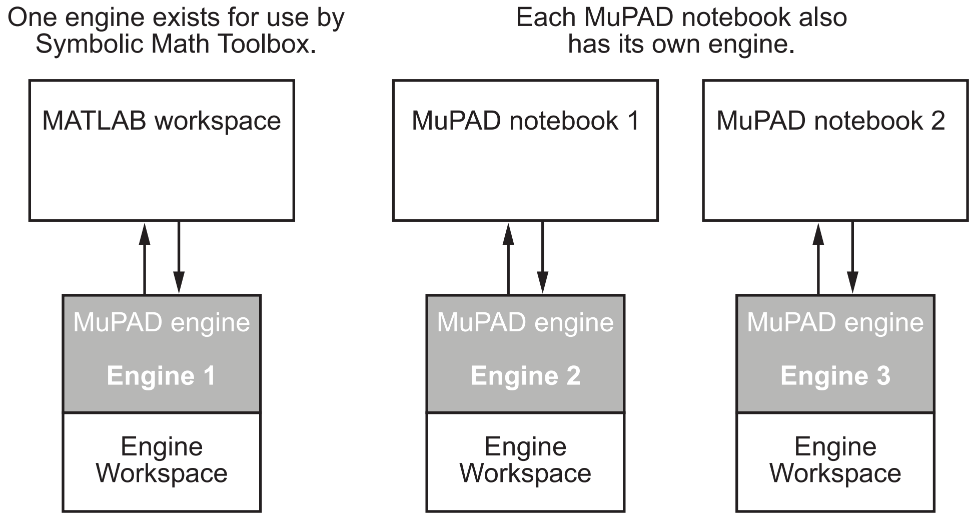
MuPAD Engines and MATLAB Workspace

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

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 3-67.

Create MuPAD Notebooks

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

Before creating a MuPAD notebook, it is best to decide which interface you intend to use primarily for your task. The two approaches are:

- Perform your computations in the MATLAB Live Editor while using MuPAD notebooks as an auxiliary tool. This approach is recommended and implies that you create a MuPAD notebook, and then execute it, transfer data and results, or close it from the MATLAB Live Editor.
- Perform your computations and obtain the results in the MuPAD Notebook. This approach is not recommended and implies that you use the MATLAB Live Editor only to access MuPAD, but do not intend to copy data and results between MATLAB and MuPAD.

If you created a MuPAD notebook without creating a handle, and then realized that you need to transfer data and results between MATLAB and MuPAD, use `allMuPADNotebooks` to create a handle to this notebook:

```
mupad
nb = allMuPADNotebooks
```

```
nb = 
Notebook1
```

This approach does not require saving the notebook. Alternatively, you can save the notebook and then open it again, creating a handle.

If You Need Communication Between Interfaces

If you perform computations in both interfaces, use handles to notebooks. The toolbox uses this handle for communication between the MATLAB workspace and the MuPAD notebook.

To create a blank MuPAD notebook from the MATLAB Command Window, type

```
nb = mupad
```

The variable `nb` is a handle to the notebook. You can use any variable name instead of `nb`.

To create several notebooks, use this syntax repeatedly, assigning a notebook handle to different variables. For example, use the variables `nb1`, `nb2`, and so on.

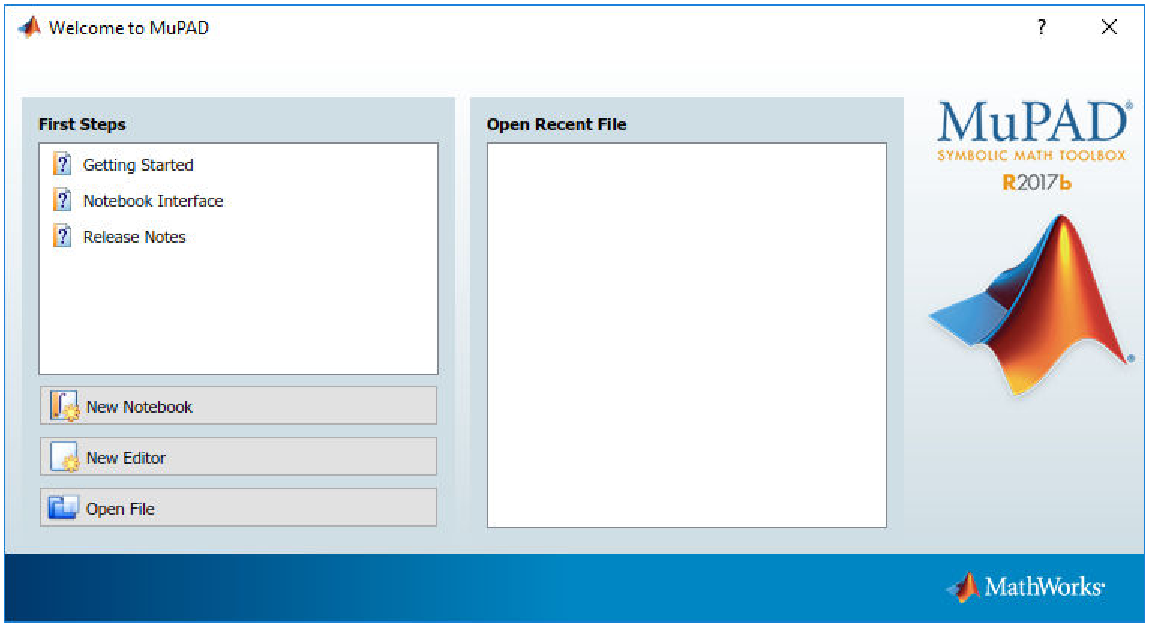
If You Use MATLAB to Access MuPAD

Use the mupad Command

To create a new blank notebook, type `mupad` in the MATLAB Command Window.

Use the Welcome to MuPAD Dialog Box

The Welcome to MuPAD dialog box lets you create a new notebook or program file, open an existing notebook or program file, and access documentation. To open this dialog box, type `mupadwelcome` in the MATLAB Command Window.



Create New Notebooks from MuPAD

If you already opened a notebook, you can create new notebooks and program files without switching to the MATLAB Live Editor:

- To create a new notebook, select **File** > **New Notebook** from the main menu or use the toolbar.
- To open a new Editor window, where you can create a program file, select **File** > **New Editor** from the main menu or use the toolbar.

Open MuPAD Notebooks

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

Before opening a MuPAD notebook, it is best to decide which interface you intend to use primarily for your task. The two approaches are:

- Perform your computations in the MATLAB Live Editor using MuPAD notebooks as an auxiliary tool. This approach is recommended and implies that you open a MuPAD notebook, and then execute it, transfer data and results, or close it from the MATLAB Live Editor. If you perform computations in both interfaces, use handles to notebooks. The toolbox uses these handles for communication between the MATLAB workspace and the MuPAD notebook.
- Perform your computations and obtain the results in MuPAD. This approach is not recommended. It implies that you use the MATLAB Live Editor only to access the MuPAD Notebook app, but do not intend to copy data and results between MATLAB and MuPAD. If you use the MATLAB Live Editor only to open a notebook, and then perform all your computations in that notebook, you can skip using a handle.

  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** or use `evaluateMuPADNotebook`. For details, see "Evaluate MuPAD Notebooks from MATLAB" on page 3-13.

If you opened a MuPAD notebook without creating a handle, and then realized that you need to transfer data and results between MATLAB and MuPAD, use `allMuPADNotebooks` to create a handle to this notebook:

```
mupad
nb = allMuPADNotebooks

nb =
Notebook1
```

This approach does not require saving changes in the notebook. Alternatively, you can save the notebook and open it again, this time creating a handle.

If You Need Communication Between Interfaces

The following commands are also useful if you lose the handle to a notebook, in which case, you can save the notebook file and then reopen it with a new handle.

Use the mupad or openmn Command

Open an existing MuPAD notebook file and create a handle to it by using `mupad` or `openmn` in the MATLAB Command Window:

```
nb = mupad('file_name')
```

```
nb1 = openmn('file_name')
```

Here, *`file_name`* must be a full path, such as `H:\Documents\Notes\myNotebook.mn`, unless the notebook is in the current folder.

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:

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

```
nb = openmn('file_name#linktarget_name')
```

Use the open Command

Open an existing MuPAD notebook file and create a handle to it by using the `open` function in the MATLAB Command Window:

```
nb1 = open('file_name')
```

Here, `file_name` must be a full path, such as `H:\Documents\Notes\myNotebook.mn`, unless the notebook is in the current folder.

If You Use MATLAB to Access MuPAD

Double-Click the File Name

You can open an existing MuPAD notebook, program file, or graphic file (`.xvc` or `.xvz`) by double-clicking the file name. The system opens the file in the appropriate interface.

Use the mupad or openmn Command

Open an existing MuPAD notebook file by using the `mupad` or `openmn` function in the MATLAB Command Window:

```
mupad('file_name')
```

```
openmn('file_name')
```

Here, `file_name` must be a full path, such as `H:\Documents\Notes\myNotebook.mn`, unless the notebook is in the current folder.

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:

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

```
openmn('file_name#linktarget_name')
```

Use the open Command

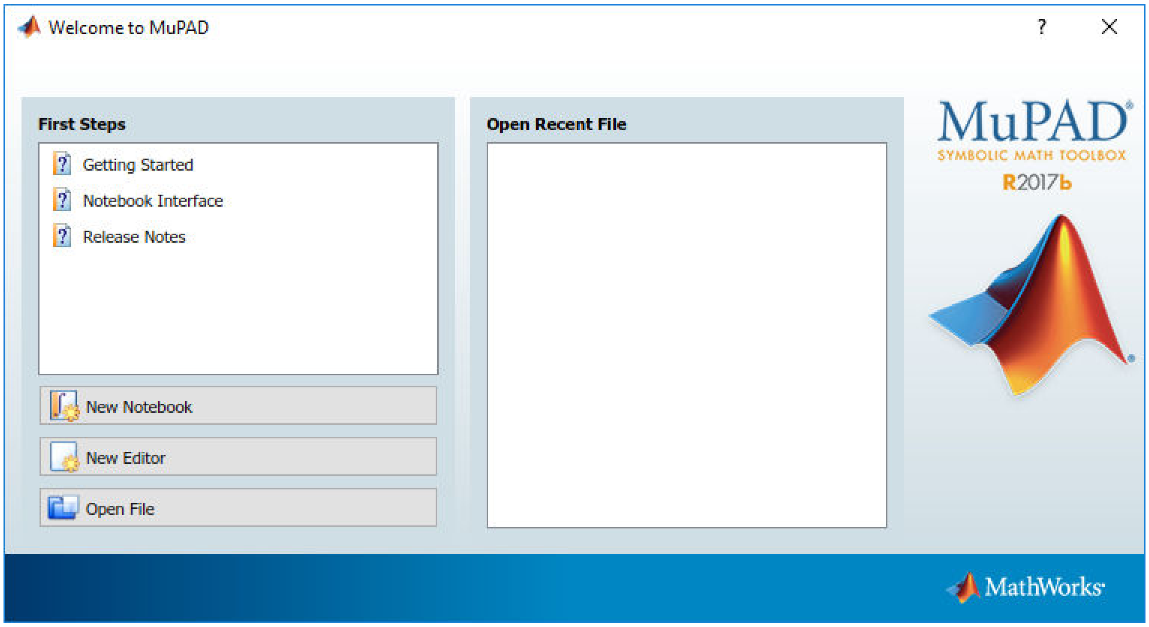
Open an existing MuPAD notebook file by using `open` in the MATLAB Command Window:

```
open('file_name')
```

Here, `file_name` must be a full path, such as `H:\Documents\Notes\myNotebook.mn`, unless the notebook is in the current folder.

Use the Welcome to MuPAD Dialog Box

The Welcome to MuPAD dialog box lets you create a new notebook or program file, open an existing notebook or program file, and access documentation. To open this dialog box, type `mupadwelcome` in the MATLAB Command Window.



Open Notebooks in MuPAD

If you already opened a notebook, you can start new notebooks and open existing ones without switching to the MATLAB Live Editor. To open an existing notebook, select **File** > **Open** from the main menu or use the toolbar. Also, you can open the list of notebooks you recently worked with.

Open MuPAD Program Files and Graphics

Besides notebooks, MuPAD lets you create and use program files (`.mu`) and graphic files (`.xvc` or `.xvz`). Also, you can use the MuPAD Debugger to diagnose problems in your MuPAD code.

Do not use a handle when opening program files and graphic files because there is no communication between these files and the MATLAB Live Editor.

Double-Click the File Name

You can open an existing MuPAD notebook, program file, or graphic file by double-clicking the file name. The system opens the file in the appropriate interface.

Use the openmn Command

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

- `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.

For example, open an existing MuPAD program file by using the `openmu` function in the MATLAB Command Window:

```
openmu('H:\Documents\Notes\myProcedure.mu')
```

You must specify a full path unless the file is in the current folder.

Use the open Command

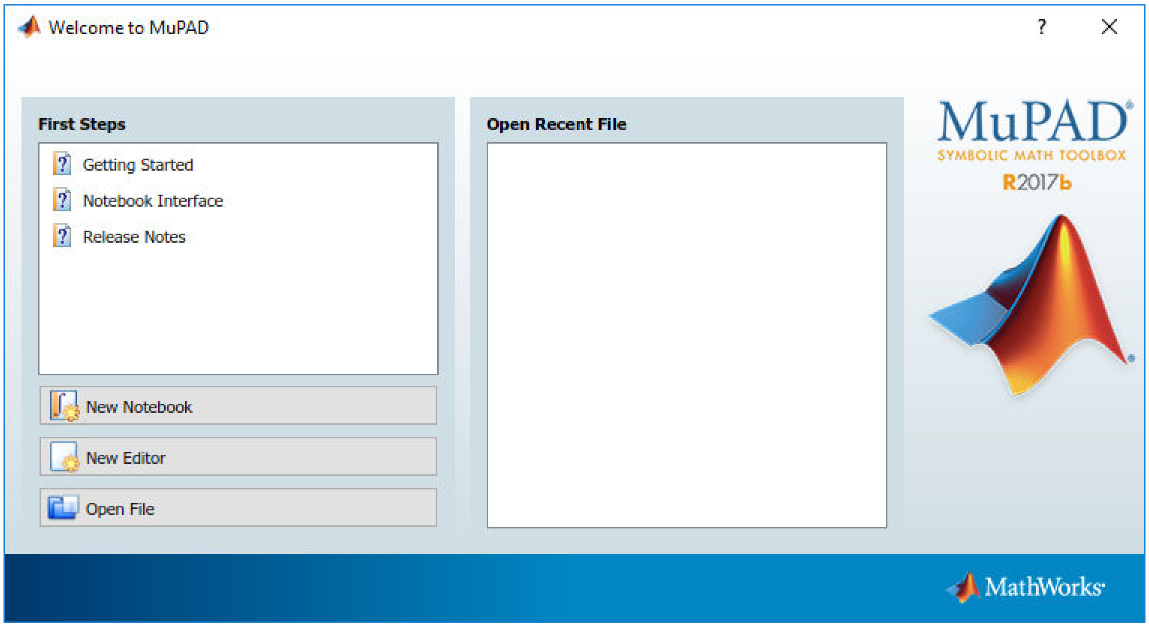
Open an existing MuPAD file by using `open` in the MATLAB Command Window:

```
open('file_name')
```

Here, `file_name` must be a full path, such as `H:\Documents\Notes \myProcedure.mu`, unless the notebook is in the current folder.

Use the Welcome to MuPAD Dialog Box

The Welcome to MuPAD dialog box lets you create a new notebook or program file, open an existing notebook or program file, and access documentation. To open this dialog box, type `mupadwelcome` in the MATLAB Command Window.



Open Program Files and Graphics from MuPAD

If you already opened a notebook, you can create new notebooks and program files and open existing ones without switching to the MATLAB Command Window. To open an existing file, select **File** > **Open** from the main menu or use the toolbar.

You also can open the Debugger window from within a MuPAD notebook. For details, see “Open the Debugger”.

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

Save MuPAD Notebooks

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

To save changes in a notebook:

1 Switch to the notebook. (You cannot save changes in a MuPAD notebook from the MATLAB Command Window.)
2 Select **File** > **Save** or **File** > **Save As** from the main menu or use the toolbar.

If you want to save and close a notebook, you can use the `close` function in the MATLAB Command Window. If the notebook has been modified, then MuPAD brings up the dialog box asking if you want to save changes. Click **Yes** to save the modified notebook.

Note 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,...)`.

Evaluate MuPAD Notebooks from MATLAB

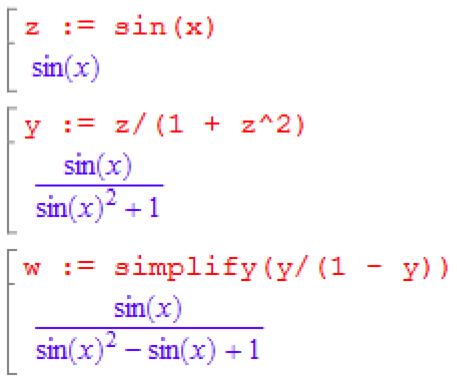
Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

When you open a saved MuPAD notebook file, the notebook displays the results (outputs), but the engine does not "remember" them. For example, suppose that you saved the notebook `myFile1.mn` in your current folder and then opened it:

```
nb = mupad('myFile1.mn');
```

Suppose that `myFile1.mn` performs these computations.



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

$$\sin(x)$$

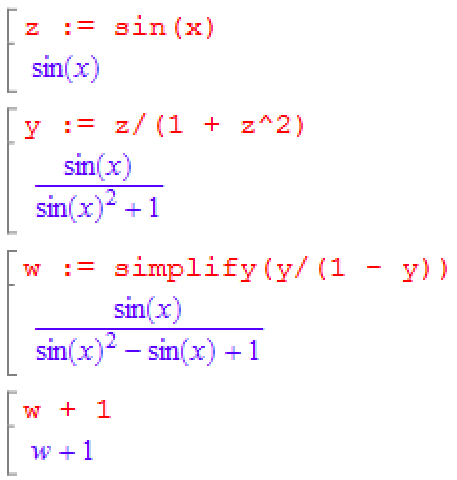
```
y := z/(1 + z^2)
```

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

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

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

Open that file and try to use the value `w` without synchronizing the notebook with its engine. The variable `w` currently has no assigned value.



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

$$\sin(x)$$

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

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

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

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

```
w + 1
```

$$w + 1$$

To synchronize a MuPAD notebook with its engine, you must evaluate the notebook as follows:

1 Open the notebooks that you want to evaluate. Symbolic Math Toolbox cannot evaluate MuPAD notebooks without opening them.

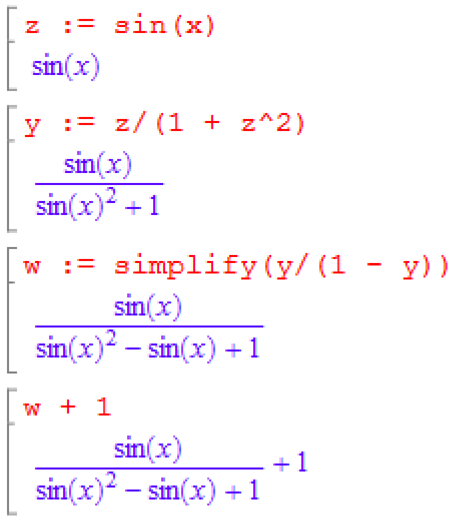
2 Use `evaluateMuPADNotebook`. Alternatively, you can evaluate the notebook by selecting **Notebook** > **Evaluate All** from the main menu of the MuPAD notebook.

3 Perform your computations using data and results obtained from MuPAD notebooks.

4 Close the notebooks. This step is optional.

For example, evaluate the notebook `myFile1.mn` located in your current folder:

```
evaluateMuPADNotebook(nb)
```



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

$$\sin(x)$$

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

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

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

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

```
w + 1
```

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

Now, you can use the data and results from that notebook in your computations. For example, copy the variables `y` and `w` to the MATLAB workspace:

```
y = getVar(nb,'y')
w = getVar(nb,'w')

y =
sin(x)/(sin(x)^2 + 1)

w =
sin(x)/(sin(x)^2 - sin(x) + 1)
```

You can evaluate several notebooks in a single call by passing a vector of notebook handles to `evaluateMuPADNotebook`:

```
nb1 = mupad('myFile1.mn');
nb2 = mupad('myFile2.mn');
evaluateMuPADNotebook([nb1,nb2])
```

Also, you can use `allMuPADNotebooks` that returns handles to all currently open notebooks. For example, if you want to evaluate the notebooks with the handles `nb1` and `nb2`, and no other notebooks are currently open, then enter:

```
evaluateMuPADNotebook(allMuPADNotebooks)
```

If any calculation in a notebook throws an error, then `evaluateMuPADNotebook` stops. The error messages appear in the MATLAB Command Window and in the MuPAD notebook. When you evaluate several notebooks and one of them throws an error, `evaluateMuPADNotebook` does not proceed to the next notebook. It stops and displays an error message immediately. If you want to skip calculations that cause errors and evaluate all input regions that run without errors, use `'IgnoreErrors',true`:

```
evaluateMuPADNotebook(allMuPADNotebooks,'IgnoreErrors',true)
```

Close MuPAD Notebooks from MATLAB

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

To close notebooks from the MATLAB Command Window, use the `close` function and specify the handle to that notebook. For example, create the notebook with the handle `nb`:

```
nb = mupad;
```

Now, close the notebook:

```
close(nb)
```

If you do not have a handle to the notebook (for example, if you created it without specifying a handle or accidentally deleted the handle later), use `allMuPADNotebooks` to return handles to all currently open notebooks. This function returns a vector of handles. For example, create three notebooks without handles:

```
mupad
mupad
mupad
```

Use `allMuPADNotebooks` to get a vector of handles to these notebooks:

```
nbhandles = allMuPADNotebooks
```

```
nbhandles =
Notebook1
Notebook2
Notebook3
```

Close the first notebook (`Notebook1`):

```
close(nbhandles(1))
```

Close all notebooks:

```
close(allMuPADNotebooks)
```

If you modify a notebook and then try to close it, MuPAD brings up the dialog box asking if you want to save changes. To suppress this dialog box, call `close` with the `'force'` flag. You might want to use this flag if your task requires opening many notebooks, evaluating them, and then closing them. For example, suppose that you want to evaluate the notebooks `myFile1.mn`, `myFile2.mn`, ..., `myFile10.mn` located in your current folder. First, open the notebooks. If you do not have any other notebooks open, you can skip specifying the handles and later use `allMuPADNotebooks`. Otherwise, do not forget to specify the handles.

```
mupad('myFile1.mn')
mupad('myFile2.mn')
...
mupad('myFile10.mn')
```

Evaluate all notebooks:

```
evaluateMuPADNotebook(allMuPADNotebooks)
```

When you evaluate MuPAD notebooks, you also modify them. Therefore, when you try to close them, the dialog box asking you to save changes will appear for each notebook. To suppress the dialog box and discard changes, use the `'force'` flag:

```
close(allMuPADNotebooks,'force')
```

Convert MuPAD Notebooks to MATLAB Live Scripts

Migrate MuPAD notebooks to MATLAB live scripts that use MATLAB code. Live scripts are an interactive way to run MATLAB code. For details, see “What Is a Live Script?” (MATLAB) MuPAD notebooks are converted to live scripts by using Symbolic Math Toolbox. For more information, see “Getting Started with Symbolic Math Toolbox”.

Convert a MuPAD Notebook .mn to a MATLAB Live Script .mlx

1 **Prepare the notebook:** This step is optional, but helps avoid conversion errors and warnings. Check if your notebook contains untranslatable objects from “MuPAD Objects That Are Not Converted” on page 3-20. These objects cause translation errors or warnings.

2 **Convert the notebook:** Use `convertMuPADNotebook`. For example, convert `myNotebook.mn` in the current folder to `myScript.mlx` in the same folder.

```
convertMuPADNotebook('myNotebook.mn','myScript.mlx')
```

Alternatively, right-click the notebook in the Current Folder browser and select **Open as Live Script**.

3 **Check for errors or warnings:** Check the output of `convertMuPADNotebook` for errors or warnings. If there are none, go to step 7. For example, this output means that the converted live script `myScript.mlx` has `4` errors and `1` warning.

```
Created ''myScript.mlx': 4 translation errors, 1 warnings. For verifying...
 the document, see help.
```

A translation error means that the translated code will not run correctly while a translation warning indicates that the code requires inspection. If the code only contains warnings, it will likely run without issues.

4 **Fix translation errors**: Open the converted live script by clicking the link in the output. Find errors by searching for `ERROR`. The error explains which MuPAD command did not translate correctly. For details and fixes, click `ERROR`. After fixing the error, delete the error message. For the list of translation errors, see “Troubleshoot MuPAD to MATLAB Translation Errors” on page 3-25. If you cannot fix your error, and the “Known Issues” on page 3-20 do not help, please contact technical support.

5 **Fix translation warnings:** Find warnings by searching for `WARNING`. The warning text explains the issue. For details and fixes, click `WARNING`. Decide to either adapt

the code or ignore the warning. Then delete the warning message. For the list of translation warnings, see “Troubleshoot MuPAD to MATLAB Translation Warnings” on page 3-35.

6 **Verify the live script:** Open the live script and check for unexpected commands, comments, formatting, and so on. For readability, the converted code may require manual cleanup, such as eliminating auxiliary variables.

7 **Execute the live script:** Ensure that the code runs properly and returns expected results. If the results are not expected, check your MuPAD code for the “Known Issues” on page 3-20 listed below.

Known Issues

These are the known issues when converting MuPAD notebooks to MATLAB live scripts with the `convertMuPADNotebook` function. If your issue is not described, please contact technical support.

- “MuPAD Objects That Are Not Converted” on page 3-20
- “No Automatic Substitution in MATLAB” on page 3-21
- “last(1) in MuPAD Is Not ans in MATLAB” on page 3-21
- “Some solve Results Are Wrongly Accessed” on page 3-22
- “break Inside case Is Wrongly Translated” on page 3-22
- “Some MuPAD Graphics Options Are Not Translated” on page 3-23
- “Some Operations on Matrices Are Wrongly Translated” on page 3-23
- “indets Behavior in MATLAB Differs” on page 3-24
- “Return Type of factor Differs in MATLAB” on page 3-24
- “Layout Issues” on page 3-24
- “Syntax Differences Between MATLAB and MuPAD” on page 3-24

MuPAD Objects That Are Not Converted

Expand the list to view MuPAD objects that are not converted. To avoid conversion errors and warnings, remove these objects or commands from your notebook before conversion.

Objects Not Converted

- Reading code from files. Replace commands such as `read("filename.mu")` by the content of `filename.mu`.
- Function calls with expression sequences as input arguments.
- Function calls where the function is generated by the preceding code instead of being specified explicitly.
- Domains, and commands that create domains and their elements.
- Assignments to slots of domains and function environments.
- Commands using the history mechanism, such as `last(2)` or `HISTORY := 30`.
- MuPAD environment variables, such as `ORDER`, `HISTORY`, and `LEVEL`.

No Automatic Substitution in MATLAB

In MATLAB, when symbolic variables are assigned values, then expressions containing those values are not automatically updated.

Fixing This Issue

When values are assigned to variables, update any expressions that contain those variables by calling `subs` on those expressions.

```
syms a b
f = a + b;
a = 1;
b = 2;
f        % f is still a + b
subs(f)        % f is updated

f =
a + b
ans =
3
```

last(1) in MuPAD Is Not ans in MATLAB

In MuPAD, `last(1)` always returns the last result. In MATLAB, `ans` returns the result of the last *unassigned* command. For example, in MATLAB if you run `x = 1`, then calling `ans` does not return `1`.

Fixing This Issue

Instead of using `ans`, assign the result to a variable and use that variable.

Some solve Results Are Wrongly Accessed

When results of MuPAD `solve` are accessed, `convertMuPADNotebook` assumes that the result is a finite set. However, if the result is a non-finite set then the code is wrongly translated.

Fixing This Issue

There is no general solution. Further, non-finite solution sets are not translatable.

If you are accessing parameters or conditions, use the `parameters` or `conditions` output arguments of MATLAB `solve`.

```
syms x
S = solve(sin(x) == 1, x, 'ReturnConditions', true);
S.x        % solution
S.parameters    % parameters in solution
S.conditions    % conditions on solution
```

```
ans =
pi/2 + 2*pi*k
ans =
k
ans =
in(k, 'integer')
```

break Inside case Is Wrongly Translated

In MuPAD, a `break` ends a case in a switch case. However, MATLAB does not require a `break` to end a case. Thus, a MuPAD `break` introduces an unnecessary `break` in MATLAB. Also, if a MuPAD case omits a `break`, then the MATLAB case will not fall-through.

Fixing This Issue

In the live script, delete `break` statements that end cases in a switch-case statement.

For fall-through in MATLAB, specify all values with their conditions in one case.

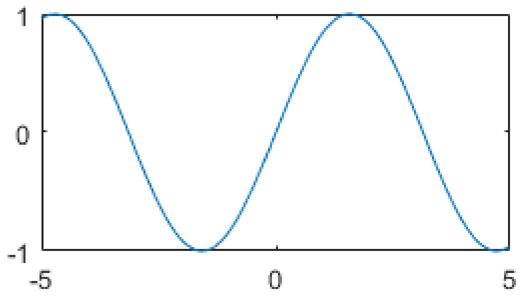
Some MuPAD Graphics Options Are Not Translated

While the most commonly used MuPAD graphics options are translated, there are some options that are not translated.

Fixing This Issue

Find the corresponding option in MATLAB by using the properties of the figure handle `gcf` or axis handle `gca`. For example, the MuPAD command `plot(sin(x), Width = 80*unit::mm, Height = 4*unit::cm)` sets height and width. Translate it to MATLAB code.

```
syms x
fplot(sin(x));
g = gcf;
g.Units = 'centimeters';
g.Position(3:4) = [8 4];
```



Some Operations on Matrices Are Wrongly Translated

Operations on matrices are not always translated correctly. For example, if `M` is a matrix, then `exp(M)` in MuPAD is wrongly translated to `exp(M)` instead of the matrix exponential `expm(M)`.

Fixing This Issue

When performing operations on matrices, search for the matrix operation and use it instead. For example, in MATLAB:

- Use `expm` instead of `exp`.
- Use `funm(M,'sin')` instead of `sin(M)`.

- `A == [1 2; 3 4]` displays differently from `A = matrix([[1, 2], [3, 4]])` in MuPAD but is programmatically equivalent.

indets Behavior in MATLAB Differs

`indets` is translated to MATLAB `symvar`. However, `symvar` does not find bound variables or constant identifiers like `PI`.

Return Type of factor Differs in MATLAB

The return type of MuPAD `factor` has no equivalent in MATLAB. Subsequent operations on the results of `factor` in MATLAB might return incorrect results.

Fixing This Issue

Check and modify the output of `factor` in MATLAB as required such that subsequent commands run correctly.

Layout Issues

- MuPAD notebook frames are not converted.
- MuPAD notebook tables are not converted.
- MuPAD plots are not interactive in live scripts.
- Titles or headings in MuPAD notebooks are not always detected.
- MuPAD text attribute `underline` is not converted
- Text formatting: Font, font size, and color are not converted. All text in live scripts looks the same.

Syntax Differences Between MATLAB and MuPAD

For the syntax differences between MATLAB and MuPAD, see “Differences Between MATLAB and MuPAD Syntax” on page 3-49.

Troubleshoot MuPAD to MATLAB Translation Errors

This page helps troubleshoot all errors generated by the `convertMuPADNotebook` function when converting MuPAD notebooks to MATLAB live scripts. For the conversion steps, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19. To troubleshoot warnings, see “Troubleshoot MuPAD to MATLAB Translation Warnings” on page 3-35.

Error Message	Details	Recommendations
No equivalent code in MATLAB.	`convertMuPADNotebook` cannot find the corresponding functionality in MATLAB.	Adjust the code so that it uses only the functionality that can be expressed in the MATLAB language. Alternatively, in the target `.mlx` file, some functionality can be replaced with MATLAB functionality, such as in statistics or file input-output.
Unable to translate the second and higher derivatives of Airy functions. Express these derivatives in terms of Airy functions and their first derivatives.	The MATLAB `airy` function represents Airy functions of the first and second kind and their first derivatives. In MuPAD, `airyAi(z,n)` and `airyBi(z,n)` can represent second and higher derivatives of Airy functions, that is, `n` can be greater than 1.	Rewrite second and higher derivatives of Airy functions in terms of Airy functions and their first derivatives. Then convert the result to MATLAB code. The MuPAD `airyAi` and `airyBi` functions return results in terms of Airy functions and their first derivatives. You can replace second and higher derivatives by their outputs in MuPAD, before converting the code to MATLAB.

Error Message	Details	Recommendations
Unable to translate assignment to MuPAD environment variable.	Environment variables are global variables, such as `HISTORY`, `LEVEL`, `ORDER`, and so on, that affect the behavior of MuPAD algorithms.	In some cases, you can use name-value pair arguments in each function call, such as setting the value `Order` in the `taylor` function call. In other cases, there is no appropriate replacement. Adjust the code so that it does not require a global setting.
Unable to translate assignments to the remember table of a procedure.	MuPAD uses *remember tables* to speed up computations, especially when you use recursive procedure calls. The system stores the arguments of a procedure call as indices of the remember table entries, and the corresponding results as values of these entries. When you call a procedure using the same arguments as in previous calls, MuPAD accesses the remember table of that procedure. If the remember table contain s the entry with the required arguments, MuPAD returns the value of that entry. For details, see “Remember Mechanism”. The remember tables are not available in MATLAB.	Adjust the code so that it does not use remember tables.

Error Message	Details	Recommendations
Unable to translate assignments to slots of domains and function environments.	In MuPAD, the `slot` function defines methods and entries of data types (domains) or for defining attributes of `function environments`. These methods and entries (slots) let you overload system functions by your own domains and function environments. Domains, function environments, and their slots are not available in MATLAB.	Adjust the code so that it does not use assignments to slots of domains and function environments.
Unable to translate explicitly given coefficient ring.	MuPAD lets you use special coefficient rings that cannot be represented by arithmetical expressions. Specifying coefficient rings of polynomials is not available in MATLAB.	Adjust the code so that it does not use polynomials over special rings.
Unable to translate complexInfinity.	MuPAD uses the value `complexInfinity`. This value is not available in MATLAB.	Adjust the code so that it does not use `complexInfinity`.
Unable to translate MuPAD code because it uses an obsolete calling syntax.	MuPAD syntax has changed and the code uses obsolete syntax that is no longer supported.	Update code to use current MuPAD syntax by checking MuPAD documentation and then run `convertMuPADNotebook` again.

Error Message	Details	Recommendations
Unable to translate a call to the function 'D' with more than one argument.	The indices in the first argument of `D` cannot be translated to variable names in MATLAB.	Use the MuPAD `diff` function instead of `D`.
Unable to translate MuPAD domains, or commands to create domains or their elements.	Domains represent data types in MuPAD. They are not available in MATLAB.	Adjust the code so that it does not create or explicitly use domains and their elements.
Unable to translate the MuPAD environment variable "{0}".	Environment variables are global variables, such as `HISTORY`, `LEVEL`, `ORDER`, and so on, that affect the behavior of MuPAD algorithms. `convertMuPADNotebook` cannot translate MuPAD environment variables because they are not available in MATLAB.	Adjust the code so that it does not require accessing MuPAD environment variables.
Unable to translate function calls with expression sequences as input arguments.	In MuPAD, a function call `f(x)`, where `x` is a sequence of `n` operands, resolves to a call with `n` arguments. MATLAB cannot resolve function calls with expression sequences to calls with multiple arguments.	Adjust the code so that it does not contain function calls with expression sequences as input arguments.

Error Message	Details	Recommendations
Unable to translate infinite sets.	MuPAD recognizes infinite sets. For example, `solve` can return a solution as an infinite set: `solve(sin(x*PI/2) = 0, x)` returns $\left\{2k \middle\| k \in \mathbb{Z}\right\}$. You can create such sets by using `Dom::ImageSet`. MATLAB does not support infinite sets.	Adjust the code so that it does not use infinite sets as inputs.
Unable to translate a call accessing previously computed results. The MATLAB ans function lets you access only the most recent result.	The MuPAD `last` function and its shortcut `%` typically let you access the last 20 commands stored in an internal history table. In MATLAB, `ans` lets you access only one most recent command.	Adjust the code so that it uses assignments instead of relying on `last` or `%`.
Unable to translate the variable "{0}" representing a MuPAD library.	Libraries contain most of the MuPAD functionality. Each library includes a collection of functions for solving particular types of mathematical problems. While MuPAD library functions are translated to MATLAB code, the libraries themselves are not.	Adjust the code so that it does not use MuPAD library names as identifiers.
Unable to map a function to objects of this class.	Objects of this class do not have an equivalent representation in MATLAB. The mapping cannot be translated.	In the target `.mlx` file, implement the mapping by writing a loop.

Error Message	Details	Recommendations
Unable to translate this form of matrix definition.	MuPAD provides a few different approaches for creating a matrix. You can create a matrix from an array, list of elements, a nested list of rows, or a table. Also, you can create a matrix by specifying only the nonzero entries, such as `A[i1,j1] = value1`, `A[i2,j2] = value2`, and so on. Some of these approaches cannot be translated to MATLAB code.	Adjust the code so that it defines matrices by using an array, list of elements, or a nested list of rows.
Cannot translate division with respect to several variables.	Polynomial division with respect to several variables is not available in MATLAB.	Adjust the code so that it does not use polynomial division with respect to several variables.
Unable to translate nested indexed assignment.	Nested indexed assignment is not available in MATLAB.	Replace the nested indexed assignment with multiple assignments.
Unable to create a polynomial from a coefficient list.	Cannot translate polynomial creation from the given coefficient list.	Make the first argument to `poly` an arithmetical expression instead of a list.

Error Message	Details	Recommendations
Unable to translate nontrivial procedures.	For code that you want to execute repeatedly, MuPAD lets you create procedures by using the `proc` command. `convertMuPADNotebook` can translate simple procedures to anonymous functions. Simple procedures do not contain loops, assignments, multiple statements, or nested functions where the inner function accesses variables of the outer function. More complicated procedures cannot be translated to MATLAB code.	Adjust the code so that it does not use complicated procedures.
Unable to translate the global table of properties.	`convertMuPADNotebook` cannot translate the MuPAD global table of properties, `PROPERTIES`, because this functionality is not available in MATLAB.	Set properties and assumptions as described in “Properties and Assumptions”.
Unable to create random generators with individual seed values.	MuPAD lets you set a separate seed value for each random number generator. MATLAB has one seed value for all random number generators. See `rng` for details.	Adjust the code so that it does not rely on individual seed values for different random number generators.

Error Message	Details	Recommendations
Unable to translate target "{0}" for MATLAB function "rewrite".	The MuPAD `rewrite` function can rewrite an expression in terms of the following targets: `andor`, `arccos`, `arccosh`, `arccot`, `arccoth`, `arcsin`, `arcsinh`, `arctan`, `arctanh`, `arg`, `bernoulli`, `cos`, `cosh`, `cot`, `coth`, `diff`, `D`, `erf`, `erfc`, `erfi`, `exp`, `fact`, `gamma`, `harmonic`, `heaviside`, `inverf`, `inverfc`, `lambertW`, `ln`, `max`, `min`, `piecewise`, `psi`, `sign`, `sin`, `sincos`, `sinh`, `sinhcosh`, `tan`, `tanh`. The MATLAB `rewrite` function supports fewer targets: `exp`, `log`, `sincos`, `sin`, `cos`, `tan`, `cot`, `sqrt`, `heaviside`, `asin`, `acos`, `atan`, `acot`, `sinh`, `cosh`, `tanh`, `coth`, `sinhcosh`, `asinh`, `acosh`, `atanh`, `acoth`, `piecewise`.	Adjust the code so that it uses the target options available in MATLAB. If needed, use a sequence of function calls to `rewrite` with different target options.
Unable to translate slots of domains and function environments.	Slots and domains are not available in MATLAB.	Adjust the code so that it does not use slots or domains.

Error Message	Details	Recommendations
Unable to substitute only one occurrence of a subexpression.	Substituting only one occurrence of a subexpression is not available in MATLAB.	In the target `.mlx` file, break up the expression using the function `children` to get the subexpression, and then substitute for it using the function `subs`.
Syntax error in MuPAD code.	MuPAD code contains a syntax error, for example, a missing bracket.	Check and correct the MuPAD code that you are translating.
Test environment of MuPAD not available in MATLAB.	The MuPAD test environment is not available in MATLAB.	Adjust the code so that it does not use the MuPAD test environment.
Unknown domain or library "{0}".	Most likely, a custom domain or library is used and cannot be translated.	Check and correct the MuPAD code that you are translating.
Unknown MuPAD function "{0}".	The function is not available in MuPAD.	Check and correct the MuPAD code that you are translating.
Unable to translate calls to the function "{0}".	The function is a valid MuPAD function, but the function call is invalid. For example, the number of input arguments or types of arguments can be incorrect.	Check and correct the MuPAD code that you are translating.
Unable to translate calls to functions of the library "{0}".	The functions of this library are available in MuPAD, but there are no corresponding functions in MATLAB.	Adjust the code so that it does not use the functions of this library.
MuPAD function "{0}" cannot be converted to function handle.	The MuPAD function does not have an equivalent function handle in MATLAB.	Adjust the code to use a function that has an equivalent in MATLAB.

Error Message	Details	Recommendations
Unable to translate option "{0}".	Most likely, this option is available in MuPAD, but there are no corresponding options in MATLAB.	Adjust the code so that it does not use this option.
Unable to translate MuPAD code because it uses invalid calling syntax.	Most likely, the function call in the MuPAD code has an error.	Check and correct the MuPAD code that you are translating.

Troubleshoot MuPAD to MATLAB Translation Warnings

This page helps troubleshoot all warnings generated by the `convertMuPADNotebook` function when converting MuPAD notebooks to MATLAB live scripts. For the conversion steps, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19. To troubleshoot errors, see “Troubleshoot MuPAD to MATLAB Translation Errors” on page 3-25.

Warning Message	Meaning	Recommendations
Translating the alias function as an assignment, and the unalias function as deletion of an assignment.	The MuPAD `alias` and `unalias` function let you create and delete an alias (abbreviation) for any MuPAD object. For example, you can create an alias `d` for the `diff` function: `alias(d = diff)`. Creating aliases is not available in MATLAB. When translating a notebook file, `convertMuPADNotebook` replaces aliases with assignments.	Verify the resulting code. If you do not want a MuPAD alias to be converted to an assignment in MATLAB, adjust the code so that it does not use aliases.
Replacing animation by its last frame.	MuPAD animations cannot be correctly reproduced in MATLAB. When translating a notebook file, `convertMuPADNotebook` replaces an animation with a static image showing the last frame of the animation.	Verify the resulting code. The last frame might not be ideal for some animations. If you want the static image to show any other frame of the animation, rewrite the MuPAD code so that it creates a static plot showing that image.

Warning Message	Meaning	Recommendations
Potentially incorrect MuPAD code "{0}". Replacing it by "{1}".	When translating a notebook file, `convertMuPADNotebook` detected that the part of the code in the MuPAD notebook might be incorrect. For example, the code appears to have a typo, or a commonly used argument is missing. `convertMuPADNotebook` corrected it.	Verify the corrected code. Then delete this warning.
Invalid assignment to remember table. Replacing it by procedure definition.	When translating a notebook file, `convertMuPADNotebook` considered an assignment to a remember table in a MuPAD notebook as unintentional, and replaced it by a procedure definition. For example, an assignment such as `f(x):=x^2` gets replaced by `f:= x->x^2`.	Verify the corrected code. Then delete this warning.

Warning Message	Meaning	Recommendations
Replacing MuPAD domain by an anonymous function that creates objects similar to the elements of this domain.	Domains represent data types in MuPAD. They are not available in MATLAB. `convertMuPADNotebook` translated a MuPAD domain to a MATLAB anonymous function that creates objects similar to the elements of the domain. For example, the code line `f:=Dom::IntegerMod(7)` gets translated to a MATLAB anonymous function `f = @(X)mod(X,sym(7))`.	Verify the resulting code. Check if an anonymous MATLAB function is the correct translation of the domain in this case, and that the code still has the desired functionality.
Ignoring addpattern command. Configurable pattern matcher not available in MATLAB.	`addpattern` functionality is not available in MATLAB.	Adjust the code to avoid using `addpattern`.
Ignoring assertions.	Assertions are not available in MATLAB. When translating a notebook file, `convertMuPADNotebook` ignores assertions.	Verify the resulting code. If assertions are not essential part of your code, you can ignore this warning. However, if your code relies on assertions, you can implement them using conditional statements, such as `if-then`.

Warning Message	Meaning	Recommendations
Ignoring assignment to a MuPAD environment variable.	Environment variables are global variables, such as `HISTORY`, `LEVEL`, `ORDER`, and so on, that affect the behavior of MuPAD algorithms.	Verify the resulting code. If an assignment to an environment variable is not essential for your code, simply delete the warning. In some cases, you can use name-value pair arguments in each function call, such as setting the value `Order` in the `taylor` function call. In other cases, there is no appropriate replacement. Adjust the code so that it does not require a global setting.
Ignoring assignment to a protected MuPAD constant or function.	The names of the built-in MuPAD functions, options, and constants are protected. If you try to assign a value to a MuPAD function, option, or constant, the system throws an error. This approach ensures that you will not overwrite a built-in functionality accidentally. See “Protect Function and Option Names”.	Verify the resulting code. Check if the ignored assignment is essential for the correctness of the code and results. If it is, adjust the code so that it does not use this assignment, but still has the desired functionality. If it is not essential, simply delete this warning.
Ignoring option "hold".	`hold` is not available in MATLAB.	Adjust the code to avoid using `hold`.
Ignoring info command. Information not available in MATLAB.	MATLAB functions do not have associated information.	For information on a function, refer to MATLAB documentation.

Warning Message	Meaning	Recommendations
Ignoring options "{0}".	These options are available in MuPAD, but are not available in MATLAB. Because they do not appear to be essential for this code, `convertMuPADNotebook` ignores them.	Verify the resulting code. Check if the ignored options are essential for the correctness of the code and results. If they are, adjust the code so that it does not use these options, but still has the desired functionality. If they are not essential, simply delete this warning.
Ignoring MuPAD path variables.	The MuPAD environment variables `FILEPATH`, `NOTEBOOKPATH`, `WRITEPATH`, and `READPATH` let you specify the working folders for writing new files, searching for files, loading files, and so on if you do not specify the full path to the file. These environment variables are not available in MATLAB.	Verify the resulting code. Check if the ignored path variables are essential for the correctness of the code and results. If they are, adjust the code so that it does not use these preferences, but still has the desired functionality. If they are not essential, simply delete this warning.

Warning Message	Meaning	Recommendations
Ignoring MuPAD preference because there is no equivalent setting in MATLAB.	The MuPAD `Pref` library provides a collection of functions which can be used to set and restore preferences, such as use of abbreviations in outputs, representation of floating-point numbers, memory limit on a MuPAD session, and so on. MATLAB uses `sympref` for a few preferences, such as specifying parameters of Fourier transforms, specifying the value of the Heaviside function at 0, or enabling and disabling abbreviations in outputs. Most preferences cannot be translated to MATLAB code.	Verify the resulting code. Check if the ignored preferences are essential for the correctness of the code and results. If they are not essential, simply delete this warning.

Warning Message	Meaning	Recommendations
Ignoring call to variable protection mechanism.	The names of the built-in MuPAD functions, options, and constants are protected. If you try to assign a value to a MuPAD function, option, or constant, the system throws an error. This approach ensures that you will not overwrite a built-in functionality accidentally. See “Protect Function and Option Names”. Protecting procedures and functions from overwriting is not available in MATLAB. When translating a notebook file, `convertMuPADNotebook` ignores the corresponding MuPAD code.	Verify the resulting code. Check if the ignored call to variable protection mechanism is essential for the correctness of the code and results. If it is, adjust the code so that it does not use this call, but still has the desired functionality. If it is not essential, simply delete this warning.

Warning Message	Meaning	Recommendations
Ignoring default value when translating a table.	MuPAD tables let you set the default value. This value is returned when you index into a table using the index for which the entry does not exist. For example, if you create the table using `T := table(a = 13,c = 42,10)`, and then index into it using `T[b]`, the result is `10`. Default values for tables cannot be translated to MATLAB. When translating a notebook file, `convertMuPADNotebook` ignores the corresponding value.	Verify the resulting code. Check if the ignored value is essential for the correctness of the code and results. If default values for the tables are not essential, simply delete this warning. Otherwise, you can create a MATLAB function that checks if the `containers.Map` object corresponding to the MuPAD table has a certain key, and if it does not, returns the default value.
Unable to decide which object the indexing refers to, instead using generic translation.	When the class of the object being indexed into is ambiguous, then `convertMuPADNotebook` defaults to a generic translation for the indexing.	Verify that the generic translation returns the correct result. If not, adjust the code.
Possibly missing a multiplication sign.	Do not skip multiplication signs in MuPAD and MATLAB code. Both languages require you to type multiplication signs explicitly. For example, the expression `x(x + 1)` must be typed as `x*(x + 1)`.	Verify the converted code. Check if you missed a multiplication sign. Correct the code if needed.
Expression used as operator. Possibly "subs" was intended.	An arithmetical expression is used as a function. `convertMuPADNotebook` attempted to fix the error.	Verify that the translation returns the correct result. If not, adjust the code.

Warning Message	Meaning	Recommendations
MuPAD package mechanism not available in MATLAB.	The MuPAD package mechanism is not available in MATLAB.	Adjust the code to avoid using the MuPAD package mechanism.

Edit MuPAD Code in MATLAB Editor

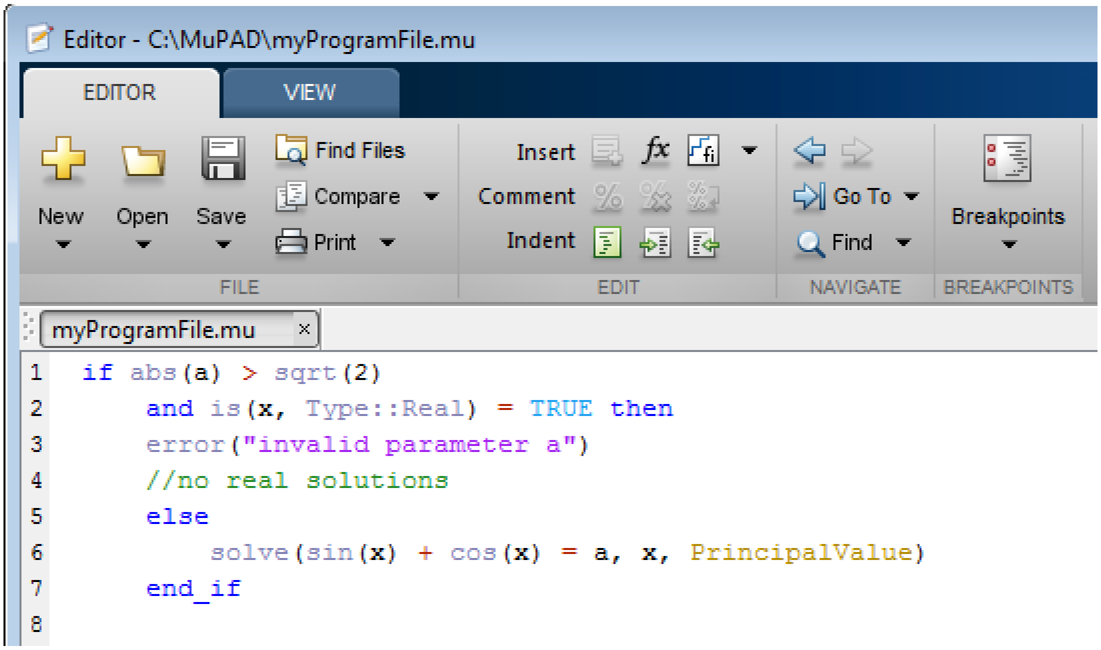
Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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.

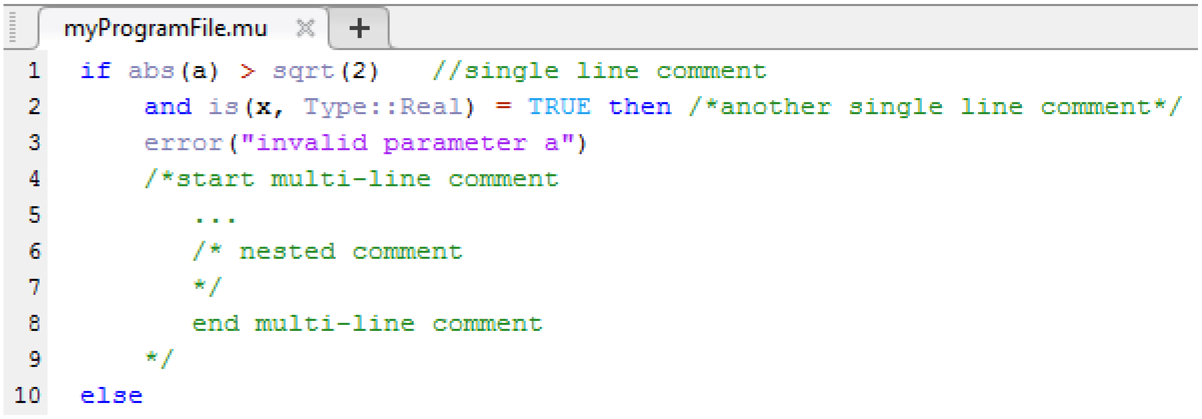


```
if abs(a) > sqrt(2)
    and is(x, Type::Real) = TRUE then
    error("invalid parameter a")
    //no real solutions
    else
        solve(sin(x) + cos(x) = a, x, PrincipalValue)
    end_if

```

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`.

Comments in MuPAD Procedures

Enter a comment in a `.mu` file by entering the `//` characters. All text following the `//` on the same line is ignored. The `//` characters do not affect text on succeeding lines. To create a multi-line comment, start with the `/*` characters and end the comment with the `*/` characters. All text between these characters is ignored. You can nest comments using `/*` and `*/`.



```
if abs(a) > sqrt(2)   //single line comment
    and is(x, Type::Real) = TRUE then /*another single line comment*/
    error("invalid parameter a")
    /*start multi-line comment
       ...
       /* nested comment
       */
       end multi-line comment
    */
else
```

Notebook Files and Program Files

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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 MuPAD Notebook. 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 MuPAD Notebook. 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 3-63.

Source Code of the MuPAD Library Functions

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

You can display the source code of the MuPAD built-in library functions. If you work in the MuPAD Notebook app, enter `expose(name)`, where `name` is the library function name. The MuPAD Notebook 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 ~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

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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	MuPAD Syntax	MATLAB Syntax
Assignment	`:=`	`=`
List variables	`anames(All, User)`	`whos`
Numerical value of expression	`float(`*`expression`*`)`	`double(`*`expression`*`)`
Suppress output	`:`	`;`
Enter matrix	`matrix([[x11,x12,x13], [x21,x22,x23]])`	`[x11,x12,x13; x21,x22,x23]`
Translate MuPAD set	`{a,b,c}`	`unique([1 2 3])`
Auto-completion	**Ctrl+space bar**	**Tab**
Equality, inequality comparison	=, <>	==, ~=

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

MATLAB vs. MuPAD Expressions

MuPAD Expression	MATLAB Expression
`infinity`	`Inf`
`PI`	`pi`
`I`	`i`
`undefined`	`NaN`
`trunc`	`fix`
`arcsin`, `arccos` etc.	`asin`, `acos` etc.
`numeric::int`	`vpasolve`
`normal`	`simplifyFraction`
`besselJ`, `besselY`, `besselI`, `besselK`	`besselj`, `bessely`, `besseli`, `besselk`
`lambertW`	`lambertw`
`Si`, `Ci`	`sinint`, `cosint`
`EULER`	`eulergamma`
`conjugate`	`conj`
`CATALAN`	`catalan`

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

	Symbolic Math Toolbox Definition	MuPAD Definition
Exponential integral	$\mathrm{expint}(x) = -\mathrm{Ei}(-x) =$ $\int_x^\infty \frac{\exp(-t)}{t}dt \text{ for } x > 0 =$ $\mathrm{Ei}(1, x).$	$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}dt \text{ for } x < 0.$ $\mathrm{Ei}(n, x) = \int_1^\infty \frac{\exp(-xt)}{t^n}dt.$ 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

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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,'MuPADvar',MATLABvar)
  ```

  For example, if `nb` is the handle to the notebook and `z` is the variable, enter:

  ```
  setVar(nb,'z',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 this command in the notebook:

  ```
  anames(All, User)
  ```

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

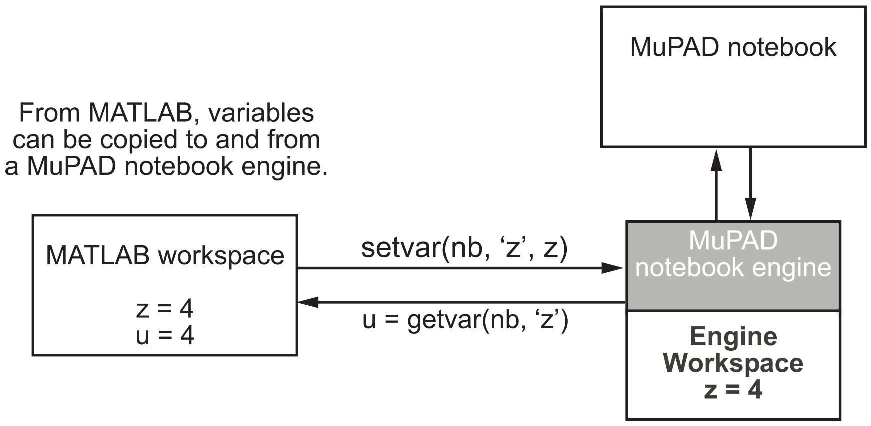
  ```
  MATLABvar = getVar(notebook_handle,'variable');
  ```

  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 “Evaluate MuPAD Notebooks from MATLAB” on page 3-13.

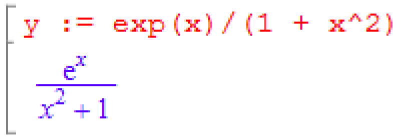
Tip Do all copying and assignments from the MATLAB workspace, not from a MuPAD notebook.



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.

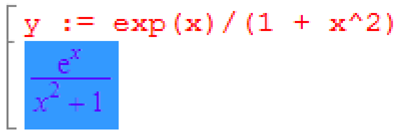
For example, consider this MuPAD expression:



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

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

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



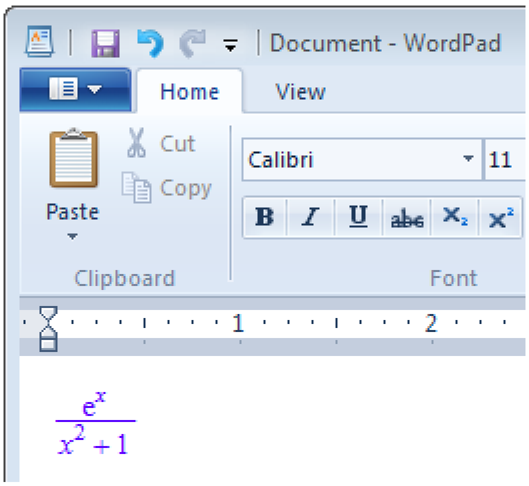
```
y := exp(x)/(1 + x^2)
```

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

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.


Document - WordPad
Home
View
Paste
Cut
Copy
Clipboard
Calibri
11
Font

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

Reserved Variable and Function Names

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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.

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 an error:

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

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

```
Error using solve (line 263)
Specify a variable for which you solve.
```

To fix this issue, 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()
```

Call Built-In MuPAD Functions from MATLAB

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

To access built-in MuPAD functions 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` and `feval` do not open a MuPAD notebook, and therefore, you cannot use these functions to access MuPAD graphics capabilities.

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 character vectors. 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
```

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 character vector. 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 character vectors.) 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
```

Use Your Own MuPAD Procedures

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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 on page 3-44.

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, combine them into a list or other data structure, or use the `print` function.

- 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 3-46. 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 the code defining that procedure, 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 file into a notebook. *Reading* a file means finding it and executing the commands inside it.

Read MuPAD Procedures

If you work in the MuPAD Notebook 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)

If you want to call the procedure from the MATLAB Live Editor, you still need to execute that procedure before calling it. See “Call Your Own MuPAD Procedures” on page 3-64.

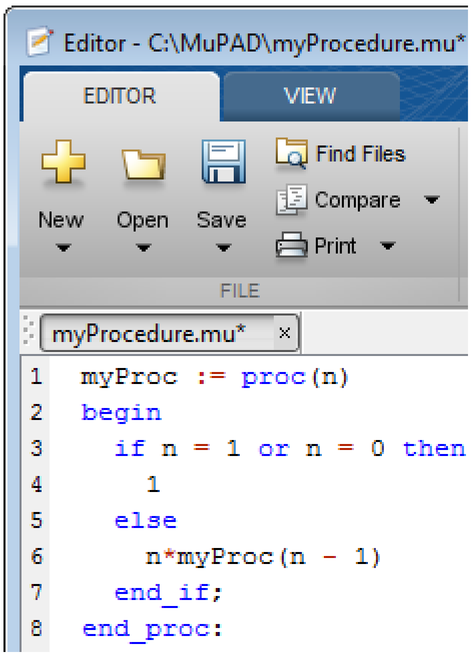
Use Startup Commands and Scripts

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:

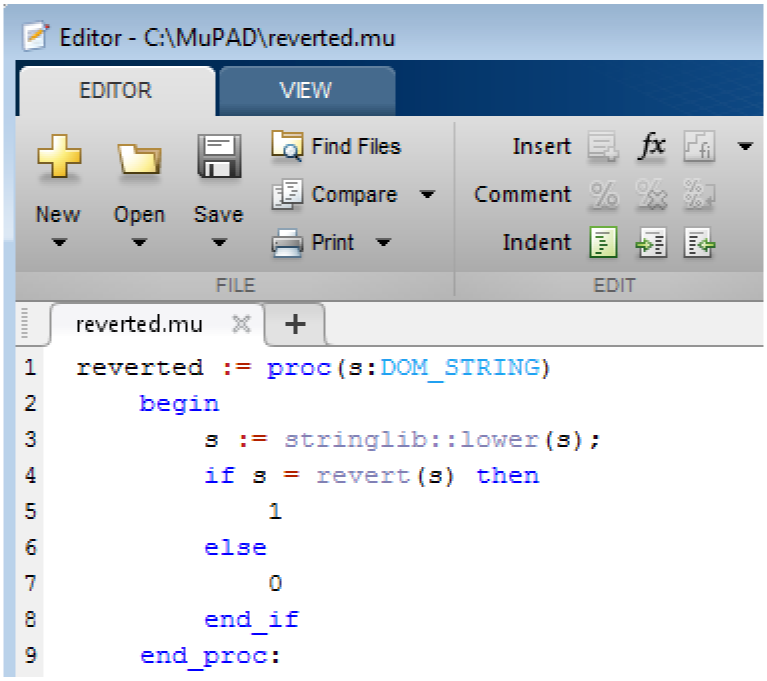
```
read(symengine, 'C:/MuPAD/myProcedure.mu')
```

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
```

If your MuPAD procedure accepts character vector arguments, enclose these arguments in two sets of quotes: double quotes inside single quotes. Single quotes suppress evaluation of the argument before passing it to the MuPAD procedure, and double quotes let MuPAD recognize that the argument is a character vector. For example, this MuPAD procedure converts a character vector to lowercase and checks if reverting that character vector changes it.



```
reverted := proc(s:DOM_STRING)
    begin
        s := stringlib::lower(s);
        if s = revert(s) then
            1
        else
            0
        end_if
    end_proc:
```

In the MATLAB Command Window, use the `read` command to read and execute `reverted.mu`.

```
read(symengine, 'C:/MuPAD/reverted.mu')
```

Now, use `feval` to call the procedure `reverted`. To pass a character vector argument to the procedure, use double quotes inside single quotes.

```
feval(symengine, 'reverted', '"Abccba"')
```

```
ans =
1
```

Clear Assumptions and Reset the Symbolic Engine

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see "Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19.

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 3-69.

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

```
assume(x,'clear')
```

For details, see “Check Assumptions Set On Variables” on page 3-68 and “Effects of Assumptions on Computations” on page 3-69.

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	`assume(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 Live Editor:

```
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 =
in(x, 'real')
```

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 =
[ in(x, 'real'), z ~= 0]
```

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

```
assume([x z],'clear')

assumptions

ans =
Empty sym: 1-by-0
```

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
 -1i
  1i
```

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 =
[ in(x, 'real'), 0 < x]
ans =
1
```

Clearing x with assume(x,'clear') clears the assumption:

```
assume(x,'clear')
assumptions(x)
```

```
ans =
Empty sym: 1-by-0
```

Create MATLAB Functions from MuPAD Expressions

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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 Live Editor, see “Generate MATLAB Functions from Symbolic Expressions” on page 2-254.

When you use the MuPAD Notebook app, 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 Live Editor:

```
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:

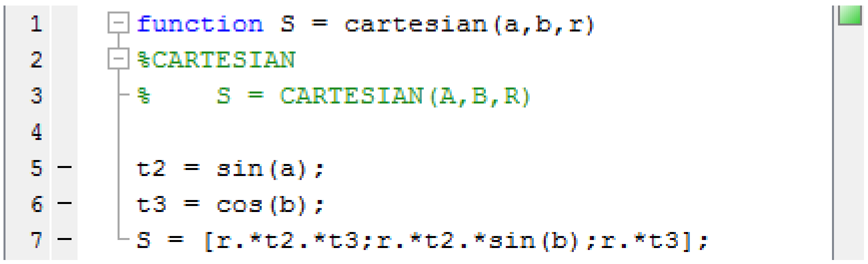
```
h = matlabFunction(S)
```

```
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 character vector representing a MuPAD expression, and then add appropriate syntax to create a valid MATLAB function.

1 In the MuPAD Notebook app, 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:

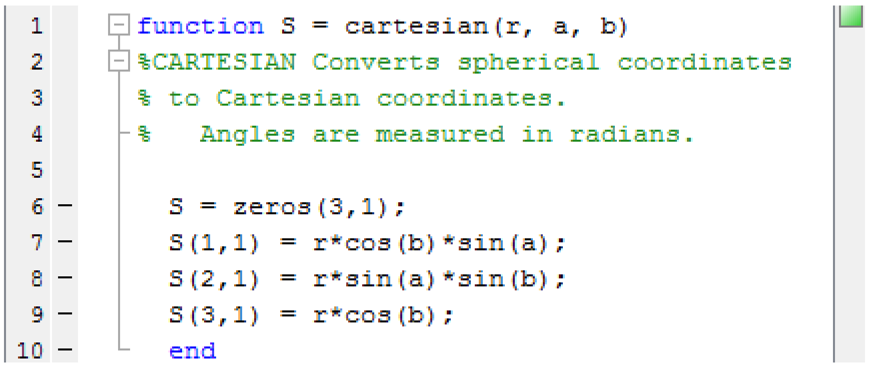
```
fprint(Unquoted, Text, "cartesian.m", generate::MATLAB(S)):
```

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 character vector 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

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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 Live Editor, see “Generate MATLAB Function Blocks from Symbolic Expressions” on page 2-259. Working in the MATLAB Live Editor is recommended.

The MuPAD Notebook 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 3-72.

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 Live Editor:

```
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

Note MuPAD notebooks are not recommended. Use MATLAB live scripts instead.

MATLAB live scripts support most MuPAD functionality, though there are some differences. For more information, see “Convert MuPAD Notebooks to MATLAB Live Scripts” on page 3-19.

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 Live Editor, see “Generate Simscape Equations from Symbolic Expressions” on page 2-261. Working in the MATLAB Live Editor is recommended.

If you work in the MuPAD Notebook app, 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 MuPAD Notebook app:

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 or in the MATLAB Command Window.

GenerateSimscape Equations in the MuPAD Notebook App

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(diff(y(t), 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

Absolute value of real or complex value

Syntax

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

Description

`abs(z)` returns the absolute value (or complex modulus) of `z`. Because symbolic variables are assumed to be complex by default, `abs` returns the complex modulus (magnitude) by default.

`abs(A)` returns the absolute value (or complex modulus) of each element of `A`.

Input Arguments

`z`

Symbolic number, variable, or expression.

`A`

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

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 abs(x)^2 and simplify the result. Because symbolic variables are assumed to be complex by default, the result does not simplify to x^2.

```
syms x
simplify(abs(x)^2)

ans =
abs(x)^2
```

Assume x is real, and repeat the calculation. Now, the result is simplified to x^2.

```
assume(x,'real')
simplify(abs(x)^2)

ans =
x^2
```

Remove assumptions on x for further calculations. For details, see “Use Assumptions on Symbolic Variables” on page 1-28.

```
assume(x,'clear')
```

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
```

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.

Tips

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

See Also

`angle` | `imag` | `real` | `sign` | `signIm`

Introduced before R2006a

acos

Symbolic inverse cosine function

Syntax

```
acos(X)
```

Description

`acos(X)` returns the inverse cosine function (arccosine function) of `X`.

Examples

Inverse Cosine Function for Numeric and Symbolic Arguments

Depending on its arguments, `acos` returns floating-point or exact symbolic results.

Compute the inverse cosine function for these numbers. Because these numbers are not symbolic objects, `acos` returns floating-point results.

```
A = acos([-1, -1/3, -1/2, 1/4, 1/2, sqrt(3)/2, 1])
```

```
A =
    3.1416    1.9106    2.0944    1.3181    1.0472    0.5236         0
```

Compute the inverse cosine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acos` returns unresolved symbolic calls.

```
symA = acos(sym([-1, -1/3, -1/2, 1/4, 1/2, sqrt(3)/2, 1]))
```

```
symA =
[ pi, pi - acos(1/3), (2*pi)/3, acos(1/4), pi/3, pi/6, 0]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

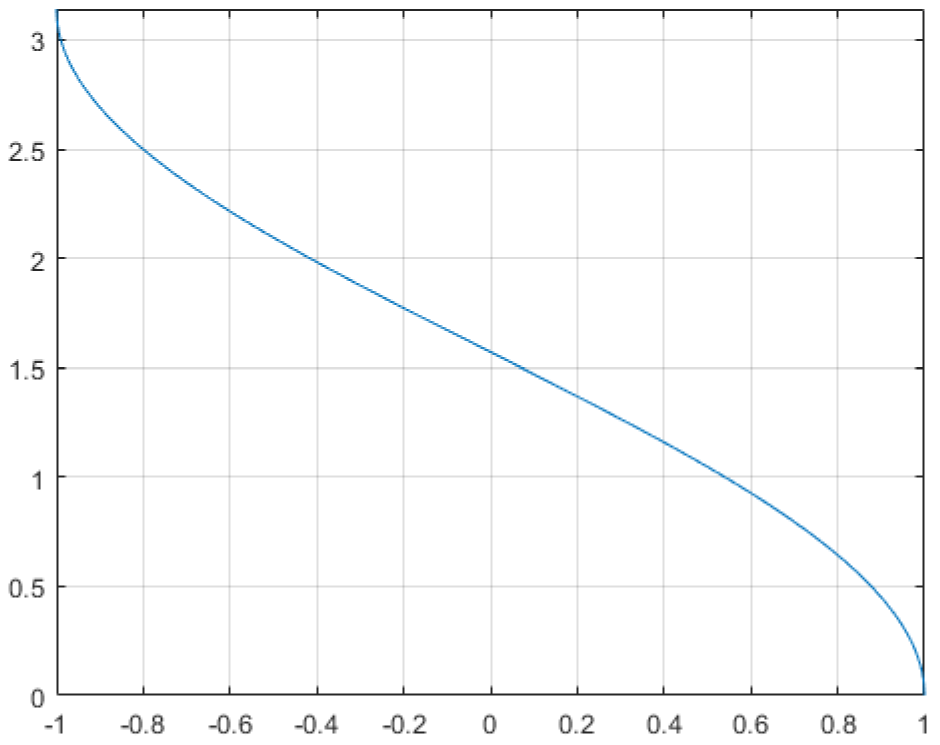
```
vpa(symA)
```

```
ans =
[ 3.1415926535897932384626433832795,...
1.9106332362490185563277142050315,...
2.0943951023931954923084289221863,...
1.318116071652817965745664254646,...
1.0471975511965977461542144610932,...
0.52359877559829887307710723054658,...
0]
```

Plot Inverse Cosine Function

Plot the inverse cosine function on the interval from -1 to 1. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acos(x), [-1, 1])
grid on
```



Handle Expressions Containing Inverse Cosine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acos`.

Find the first and second derivatives of the inverse cosine function:

```
syms x
diff(acos(x), x)
diff(acos(x), x, x)
```

```
ans =
-1/(1 - x^2)^(1/2)
```

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

Find the indefinite integral of the inverse cosine function:

```
int(acos(x), x)
```

```
ans =
x*acos(x) - (1 - x^2)^(1/2)
```

Find the Taylor series expansion of `acos(x)`:

```
taylor(acos(x), x)
```

```
ans =
- (3*x^5)/40 - x^3/6 - x + pi/2
```

Rewrite the inverse cosine function in terms of the natural logarithm:

```
rewrite(acos(x), 'log')
```

```
ans =
-log(x + (1 - x^2)^(1/2)*1i)*1i
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

acosh

Symbolic inverse hyperbolic cosine function

Syntax

```
acosh(X)
```

Description

`acosh(X)` returns the inverse hyperbolic cosine function of `X`.

Examples

Inverse Hyperbolic Cosine Function for Numeric and Symbolic Arguments

Depending on its arguments, `acosh` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic cosine function for these numbers. Because these numbers are not symbolic objects, `acosh` returns floating-point results.

```
A = acosh([-1, 0, 1/6, 1/2, 1, 2])
```

```
A =
   0.0000 + 3.1416i   0.0000 + 1.5708i   0.0000 + 1.4033i...
   0.0000 + 1.0472i   0.0000 + 0.0000i   1.3170 + 0.0000i
```

Compute the inverse hyperbolic cosine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acosh` returns unresolved symbolic calls.

```
symA = acosh(sym([-1, 0, 1/6, 1/2, 1, 2]))
```

```
symA =
[ pi*1i, (pi*1i)/2, acosh(1/6), (pi*1i)/3, 0, acosh(2)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

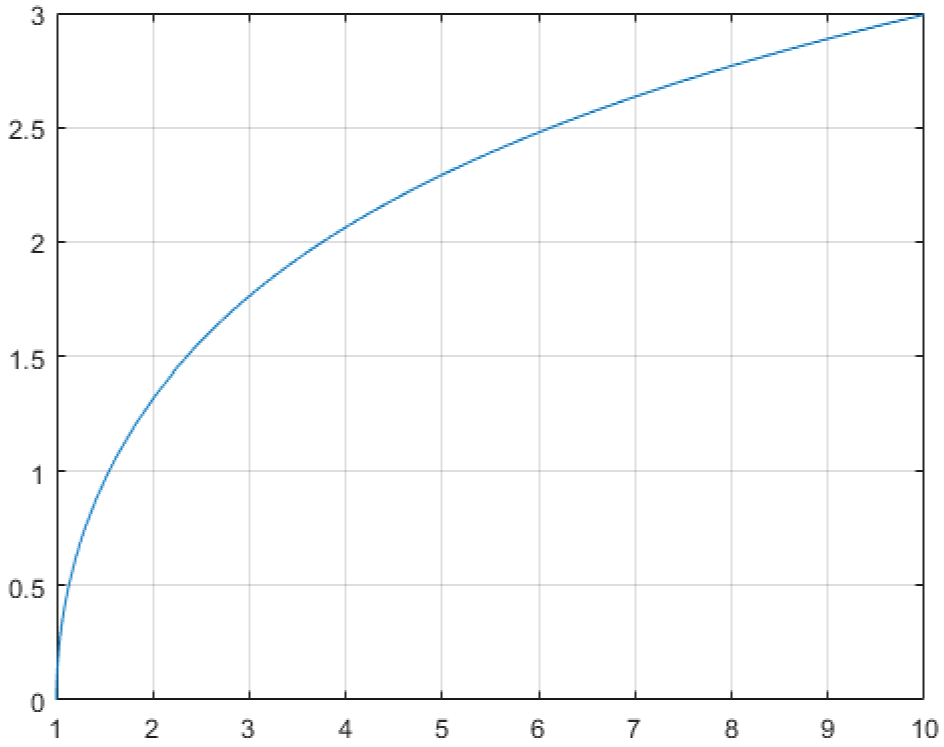
```
vpa(symA)
```

```
ans =

[ 3.1415926535897932384626433832795i,...
  1.5707963267948966192313216916398i,...
  1.4033482475752072886780470855961i,...
  1.0471975511965977461542144610932i,...
  0,...
  1.316957896924816708625046347308]
```

Plot Inverse Hyperbolic Cosine Function

Plot the inverse hyperbolic cosine function on the interval from 1 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acosh(x), [1, 10])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Cosine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acosh`.

Find the first and second derivatives of the inverse hyperbolic cosine function. Simplify the second derivative by using `simplify`.

```
syms x
diff(acosh(x), x)
simplify(diff(acosh(x), x, x))
```

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

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

Find the indefinite integral of the inverse hyperbolic cosine function. Simplify the result by using `simplify`.

```
int(acosh(x), x)

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

Find the Taylor series expansion of `acosh(x)` for `x > 1`:

```
assume(x > 1)
taylor(acosh(x), x)

ans =
(x^5*3i)/40 + (x^3*1i)/6 + x*1i - (pi*1i)/2
```

For further computations, clear the assumption:

```
syms x clear
```

Rewrite the inverse hyperbolic cosine function in terms of the natural logarithm:

```
rewrite(acosh(x), 'log')

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

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

acot

Symbolic inverse cotangent function

Syntax

```
acot(X)
```

Description

`acot(X)` returns the inverse cotangent function (arccotangent function) of `X`.

Examples

Inverse Cotangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `acot` returns floating-point or exact symbolic results.

Compute the inverse cotangent function for these numbers. Because these numbers are not symbolic objects, `acot` returns floating-point results.

```
A = acot([-1, -1/3, -1/sqrt(3), 1/2, 1, sqrt(3)])
```

```
A =
   -0.7854   -1.2490   -1.0472    1.1071    0.7854    0.5236
```

Compute the inverse cotangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acot` returns unresolved symbolic calls.

```
symA = acot(sym([-1, -1/3, -1/sqrt(3), 1/2, 1, sqrt(3)]))
```

```
symA =
[ -pi/4, -acot(1/3), -pi/3, acot(1/2), pi/4, pi/6]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

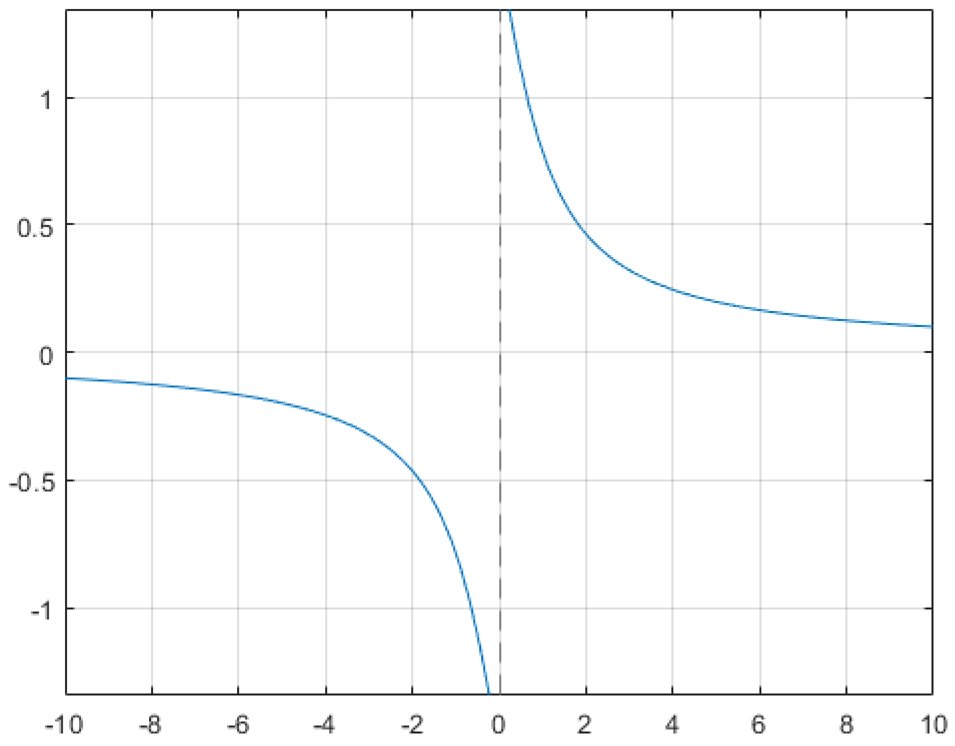
```
vpa(symA)
```

```
ans =
[ -0.78539816339744830961566084581988,...
-1.2490457723982544258299170772811,...
-1.0471975511965977461542144610932,...
1.1071487177940905030170654601785,...
0.78539816339744830961566084581988,...
0.52359877559829887307710723054658]
```

Plot Inverse Cotangent Function

Plot the inverse cotangent function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acot(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Cotangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acot`.

Find the first and second derivatives of the inverse cotangent function:

```
syms x
diff(acot(x), x)
diff(acot(x), x, x)
```

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

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

Find the indefinite integral of the inverse cotangent function:

```
int(acot(x), x)
```

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

Find the Taylor series expansion of `acot(x)` for `x > 0`:

```
assume(x > 0)
taylor(acot(x), x)
```

```
ans =
- x^5/5 + x^3/3 - x + pi/2
```

For further computations, clear the assumption:

```
syms x clear
```

Rewrite the inverse cotangent function in terms of the natural logarithm:

```
rewrite(acot(x), 'log')
```

```
ans =
(log(1 - 1i/x)*1i)/2 - (log(1i/x + 1)*1i)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acos` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

acoth

Symbolic inverse hyperbolic cotangent function

Syntax

```
acoth(X)
```

Description

`acoth(X)` returns the inverse hyperbolic cotangent function of `X`.

Examples

Inverse Hyperbolic Cotangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `acoth` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic cotangent function for these numbers. Because these numbers are not symbolic objects, `acoth` returns floating-point results.

```
A = acoth([-pi/2, -1, 0, 1/2, 1, pi/2])
```

```
A =
 -0.7525 + 0.0000i      -Inf + 0.0000i   0.0000 + 1.5708i...
   0.5493 + 1.5708i       Inf + 0.0000i   0.7525 + 0.0000i
```

Compute the inverse hyperbolic cotangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acoth` returns unresolved symbolic calls.

```
symA = acoth(sym([-pi/2, -1, 0, 1/2, 1, pi/2]))
```

```
symA =
[ -acoth(pi/2), Inf, -(pi*1i)/2, acoth(1/2), Inf, acoth(pi/2)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

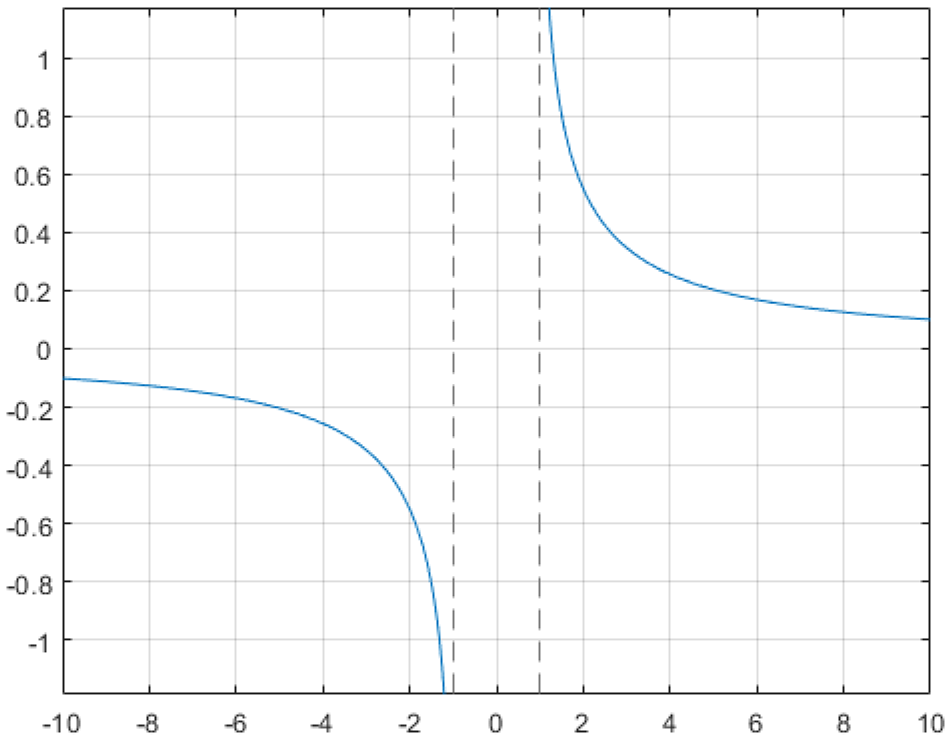
```
vpa(symA)
```

```
ans =
[ -0.75246926714192715916204347800251,...
Inf,...
-1.5707963267948966192313216916398i,...
0.54930614433405484569762261846126...
 - 1.5707963267948966192313216916398i,...
Inf,...
0.75246926714192715916204347800251]
```

Plot Inverse Hyperbolic Cotangent Function

Plot the inverse hyperbolic cotangent function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acoth(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Cotangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acoth`.

Find the first and second derivatives of the inverse hyperbolic cotangent function:

```
syms x
diff(acoth(x), x)
diff(acoth(x), x, x)
```

```
ans =
-1/(x^2 - 1)
```

```
ans =
(2*x)/(x^2 - 1)^2
```

Find the indefinite integral of the inverse hyperbolic cotangent function:

```
int(acoth(x), x)
```

```
ans =
log(x^2 - 1)/2 + x*acoth(x)
```

Find the Taylor series expansion of `acoth(x)` for `x > 0`:

```
assume(x > 0)
taylor(acoth(x), x)
```

```
ans =
x^5/5 + x^3/3 + x - (pi*1i)/2
```

For further computations, clear the assumption:

```
syms x clear
```

Rewrite the inverse hyperbolic cotangent function in terms of the natural logarithm:

```
rewrite(acoth(x), 'log')
```

```
ans =
log(1/x + 1)/2 - log(1 - 1/x)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

acsc

Symbolic inverse cosecant function

Syntax

```
acsc(X)
```

Description

`acsc(X)` returns the inverse cosecant function (arccosecant function) of `X`.

Examples

Inverse Cosecant Function for Numeric and Symbolic Arguments

Depending on its arguments, `acsc` returns floating-point or exact symbolic results.

Compute the inverse cosecant function for these numbers. Because these numbers are not symbolic objects, `acsc` returns floating-point results.

```
A = acsc([-2, 0, 2/sqrt(3), 1/2, 1, 5])
```

```
A =
  -0.5236 + 0.0000i   1.5708 -     Infi   1.0472 + 0.0000i   1.5708...
 - 1.3170i   1.5708 + 0.0000i   0.2014 + 0.0000i
```

Compute the inverse cosecant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acsc` returns unresolved symbolic calls.

```
symA = acsc(sym([-2, 0, 2/sqrt(3), 1/2, 1, 5]))
```

```
symA =
[ -pi/6, Inf, pi/3, asin(2), pi/2, asin(1/5)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

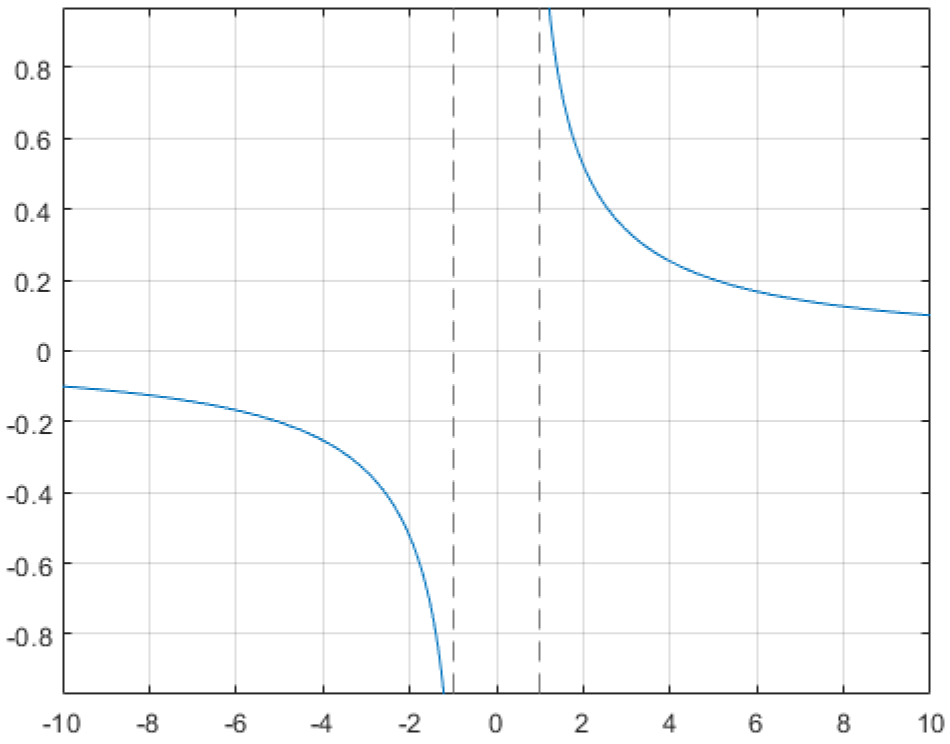
```
vpa(symA)
```

```
ans =
[ -0.52359877559829887307710723054658,...
Inf,...
1.0471975511965977461542144610932,...
1.5707963267948966192313216916398...
 - 1.3169578969248165734029498707969i,...
1.5707963267948966192313216916398,...
0.20135792079033079660099758712022]
```

Plot Inverse Cosecant Function

Plot the inverse cosecant function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acsc(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Cosecant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acsc`.

Find the first and second derivatives of the inverse cosecant function:

```
syms x
diff(acsc(x), x)
diff(acsc(x), x, x)
```

```
ans =
-1/(x^2*(1 - 1/x^2)^(1/2))
```

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

Find the indefinite integral of the inverse cosecant function:

```
int(acsc(x), x)

ans =
x*asin(1/x) + log(x + (x^2 - 1)^(1/2))*sign(x)
```

Find the Taylor series expansion of `acsc(x)` around `x = Inf`:

```
taylor(acsc(x), x, Inf)

ans =
1/x + 1/(6*x^3) + 3/(40*x^5)
```

Rewrite the inverse cosecant function in terms of the natural logarithm:

```
rewrite(acsc(x), 'log')

ans =
-log(1i/x + (1 - 1/x^2)^(1/2))*1i
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acos` | `acot` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

acsch

Symbolic inverse hyperbolic cosecant function

Syntax

```
acsch(X)
```

Description

`acsch(X)` returns the inverse hyperbolic cosecant function of `X`.

Examples

Inverse Hyperbolic Cosecant Function for Numeric and Symbolic Arguments

Depending on its arguments, `acsch` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic cosecant function for these numbers. Because these numbers are not symbolic objects, `acsch` returns floating-point results.

```
A = acsch([-2*i, 0, 2*i/sqrt(3), 1/2, i, 3])
```

```
A =
   0.0000 + 0.5236i      Inf + 0.0000i   0.0000 - 1.0472i...
   1.4436 + 0.0000i   0.0000 - 1.5708i   0.3275 + 0.0000i
```

Compute the inverse hyperbolic cosecant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `acsch` returns unresolved symbolic calls.

```
symA = acsch(sym([-2*i, 0, 2*i/sqrt(3), 1/2, i, 3]))
```

```
symA =
[ (pi*1i)/6, Inf, -(pi*1i)/3, asinh(2), -(pi*1i)/2, asinh(1/3)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

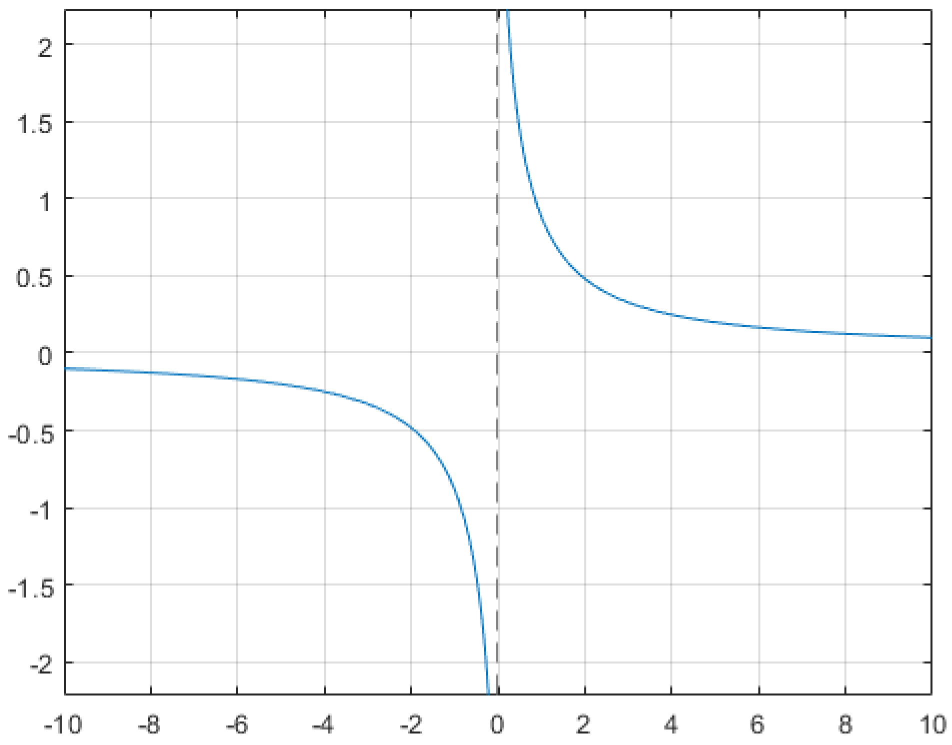
```
vpa(symA)
```

```
ans =
[ 0.52359877559829887307710723054658i,...
Inf,...
-1.0471975511965977461542144610932i,...
1.4436354751788103424932767402731,...
-1.5707963267948966192313216916398i,...
0.32745015023725844332253525998826]
```

Plot Inverse Hyperbolic Cosecant Function

Plot the inverse hyperbolic cosecant function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(acsch(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Cosecant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `acsch`.

Find the first and second derivatives of the inverse hyperbolic cosecant function:

```
syms x
diff(acsch(x), x)
diff(acsch(x), x, x)
```

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

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

Find the indefinite integral of the inverse hyperbolic cosecant function:

```
int(acsch(x), x)

ans =
x*asinh(1/x) + asinh(x)*sign(x)
```

Find the Taylor series expansion of `acsch(x)` around `x = Inf`:

```
taylor(acsch(x), x, Inf)

ans =
1/x - 1/(6*x^3) + 3/(40*x^5)
```

Rewrite the inverse hyperbolic cosecant function in terms of the natural logarithm:

```
rewrite(acsch(x), 'log')

ans =
log((1/x^2 + 1)^(1/2) + 1/x)
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

adjoint

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`.

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 =
  3×3 logical array
   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 =
  3×3 logical array
   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 =
  2×2 logical array
   1   1
   1   1
```

Definitions

Adjoint of 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 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.

See Also

`det` | `inv` | `rank`

Introduced in R2013a

airy

Airy function

Syntax

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

airy(n,x)

airy(___,1)
```

Description

`airy(x)` returns the Airy function on page 4-42 of the first kind, Ai(x), for each element of `x`.

`airy(0,x)` is the same as `airy(x)`.

`airy(1,x)` returns the derivative of Ai(x).

`airy(2,x)` returns the Airy function on page 4-42 of the second kind, Bi(x).

`airy(3,x)` returns the derivative of Bi(x).

`airy(n,x)` uses the values in vector `n` to return the corresponding Airy functions of elements of vector `x`. Both `n` and `x` must have the same size.

`airy(___,1)` returns the “Scaled Airy Functions” on page 4-43 following the syntax for the MATLAB `airy` function.

Examples

Find the Airy Function of the First Kind

Find the Airy function of the first kind, Ai(*x*), for numeric or symbolic inputs using `airy`. Approximate exact symbolic outputs using `vpa`.

Find the Airy function of the first kind, Ai(*x*), at `1.5`. Because the input is double and not symbolic, you get a double result.

```
airy(1.5)

ans =
    0.0717
```

Find the Airy function of the values of vector `v` symbolically, by converting `v` to symbolic form using `sym`. Because the input is symbolic, `airy` returns exact symbolic results. The exact symbolic results for most symbolic inputs are unresolved function calls.

```
v = sym([-1 0 25.1 1+1i]);
vAiry = airy(v)

vAiry =
[ airy(0, -1), 3^(1/3)/(3*gamma(2/3)), airy(0, 251/10), airy(0, 1 + 1i)]
```

Numerically approximate the exact symbolic result using `vpa`.

```
vpa(vAiry)

ans =
[ 0.53556088329235211879951656563887, 0.35502805388781723926006318600418,...
 4.9152763177499054787371976959487e-38,...
 0.060458308371838149196532978116646 - 0.15188956587718140235494791259223i]
```

Find the Airy function, Ai(*x*), of the symbolic input `x^2`. For symbolic expressions, `airy` returns an unresolved call.

```
syms x
airy(x^2)

ans =
airy(0, x^2)
```

Find the Airy Function of the Second Kind

Find the Airy function of the second kind, Bi(*x*), of the symbolic input `[-3 4 1+1i x^2]` by specifying the first argument as `2`. Because the input is symbolic, `airy` returns exact symbolic results. The exact symbolic results for most symbolic inputs are unresolved function calls.

```
v = sym([-3 4 1+1i x^2]);
vAiry = airy(2, v)
```

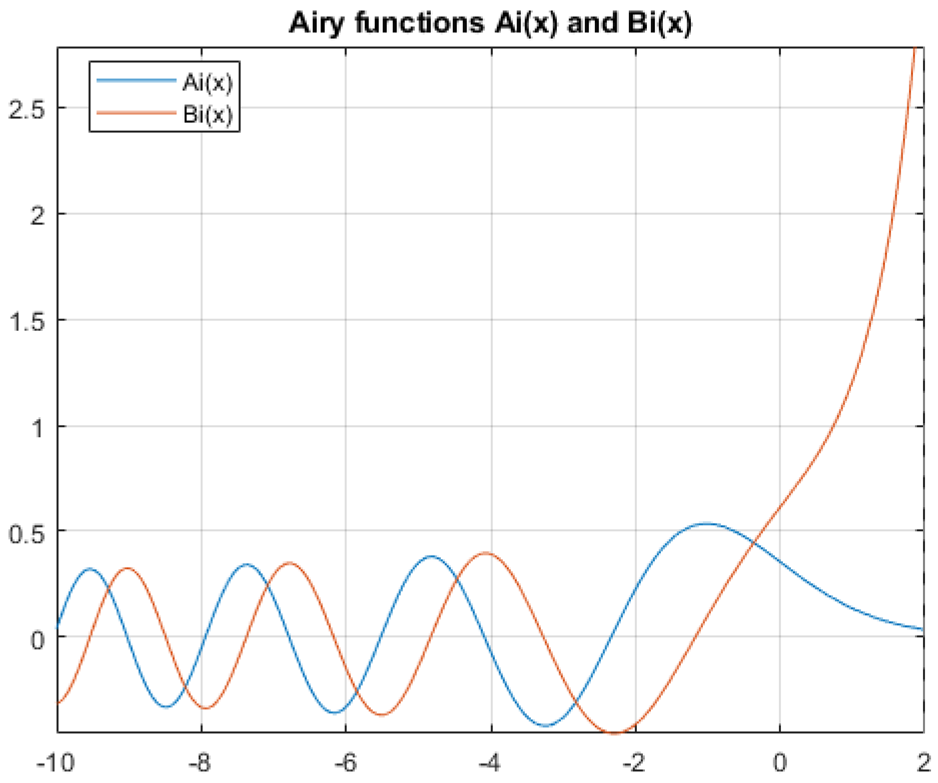
```
vAiry =
[ airy(2, -3), airy(2, 4), airy(2, 1 + 1i), airy(2, x^2)]
```

Use the syntax `airy(2,x)` like `airy(x)`, as described in the example “Find the Airy Function of the First Kind” on page 4-36.

Plot Airy Functions

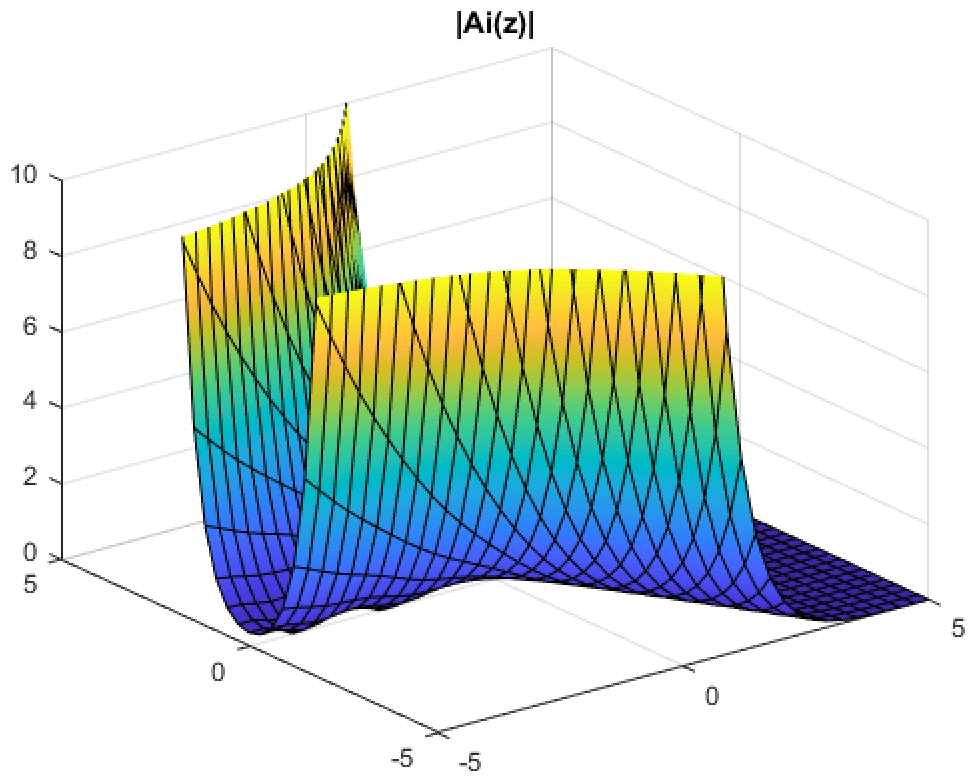
Plot the Airy Functions, $Ai(x)$ and $Bi(x)$, over the interval `[-10 2]` using `fplot`. Before R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(airy(x), [-10 2])
hold on
fplot(airy(2,x), [-10 2])
legend('Ai(x)','Bi(x)','Location','Best')
title('Airy functions Ai(x) and Bi(x)')
grid on
```



Plot the absolute value of $Ai(z)$ over the complex plane.

```
syms y
z = x + 1i*y;
figure(2)
fsurf(abs(airy(z)))
title('|Ai(z)|')
a = gca;
a.ZLim = [0 10];
caxis([0 10])
```



Find Derivatives of Airy Functions

Find the derivative of the Airy function of the first kind, $\mathrm{Ai}'(x)$, at `0` by specifying the first argument of `airy` as `1`. Then, numerically approximate the derivative using `vpa`.

```
dAi = airy(1, sym(0))
dAi_vpa = vpa(dAi)
```

```
dAi =
-(3^(1/6)*gamma(2/3))/(2*pi)
dAi_vpa =
-0.25881940379280679840518356018920
```

Find the derivative of the Airy function of the second kind, $\mathrm{Bi}'(x)$, at `x` by specifying the first argument as `3`. Then, find the derivative at $x = 5$ by substituting for `x` using `subs` and calling `vpa`.

```
syms x
dBi = airy(3, x)
dBi_vpa = vpa(subs(dBi, x, 5))
```

```
dBi =
airy(3, x)
dBi_vpa =
1435.8190802179825186717212380046
```

Solve Airy Differential Equation for Airy Functions

Show that the Airy functions $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$ are the solutions of the differential equation

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

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

```
ans =
C1*airy(0, x) + C2*airy(2, x)
```

Differentiate Airy Functions

Differentiate expressions containing `airy`.

```
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)
```

Expand Airy Function using Taylor Series

Find the Taylor series expansion of the Airy functions, Ai(*x*) and Bi(*x*), using `taylor`.

```
aiTaylor = taylor(airy(x))
biTaylor = taylor(airy(2, x))
```

```
aiTaylor =
- (3^(1/6)*gamma(2/3)*x^4)/(24*pi) + (3^(1/3)*x^3)/(18*gamma(2/3))...
 - (3^(1/6)*gamma(2/3)*x)/(2*pi) + 3^(1/3)/(3*gamma(2/3))
biTaylor =
(3^(2/3)*gamma(2/3)*x^4)/(24*pi) + (3^(5/6)*x^3)/(18*gamma(2/3))...
 + (3^(2/3)*gamma(2/3)*x)/(2*pi) + 3^(5/6)/(3*gamma(2/3))
```

Fourier Transform of Airy Function

Find the Fourier transform of the Airy function Ai(*x*) using `fourier`.

```
syms x
aiFourier = fourier(airy(x))
```

```
aiFourier =
exp((w^3*1i)/3)
```

Numeric Roots of Airy Function

Find a root of the Airy function Ai(*x*) numerically using `vpasolve`.

```
syms x
vpasolve(airy(x) == 0, x)
```

```
ans =
 -226.99630507523600716771890962744
```

Find a root in the interval `[-5 -3]`.

```
vpasolve(airy(x) == 0, x, [-5 -3])
```

```
ans =
-4.0879494441309706166369887014574
```

Input Arguments

`x` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`n` — Type of Airy function

0 (default) | number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array

Type of Airy function, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, or multidimensional array. The values of the input must be `0`, `1`, `2`, or `3`, which specify the Airy function as follows.

n	Returns
`0` (default)	Airy function, Ai(x), which is the same as `airy(x)`.
`1`	Derivative of Airy function, Ai'(x).
`2`	Airy function of the second kind, Bi(x).
`3`	Derivative of Airy function of the second kind, Bi'(x).

Definitions

Airy Functions

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

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

Ai(x) is called the Airy function of the first kind. Bi(x) is called the Airy function of the second kind.

Scaled Airy Functions

The Airy function of the first kind, Ai(*x*), is scaled as

$$e^{\left(\frac{2}{3}x^{(3/2)}\right)}\mathrm{Ai}(x).$$

The derivative, Ai'(*x*), is scaled by the same factor.

The Airy function of the second kind, Bi(*x*), is scaled as

$$e^{-\left|\frac{2}{3}\mathrm{Re}\left(x^{(3/2)}\right)\right|}\mathrm{Bi}(x).$$

The derivative, Bi'(*x*), is scaled by the same factor.

Tips

- When you call `airy` for inputs that are not symbolic objects, you call the MATLAB `airy` function.
- When you call `airy(n, x)`, at least one argument must be a scalar or both arguments must be vectors or matrices of the same size. If one argument is a scalar and the other is a vector or matrix, `airy(n,x)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to the scalar.
- `airy` returns special exact values at `0`.

See Also

`besseli` | `besselj` | `besselk` | `bessely`

Introduced in R2012a

all

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`.

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 =
  logical
   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 =
  1×2 logical array
   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 =
  1×2 logical array
   1   0
```

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

```
all(M, 2)

ans =
  2×1 logical array
   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 =
  logical
   0
```

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

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

ans =
  logical
   1
```

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.

See Also

`and` | `any` | `isAlways` | `not` | `or` | `xor`

Introduced in R2012a

allMuPADNotebooks

All open notebooks

Syntax

```
L = allMuPADNotebooks
```

Description

`L = allMuPADNotebooks` returns a vector with handles (pointers) to all currently open MuPAD notebooks.

If there are no open notebooks, `allMuPADNotebooks` returns an empty object `[ empty mupad ]`.

Examples

Identify All Open Notebooks

Get a vector of handles to all currently open MuPAD notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2
```

```
nb3 =
Notebook1
```

Suppose that there are no other open notebooks. Use `allMuPADNotebooks` to get a vector of handles to these notebooks:

```
allNBs = allMuPADNotebooks
```

```
allNBs =
myFile1
myFile2
Notebook1
```

Create Handle to Existing Notebook

If you already created a MuPAD notebook without a handle or if you lost the handle to a notebook, use `allMuPADNotebooks` to create a new handle. Alternatively, you can save the notebook, close it, and then open it again using a handle.

Create a new notebook:

```
mupad
```

Suppose that you already performed some computations in that notebook, and now want to transfer a few variables to the MATLAB workspace. To be able to do it, you need to create a handle to this notebook:

```
nb = allMuPADNotebooks
```

```
nb =
Notebook1
```

Now, you can use `nb` when transferring data and results between the notebook `Notebook1` and the MATLAB workspace. This approach does not require you to save `Notebook1`.

```
getVar(nb,'x')
```

```
ans =
x
```

- “Create MuPAD Notebooks” on page 3-3
- “Open MuPAD Notebooks” on page 3-6
- “Save MuPAD Notebooks” on page 3-12
- “Evaluate MuPAD Notebooks from MATLAB” on page 3-13
- “Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52
- “Close MuPAD Notebooks from MATLAB” on page 3-17

Output Arguments

`L` — All open MuPAD notebooks
vector of handles to notebooks

All open MuPAD notebooks, returned as a vector of handles to these notebooks.

See Also

`close` | `evaluateMuPADNotebook` | `getVar` | `mupad` | `mupadNotebookTitle` | `openmn` | `setVar`

Topics

“Create MuPAD Notebooks” on page 3-3
“Open MuPAD Notebooks” on page 3-6
“Save MuPAD Notebooks” on page 3-12
“Evaluate MuPAD Notebooks from MATLAB” on page 3-13
“Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52
“Close MuPAD Notebooks from MATLAB” on page 3-17

Introduced in R2013b

and

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`.

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 & 1/2 < 1
x2 =
0 < 10 & 10 < 1
```

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

```
isAlways(x1)
isAlways(x2)
```

```
ans =
  logical
   1
ans =
  logical
   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 `isAlways`:

```
isAlways(s1)
isAlways(s2)
```

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

Evaluate these inequalities to logical `1` or `0` using `isAlways`.

```
isAlways(x1)
isAlways(x2)
```

```
ans =
  logical
   1
ans =
  logical
   0
```

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 `isAlways`.

See Also

`all` | `any` | `isAlways` | `not` | `or` | `piecewise` | `xor`

Introduced in R2012a

angle

Symbolic polar angle

Syntax

```
angle(Z)
```

Description

`angle(Z)` computes the polar angle of the complex value `Z`.

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), pi/4]
```

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]
```

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`.

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` | `sign` | `signIm`

Introduced in R2013a

any

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`.

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 =
  logical
   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 =
  1×2 logical array
   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 =
  1×2 logical array
   1   0
```

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

```
any(M, 2)

ans =
  2×1 logical array
   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 =
  logical
   0
```

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

```
any([-4 + i, x == x + 1])

ans =
  logical
   1
```

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.

See Also

`all` | `and` | `isAlways` | `not` | `or` | `xor`

Introduced in R2012a

argnames

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`

Introduced in R2012a

asec

Symbolic inverse secant function

Syntax

```
asec(X)
```

Description

`asec(X)` returns the inverse secant function (arcsecant function) of `X`.

Examples

Inverse Secant Function for Numeric and Symbolic Arguments

Depending on its arguments, `asec` returns floating-point or exact symbolic results.

Compute the inverse secant function for these numbers. Because these numbers are not symbolic objects, `asec` returns floating-point results.

```
A = asec([-2, 0, 2/sqrt(3), 1/2, 1, 5])
```

```
A =
   2.0944 + 0.0000i   0.0000 +    Infi   0.5236 + 0.0000i...
   0.0000 + 1.3170i   0.0000 + 0.0000i   1.3694 + 0.0000i
```

Compute the inverse secant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `asec` returns unresolved symbolic calls.

```
symA = asec(sym([-2, 0, 2/sqrt(3), 1/2, 1, 5]))
```

```
symA =
[ (2*pi)/3, Inf, pi/6, acos(2), 0, acos(1/5)]
```

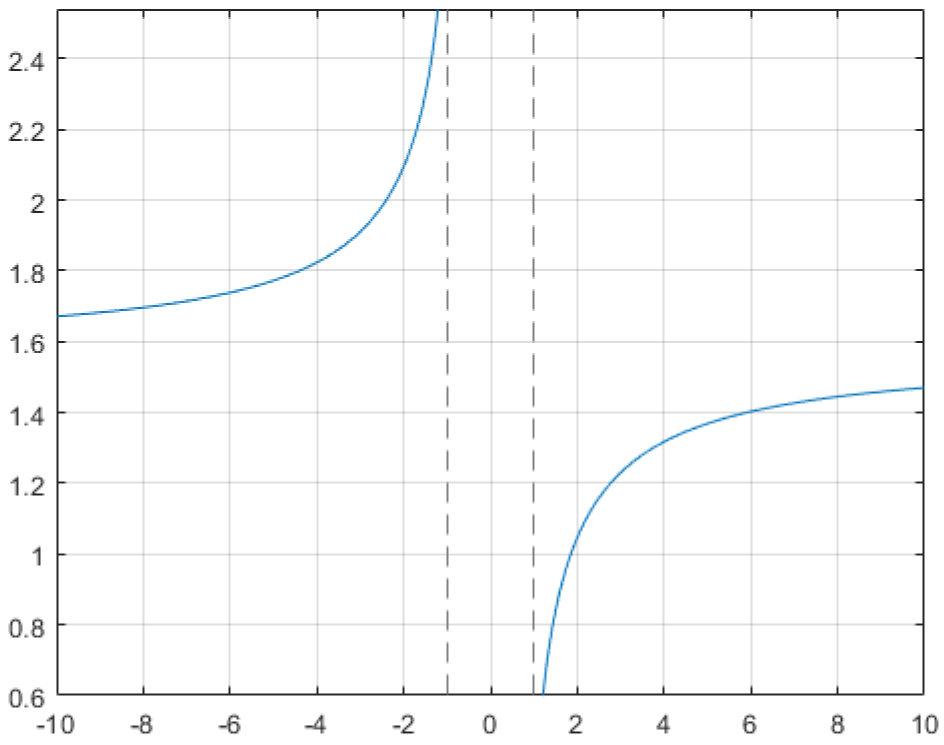
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 2.0943951023931954923084289221863,...
Inf,...
0.52359877559829887307710723054658,...
1.3169578969248165734029498707969i,...
0,...
1.3694384060045659001758622252964]
```

Plot Inverse Secant Function

Plot the inverse secant function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(asec(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Secant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `asec`.

Find the first and second derivatives of the inverse secant function:

```
syms x
diff(asec(x), x)
diff(asec(x), x, x)
```

```
ans =
1/(x^2*(1 - 1/x^2)^(1/2))
```

```
ans =
- 2/(x^3*(1 - 1/x^2)^(1/2)) - 1/(x^5*(1 - 1/x^2)^(3/2))
```

Find the indefinite integral of the inverse secant function:

```
int(asec(x), x)

ans =
x*acos(1/x) - log(x + (x^2 - 1)^(1/2))*sign(x)
```

Find the Taylor series expansion of `asec(x)` around `x = Inf`:

```
taylor(asec(x), x, Inf)

ans =
pi/2 - 1/x - 1/(6*x^3) - 3/(40*x^5)
```

Rewrite the inverse secant function in terms of the natural logarithm:

```
rewrite(asec(x), 'log')

ans =
-log(1/x + (1 - 1/x^2)^(1/2)*1i)*1i
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acos` | `acot` | `acsc` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

asech

Symbolic inverse hyperbolic secant function

Syntax

```
asech(X)
```

Description

`asech(X)` returns the inverse hyperbolic secant function of `X`.

Examples

Inverse Hyperbolic Secant Function for Numeric and Symbolic Arguments

Depending on its arguments, `asech` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic secant function for these numbers. Because these numbers are not symbolic objects, `asech` returns floating-point results.

```
A = asech([-2, 0, 2/sqrt(3), 1/2, 1, 3])
```

```
A =
   0.0000 + 2.0944i      Inf + 0.0000i   0.0000 + 0.5236i...
   1.3170 + 0.0000i   0.0000 + 0.0000i   0.0000 + 1.2310i
```

Compute the inverse hyperbolic secant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `asech` returns unresolved symbolic calls.

```
symA = asech(sym([-2, 0, 2/sqrt(3), 1/2, 1, 3]))
```

```
symA =
[ (pi*2i)/3, Inf, (pi*1i)/6, acosh(2), 0, acosh(1/3)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

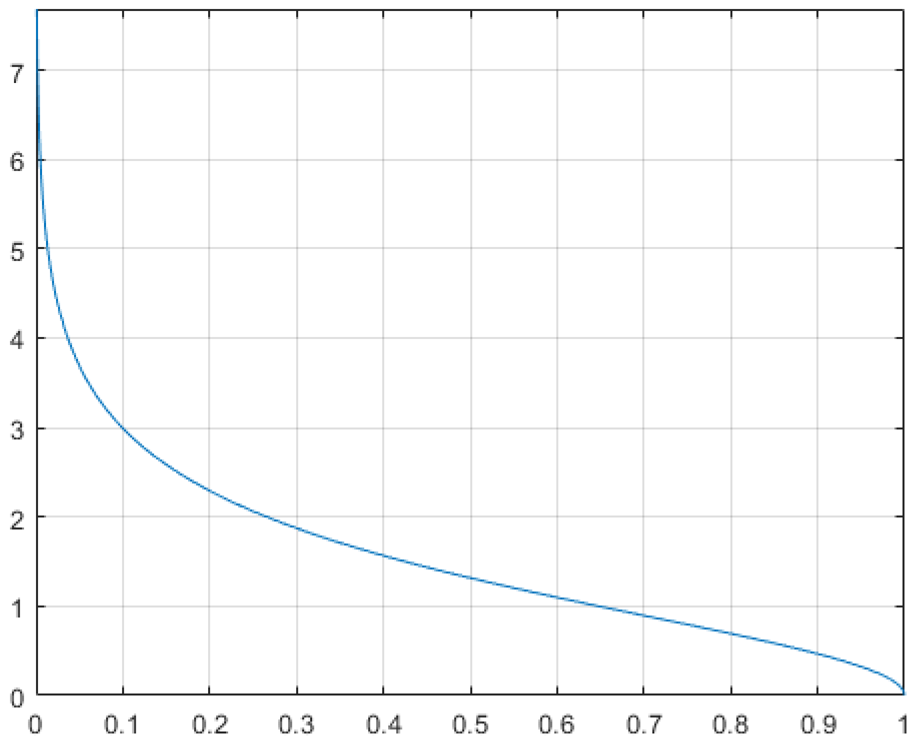
```
vpa(symA)
```

```
ans =
[ 2.0943951023931954923084289221863i,...
 Inf,...
 0.52359877559829887307710723054658i,...
 1.3169578969248167086250463473080,...
 0,...
 1.2309594173407746821349291782483i]
```

Plot Inverse Hyperbolic Secant Function

Plot the inverse hyperbolic secant function on the interval from 0 to 1. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(asech(x), [0, 1])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Secant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `asech`.

Find the first and second derivatives of the inverse hyperbolic secant function. Simplify the second derivative by using `simplify`.

```
syms x
diff(asech(x), x)
simplify(diff(asech(x), x, x))
```

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

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

Find the indefinite integral of the inverse hyperbolic secant function:

```
int(asech(x), x)
```

```
ans =
atan(1/((1/x - 1)^(1/2)*(1/x + 1)^(1/2))) + x*acosh(1/x)
```

Find the Taylor series expansion of `asech(x)` around `x = Inf`:

```
taylor(asech(x), x, Inf)
```

```
ans =
(pi*1i)/2 - 1i/x - 1i/(6*x^3) - 3i/(40*x^5)
```

Rewrite the inverse hyperbolic secant function in terms of the natural logarithm:

```
rewrite(asech(x), 'log')
```

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

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

asin

Symbolic inverse sine function

Syntax

`asin(X)`

Description

`asin(X)` returns the inverse sine function (arcsine function) of `X`.

Examples

Inverse Sine Function for Numeric and Symbolic Arguments

Depending on its arguments, `asin` returns floating-point or exact symbolic results.

Compute the inverse sine function for these numbers. Because these numbers are not symbolic objects, `asin` returns floating-point results.

```
A = asin([-1, -1/3, -1/2, 1/4, 1/2, sqrt(3)/2, 1])
```

```
A =
   -1.5708   -0.3398   -0.5236    0.2527    0.5236    1.0472    1.5708
```

Compute the inverse sine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `asin` returns unresolved symbolic calls.

```
symA = asin(sym([-1, -1/3, -1/2, 1/4, 1/2, sqrt(3)/2, 1]))
```

```
symA =
[ -pi/2, -asin(1/3), -pi/6, asin(1/4), pi/6, pi/3, pi/2]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

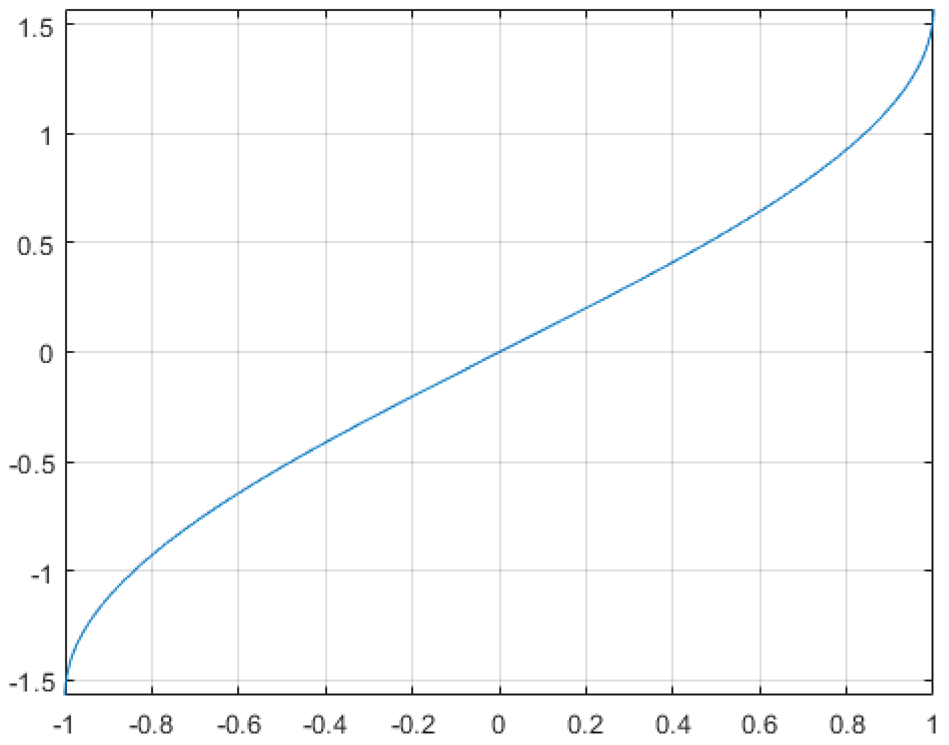
```
vpa(symA)
```

```
ans =
[ -1.5707963267948966192313216916398,...
-0.33983690945412193709639251339176,...
-0.52359877559829887307710723054658,...
0.25268025514207865348565743699371,...
0.52359877559829887307710723054658,...
1.0471975511965977461542144610932,...
1.5707963267948966192313216916398]
```

Plot Inverse Sine Function

Plot the inverse sine function on the interval from -1 to 1. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(asin(x), [-1, 1])
grid on
```



Handle Expressions Containing Inverse Sine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `asin`.

Find the first and second derivatives of the inverse sine function:

```
syms x
diff(asin(x), x)
diff(asin(x), x, x)
```

```
ans =
1/(1 - x^2)^(1/2)
```

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

Find the indefinite integral of the inverse sine function:

```
int(asin(x), x)

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

Find the Taylor series expansion of `asin(x)`:

```
taylor(asin(x), x)

ans =
(3*x^5)/40 + x^3/6 + x
```

Rewrite the inverse sine function in terms of the natural logarithm:

```
rewrite(asin(x), 'log')

ans =
-log((1 - x^2)^(1/2) + x*1i)*1i
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acos` | `acot` | `acsc` | `asec` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

asinh

Symbolic inverse hyperbolic sine function

Syntax

```
asinh(X)
```

Description

`asinh(X)` returns the inverse hyperbolic sine function of `X`.

Examples

Inverse Hyperbolic Sine Function for Numeric and Symbolic Arguments

Depending on its arguments, `asinh` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic sine function for these numbers. Because these numbers are not symbolic objects, `asinh` returns floating-point results.

```
A = asinh([-i, 0, 1/6, i/2, i, 2])
```

```
A =
   0.0000 - 1.5708i   0.0000 + 0.0000i   0.1659 + 0.0000i...
   0.0000 + 0.5236i   0.0000 + 1.5708i   1.4436 + 0.0000i
```

Compute the inverse hyperbolic sine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `asinh` returns unresolved symbolic calls.

```
symA = asinh(sym([-i, 0, 1/6, i/2, i, 2]))
```

```
symA =
[ -(pi*1i)/2, 0, asinh(1/6), (pi*1i)/6, (pi*1i)/2, asinh(2)]
```

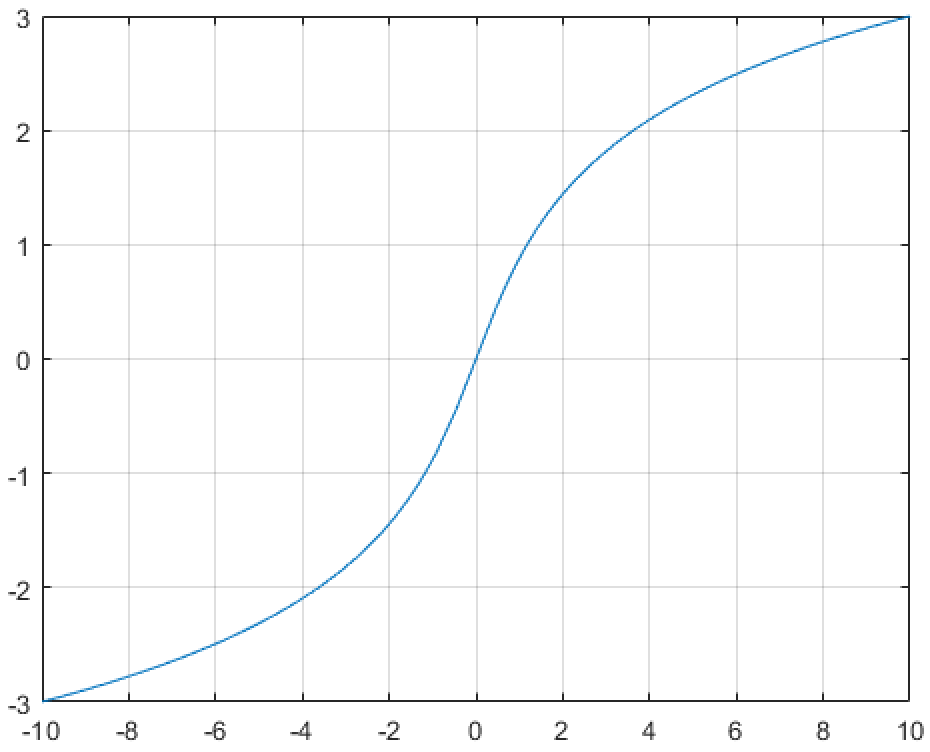
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.5707963267948966192313216916398i,...
0,...
0.1659045502693011764350217163155 3,...
0.52359877559829887307710723054658i,...
1.5707963267948966192313216916398i,...
1.4436354751788103012444253181457]
```

Plot Inverse Hyperbolic Sine Function

Plot the inverse hyperbolic sine function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(asinh(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Sine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `asinh`.

Find the first and second derivatives of the inverse hyperbolic sine function:

```
syms x
diff(asinh(x), x)
diff(asinh(x), x, x)
```

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

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

Find the indefinite integral of the inverse hyperbolic sine function:

```
int(asinh(x), x)

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

Find the Taylor series expansion of `asinh(x)`:

```
taylor(asinh(x), x)

ans =
(3*x^5)/40 - x^3/6 + x
```

Rewrite the inverse hyperbolic sine function in terms of the natural logarithm:

```
rewrite(asinh(x), 'log')

ans =
log(x + (x^2 + 1)^(1/2))
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

assume

Set assumption on symbolic object

Syntax

```
assume(condition)
assume(expr,set)
assume(expr,'clear')
```

Description

`assume(condition)` states that `condition` is valid. `assume` is not additive. Instead, it automatically deletes all previous assumptions on the variables in `condition`.

`assume(expr,set)` states that `expr` belongs to `set`. `assume` deletes previous assumptions on variables in `expr`.

`assume(expr,'clear')` clears all assumptions on all variables in `expr`.

Examples

Common Assumptions

Set an assumption using the associated syntax.

Assume 'x' is	Syntax
real	`assume(x,'real')`
rational	`assume(x,'rational')`
positive	`assume(x > 0)`
less than -1 or greater than 1	`assume(x<-1 | x>1)`

Assume 'x' is	Syntax
an integer from 2 through 10	`assume(in(x,'integer') & x>2 & x<10)`
not an integer	`assume(~in(z,'integer'))`
not equal to 0	`assume(x ~= 0)`
even	`assume(x/2,'integer')`
odd	`assume((x-1)/2,'integer')`
from 0 through 2π	`assume(x>0 & x<2*pi)`
a multiple of π	`assume(x/pi,'integer')`

Assume Variable Is Even or Odd

Assume `x` is even by assuming that `x/2` is an integer. Assume `x` is odd by assuming that `(x-1)/2` is an integer.

Assume `x` is even.

```
syms x
assume(x/2,'integer')
```

Find all even numbers between `0` and `10` using `solve`.

```
solve(x>0,x<10,x)
```

```
ans =
 2
 4
 6
 8
```

Assume `x` is odd. `assume` is not additive, but instead automatically deletes the previous assumption `in(x/2, 'integer')`.

```
assume((x-1)/2,'integer')
solve(x>0,x<10,x)
```

```
ans =
 1
 3
 5
 7
 9
```

Clear the assumptions on `x` for further computations.

```
assume(x,'clear')
```

Multiple Assumptions

Successive `assume` commands do not set multiple assumptions. Instead, each `assume` command deletes previous assumptions and sets new assumptions. Set multiple assumptions by using `assumeAlso` or the `&` operator.

Assume `x > 5` and then `x < 10` by using `assume`. Use `assumptions` to check that only the second assumption exists because `assume` deleted the first assumption when setting the second.

```
syms x
assume(x > 5)
assume(x < 10)
assumptions
```

```
ans =
x < 10
```

Assume the first assumption in addition to the second by using `assumeAlso`. Check that both assumptions exist.

```
assumeAlso(x > 5)
assumptions
```

```
ans =
[ 5 < x, x < 10]
```

Clear the assumptions on `x`.

```
assume(x,'clear')
```

Assume both conditions using the `&` operator. Check that both assumptions exist.

```
assume(x>5 & x<10)
assumptions
```

```
ans =
[ 5 < x, x < 10]
```

Clear the assumptions on `x` for future calculations.

```
assume(x,'clear')
```

Assumptions on Integrand

Compute an indefinite integral with and without the assumption on the symbolic parameter `a`.

Use `assume` to set an assumption that `a` does not equal `-1`.

```
syms x a
assume(a ~= -1)
```

Compute this integral.

```
int(x^a,x)
```

```
ans =
x^(a + 1)/(a + 1)
```

Now, clear the assumption and compute the same integral. Without assumptions, `int` returns this piecewise result.

```
assume(a,'clear')
int(x^a, x)
```

```
ans =
piecewise(a == -1, log(x), a ~= -1, x^(a + 1)/(a + 1))
```

Assumptions on Parameters and Variables of Equation

Use assumptions to restrict the returned solutions of an equation to a particular interval.

Solve this equation.

```
syms x
eqn = x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6 + (365*x)/2 + 250/3;
solve(eqn, x)
```

```
ans =
   -5
   -1
 -1/3
  1/2
  100
```

Use `assume` to restrict the solutions to the interval –1 <= *x* <= 1.

```
assume(-1 <= x <= 1)
solve(eqn, x)
```

```
ans =
   -1
 -1/3
  1/2
```

Set several assumptions simultaneously by using 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(eqn, x)
```

```
ans =
  -5
 100
```

For further computations, clear the assumptions.

```
assume(x,'clear')
```

Use Assumptions for Simplification

Setting appropriate assumptions can result in simpler expressions.

Try to simplify the expression `sin(2*pi*n)` using `simplify`. The `simplify` function cannot simplify the input and returns the input as it is.

```
syms n
simplify(sin(2*n*pi))
```

```
ans =
sin(2*pi*n)
```

Assume `n` is an integer. `simplify` now simplifies the expression.

```
assume(n,'integer')
simplify(sin(2*n*pi))

ans =
0
```

For further computations, clear the assumption.

```
assume(n,'clear')
```

Assumptions on Expressions

Set assumption on the symbolic expression.

You can set assumptions not only on variables, but also on expressions. For example, compute this integral.

```
syms x
f = 1/abs(x^2 - 1);
int(f,x)

ans =
-atanh(x)/sign(x^2 - 1)
```

Set the assumption $x^2 - 1 > 0$ to produce a simpler result.

```
assume(x^2 - 1 > 0)
int(f,x)

ans =
-atanh(x)
```

For further computations, clear the assumption.

```
assume(x,'clear')
```

Assumptions to Prove Relations

Prove relations that hold under certain conditions by first assuming the conditions and then using `isAlways`.

Prove that `sin(pi*x)` is never equal to `0` when `x` is not an integer. The `isAlways` function returns logical `1` (`true`), which means the condition holds for all values of `x` under the set assumptions.

```
syms x
assume(~in(x,'integer'))
isAlways(sin(pi*x) ~= 0)

ans =
  logical
   1
```

Assumptions on Matrix Elements

Set assumptions on all elements of a matrix using `sym`.

Create the 3-by-3 symbolic matrix `A` with auto-generated elements. Specify the `set` as `rational`.

```
A = sym('A',[3 3],'rational')

A =
[ A1_1, A1_2, A1_3]
[ A2_1, A2_2, A2_3]
[ A3_1, A3_2, A3_3]
```

Return the assumptions on the elements of `A` using `assumptions`.

```
assumptions(A)

ans =
[ in(A1_1, 'rational'), in(A1_2, 'rational'), in(A1_3, 'rational'),...
  in(A2_1, 'rational'), in(A2_2, 'rational'), in(A2_3, 'rational'),...
  in(A3_1, 'rational'), in(A3_2, 'rational'), in(A3_3, 'rational')]
```

You can also use `assume` to set assumptions on all elements of a matrix. Assume all elements of `A` are `positive` using `assume`.

```
assume(A,'positive')
```

For further computations, clear the assumptions.

```
assume(A,'clear')
```

Input Arguments

`condition` **— Assumption statement**

symbolic expression | symbolic equation | symbolic relation | vector or matrix of symbolic expressions, equations, or relations

Assumption statement, specified as a 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` **— Expression to set assumption on**

symbolic variable | symbolic expression | vector or matrix of symbolic variables or expressions

Expression to set assumption on, specified as a symbolic variable, expression, vector, or matrix. If `expr` is a vector or matrix, then `assume(expr,set)` sets an assumption that each element of `expr` belongs to `set`.

`set` **— Set of integer, rational, real, or positive numbers**

`'integer'` | `'rational'` | `'real'` | `'positive'`

Set of integer, rational, real, or positive numbers, specified as `'integer'`, `'rational'`, `'real'`, or `'positive'`.

Tips

- `assume` removes any assumptions previously set on the symbolic variables. To retain previous assumptions while adding an assumption, 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 `var`, use this command.

  ```
  assume(var,'clear')
  ```

- To delete all objects in the MATLAB workspace and close the Symbolic Math Toolbox engine associated with the MATLAB workspace clearing all assumptions, use this command:

  ```
  clear all
  ```

- MATLAB projects complex numbers in inequalities to the real axis. If `condition` 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`.) 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.
- When you create a new symbolic variable using `sym` and `syms`, you also can set an assumption that the variable is real, positive, integer, or rational.

  ```
  a = sym('a','real');
  b = sym('b','integer');
  c = sym('c','positive');
  d = sym('d','positive');
  e = sym('e','rational');
  ```

  or more efficiently

  ```
  syms a real
  syms b integer
  syms c d positive
  syms e rational
  ```

See Also

`and` | `assumeAlso` | `assumptions` | `clear all` | `in` | `isAlways` | `not` | `or` | `piecewise` | `sym` | `syms`

Topics

“Set Assumptions” on page 1-28
“Check Existing Assumptions” on page 1-29
“Delete Symbolic Objects and Their Assumptions” on page 1-29
“Default Assumption” on page 1-28

Introduced in R2012a

assumeAlso

Add assumption on symbolic object

Syntax

```
assumeAlso(condition)
assumeAlso(expr,set)
```

Description

`assumeAlso(condition)` states that `condition` is valid for all symbolic variables in `condition`. 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.

Examples

Assumptions Specified as Relations

Set assumptions using `assume`. Then add more assumptions using `assumeAlso`.

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)
```

```
Warning: Solutions are valid under the following
conditions: x <= 1;
x == 1.
 To include parameters and conditions in the
 solution, specify the 'ReturnConditions' value as
'true'.
> In solve>warnIfParams (line 508)
```

```
  In solve (line 357)
s =
  (1 - x)^(1/2)*(x + 1)^(1/2)
 -(1 - x)^(1/2)*(x + 1)^(1/2)
```

The solver warns that both solutions hold only under certain conditions.

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.

```
assume([x y],'clear')
```

Assumptions Specified as Sets

Set assumptions using `syms`. Then add more assumptions using `assumeAlso`.

When declaring the symbolic variable `n`, set an assumption that `n` is positive.

```
syms n positive
```

Using `assumeAlso`, add more assumptions on the same variable `n`. For example, assume also that n is and integer.

```
assumeAlso(n,'integer')
```

Return all assumptions affecting variable `n` using `assumptions`. In this case, `n` is a positive integer.

```
assumptions(n)
```

```
ans =
[ 0 < n, in(n, 'integer')]
```

For further computations, clear the assumptions.

```
assume(n,'clear')
```

Assumptions on Matrix Elements

Use the assumption on a matrix as a shortcut for setting the same assumption on each matrix element.

Create the 3-by-3 symbolic matrix `A` with auto-generated elements. To assume every element of `A` is rational, specify `set` as `'rational'`.

```
A = sym('A',[3 3],'rational')
```

```
A =
[ A1_1, A1_2, A1_3]
[ A2_1, A2_2, A2_3]
[ A3_1, A3_2, A3_3]
```

Now, add the assumption that each element of `A` is greater than 1.

```
assumeAlso(A > 1)
```

Return assumptions affecting elements of `A` using `assumptions`:

```
assumptions(A)
```

```
ans =
[ 1 < A1_1, 1 < A1_2, 1 < A1_3, 1 < A2_1, 1 < A2_2, 1 < A2_3,...
 1 < A3_1, 1 < A3_2, 1 < A3_3,...
 in(A1_1, 'rational'), in(A1_2, 'rational'), in(A1_3, 'rational'),...
 in(A2_1, 'rational'), in(A2_2, 'rational'), in(A2_3, 'rational'),...
 in(A3_1, 'rational'), in(A3_2, 'rational'), in(A3_3, 'rational')]
```

For further computations, clear the assumptions.

```
assume(A,'clear')
```

Contradicting Assumptions

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 an error.

Try to set contradicting assumptions. `assumeAlso` returns an error.

```
syms y
assume(y,'real')
assumeAlso(y == i)
```

```
Error using mupadengine/feval (line 163)
Assumptions are inconsistent.
Error in sym/assumeAlso (line 551)
                feval(symengine, 'assumeAlso', cond);
```

`assumeAlso` does not guarantee to detect contradicting assumptions. For example, 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')
```

Return all assumptions affecting variable `y` using `assumptions`:

```
assumptions(y)
```

```
ans =
[ in(y, 'real'), in(y*1i, 'real'), y ~= 0]
```

For further computations, clear the assumptions.

```
assume(y,'clear')
```

Input Arguments

`condition` — Assumption statement

symbolic expression | symbolic equation | relation | vector or matrix of symbolic expressions, equations, or relations

Assumption statement, specified as a 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` — Expression to set assumption on

symbolic variable | symbolic expression | vector or matrix of symbolic variables or expressions

Expression to set assumption on, specified as a symbolic variable, expression, or a vector or matrix of symbolic variables or expressions. If `expr` is a vector or matrix, then `assumeAlso(expr,set)` sets an assumption that each element of `expr` belongs to `set`.

`set` — Set of integer, rational, real, or positive numbers
`'integer'` | `'rational'` | `'real'` | `'positive'`

Set of integer, rational, real, or positive numbers, specified as `'integer'`, `'rational'`, `'real'`, or `'positive'`.

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 `var` use this command.

  ```
  assume(var,'clear')
  ```
- To clear all objects in the MATLAB workspace and close the Symbolic Math Toolbox engine associated with the MATLAB workspace resetting all its assumptions, use this command.

  ```
  clear all
  ```
- MATLAB projects complex numbers in inequalities to the real axis. If `condition` 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`.) 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.

- 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` | `in` | `isAlways` | `not` | `or` | `piecewise` | `sym` | `syms`

Topics

"Set Assumptions" on page 1-28
"Check Existing Assumptions" on page 1-29
"Delete Symbolic Objects and Their Assumptions" on page 1-29
"Default Assumption" on page 1-28

Introduced in R2012a

assumptions

Show assumptions affecting symbolic variable, expression, or function

Syntax

```
assumptions(var)
assumptions
```

Description

`assumptions(var)` returns all assumptions that affect variable `var`. If `var` is an expression or function, `assumptions` returns all assumptions that affect all variables in `var`.

`assumptions` returns all assumptions that affect all variables in MATLAB Workspace.

Examples

Assumptions on Variables

Assume that the variable `n` is an integer using `assume`. Return the assumption using `assumptions`.

```
syms n
assume(n,'integer')
assumptions
```

```
ans =
in(n, 'integer')
```

The syntax `in(n, 'integer')` indicates `n` is an integer.

Assume that `n` is less than `x` and that `x < 42` using `assume`. The `assume` function replaces old assumptions on input with the new assumptions. Return all assumptions that affect `n`.

```
syms x
assume(n<x & x<42)
assumptions(n)

ans =
[ n < x, x < 42]
```

`assumptions` returns the assumption `x < 42` because it affects `n` through the assumption `n < x`. Thus, `assumptions` returns the transitive closure of assumptions, which is all assumptions that mathematically affect the input.

Set the assumption on variable `m` that `1 < m < 3`. Return all assumptions on `m` and `x` using `assumptions`.

```
syms m
assume(1<m<3)
assumptions([m x])

ans =
[ n < x, 1 < m, m < 3, x < 42]
```

To see the assumptions that affect all variables, use `assumptions` without any arguments.

```
assumptions

ans =
[ n < x, 1 < m, m < 3, x < 42]
```

For further computations, clear the assumptions.

```
assume([m n x],'clear')
```

Multiple Assumptions on One Variable

You cannot set an additional assumption on a variable using `assume` because `assume` clears all previous assumptions on that variable. To set an additional assumption on a variable, using `assumeAlso`.

Set an assumption on `x` using `assume`. Set an additional assumption on `x` use `assumeAlso`. Use `assumptions` to return the multiple assumptions on `x`.

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

```
assumeAlso(x<0)
assumptions(x)
```

```
ans =
[ in(x, 'real'), x < 0]
```

The syntax `in(x, 'real')` indicates `x` is `real`.

For further computations, clear the assumptions.

```
assume(x,'clear')
```

Assumptions Affecting Expressions and Functions

`assumptions` accepts symbolic expressions and functions as input and returns all assumptions that affect all variables in the symbolic expressions or functions.

Set assumptions on variables in a symbolic expression. Find all assumptions that affect all variables in the symbolic expression using `assumptions`.

```
syms a b c
expr = a*exp(b)*sin(c);
assume(a+b > 3 & in(a,'integer') & in(c,'real'))
assumptions(expr)
```

```
ans =
[ 3 < a + b, in(a, 'integer'), in(c, 'real')
```

Find all assumptions that affect all variables that are inputs to a symbolic function.

```
syms f(a,b,c)
assumptions(f)
```

```
ans =
[ 3 < a + b, in(a, 'integer'), in(c, 'real')]
```

Clear the assumptions for further computations.

```
assume([a b c],'clear')
```

Restore Old Assumptions

To restore old assumptions, first store the assumptions returned by `assumptions`. Then you can restore these assumptions at any point by calling `assume` or `assumeAlso`.

Solve the equation for a spring using `dsolve` under the assumptions that the mass and spring constant are `positive`.

```
syms m k positive
syms x(t)
dsolve(m*diff(x,t,t) == -k*x, x(0)==0)
```

```
ans =
C8*sin((k^(1/2)*t)/m^(1/2))
```

Suppose you want to explore solutions unconstrained by assumptions, but want to restore the assumptions afterwards. First store the assumptions using `assumptions`, then clear the assumptions and solve the equation. `dsolve` returns unconstrained solutions.

```
tmp = assumptions;
assume([m k],'clear')
dsolve(m*diff(x,t,t) == -k*x, x(0)==0)
```

```
ans =
C10*exp((t*(-k*m)^(1/2))/m) + C10*exp(-(t*(-k*m)^(1/2))/m)
```

Restore the original assumptions using `assume`.

```
assume(tmp)
```

After computations are complete, clear assumptions using `assume`.

```
assume([m k],'clear')
```

Input Arguments

`var` — Symbolic input to check for assumptions

symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Symbolic input for which to show assumptions, specified as a symbolic variable, expression, or function, or a vector, matrix, or multidimensional array of symbolic variables, expressions, or functions.

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 you declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable `var` use this command.

  ```
  assume(var,'clear')
  ```
- To close the Symbolic Math Toolbox engine associated with the MATLAB workspace resetting all its assumptions, use this command.

  ```
  reset(symengine)
  ```

  Immediately before or after executing `reset(symengine)` you should clear all symbolic objects in the MATLAB workspace.
- To clear all objects in the MATLAB workspace and close the Symbolic Math Toolbox engine associated with the MATLAB workspace resetting all its assumptions, use this command.

  ```
  clear all
  ```

See Also

`and` | `assume` | `assumeAlso` | `clear` | `clear all` | `in` | `isAlways` | `not` | `or` | `piecewise` | `sym` | `syms`

Topics

"Set Assumptions" on page 1-28
"Check Existing Assumptions" on page 1-29
"Delete Symbolic Objects and Their Assumptions" on page 1-29
"Default Assumption" on page 1-28

Introduced in R2012a

atan

Symbolic inverse tangent function

Syntax

`atan(X)`

Description

`atan(X)` returns the inverse tangent function (arctangent function) of `X`.

Examples

Inverse Tangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `atan` returns floating-point or exact symbolic results.

Compute the inverse tangent function for these numbers. Because these numbers are not symbolic objects, `atan` returns floating-point results.

```
A = atan([-1, -1/3, -1/sqrt(3), 1/2, 1, sqrt(3)])
```

```
A =
   -0.7854   -0.3218   -0.5236    0.4636    0.7854    1.0472
```

Compute the inverse tangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `atan` returns unresolved symbolic calls.

```
symA = atan(sym([-1, -1/3, -1/sqrt(3), 1/2, 1, sqrt(3)]))
```

```
symA =
[ -pi/4, -atan(1/3), -pi/6, atan(1/2), pi/4, pi/3]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

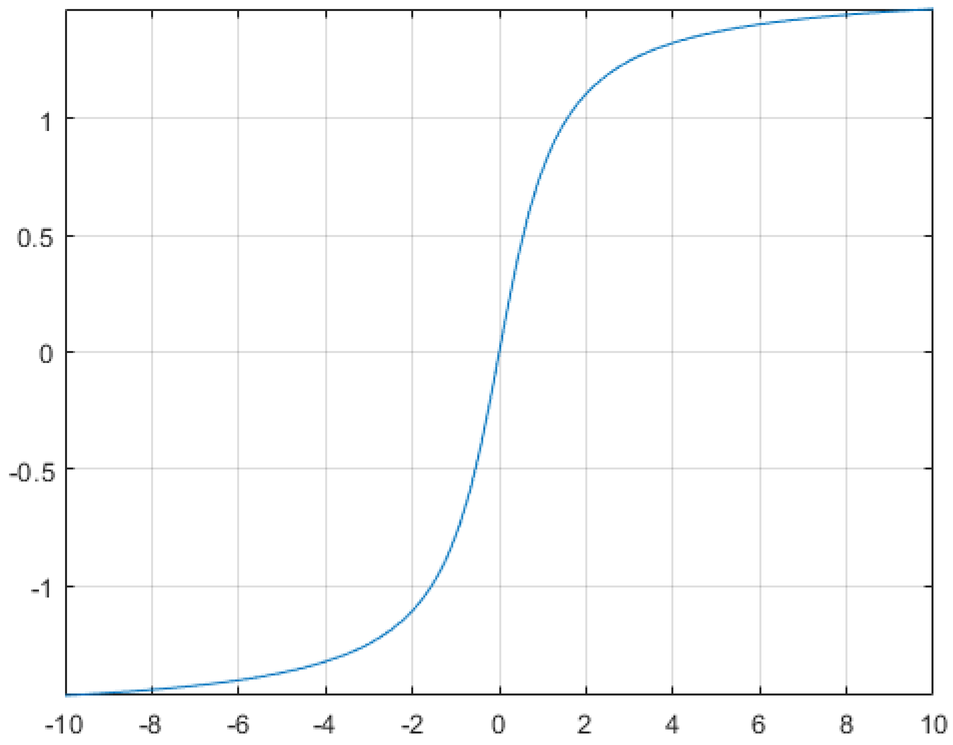
```
vpa(symA)
```

```
ans =
[ -0.78539816339744830961566084581988,...
-0.32175055439664219340140461435866,...
-0.52359877559829887307710723054658,...
0.46364760900080611621425623146121,...
0.78539816339744830961566084581988,...
1.0471975511965977461542144610932]
```

Plot Inverse Tangent Function

Plot the inverse tangent function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(atan(x), [-10, 10])
grid on
```



Handle Expressions Containing Inverse Tangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `atan`.

Find the first and second derivatives of the inverse tangent function:

```
syms x
diff(atan(x), x)
diff(atan(x), x, x)
```

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

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

Find the indefinite integral of the inverse tangent function:

```
int(atan(x), x)
```

```
ans =
x*atan(x) - log(x^2 + 1)/2
```

Find the Taylor series expansion of `atan(x)`:

```
taylor(atan(x), x)
```

```
ans =
x^5/5 - x^3/3 + x
```

Rewrite the inverse tangent function in terms of the natural logarithm:

```
rewrite(atan(x), 'log')
```

```
ans =
(log(1 - x*1i)*1i)/2 - (log(1 + x*1i)*1i)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan2` | `cos` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

atan2

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.

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.

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), pi/4]
```

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]
```

Definitions

atan2 vs. atan

If $X \neq 0$ and $Y \neq 0$, then

$$\operatorname{atan2}(Y,X) = \operatorname{atan}(\frac{Y}{X}) + \frac{\pi}{2}\operatorname{sign}(Y)\left(1 - \operatorname{sign}(X)\right)$$

Results returned by `atan2` belong to the closed interval `[-pi,pi]`. Results returned by `atan` belong to the closed interval `[-pi/2,pi/2]`.

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`.

Alternatives

For complex `Z = X + Y*i`, the call `atan2(Y,X)` is equivalent to `angle(Z)`.

See Also

`angle` | `atan` | `conj` | `imag` | `real`

Introduced in R2013a

atanh

Symbolic inverse hyperbolic tangent function

Syntax

```
atanh(X)
```

Description

`atanh(X)` returns the inverse hyperbolic tangent function of `X`.

Examples

Inverse Hyperbolic Tangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `atanh` returns floating-point or exact symbolic results.

Compute the inverse hyperbolic tangent function for these numbers. Because these numbers are not symbolic objects, `atanh` returns floating-point results.

```
A = atanh([-i, 0, 1/6, i/2, i, 2])
```

```
A =
   0.0000 - 0.7854i   0.0000 + 0.0000i   0.1682 + 0.0000i...
   0.0000 + 0.4636i   0.0000 + 0.7854i   0.5493 + 1.5708i
```

Compute the inverse hyperbolic tangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `atanh` returns unresolved symbolic calls.

```
symA = atanh(sym([-i, 0, 1/6, i/2, i, 2]))
```

```
symA =
[ -(pi*1i)/4, 0, atanh(1/6), atanh(1i/2), (pi*1i)/4, atanh(2)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

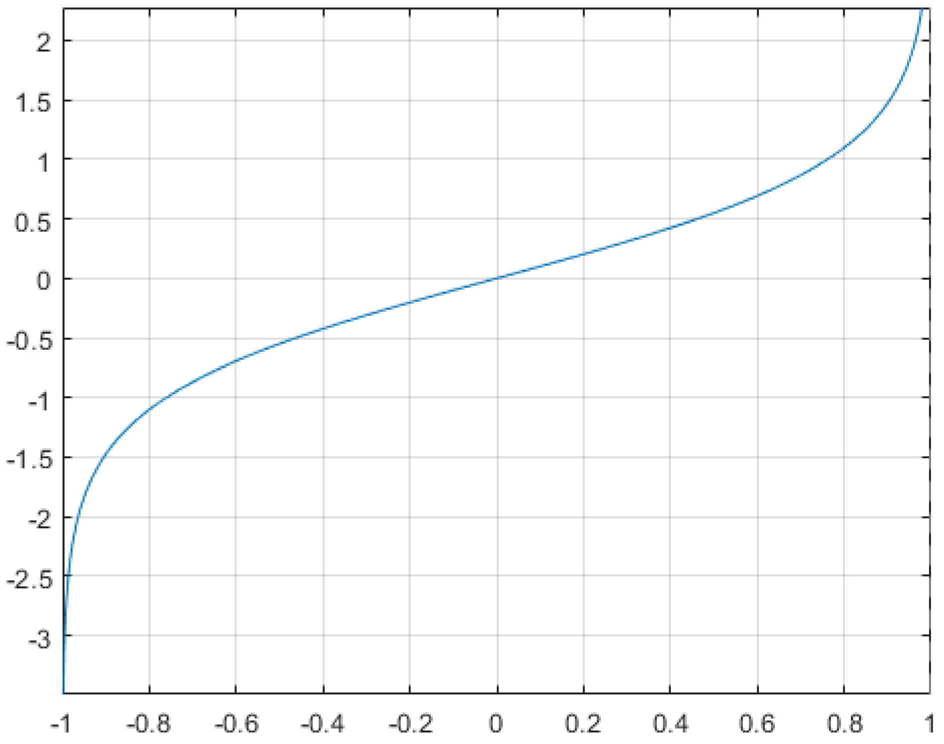
```
vpa(symA)
```

```
ans =
[ -0.78539816339744830961566084581988i,...
0,...
0.1682361183106064652522967051085,...
0.46364760900080611621425623146121i,...
0.78539816339744830961566084581988i,...
0.54930614433405484569762261846126 - 1.5707963267948966192313216916398i]
```

Plot Inverse Hyperbolic Tangent Function

Plot the inverse hyperbolic tangent function on the interval from -1 to 1. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(atanh(x), [-1, 1])
grid on
```



Handle Expressions Containing Inverse Hyperbolic Tangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `atanh`.

Find the first and second derivatives of the inverse hyperbolic tangent function:

```
syms x
diff(atanh(x), x)
diff(atanh(x), x, x)
```

```
ans =
-1/(x^2 - 1)
```

```
ans =
(2*x)/(x^2 - 1)^2
```

Find the indefinite integral of the inverse hyperbolic tangent function:

```
int(atanh(x), x)

ans =
log(x^2 - 1)/2 + x*atanh(x)
```

Find the Taylor series expansion of `atanh(x)`:

```
taylor(atanh(x), x)

ans =
x^5/5 + x^3/3 + x
```

Rewrite the inverse hyperbolic tangent function in terms of the natural logarithm:

```
rewrite(atanh(x), 'log')

ans =
log(x + 1)/2 - log(1 - x)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

baseUnits

Base units of unit system

Syntax

```
baseUnits(unitSystem)
```

Description

`baseUnits(unitSystem)` returns the base units of the unit system `unitSystem` as a vector of symbolic units. You can use the returned units to create new unit systems by using `newUnitSystem`.

Examples

Base Units of Unit System

Get the base units of a unit system by using `baseUnits`. By default, the available unit systems are `SI`, `CGS`, and `US`. Then, modify the base units and create a new unit system using the modified base units.

Get the base units of the `SI` unit system.

```
SIUnits = baseUnits('SI')
```

```
SIUnits =
[ [kg], [s], [m], [A], [cd], [mol], [K]]
```

Note Do not define a variable called `baseUnits` because the variable will prevent access to the `baseUnits` function.

Define base units that use kilometer for length and hour for time by modifying `SIUnits` using `subs`.

```
u = symunit;
newUnits = subs(SIUnits,[u.m u.s],[u.km u.hr])
```

```
newUnits =
[ [kg], [h], [km], [A], [cd], [mol], [K]]
```

Define the new unit system by using `newUnitSystem`.

```
newUnitSystem('SI_km_hr',newUnits)
```

```
ans = 
    "SI_km_hr"
```

To convert units between unit systems, see “Unit Conversions and Unit Systems” on page 2-30.

- “Units of Measurement Tutorial” on page 2-5
- “Unit Conversions and Unit Systems” on page 2-30
- “Units List” on page 2-13

Input Arguments

`unitSystem` **— Name of unit system**
string | character vector

Name of the unit system, specified as a string or character vector.

See Also

`derivedUnits` | `newUnitSystem` | `removeUnitSystem` | `rewrite` | `symunit`

Topics

“Units of Measurement Tutorial” on page 2-5
“Unit Conversions and Unit Systems” on page 2-30
“Units List” on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

bernoulli

Bernoulli numbers and polynomials

Syntax

```
bernoulli(n)
bernoulli(n,x)
```

Description

`bernoulli(n)` returns the `n`th Bernoulli number on page 4-118.

`bernoulli(n,x)` returns the `n`th Bernoulli polynomial on page 4-118.

Examples

Bernoulli Numbers with Odd and Even Indices

The 0th Bernoulli number is `1`. The next Bernoulli number can be `-1/2` or `1/2`, depending on the definition. The `bernoulli` function uses `-1/2`. The Bernoulli numbers with even indices `n > 1` alternate the signs. Any Bernoulli number with an odd index `n > 2` is `0`.

Compute the even-indexed Bernoulli numbers with the indices from `0` to `10`. Because these indices are not symbolic objects, `bernoulli` returns floating-point results.

```
bernoulli(0:2:10)
```

```
ans =
    1.0000    0.1667   -0.0333    0.0238   -0.0333    0.0758
```

Compute the same Bernoulli numbers for the indices converted to symbolic objects:

```
bernoulli(sym(0:2:10))
```

```
ans =
[ 1, 1/6, -1/30, 1/42, -1/30, 5/66]
```

Compute the odd-indexed Bernoulli numbers with the indices from `1` to `11`:

```
bernoulli(sym(1:2:11))
```

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

Bernoulli Polynomials

For the Bernoulli polynomials, use `bernoulli` with two input arguments.

Compute the first, second, and third Bernoulli polynomials in variables `x`, `y`, and `z`, respectively:

```
syms x y z
bernoulli(1, x)
bernoulli(2, y)
bernoulli(3, z)
```

```
ans =
x - 1/2

ans =
y^2 - y + 1/6

ans =
z^3 - (3*z^2)/2 + z/2
```

If the second argument is a number, `bernoulli` evaluates the polynomial at that number. Here, the result is a floating-point number because the input arguments are not symbolic numbers:

```
bernoulli(2, 1/3)
```

```
ans =
   -0.0556
```

To get the exact symbolic result, convert at least one of the numbers to a symbolic object:

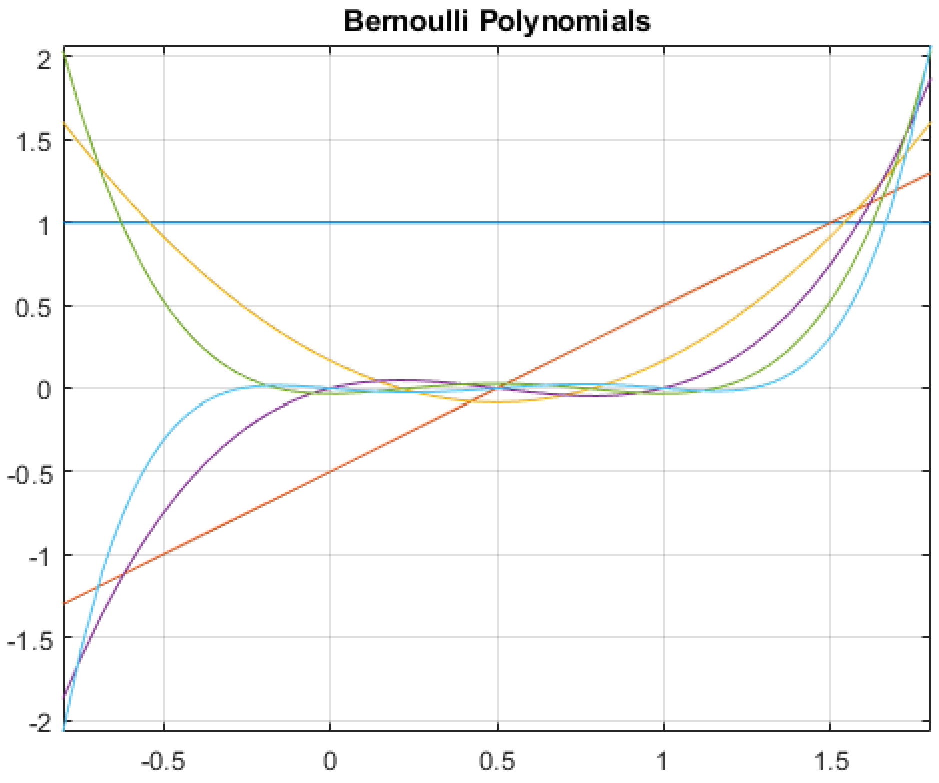
```
bernoulli(2, sym(1/3))
```

```
ans =
-1/18
```

Plot Bernoulli Polynomials

Plot the first six Bernoulli polynomials. Prior to R2016a, use `ezplot` instead of `fplot`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(bernoulli(0:5, x), [-0.8 1.8])
title('Bernoulli Polynomials')
grid on
```



Handle Expressions Containing Bernoulli Polynomials

Many functions, such as `diff` and `expand`, handles expressions containing `bernoulli`.

Find the first and second derivatives of the Bernoulli polynomial:

```
syms n x
diff(bernoulli(n,x^2), x)

ans =
2*n*x*bernoulli(n - 1, x^2)

diff(bernoulli(n,x^2), x, x)

ans =
2*n*bernoulli(n - 1, x^2) +...
4*n*x^2*bernoulli(n - 2, x^2)*(n - 1)
```

Expand these expressions containing the Bernoulli polynomials:

```
expand(bernoulli(n, x + 3))

ans =
bernoulli(n, x) + (n*(x + 1)^n)/(x + 1) +...
(n*(x + 2)^n)/(x + 2) + (n*x^n)/x

expand(bernoulli(n, 3*x))

ans =
(3^n*bernoulli(n, x))/3 + (3^n*bernoulli(n, x + 1/3))/3 +...
(3^n*bernoulli(n, x + 2/3))/3
```

Input Arguments

`n` — Index of the Bernoulli number or polynomial

nonnegative integer | symbolic nonnegative integer | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Index of the Bernoulli number or polynomial, specified as a nonnegative integer, symbolic nonnegative integer, variable, expression, function, vector, or matrix. If `n` is a vector or matrix, `bernoulli` returns Bernoulli numbers or polynomials for each element of `n`. If one input argument is a scalar and the other one is a vector or a matrix, `bernoulli(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.

`x` — Polynomial variable
symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Polynomial variable, specified as a symbolic variable, expression, function, vector, or matrix. If `x` is a vector or matrix, `bernoulli` returns Bernoulli numbers or polynomials for each element of `x`. When you use the `bernoulli` function to find Bernoulli polynomials, at least one argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `bernoulli(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.

Definitions

Bernoulli Polynomials

The Bernoulli polynomials are defined as follows:

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} \text{bernoulli}(n,x) \frac{t^n}{n!}$$

Bernoulli Numbers

The Bernoulli numbers are defined as follows:

$$\text{bernoulli}(n) = \text{bernoulli}(n,0)$$

See Also

`euler`

Introduced in R2014a

bernstein

Bernstein polynomials

Syntax

```
bernstein(f,n,t)
bernstein(g,n,t)
bernstein(g,var,n,t)
```

Description

`bernstein(f,n,t)` with a function handle `f` returns the `n`th-order Bernstein polynomial on page 4-125 `symsum(nchoosek(n,k)*t^k*(1-t)^(n-k)*f(k/n),k, 0,n)`, evaluated at the point `t`. This polynomial approximates the function `f` over the interval `[0,1]`.

`bernstein(g,n,t)` with a symbolic expression or function `g` returns the `n`th-order Bernstein polynomial, evaluated at the point `t`. This syntax regards `g` as a univariate function of the variable determined by `symvar(g,1)`.

If any argument is symbolic, `bernstein` converts all arguments except a function handle to symbolic, and converts a function handle's results to symbolic.

`bernstein(g,var,n,t)` with a symbolic expression or function `g` returns the approximating `n`th-order Bernstein polynomial, regarding `g` as a univariate function of the variable `var`.

Examples

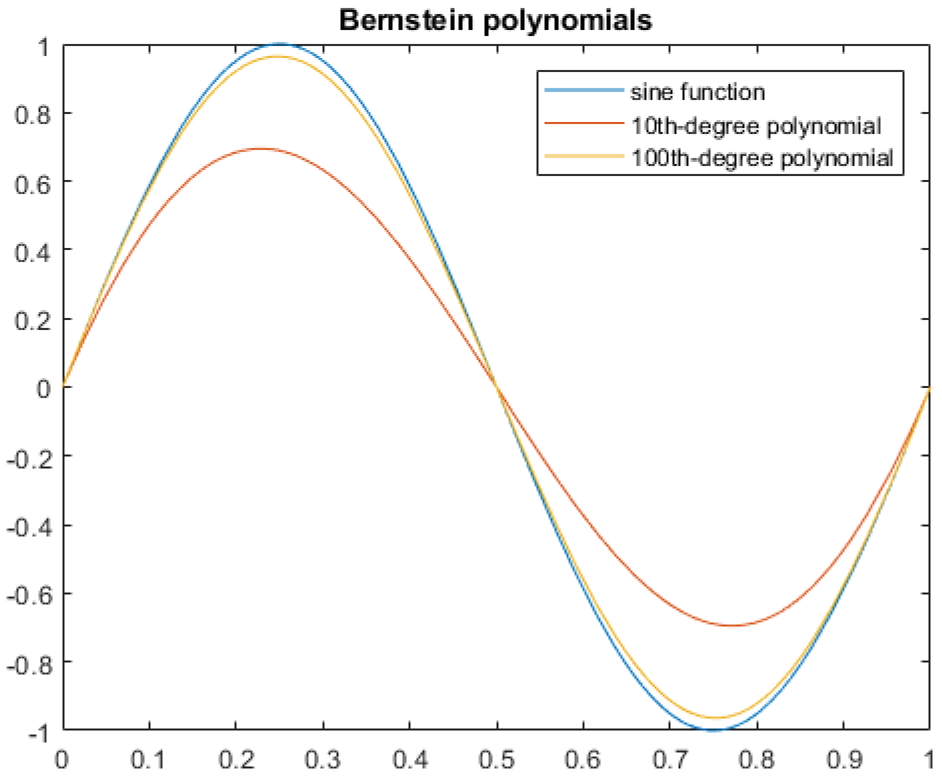
Approximation of Sine Function Specified as Function Handle

Approximate the sine function by the 10th- and 100th-degree Bernstein polynomials:

```
syms t
b10 = bernstein(@(t) sin(2*pi*t), 10, t);
b100 = bernstein(@(t) sin(2*pi*t), 100, t);
```

Plot `sin(2*pi*t)` and its approximations:

```
fplot(sin(2*pi*t),[0,1])
hold on
fplot(b10,[0,1])
fplot(b100,[0,1])

legend('sine function','10th-degree polynomial',...
                      '100th-degree polynomial')
title('Bernstein polynomials')
hold off
```



Approximation of Exponential Function Specified as Symbolic Expression

Approximate the exponential function by the second-order Bernstein polynomial in the variable `t`:

```
syms x t
bernstein(exp(x), 2, t)
```

```
ans =
(t - 1)^2 + t^2*exp(1) - 2*t*exp(1/2)*(t - 1)
```

Approximate the multivariate exponential function. When you approximate a multivariate function, `bernstein` regards it as a univariate function of the default variable determined by `symvar`. The default variable for the expression `y*exp(x*y)` is `x`:

```
syms x y t
symvar(y*exp(x*y), 1)

ans =
x
```

`bernstein` treats this expression as a univariate function of `x`:

```
bernstein(y*exp(x*y), 2, t)

ans =
y*(t - 1)^2 + t^2*y*exp(y) - 2*t*y*exp(y/2)*(t - 1)
```

To treat `y*exp(x*y)` as a function of the variable `y`, specify the variable explicitly:

```
bernstein(y*exp(x*y), y, 2, t)

ans =
t^2*exp(x) - t*exp(x/2)*(t - 1)
```

Approximation of Linear Ramp Specified as Symbolic Function

Approximate function `f` representing a linear ramp by the fifth-order Bernstein polynomials in the variable `t`:

```
syms f(t)
f(t) = triangularPulse(1/4, 3/4, Inf, t);
p = bernstein(f, 5, t)

p =
7*t^3*(t - 1)^2 - 3*t^2*(t - 1)^3 - 5*t^4*(t - 1) + t^5
```

Simplify the result:

```
simplify(p)

ans =
-t^2*(2*t - 3)
```

Numerical Stability of Simplified Bernstein Polynomials

When you simplify a high-order symbolic Bernstein polynomial, the result often cannot be evaluated in a numerically stable way.

Approximate this rectangular pulse function by the 100th-degree Bernstein polynomial, and then simplify the result:

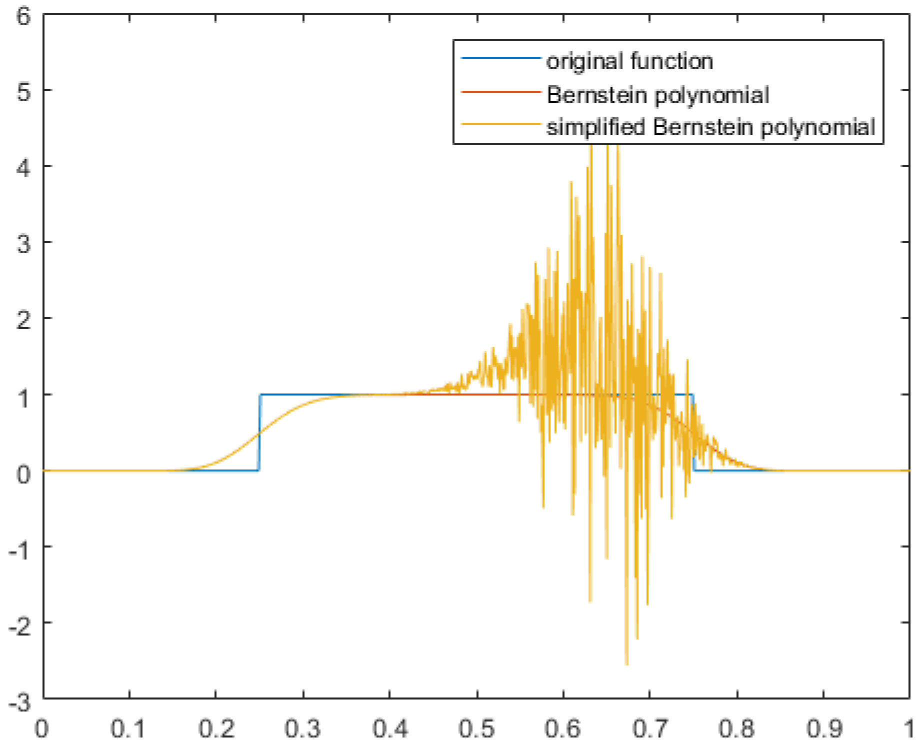
```
f = @(x)rectangularPulse(1/4,3/4,x);
b1 = bernstein(f, 100, sym('t'));
b2 = simplify(b1);
```

Convert the polynomial `b1` and the simplified polynomial `b2` to MATLAB functions:

```
f1 = matlabFunction(b1);
f2 = matlabFunction(b2);
```

Compare the plot of the original rectangular pulse function, its numerically stable Bernstein representation `f1`, and its simplified version `f2`. The simplified version is not numerically stable.

```
t = 0:0.001:1;
plot(t, f(t), t, f1(t), t, f2(t))
hold on
legend('original function','Bernstein polynomial',...
                'simplified Bernstein polynomial')
hold off
```



Input Arguments

`f` — Function to be approximated by a polynomial

function handle

Function to be approximated by a polynomial, specified as a function handle. `f` must accept one scalar input argument and return a scalar value.

`g` — Function to be approximated by a polynomial

symbolic expression | symbolic function

Function to be approximated by a polynomial, specified as a symbolic expression or function.

`n` — Bernstein polynomial order

nonnegative integer

Bernstein polynomial order, specified as a nonnegative number.

`t` — Evaluation point

number | symbolic number | symbolic variable | symbolic expression | symbolic function

Evaluation point, specified as a number, symbolic number, variable, expression, or function. If `t` is a symbolic function, the evaluation point is the mathematical expression that defines `t`. To extract the mathematical expression defining `t`, `bernstein` uses `formula(t)`.

`var` — Free variable

symbolic variable

Free variable, specified as a symbolic variable.

Definitions

Bernstein Polynomials

A Bernstein polynomial is a linear combination of Bernstein basis polynomials.

A Bernstein polynomial of degree `n` is defined as follows:

$$B(t) = \sum_{k=0}^{n} \beta_k b_{k,n}(t).$$

Here,

$$b_{k,n}(t) = \binom{n}{k} t^k (1-t)^{n-k}, \quad k = 0, \ldots, n$$

are the Bernstein basis polynomials, and $\binom{n}{k}$ is a binomial coefficient.

The coefficients β_k are called Bernstein coefficients or Bezier coefficients.

If `f` is a continuous function on the interval `[0, 1]` and

$$B_n(f)(t) = \sum_{k=0}^{n} f\left(\frac{k}{n}\right) b_{k,n}(t)$$

is the approximating Bernstein polynomial, then

$$\lim_{n \to \infty} B_n(f)(t) = f(t)$$

uniformly in `t` on the interval `[0, 1]`.

Tips

- Symbolic polynomials returned for symbolic `t` are numerically stable when substituting numerical values between `0` and `1` for `t`.
- If you simplify a symbolic Bernstein polynomial, the result can be unstable when substituting numerical values for the curve parameter `t`.

See Also

`bernsteinMatrix` | `formula` | `nchoosek` | `symsum` | `symvar`

Introduced in R2013b

bernsteinMatrix

Bernstein matrix

Syntax

```
B = bernsteinMatrix(n,t)
```

Description

`B = bernsteinMatrix(n,t)`, where `t` is a vector, returns the `length(t)`-by-`(n+1)` Bernstein matrix `B`, such that `B(i,k+1)= nchoosek(n,k)*t(i)^k*(1-t(i))^(n-k)`. Here, the index `i` runs from 1 to `length(t)`, and the index `k` runs from `0` to `n`.

The Bernstein matrix is also called the Bezier matrix.

Use Bernstein matrices to construct Bezier curves:

```
bezierCurve = bernsteinMatrix(n, t)*P
```

Here, the `n+1` rows of the matrix `P` specify the control points of the Bezier curve. For example, to construct the second-order 3-D Bezier curve, specify the control points as:

```
P = [p0x, p0y, p0z;  p1x, p1y, p1z;  p2x, p2y, p2z]
```

Examples

2-D Bezier Curve

Plot the fourth-order Bezier curve specified by the control points `p0 = [0 1]`,`p1 = [4 3]`,`p2 = [6 2]`,`p3 = [3 0]`,`p4 = [2 4]`. Create a matrix with each row representing a control point:

```
P = [0 1; 4 3; 6 2; 3 0; 2 4];
```

Compute the fourth-order Bernstein matrix `B`:

```
syms t
B = bernsteinMatrix(4, t)
```

```
B =
[ (t - 1)^4, -4*t*(t - 1)^3, 6*t^2*(t - 1)^2, -4*t^3*(t - 1), t^4]
```

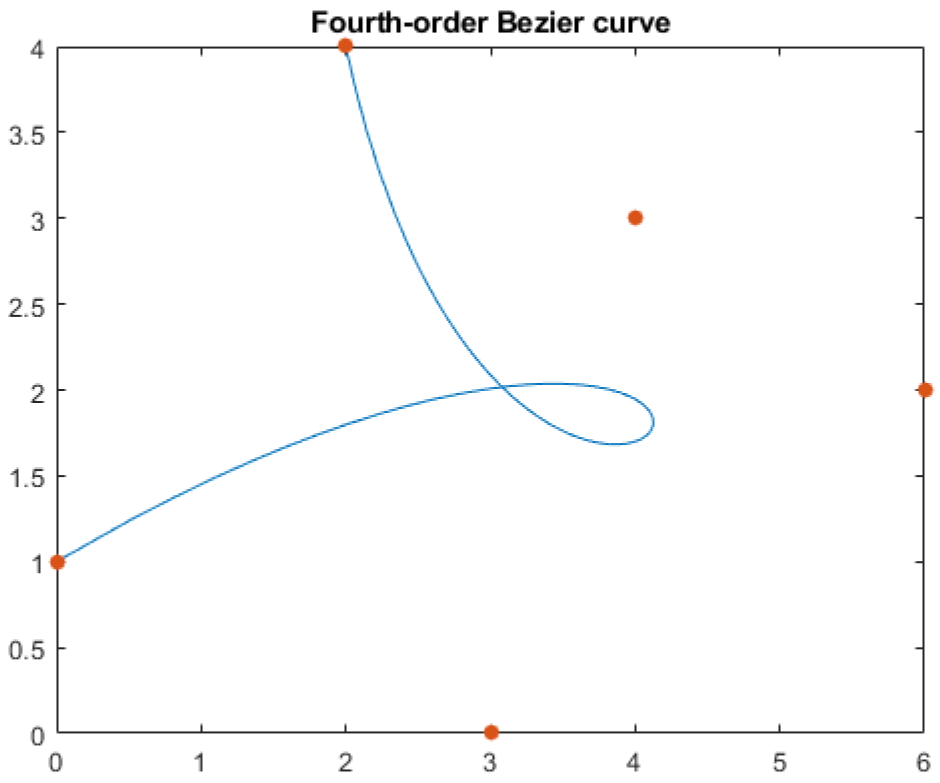
Construct the Bezier curve:

```
bezierCurve = simplify(B*P)
```

```
bezierCurve =
[ -2*t*(- 5*t^3 + 6*t^2 + 6*t - 8), 5*t^4 + 8*t^3 - 18*t^2 + 8*t + 1]
```

Plot the curve adding the control points to the plot:

```
fplot(bezierCurve(1), bezierCurve(2), [0, 1])
hold on
scatter(P(:,1), P(:,2),'filled')
title('Fourth-order Bezier curve')
hold off
```



3-D Bezier Curve

Construct the third-order Bezier curve specified by the 4-by-3 matrix `P` of control points. Each control point corresponds to a row of the matrix `P`.

```
P = [0 0 0; 2 2 2; 2 -1 1; 6 1 3];
```

Compute the third-order Bernstein matrix:

```
syms t
B = bernsteinMatrix(3,t)
```

```
B =
[ -(t - 1)^3, 3*t*(t - 1)^2, -3*t^2*(t - 1), t^3]
```

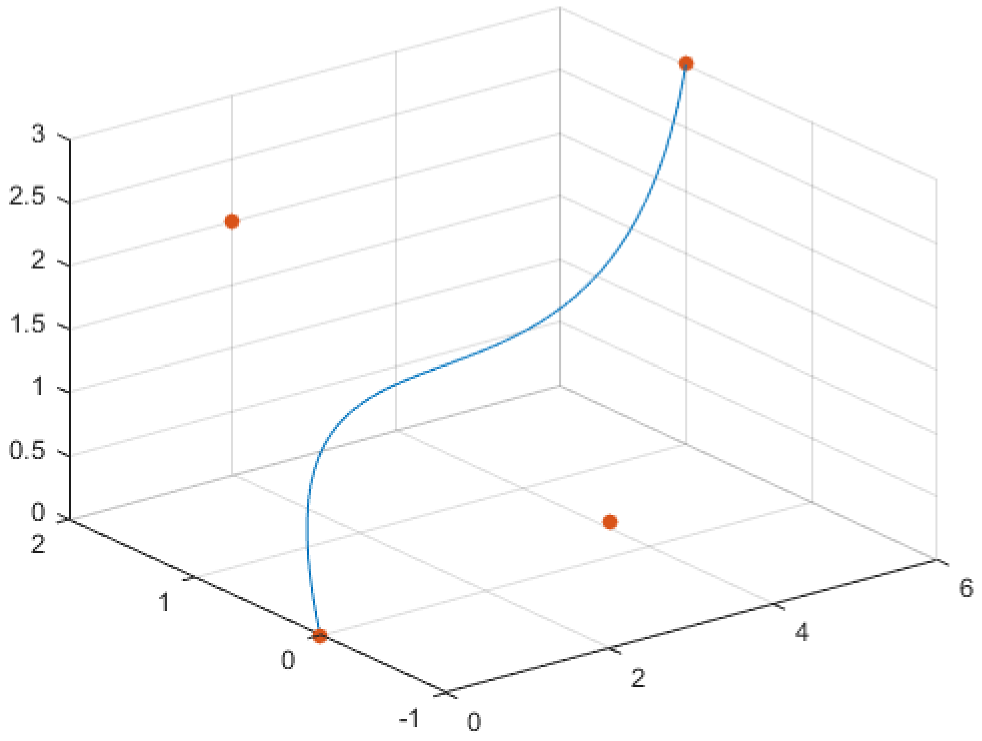
Construct the Bezier curve:

```
bezierCurve = simplify(B*P)
```

```
bezierCurve =
[ 6*t*(t^2 - t + 1), t*(10*t^2 - 15*t + 6), 3*t*(2*t^2 - 3*t + 2)]
```

Plot the curve adding the control points to the plot:

```
fplot3(bezierCurve(1), bezierCurve(2), bezierCurve(3), [0, 1])
hold on
scatter3(P(:,1), P(:,2), P(:,3),'filled')
hold off
```



3-D Bezier Curve with Evaluation Point Specified as Vector

Construct the third-order Bezier curve with the evaluation point specified by the following `1`-by-`101` vector `t`:

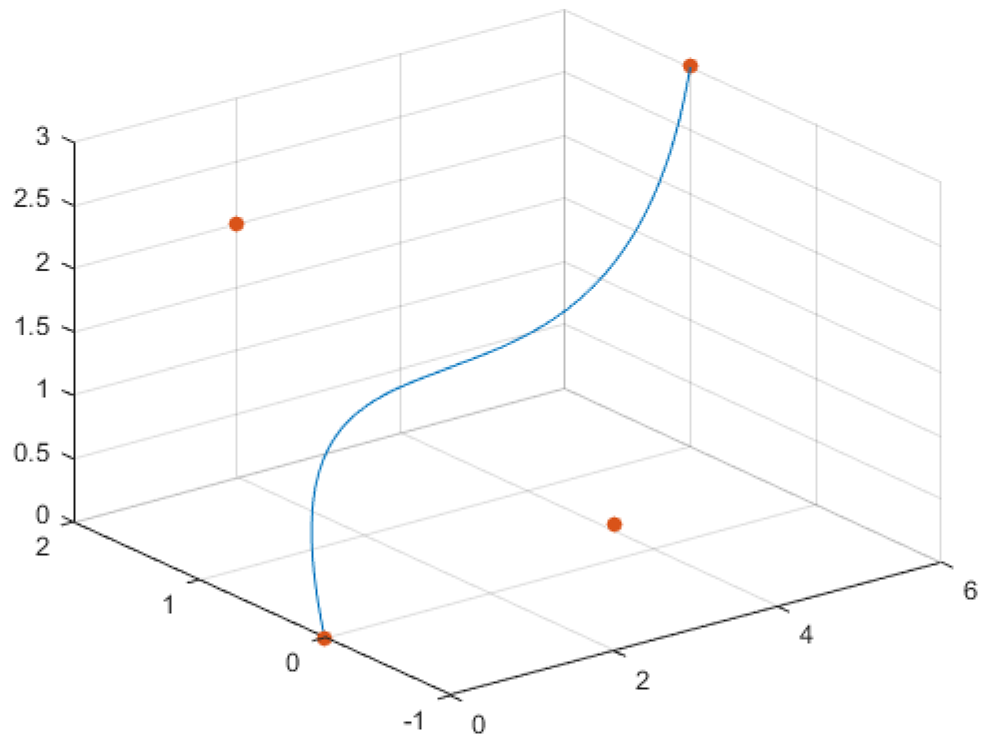
```
t = 0:1/100:1;
```

Compute the third-order `101`-by-`4` Bernstein matrix and specify the control points:

```
B = bernsteinMatrix(3,t);
P = [0 0 0; 2 2 2; 2 -1 1; 6 1 3];
```

Construct and plot the Bezier curve. Add grid lines and control points to the plot.

```
bezierCurve = B*P;
plot3(bezierCurve(:,1), bezierCurve(:,2), bezierCurve(:,3))
hold on
grid
scatter3(P(:,1), P(:,2), P(:,3),'filled')
hold off
```



Input Arguments

`n` — Approximation order

nonnegative integer

Approximation order, specified as a nonnegative integer.

`t` — Evaluation point

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic vector

Evaluation point, specified as a number, symbolic number, variable, expression, or vector.

Output Arguments

`B` — Bernstein matrix

matrix

Bernstein matrix, returned as a `length(t)`-by-`n+1` matrix.

See Also

`bernstein` | `nchoosek` | `symsum` | `symvar`

Introduced in R2013b

besseli

Modified Bessel function of the first kind

Syntax

```
besseli(nu,z)
```

Description

`besseli(nu,z)` returns the modified Bessel function of the first kind on page 4-138, $I_v(z)$.

Input Arguments

`nu`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `nu` is a vector or matrix, `besseli` returns the modified Bessel function of the first kind for each element of `nu`.

`z`

Symbolic number, variable, expression, or function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `z` is a vector or matrix, `besseli` returns the modified Bessel function of the first kind for each element of `z`.

Examples

Find Modified Bessel Function of First Kind

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 + 0.0000i   1.5906 + 0.0000i   1.7951 + 0.0000i  -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 + 2i)]
```

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)]
```

Solve Bessel Differential Equation for Modified Bessel Functions

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
isAlways(z^2*diff(besseli(nu, z), z, 2) + z*diff(besseli(nu, z), z)...
 - (z^2 + nu^2)*besseli(nu, z) == 0)

ans =
  logical
   1
```

Special Values of Modified Bessel Function of First Kind

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))/(x^(1/2)*pi^(1/2))

besseli(-1/2, x)

ans =
(2^(1/2)*cosh(x))/(x^(1/2)*pi^(1/2))

besseli(-3/2, x)

ans =
(2^(1/2)*(sinh(x) - cosh(x)/x))/(x^(1/2)*pi^(1/2))

besseli(5/2, x)

ans =
-(2^(1/2)*((3*cosh(x))/x - sinh(x)*(3/x^2 + 1)))/(x^(1/2)*pi^(1/2))
```

Differentiate Modified Bessel Function of First Kind

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))
```

Bessel Function for Matrix Input

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))`.

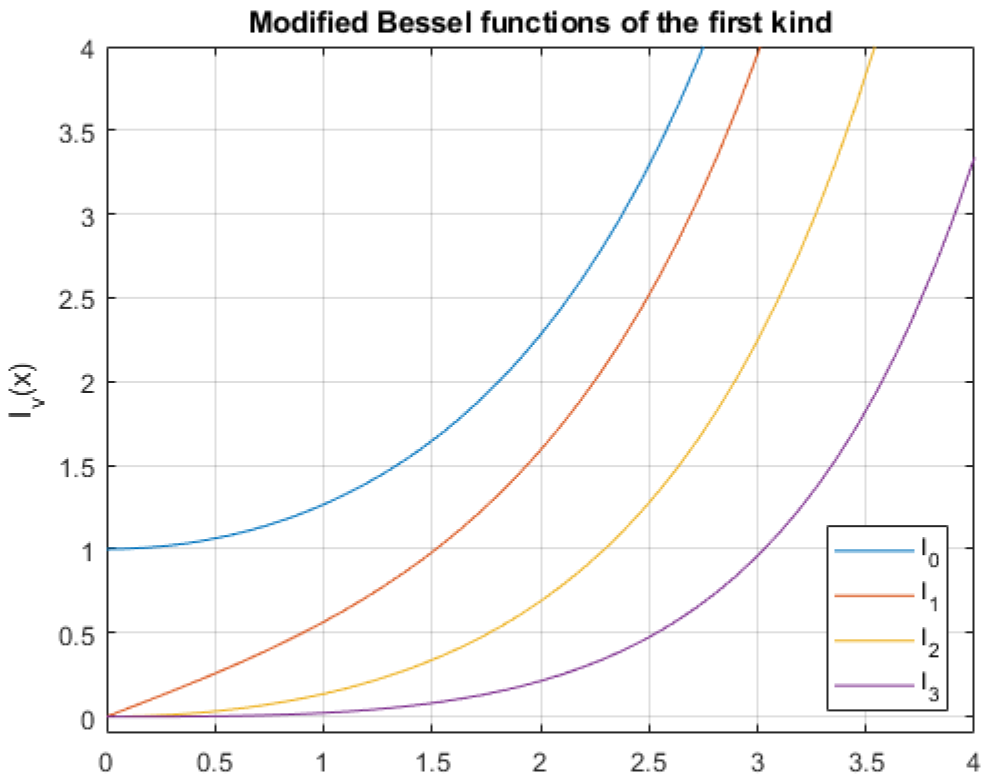
```
syms x
A = [-1, pi; x, 0];
besseli(1/2, A)
```

```
ans =
[        (2^(1/2)*sinh(1)*1i)/pi^(1/2), (2^(1/2)*sinh(pi))/pi]
[ (2^(1/2)*sinh(x))/(x^(1/2)*pi^(1/2)),                     0]
```

Plot the Modified Bessel Functions of the First Kind

Plot the modified Bessel functions of the first kind for $\nu = 0, 1, 2, 3$. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(besseli(0:3, x))
axis([0 4 -0.1 4])
grid on

ylabel('I_v(x)')
legend('I_0','I_1','I_2','I_3', 'Location','Best')
title('Modified Bessel functions of the first kind')
```



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, $I_v(z)$, and the modified Bessel functions of the second kind, $K_v(z)$:

$$w(z) = C_1 I_\nu(z) + C_2 K_\nu(z)$$

This formula is the integral representation of the modified Bessel functions of the first kind:

$$I_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi e^{z\cos(t)} \sin(t)^{2\nu}\, dt$$

Tips

- Calling `besseli` for a number that is not a symbolic object invokes the MATLAB `besseli` function.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `besseli(nu,z)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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`

Introduced in R2014a

besselj

Bessel function of the first kind

Syntax

```
besselj(nu,z)
```

Description

`besselj(nu,z)` returns the Bessel function of the first kind on page 4-144, $J_v(z)$.

Input Arguments

`nu`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `nu` is a vector or matrix, `besseli` returns the Bessel function of the first kind for each element of `nu`.

`z`

Symbolic number, variable, expression, or function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `z` is a vector or matrix, `besseli` returns the Bessel function of the first kind for each element of `z`.

Examples

Find Bessel Function of First Kind

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 + 0.0000i  -0.5767 + 0.0000i   0.5496 + 0.0000i   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 + 2i)]
```

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)]
```

Solve Bessel Differential Equation for Bessel Functions

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
isAlways(z^2*diff(besselj(nu, z), z, 2) + z*diff(besselj(nu, z), z)...
 + (z^2 - nu^2)*besselj(nu, z) == 0)

ans =
  logical
   1
```

Special Values of Bessel Function of First Kind

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))/(x^(1/2)*pi^(1/2))

besselj(-1/2, x)

ans =
(2^(1/2)*cos(x))/(x^(1/2)*pi^(1/2))

besselj(-3/2, x)

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

besselj(5/2, x)

ans =
-(2^(1/2)*((3*cos(x))/x - sin(x)*(3/x^2 - 1)))/(x^(1/2)*pi^(1/2))
```

Differentiate Bessel Function of First Kind

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))
```

Find Bessel Function for Matrix Input

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))`.

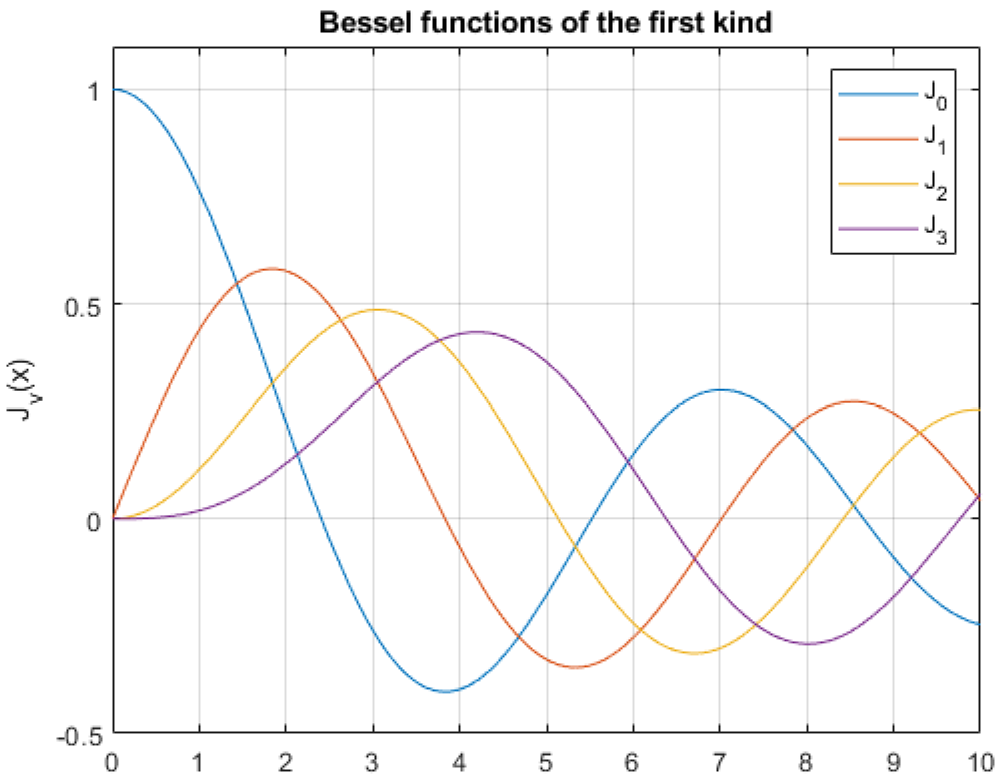
```
syms x
A = [-1, pi; x, 0];
besselj(1/2, A)
```

```
ans =
[        (2^(1/2)*sin(1)*1i)/pi^(1/2), 0]
[ (2^(1/2)*sin(x))/(x^(1/2)*pi^(1/2)), 0]
```

Plot Bessel Functions of First Kind

Plot the Bessel functions of the first kind for $0, 1, 2, 3$. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(besselj(0:3, x))
axis([0 10 -0.5 1.1])
grid on

ylabel('J_v(x)')
legend('J_0','J_1','J_2','J_3', 'Location','Best')
title('Bessel functions of the first kind')
```



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, $J_\nu(z)$, and the Bessel functions of the second kind, $Y_\nu(z)$:

$$w(z) = C_1 J_\nu(z) + C_2 Y_\nu(z)$$

This formula is the integral representation of the Bessel functions of the first kind:

$$J_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi \cos(z\cos(t))\sin(t)^{2\nu}\,dt$$

Tips

- Calling `besselj` for a number that is not a symbolic object invokes the MATLAB `besselj` function.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `besselj(nu,z)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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`

Introduced in R2014a

besselk

Modified Bessel function of the second kind

Syntax

```
besselk(nu,z)
```

Description

`besselk(nu,z)` returns the modified Bessel function of the second kind on page 4-150, $K_v(z)$.

Input Arguments

`nu`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `nu` is a vector or matrix, `besseli` returns the modified Bessel function of the second kind for each element of `nu`.

`z`

Symbolic number, variable, expression, or function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `z` is a vector or matrix, `besseli` returns the modified Bessel function of the second kind for each element of `z`.

Examples

Find Modified Bessel Function of Second Kind

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 + 0.0000i   0.1399 + 0.0000i   0.1594 + 0.0000i  -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 + 2i)]
```

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)]
```

Special Values of Modified Bessel Function of Second Kind

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))
```

Solve Bessel Differential Equation for Bessel Functions

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
isAlways(z^2*diff(besselk(nu, z), z, 2) + z*diff(besselk(nu, z), z)...
 - (z^2 + nu^2)*besselk(nu, z) == 0)

ans =
  logical
   1
```

Differentiate Modified Bessel Function of Second Kind

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)
```

Find Bessel Function for Matrix Input

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))`.

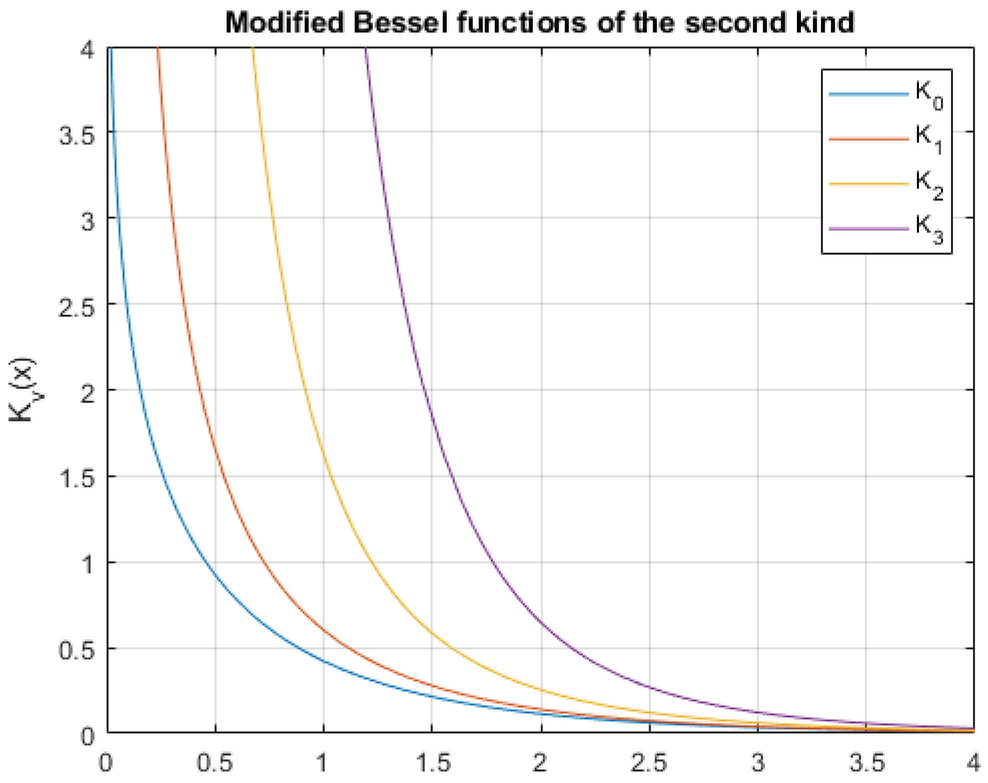
```
syms x
A = [-1, pi; x, 0];
besselk(1/2, A)
```

```
ans =
[         -(2^(1/2)*pi^(1/2)*exp(1)*1i)/2, (2^(1/2)*exp(-pi))/2]
[ (2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2)),                  Inf]
```

Plot Modified Bessel Functions of Second Kind

Plot the modified Bessel functions of the second kind for $v = 0, 1, 2, 3$. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(besselk(0:3, x))
axis([0 4 0 4])
grid on

ylabel('K_v(x)')
legend('K_0','K_1','K_2','K_3', 'Location','Best')
title('Modified Bessel functions of the second kind')
```



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 $I_v(z)$ are the modified Bessel functions of the first kind:

$$I_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu+1/2)}\int_0^\pi e^{z\cos(t)}\sin(t)^{2\nu}\,dt$$

Tips

- Calling `besselk` for a number that is not a symbolic object invokes the MATLAB `besselk` function.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `besselk(nu,z)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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`

Introduced in R2014a

bessely

Bessel function of the second kind

Syntax

```
bessely(nu,z)
```

Description

`bessely(nu,z)` returns the Bessel function of the second kind on page 4-157, $Y_v(z)$.

Input Arguments

`nu`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `nu` is a vector or matrix, `bessely` returns the Bessel function of the second kind for each element of `nu`.

`z`

Symbolic number, variable, expression, or function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `z` is a vector or matrix, `bessely` returns the Bessel function of the second kind for each element of `z`.

Examples

Find Bessel Function of Second Kind

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 + 0.0000i   0.1070 + 0.0000i   0.2358 + 0.0000i  -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 + 2i)]
```

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)]
```

Solve Bessel Differential Equation for Bessel Functions

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
isAlways(z^2*diff(bessely(nu, z), z, 2) + z*diff(bessely(nu, z), z)...
 + (z^2 - nu^2)*bessely(nu, z) == 0)

ans =
  logical
   1
```

Special Values of Bessel Function of Second Kind

If the first parameter is an odd integer multiplied by 1/2, `bessely` rewrites the Bessel functions in terms of elementary functions:

```
syms x
bessely(1/2, x)

ans =
-(2^(1/2)*cos(x))/(x^(1/2)*pi^(1/2))

bessely(-1/2, x)

ans =
(2^(1/2)*sin(x))/(x^(1/2)*pi^(1/2))

bessely(-3/2, x)

ans =
(2^(1/2)*(cos(x) - sin(x)/x))/(x^(1/2)*pi^(1/2))

bessely(5/2, x)

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

Differentiate Bessel Functions of Second Kind

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))
```

Find Bessel Function for Matrix Input

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))`.

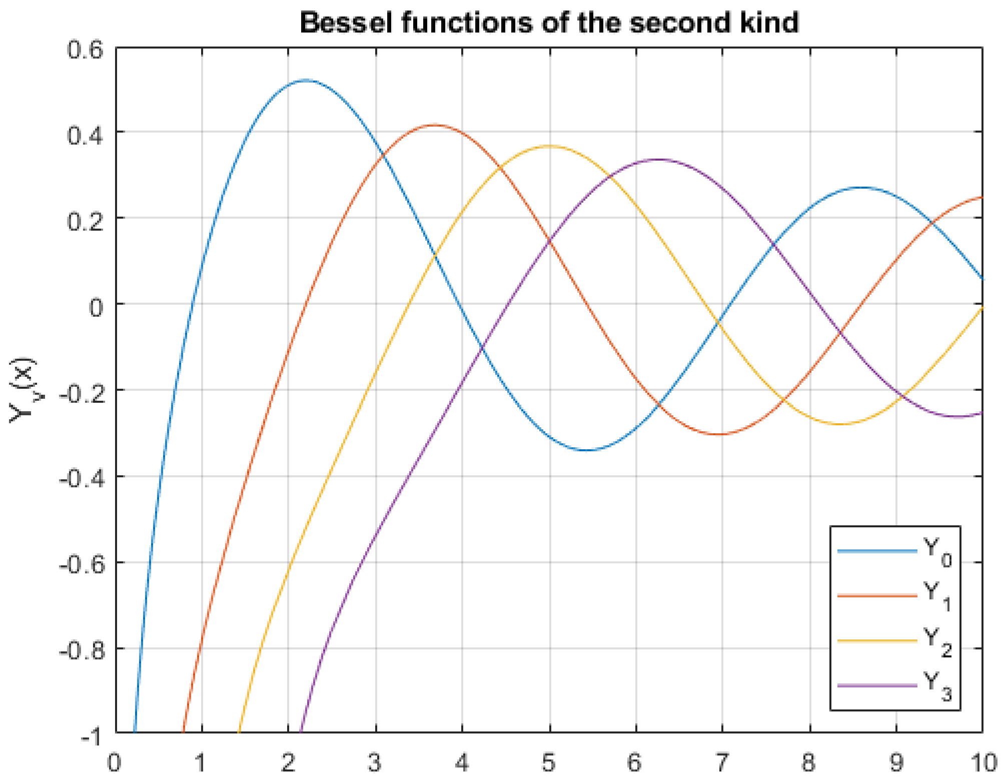
```
syms x
A = [-1, pi; x, 0];
bessely(1/2, A)
```

```
ans =
[          (2^(1/2)*cos(1)*1i)/pi^(1/2), 2^(1/2)/pi]
[ -(2^(1/2)*cos(x))/(x^(1/2)*pi^(1/2)),        Inf]
```

Plot Bessel Functions of Second Kind

Plot the Bessel functions of the second kind for $\nu = 0, 1, 2, 3$. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(bessely(0:3, x))
axis([0 10 -1 0.6])
grid on

ylabel('Y_v(x)')
legend('Y_0','Y_1','Y_2','Y_3', 'Location','Best')
title('Bessel functions of the second kind')
```



Definitions

Bessel Function of the Second 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, $J_\nu(z)$, and the Bessel functions of the second kind, $Y_\nu(z)$:

$$w(z) = C_1 J_\nu(z) + C_2 Y_\nu(z)$$

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:

$$J_\nu(z) = \frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu + 1/2)} \int_0^\pi \cos(z\cos(t))\sin(t)^{2\nu}\,dt$$

Tips

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

  At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `bessely(nu,z)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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`

Introduced in R2014a

beta

Beta function

Syntax

```
beta(x,y)
```

Description

`beta(x,y)` returns the beta function on page 4-162 of `x` and `y`.

Input Arguments

`x`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `x` is a vector or matrix, `beta` returns the beta function for each element of `x`.

`y`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `y` is a vector or matrix, `beta` returns the beta function for each element of `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(1i))/(2*gamma(3/2 + 1i)), gamma(1i)^2/gamma(2i),...
 (pi*(1/2 + 1i/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))
```

Definitions

Beta Function

This integral defines the beta function:

$$\mathrm{B}(x,y)=\int_0^1 t^{x-1}(1-t)^{y-1}\,dt=\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

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(sym(0),0)`, `beta(0,sym(0))`, and `beta(sym(0),sym(0))` return `NaN`.

- `beta(x,y) = beta(y,x)` and `beta(x,A) = beta(A,x)`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `beta(x,y)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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

`factorial` | `gamma` | `nchoosek` | `psi`

Introduced in R2014a

cat

Concatenate symbolic arrays along specified dimension

Syntax

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

Description

`cat(dim,A1,...,AN)` concatenates the arrays `A1,...,AN` along dimension `dim`. The remaining dimensions must be the same size.

Examples

Concatenate Two Vectors into Matrix

Create vectors `A` and `B`.

```
A = sym('a%d',[1 4])
B = sym('b%d',[1 4])
```

```
A =
[ a1, a2, a3, a4]
B =
[ b1, b2, b3, b4]
```

To concatenate `A` and `B` into a matrix, specify dimension `dim` as `1`.

```
cat(1,A,B)
```

```
ans =
[ a1, a2, a3, a4]
[ b1, b2, b3, b4]
```

Alternatively, use the syntax `[A;B]`.

```
[A;B]
```

```
ans =
[ a1, a2, a3, a4]
[ b1, b2, b3, b4]
```

Concatenate Two Vectors into One Vector

To concatenate two vectors into one vector, specify dimension `dim` as `2`.

```
A = sym('a%d',[1 4]);
B = sym('b%d',[1 4]);
cat(2,A,B)
```

```
ans =
[ a1, a2, a3, a4, b1, b2, b3, b4]
```

Alternatively, use the syntax `[A B]`.

```
[A B]
```

```
ans =
[ a1, a2, a3, a4, b1, b2, b3, b4]
```

Concatenate Multidimensional Arrays Along Their Third Dimension

Create arrays `A` and `B`.

```
A = sym('a%d%d',[2 2]);
A(:,:,2) = -A
B = sym('b%d%d', [2 2]);
B(:,:,2) = -B
```

```
A(:,:,1) =
[ a11, a12]
[ a21, a22]
A(:,:,2) =
[ -a11, -a12]
[ -a21, -a22]

B(:,:,1) =
[ b11, b12]
[ b21, b22]
B(:,:,2) =
[ -b11, -b12]
[ -b21, -b22]
```

Concatenate `A` and `B` by specifying dimension `dim` as `3`.

```
cat(3,A,B)
```

```
ans(:,:,1) =
[ a11, a12]
[ a21, a22]
ans(:,:,2) =
[ -a11, -a12]
[ -a21, -a22]
ans(:,:,3) =
[ b11, b12]
[ b21, b22]
ans(:,:,4) =
[ -b11, -b12]
[ -b21, -b22]
```

Input Arguments

`dim` — Dimension to concatenate arrays along

positive integer

Dimension to concatenate arrays along, specified as a positive integer.

`A1,...,AN` — Input arrays

symbolic variables | symbolic vectors | symbolic matrices | symbolic multidimensional arrays

Input arrays, specified as symbolic variables, vectors, matrices, or multidimensional arrays.

See Also

`horzcat` | `reshape` | `vertcat`

Introduced in R2010b

catalan

Catalan constant

Syntax

```
catalan
```

Description

`catalan` represents the Catalan constant on page 4-168. To get a floating-point approximation with the current precision set by `digits`, use `vpa(catalan)`.

Examples

Approximate Catalan Constant

Find a floating-point approximation of the Catalan constant with the default number of digits and with the 10-digit precision.

Use `vpa` to approximate the Catalan constant with the default 32-digit precision:

```
vpa(catalan)
```

```
ans =
0.91596559417721901505460351493238
```

Set the number of digits to 10 and approximate the Catalan constant:

```
old = digits(10);
vpa(catalan)
```

```
ans =
0.9159655942
```

Restore the default number of digits:

```
digits(old)
```

Definitions

Catalan Constant

The Catalan constant is defined as follows:

$$\text{catalan} = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)^2} = \frac{1}{1^2} - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \ldots$$

See Also

`dilog` | `eulergamma`

Introduced in R2014a

ccode

C code representation of symbolic expression

Syntax

```
ccode(f)
ccode(f,Name,Value)
```

Description

`ccode(f)` returns C code for the symbolic expression `f`.

`ccode(f,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Generate C Code from Symbolic Expression

Generate C code from the symbolic expression `log(1+x)`.

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

```
ans =
    '  t0 = log(x+1.0);'
```

Generate C code for the 3-by-3 Hilbert matrix.

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

```
ans =
    '  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;'
```

Initialize Arrays Efficiently

Because generated C code initializes only non-zero elements, you can efficiently initialize arrays by setting all elements to `0` directly in your C code. Then, use the generated C code to initialize only nonzero elements. This approach enables efficient initialization of matrices, especially sparse matrices.

Initialize the 3-by-3 identity matrix. First initialize the matrix with all elements set to `0` in your C code. Then use the generated C code to initialize the nonzero values.

```
I3 = sym(eye(3));
I3code = ccode(I3)
```

```
I3code =
    '  I3[0][0] = 1.0;
       I3[1][1] = 1.0;
       I3[2][2] = 1.0;'
```

Write Optimized C Code to File with Comments

Write C code to the file `ccodetest.c` by specifying the `File` option. When writing to a file, `ccode` optimizes the code by using intermediate variables named `t0`, `t1`, and so on.

```
syms x
f = diff(tan(x));
ccode(f,'File','ccodetest.c')
```

```
  t2 = tan(x);
  t0 = t2*t2+1.0;
```

Include the comment `Version: 1.1` in the file by using the `Comments` option. `ccode` uses block comments.

```
ccode(f,'File','ccodetest.c','Comments','Version: 1.1')
```

```
/*
Version: 1.1
*/
  t2 = tan(x);
  t0 = t2*t2+1.0;
```

Input Arguments

`f` — Symbolic input
symbolic expression

Symbolic input, specified as a symbolic expression.

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`.

Example: `ccode(x^2,'File','ccode.c','Comments','V1.2')`

`File` — File to write to
character vector | string

File to write to, specified as a character vector or string. When writing to a file, `ccode` optimizes the code by using intermediate variables named `t0`, `t1`, and so on.

`Comments` — Comments to include in file header
character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector. Because `ccode` uses block comments, the comments must not contain `/*` or `*/`.

See Also

`fortran` | `latex` | `matlabFunction`

Introduced before R2006a

ceil

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

`fix` | `floor` | `frac` | `round`

Introduced before R2006a

cell2sym

Convert cell array to symbolic array

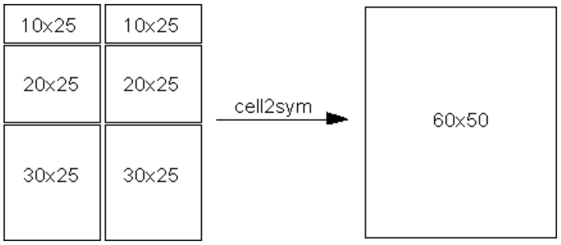
Syntax

```
S = cell2sym(C)
S = cell2sym(C,flag)
```

Description

`S = cell2sym(C)` converts a cell array `C` to a symbolic array `S`. The elements of `C` must be convertible to symbolic objects.

If each element of the input cell array `C` is a scalar, then `size(S) = size(C)`, and `S(k) = sym(C(k))` for all indices `k`. If the cell array `C` contains nonscalar elements, then the contents of `C` must support concatenation into an N-dimensional rectangle. Otherwise, the results are undefined. For example, the contents of cells in the same column must have the same number of columns. However, they do not need to have the same number of rows. See figure.



`S = cell2sym(C,flag)` uses the technique specified by flag for converting floating-point numbers to symbolic numbers.

Examples

Convert Cell Array of Scalars

Convert a cell array of only scalar elements to a symbolic array.

Create a cell array of scalar elements.

```
C = {'x','y','z'; 1 2 3}
```

```
C =
  2×3 cell array
    {'x'}    {'y'}    {'z'}
    {[1]}    {[2]}    {[3]}
```

Convert this cell array to a symbolic array.

```
S = cell2sym(C)
```

```
S =
[ x, y, z]
[ 1, 2, 3]
```

`cell2sym` does not create symbolic variables `x`, `y`, and `z` in the MATLAB workspace. To access an element of `S`, use parentheses.

```
S(1,1)
```

```
ans =
x
```

Convert Cell Array Containing Nonscalar Elements

Convert a cell array whose elements are scalars, vectors, and matrices into a symbolic array. Such conversion is possible only if the contents of the cell array can be concatenated into an N-dimensional rectangle.

Create a cell array, the elements of which are a scalar, a row vector, a column vector, and a matrix.

```
C = {'x' [2 3 4]; ['y'; sym(9)] [6 7 8; 10 11 12]}
```

```
C =
  2×2 cell array
```

```
    {'x'    }    {1×3 double}
    {2×1 sym}    {2×3 double}
```

Convert this cell array to a symbolic array.

```
S = cell2sym(C)
```

```
S =
[ x,  2,  3,  4]
[ y,  6,  7,  8]
[ 9, 10, 11, 12]
```

Choose Conversion Technique for Floating-Point Values

When converting a cell array containing floating-point numbers, you can explicitly specify the conversion technique.

Create a cell array `pi` with two elements: the double-precision value of the constant `pi` and the exact value `pi`.

```
C = {pi, sym(pi)}
```

```
C =
  1×2 cell array
    {[3.1416]}    {1×1 sym}
```

Convert this cell array to a symbolic array. By default, `cell2sym` uses the rational conversion mode. Thus, results returned by `cell2sym` without a flag are the same as results returned by `cell2sym` with the flag `'r'`.

```
S = cell2sym(C)
```

```
S =
[ pi, pi]
```

```
S = cell2sym(C,'r')
```

```
S =
[ pi, pi]
```

Convert the same cell array to a symbolic array using the flags `'d'`, `'e'`, and `'f'`. See the “Input Arguments” on page 4-177 section for the details about conversion techniques.

```
S = cell2sym(C,'d')
```

```
S =
[ 3.1415926535897931159979634685442, pi]

S = cell2sym(C,'e')

S =
[ pi - (198*eps)/359, pi]

S = cell2sym(C,'f')

S =
[ 884279719003555/281474976710656, pi]
```

Input Arguments

C — Input cell array

cell array

Input cell array, specified as a cell array. The elements of `C` must be convertible to symbolic objects.

`flag` — Conversion technique

`'r'` (default) | `'d'` | `'e'` | `'f'`

Conversion technique, specified as one of the characters listed in this table.

`'r'`	In the *rational* mode, `cell2sym` converts 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` to the corresponding symbolic form. This approach effectively compensates for the round-off error involved in the original evaluation, but might not represent the floating-point value precisely. If `cell2sym` cannot find simple rational approximation, then it uses the same technique as it would use with the flag `'f'`.
`'d'`	In the *decimal* mode, `cell2sym` takes the number of digits from the current setting of `digits`. Conversions with fewer than 16 digits lose some accuracy, while more than 16 digits might not be warranted. For example, `cell2sym({4/3},'d')` with the 10-digit accuracy returns `1.333333333`, while with the 20-digit accuracy it returns `1.3333333333333332593`. The latter does not end in `3`s, but it is an accurate decimal representation of the floating-point number nearest to `4/3`.

`'e'`	In the *estimate error* mode, `cell2sym` supplements a result obtained in the rational mode by a term involving the variable `eps`. This term estimates the difference between the theoretical rational expression and its actual floating-point value. For example, `cell2sym({3*pi/4},'e')` returns `(3*pi)/4 - (103*eps)/249`.
`'f'`	In the *floating-point* mode, `cell2sym` represents all values in the form `N*2^e` or `-N*2^e`, where `N >= 0` and `e` are integers. For example, `cell2sym({1/10},'f')` returns `3602879701896397/36028797018963968`. The returned rational value is the exact value of the floating-point number that you convert to a symbolic number.

Output Arguments

`s` — Resulting symbolic array

symbolic array

Resulting symbolic array, returned as a symbolic array.

See Also

`cell2mat` | `mat2cell` | `num2cell` | `sym2cell`

Introduced in R2016a

char

Convert symbolic objects to character vectors

Syntax

```
char(A)
```

Description

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

Input Arguments

`A`

Symbolic scalar or symbolic array.

Examples

Convert symbolic expressions to character vectors, and then concatenate the character vectors:

```
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 character vector.

Convert a symbolic matrix to a character vector:

```
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]])'
```

Tips

- `char` can change term ordering in an expression.

See Also

`double` | `pretty` | `sym`

Introduced before R2006a

charpoly

Characteristic polynomial of matrix

Syntax

```
charpoly(A)
charpoly(A,var)
```

Description

`charpoly(A)` returns a vector of the coefficients of the characteristic polynomial on page 4-182 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.

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
```

Definitions

Characteristic Polynomial of 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.

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`

Introduced in R2012b

chebyshevT

Chebyshev polynomials of the first kind

Syntax

```
chebyshevT(n,x)
```

Description

`chebyshevT(n,x)` represents the `n`th degree Chebyshev polynomial of the first kind on page 4-188 at the point `x`.

Examples

First Five Chebyshev Polynomials of the First Kind

Find the first five Chebyshev polynomials of the first kind for the variable `x`.

```
syms x
chebyshevT([0, 1, 2, 3, 4], x)
```

```
ans =
[ 1, x, 2*x^2 - 1, 4*x^3 - 3*x, 8*x^4 - 8*x^2 + 1]
```

Chebyshev Polynomials for Numeric and Symbolic Arguments

Depending on its arguments, `chebyshevT` returns floating-point or exact symbolic results.

Find the value of the fifth-degree Chebyshev polynomial of the first kind at these points. Because these numbers are not symbolic objects, `chebyshevT` returns floating-point results.

```
chebyshevT(5, [1/6, 1/4, 1/3, 1/2, 2/3, 3/4])
```

```
ans =
    0.7428    0.9531    0.9918    0.5000   -0.4856   -0.8906
```

Find the value of the fifth-degree Chebyshev polynomial of the first kind for the same numbers converted to symbolic objects. For symbolic numbers, `chebyshevT` returns exact symbolic results.

```
chebyshevT(5, sym([1/6, 1/4, 1/3, 1/2, 2/3, 3/4]))
```

```
ans =
[ 361/486, 61/64, 241/243, 1/2, -118/243, -57/64]
```

Evaluate Chebyshev Polynomials with Floating-Point Numbers

Floating-point evaluation of Chebyshev polynomials by direct calls of `chebyshevT` is numerically stable. However, first computing the polynomial using a symbolic variable, and then substituting variable-precision values into this expression can be numerically unstable.

Find the value of the 500th-degree Chebyshev polynomial of the first kind at `1/3` and `vpa(1/3)`. Floating-point evaluation is numerically stable.

```
chebyshevT(500, 1/3)
chebyshevT(500, vpa(1/3))
```

```
ans =
    0.9631

ans =
0.963114126817085233778571286718
```

Now, find the symbolic polynomial `T500 = chebyshevT(500, x)`, and substitute `x = vpa(1/3)` into the result. This approach is numerically unstable.

```
syms x
T500 = chebyshevT(500, x);
subs(T500, x, vpa(1/3))
```

```
ans =
-3293905791337500897482813472768.0
```

Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

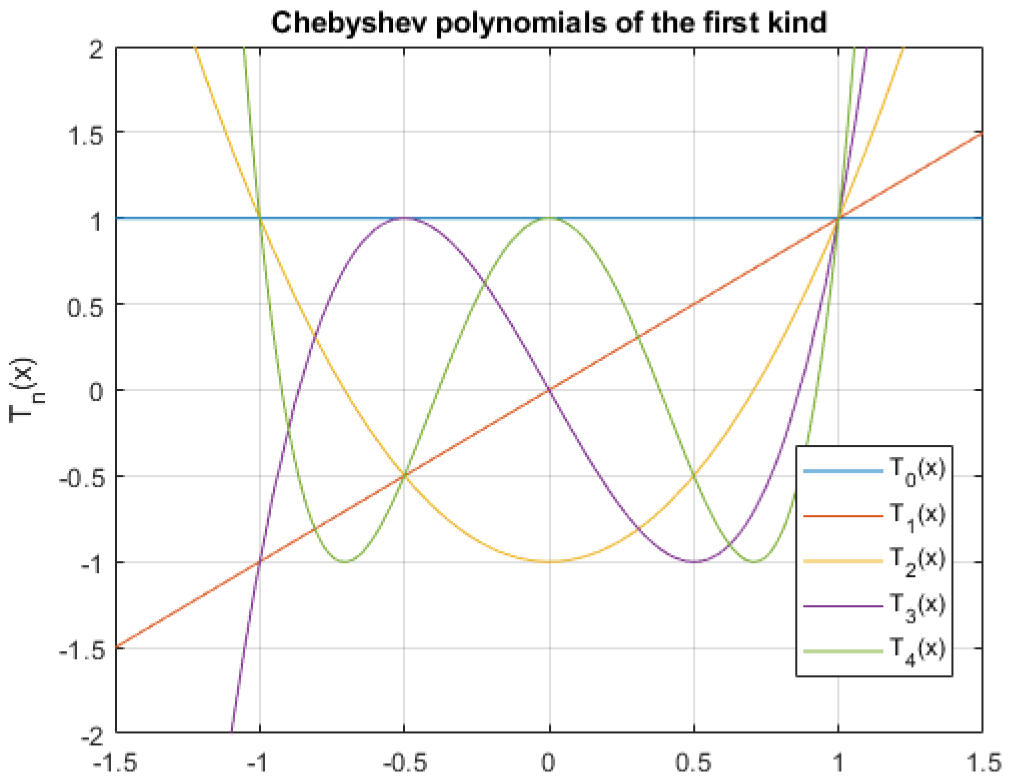
```
subs(vpa(T500), x, sym(1/3))
```

```
ans =
1202292431349342132757038366720.0
```

Plot Chebyshev Polynomials of the First Kind

Plot the first five Chebyshev polynomials of the first kind. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(chebyshevT(0:4, x))
axis([-1.5 1.5 -2 2])
grid on

ylabel('T_n(x)')
legend('T_0(x)', 'T_1(x)', 'T_2(x)', 'T_3(x)', 'T_4(x)', 'Location', 'Best')
title('Chebyshev polynomials of the first kind')
```



Input Arguments

n — Degree of polynomial

nonnegative integer | symbolic variable | symbolic expression | symbolic function | vector | matrix

Degree of the polynomial, specified as a nonnegative integer, symbolic variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

x — Evaluation point
number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Evaluation point, specified as a number, symbolic number, variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Chebyshev Polynomials of the First Kind

Chebyshev polynomials of the first kind are defined as $T_n(x) = \cos(n*\arccos(x))$.

These polynomials satisfy the recursion formula

$$T(0,x)=1, \quad T(1,x)=x, \quad T(n,x)=2xT(n-1,x)-T(n-2,x)$$

Chebyshev polynomials of the first kind are orthogonal on the interval $-1 \le x \le 1$ with respect to the weight function

$$w(x)=\frac{1}{\sqrt{1-x^2}}$$

Chebyshev polynomials of the first kind are a special case of the Jacobi polynomials

$$T(n,x)=\frac{2^{2n}(n!)^2}{(2n)!}P\left(n,-\frac{1}{2},-\frac{1}{2},x\right)$$

and Gegenbauer polynomials

$$T(n,x)=\frac{n}{2}G(n,0,x)$$

Tips

- `chebyshevT` returns floating-point results for numeric arguments that are not symbolic objects.

- `chebyshevT` acts element-wise on nonscalar inputs.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, then `chebyshevT` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

References

[1] Hochstrasser, U. W. “Orthogonal Polynomials.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`chebyshevU` | `gegenbauerC` | `hermiteH` | `jacobiP` | `laguerreL` | `legendreP`

Introduced in R2014b

chebyshevU

Chebyshev polynomials of the second kind

Syntax

```
chebyshevU(n,x)
```

Description

`chebyshevU(n,x)` represents the `n`th degree Chebyshev polynomial of the second kind on page 4-194 at the point `x`.

Examples

First Five Chebyshev Polynomials of the Second Kind

Find the first five Chebyshev polynomials of the second kind for the variable `x`.

```
syms x
chebyshevU([0, 1, 2, 3, 4], x)
```

```
ans =
[ 1, 2*x, 4*x^2 - 1, 8*x^3 - 4*x, 16*x^4 - 12*x^2 + 1]
```

Chebyshev Polynomials for Numeric and Symbolic Arguments

Depending on its arguments, `chebyshevU` returns floating-point or exact symbolic results.

Find the value of the fifth-degree Chebyshev polynomial of the second kind at these points. Because these numbers are not symbolic objects, `chebyshevU` returns floating-point results.

```
chebyshevU(5, [1/6, 1/3, 1/2, 2/3, 4/5])
```

```
ans =
    0.8560    0.9465    0.0000   -1.2675   -1.0982
```

Find the value of the fifth-degree Chebyshev polynomial of the second kind for the same numbers converted to symbolic objects. For symbolic numbers, `chebyshevU` returns exact symbolic results.

```
chebyshevU(5, sym([1/6, 1/4, 1/3, 1/2, 2/3, 4/5]))
```

```
ans =
[ 208/243, 33/32, 230/243, 0, -308/243, -3432/3125]
```

Evaluate Chebyshev Polynomials with Floating-Point Numbers

Floating-point evaluation of Chebyshev polynomials by direct calls of `chebyshevU` is numerically stable. However, first computing the polynomial using a symbolic variable, and then substituting variable-precision values into this expression can be numerically unstable.

Find the value of the 500th-degree Chebyshev polynomial of the second kind at `1/3` and `vpa(1/3)`. Floating-point evaluation is numerically stable.

```
chebyshevU(500, 1/3)
chebyshevU(500, vpa(1/3))
```

```
ans =
    0.8680

ans =
0.86797529488884242798157148968078
```

Now, find the symbolic polynomial `U500 = chebyshevU(500, x)`, and substitute `x = vpa(1/3)` into the result. This approach is numerically unstable.

```
syms x
U500 = chebyshevU(500, x);
subs(U500, x, vpa(1/3))
```

```
ans =
63080680195950160912110845952.0
```

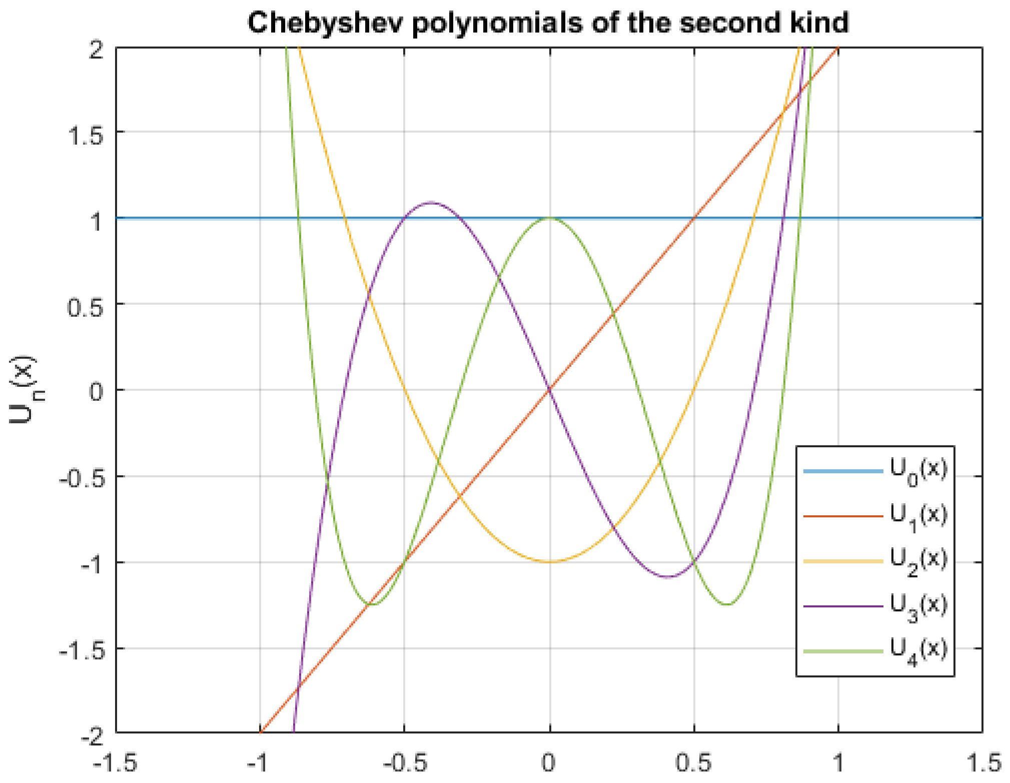
Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

```
subs(vpa(U500), x, sym(1/3))

ans =
-1878009301399851172833781612544.0
```

Plot Chebyshev Polynomials of the Second Kind

Plot the first five Chebyshev polynomials of the second kind. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(chebyshevU(0:4, x))
axis([-1.5 1.5 -2 2])
grid on

ylabel('U_n(x)')
legend('U_0(x)', 'U_1(x)', 'U_2(x)', 'U_3(x)', 'U_4(x)', 'Location', 'Best')
title('Chebyshev polynomials of the second kind')
```



Input Arguments

n — Degree of polynomial

nonnegative integer | symbolic variable | symbolic expression | symbolic function | vector | matrix

Degree of the polynomial, specified as a nonnegative integer, symbolic variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

x — Evaluation point
number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Evaluation point, specified as a number, symbolic number, variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Chebyshev Polynomials of the Second Kind

Chebyshev polynomials of the second kind are defined as follows:

$$U(n,x) = \frac{\sin\left((n+1)a\cos(x)\right)}{\sin\left(a\cos(x)\right)}$$

These polynomials satisfy the recursion formula

$$U(0,x) = 1, \quad U(1,x) = 2x, \quad U(n,x) = 2xU(n-1,x) - U(n-2,x)$$

Chebyshev polynomials of the second kind are orthogonal on the interval $-1 \le x \le 1$ with respect to the weight function

$$w(x) = \sqrt{1-x^2}$$

Chebyshev polynomials of the second kind are a special case of the Jacobi polynomials

$$U(n,x) = \frac{2^{2n}n!(n+1)!}{(2n+1)!}P\left(n,\frac{1}{2},\frac{1}{2},x\right)$$

and Gegenbauer polynomials

$$U(n,x) = G(n,1,x)$$

Tips

- `chebyshevU` returns floating-point results for numeric arguments that are not symbolic objects.
- `chebyshevU` acts element-wise on nonscalar inputs.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, then `chebyshevU` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

References

[1] Hochstrasser, U. W. "Orthogonal Polynomials." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`chebyshevT` | `gegenbauerC` | `hermiteH` | `jacobiP` | `laguerreL` | `legendreP`

Introduced in R2014b

checkUnits

Check for compatible dimensions and consistent units

Syntax

```
C = checkUnits(expr)
C = checkUnits(expr,'Compatible')
C = checkUnits(expr,'Consistent')
```

Description

`C = checkUnits(expr)` checks `expr` for compatible dimensions and consistent units and returns a structure containing the fields `Consistent` and `Compatible`. The fields contain logical `0` (`false`) or logical `1` (`true`) depending on the check results.

`expr` has compatible dimensions if all terms have the same dimensions, such as length or time. `expr` has consistent units if all units of the same dimension can be converted to each other with a conversion factor of 1.

`C = checkUnits(expr,'Compatible')` only checks `expr` for compatible dimensions.

`C = checkUnits(expr,'Consistent')` only checks `expr` for consistent units.

Examples

Check Dimensions of Units

Check the dimensions of an equation or expression. The dimensions are checked to confirm that the equation or expression is valid.

Verify the dimensions of the equation

$$A\ \frac{\mathrm{m}}{\mathrm{s}} = B\ \frac{\mathrm{kg}}{\mathrm{s}}$$

by using checkUnits with the option 'Compatible'. MATLAB assumes that symbolic variables are dimensionless. The checkUnits function returns logical 0 (false) because the dimensions of the equation are not compatible.

```
u = symunit;
syms A B
eqn = A*u.m/u.s == B*u.kg/u.s;
checkUnits(eqn,'Compatible')

ans =
  logical
   0
```

Replace u.kg with u.m by using subs and repeat the check. Because the dimensions are now compatible, checkUnits returns logical 1 (true).

```
eqn = subs(eqn,u.kg,u.m);
checkUnits(eqn,'Compatible')

ans =
  logical
   1
```

Check Consistency of Units

Checking units for consistency is a stronger check than compatibility. Units are consistent when all units of the same dimension can be converted to each other with a conversion factor of 1. For example, 1 Newton is consistent with 1 kg m/s^2 but not with 1 kg cm/s^2.

Show that 1 Newton is consistent with 1 kg m/s^2 by checking expr1 but not with 1 kg cm/s^2 by checking expr2.

```
u = symunit;
expr1 = 1*u.N + 1*u.kg*u.m/u.s^2;
expr2 = 1*u.N + 1*u.kg*u.cm/u.s^2;
checkUnits(expr1,'Consistent')

ans =
  logical
   1

checkUnits(expr2,'Consistent')
```

```
ans =
  logical
   0
```

Show the difference between compatibility and consistency by showing that `expr2` has compatible dimensions but not consistent units.

```
checkUnits(expr2,'Compatible')
```

```
ans =
  logical
   1
```

Check Multiple Equations or Expressions

Check multiple equations or expressions by placing them in an array. `checkUnits` returns an array whose elements correspond to the elements of the input.

Check multiple equations for compatible dimensions. `checkUnits` returns `[1 0]`, meaning that the first equation has compatible dimensions while the second equation does not.

```
u = symunit;
syms x y z
eqn1 = x*u.m == y*u.m^2/(z*u.m);
eqn2 = x*u.m + y*u.s == z*u.m;
eqns = [eqn1 eqn2];
compatible = checkUnits(eqns,'Compatible')
```

```
compatible =
  1×2 logical array
   1   0
```

Check Dimensions and Consistency of Units

Check for both compatible dimensions and consistent units of the equation or expression by using `checkUnits`.

Define the equations for x- and y-displacement of a moving projectile. Check their units for compatibility and consistency.

```
u = symunit;
g = 9.81*u.cm/u.s^2;
```

```
v = 10*u.m/u.s^2;
syms theta x(t) y(t)
x(t) = v*cos(theta)*t;
y(t) = v*sin(theta)*t + (-g*t^2)/2;
S = checkUnits([x y])

S = 
  struct with fields:

    Consistent: [1 0]
    Compatible: [1 1]
```

The second equation has compatible dimensions but inconsistent units. This inconsistency is because `g` incorrectly uses cm instead of m. Redefine `g` and check the equations again. The second equation now has consistent units.

```
g = 9.81*u.m/u.s^2;
y(t) = v*sin(theta)*t + (-g*t^2)/2;
S = checkUnits([x y])

S = 
  struct with fields:

    Consistent: [1 1]
    Compatible: [1 1]
```

Input Arguments

`expr` — Input expression

symbolic expression | symbolic equation | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Input expression, specified as a symbolic expression, equation, function, vector, matrix, or multidimensional array.

See Also

`findUnits` | `isUnit` | `newUnit` | `separateUnits` | `str2symunit` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

Introduced in R2017a

children

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
```

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 =
  2×2 cell array
    {1×2 sym}    {1×2 sym}
    {1×2 sym}    {1×1 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` | `lhs` | `numden` | `rhs` | `subs`

Topics

“Create Symbolic Numbers, Variables, and Expressions” on page 1-3

Introduced in R2012a

chol

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 on page 4-211. Otherwise, this syntax throws an error.

`[T,p] = chol(A)` computes the Cholesky factorization on page 4-211 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 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.

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.

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. To make `isAlways` return logical `0` (`false`) for undecidable conditions, set `Unknown` to `false`.

```
syms a
A = [a 0; 0 a];
isAlways(A == A','Unknown','false')
```

```
ans =
  2×2 logical array
     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. To make `isAlways` return logical `0` (`false`) for undecidable conditions, set `Unknown` to `false`.

```
T = chol(sym([1 1; 2 1]), 'nocheck')
```

```
T =
[ 1,         2]
[ 0, 3^(1/2)*1i]

isAlways(A == T'*T,'Unknown','false')

ans =
  2×2 logical array
     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`. To make `isAlways` return logical `0` (`false`) for undecidable conditions, set `Unknown` to `false`.

```
isAlways(A == T.'*T)

ans =
  2×2 logical array
     1     1
     1     1

isAlways(A == T'*T,'Unknown','false')

ans =
  2×2 logical array
```

```
     0         0
     0         0
```

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`.

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`.

See Also

`chol` | `ctranspose` | `eig` | `isAlways` | `lu` | `qr` | `svd` | `transpose` | `vpa`

Introduced in R2013a

clear all

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`

Introduced in R2008b

close

Close MuPAD notebook

Syntax

```
close(nb)
close(nb,'force')
```

Description

`close(nb)` closes the MuPAD notebook with the handle `nb`. If you modified the notebook, `close(nb)` brings up a dialog box asking if you want to save the changes.

`close(nb,'force')` closes notebook `nb` without prompting you to save the changes. If you modified the notebook, `close(nb,'force')` discards the changes.

This syntax can be helpful when you evaluate MuPAD notebooks by using `evaluateMuPADNotebook`. When you evaluate a notebook, MuPAD inserts results in the output regions or at least inserts the new input region at the bottom of the notebook, thus modifying the notebook. If you want to close the notebook quickly without saving such changes, use `close(nb,'force')`.

Examples

Close One Notebook

Open and close an existing notebook.

Suppose that your current folder contains a MuPAD notebook named `myFile1.mn`. Open this notebook keeping its handle in the variable `nb1`:

```
nb1 = mupad('myFile1.mn');
```

Suppose that you finished using this notebook and now want to close it. Enter this command in the MATLAB Command Window. If you have unsaved changes in that notebook, then this command will bring up a dialog box asking if you want to save the changes.

```
close(nb1)
```

Close Several Notebooks

Use a vector of notebook handles to close several notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Close `myFile1.mn` and `myFile2.mn`. If you have unsaved changes in any of these two notebooks, then this command will bring up a dialog box asking if you want to save the changes.

```
close([nb1, nb2])
```

Close All Open Notebooks

Identify and close all currently open MuPAD notebooks.

Get a list of all currently open notebooks:

```
allNBs = allMuPADNotebooks;
```

Close all notebooks. If you have unsaved changes in any notebook, then this command will bring up a dialog box asking if you want to save the changes.

```
close(allNBs)
```

Close All Open Notebooks and Discard Modifications

Identify and close all currently open MuPAD notebooks without saving changes.

Get a list of all currently open notebooks:

```
allNBs = allMuPADNotebooks;
```

Close all notebooks using the `force` flag to suppress the dialog box that offers you to save changes:

```
close(allNBs,'force')
```

- “Create MuPAD Notebooks” on page 3-3
- “Open MuPAD Notebooks” on page 3-6
- “Save MuPAD Notebooks” on page 3-12
- “Evaluate MuPAD Notebooks from MATLAB” on page 3-13
- “Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52
- “Close MuPAD Notebooks from MATLAB” on page 3-17

Input Arguments

`nb` — Pointer to MuPAD notebook
handle to notebook | vector of handles to notebooks

Pointer to notebook, specified as a MuPAD notebook handle or a vector of handles. You create the notebook handle when opening a notebook with the `mupad` or `openmn` function.

You can get the list of all open notebooks using the `allMuPADNotebooks` function. `close` accepts a vector of handles returned by `allMuPADNotebooks`.

See Also

`allMuPADNotebooks` | `evaluateMuPADNotebook` | `getVar` | `mupad` | `mupadNotebookTitle` | `openmn` | `setVar`

Topics

“Create MuPAD Notebooks” on page 3-3
“Open MuPAD Notebooks” on page 3-6
“Save MuPAD Notebooks” on page 3-12
“Evaluate MuPAD Notebooks from MATLAB” on page 3-13
“Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52
“Close MuPAD Notebooks from MATLAB” on page 3-17

Introduced in R2013b

coeffs

Coefficients of polynomial

Syntax

```
C = coeffs(p)
C = coeffs(p,var)
C = coeffs(p,vars)
[C,T] = coeffs( ___ )
 ___  = coeffs( ___ ,'All')
```

Description

`C = coeffs(p)` returns coefficients of the polynomial `p` with respect to all variables determined in `p` by `symvar`.

`C = coeffs(p,var)` returns coefficients of the polynomial `p` with respect to the variable `var`.

`C = coeffs(p,vars)` returns coefficients of the multivariate polynomial `p` with respect to the variables `vars`.

`[C,T] = coeffs( ___ )` returns the coefficient `C` and the corresponding terms `T` of the polynomial `p`.

` ___  = coeffs( ___ ,'All')` returns all coefficients, including coefficients that are 0. For example, `coeffs(2*x^2,'All')` returns `[ 2, 0, 0]` instead of `2`.

Examples

Coefficients of Univariate Polynomial

Find the coefficients of this univariate polynomial. The coefficients are ordered from the lowest degree to the highest degree.

```
syms x
c = coeffs(16*x^2 + 19*x + 11)

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

Reverse the ordering of coefficients by using `fliplr`.

```
c = fliplr(c)

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

Coefficients of Multivariate Polynomial with Respect to Particular Variable

Find the coefficients of this polynomial with respect to variable `x` and variable `y`.

```
syms x y
cx = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, x)
cy = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, y)

cx =
[ 4*y^3, 3*y^2, 2*y, 1]

cy =
[ x^3, 2*x^2, 3*x, 4]
```

Coefficients of Multivariate Polynomial with Respect to Two Variables

Find the coefficients of this polynomial with respect to both variables `x` and `y`.

```
syms x y
cxy = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, [x y])
cyx = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, [y x])

cxy =
[ 4, 3, 2, 1]

cyx =
[ 1, 2, 3, 4]
```

Coefficients and Corresponding Terms of Univariate Polynomial

Find the coefficients and the corresponding terms of this univariate polynomial. When two outputs are provided, the coefficients are ordered from the highest degree to the lowest degree.

```
syms x
[c,t] = coeffs(16*x^2 + 19*x + 11)
```

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

t =
[ x^2, x, 1]
```

Coefficients and Corresponding Terms of Multivariate Polynomial

Find the coefficients and the corresponding terms of this polynomial with respect to variable `x` and variable `y`.

```
syms x y
[cx,tx] = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, x)
[cy,ty] = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, y)
```

```
cx =
[ 1, 2*y, 3*y^2, 4*y^3]

tx =
[ x^3, x^2, x, 1]

cy =
[ 4, 3*x, 2*x^2, x^3]

ty =
[ y^3, y^2, y, 1]
```

Find the coefficients of this polynomial with respect to both variables `x` and `y`.

```
syms x y
[cxy, txy] = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, [x,y])
[cyx, tyx] = coeffs(x^3 + 2*x^2*y + 3*x*y^2 + 4*y^3, [y,x])
```

```
cxy =
[ 1, 2, 3, 4]
```

```
txy =
[ x^3, x^2*y, x*y^2, y^3]

cyx =
[ 4, 3, 2, 1]

tyx =
[ y^3, x*y^2, x^2*y, x^3]
```

All Coefficients of Polynomial

Find all coefficients of a polynomial, including coefficients that are `0`, by specifying the option `'All'`.

Find all coefficients of $3x^2$.

```
syms x
c = coeffs(3*x^2, 'All')
```

```
c =
[ 3, 0, 0]
```

If you find coefficients with respect to multiple variables and specify `'All'`, then `coeffs` returns coefficients for all combinations of the variables.

Find all coefficients and corresponding terms of $ax^2 + by$.

```
syms a b y
[cxy, txy] = coeffs(a*x^2 + b*y, [y x], 'All')
```

```
cxy =
[ 0, 0, b]
[ a, 0, 0]
txy =
[ x^2*y, x*y, y]
[   x^2,   x, 1]
```

Input Arguments

`p` — Polynomial

symbolic expression | symbolic function

Polynomial, specified as a symbolic expression or function.

`var` — Polynomial variable
symbolic variable

Polynomial variable, specified as a symbolic variable.

`vars` — Polynomial variables
vector of symbolic variables

Polynomial variables, specified as a vector of symbolic variables.

Output Arguments

`C` — Coefficients of polynomial
symbolic number | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic multidimensional array

Coefficients of polynomial, returned as a symbolic number, variable, expression, vector, matrix, or multidimensional array. If there is only one coefficient and one corresponding term, then `C` is returned as a scalar.

`T` — Terms of polynomial
symbolic number | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic multidimensional array

Terms of polynomial, returned as a symbolic number, variable, expression, vector, matrix, or multidimensional array. If there is only one coefficient and one corresponding term, then `T` is returned as a scalar.

See Also

`poly2sym` | `sym2poly`

Introduced before R2006a

collect

Collect coefficients

Syntax

```
collect(P)
collect(P,expr)
```

Description

`collect(P)` collects coefficients in `P` of the powers of the default variable of `P`. The default variable is found by `symvar`.

`collect(P,expr)` collects coefficients in `P` of the powers of the symbolic expression `expr`. If `P` is a vector or matrix, then `collect` acts element-wise on `P`. If `expr` is a vector, then `collect` finds coefficients in terms of all expressions in `expr`.

Examples

Collect Coefficients of Powers of Default Variable

Collect the coefficients of a symbolic expression.

```
syms x
coeffs = collect((exp(x) + x)*(x + 2))
```

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

Because you did not specify the variable, `collect` uses the default variable defined by `symvar`. For this expression, the default variable is `x`.

```
symvar((exp(x) + x)*(x + 2), 1)
```

```
ans =
x
```

Collect Coefficients of Powers of a Particular Variable

Collect coefficients of a particular variable by specifying the variable as the second argument to `collect`.

Collect coefficients of an expression in powers of `x`, and then in powers of `y`.

```
syms x y
coeffs_x = collect(x^2*y + y*x - x^2 - 2*x, x)
coeffs_y = collect(x^2*y + y*x - x^2 - 2*x, y)
```

```
coeffs_x =
(y - 1)*x^2 + (y - 2)*x
coeffs_y =
(x^2 + x)*y - x^2 - 2*x
```

Collect coefficients in powers of both `x` and `y` by specifying the second argument as a vector of variables.

```
syms a b
coeffs_xy = collect(a^2*x*y + a*b*x^2 + a*x*y + x^2, [x y])
```

```
coeffs_xy =
(a*b + 1)*x^2 + (a^2 + a)*x*y
```

Collect Coefficients in Terms of `i` and `pi`

Collect coefficients of an expression in terms of `i`, and then in terms of `pi`.

```
syms x y
coeffs_i = collect(2*x*i - 3*i*y, i)
coeffs_pi = collect(x*pi*(pi - y) + x*(pi + i) + 3*pi*y, pi)
```

```
coeffs_i =
(2*x - 3*y)*1i
coeffs_pi =
x*pi^2 + (x + 3*y - x*y)*pi + x*1i
```

Collect Coefficients of Symbolic Expressions and Functions

Collect coefficients of expressions and functions by specifying the second argument as an expression or function. Collect coefficients of multiple expressions or functions by using vector input.

Expand `sin(x + 3*y)` and collect coefficients of `cos(y)`, and then of both `sin(x)` and `sin(y)`.

```
syms x y
f = expand(sin(x + 3*y));
coeffs_cosy = collect(f, cos(y))
```

```
coeffs_cosy =
4*sin(x)*cos(y)^3 + 4*cos(x)*sin(y)*cos(y)^2 + (-3*sin(x))*cos(y) - cos(x)*sin(y)
```

```
coeffs_sinxsiny = collect(f, [sin(x) sin(y)])
```

```
coeffs_sinxsiny =
(4*cos(y)^3 - 3*cos(y))*sin(x) + (4*cos(x)*cos(y)^2 - cos(x))*sin(y)
```

Collect coefficients of the symbolic function `y(x)` in a symbolic expression.

```
syms y(x)
f = y^2*x + y*x^2 + y*sin(x) + x*y;
coeffs_y = collect(f, y)
```

```
coeffs_y(x) =
x*y(x)^2 + (x + sin(x) + x^2)*y(x)
```

Collect Coefficients for Each Element of Matrix

Call `collect` on a matrix. `collect` acts element-wise on the matrix.

```
syms x y
collect([(x + 1)*(y + 1), x^2 + x*(x -y); 2*x*y - x, x*y + x/y], x)
```

```
ans =
[ (y + 1)*x + y + 1, 2*x^2 - y*x]
[       (2*y - 1)*x, (y + 1/y)*x]
```

Collect Coefficients of Function Calls

Collect coefficients of calls to a particular function by specifying the function name as the second argument. Collect coefficients of function calls with respect to multiple functions by specifying the multiple functions as a cell array of character vectors.

Collect coefficients of calls to the `sin` function in `F`, where `F` contains multiple calls to different functions.

```
syms a b c d e f x
F = a*sin(2*x) + b*sin(2*x) + c*cos(x) + d*cos(x) + e*sin(3*x) +f*sin(3*x);
collect(F, 'sin')

ans =
(a + b)*sin(2*x) + (e + f)*sin(3*x) + c*cos(x) + d*cos(x)
```

Collect coefficients of calls to both the `sin` and `cos` functions in `F`.

```
collect(F, {'sin' 'cos'})

ans =
(c + d)*cos(x) + (a + b)*sin(2*x) + (e + f)*sin(3*x)
```

Input Arguments

`P` — Input expression

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input expression, specified as a symbolic expression, function, vector, or matrix.

`expr` — Expression in terms of which you collect coefficients

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | character vector | cell array of character vectors

Expression in terms of which you collect the coefficients, specified as a symbolic number, variable, expression, function, or vector; a character vector; a cell array of character vectors.

Example: `i`, `pi`, `x`, `sin(x)`, `y(x)`, `[sin(x) cos(y)]`, `{'sin' 'cos'}`.

See Also

`combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction` | `symvar`

Introduced before R2006a

colon:

Create symbolic vectors, array subscripting, and `for`-loop iterators

Syntax

```
m:n
m:d:n
x:x+r
x:d:x+r
```

Description

`m:n` returns a symbolic vector of values `[m,m+1,...,n]`, where `m` and `n` are symbolic constants. If `n` is not an increment of `m`, then the last value of the vector stops before `n`. This behavior holds for all syntaxes.

`m:d:n` returns a symbolic vector of values `[m,m+d,...,n]`, where `d` is a rational number.

`x:x+r` returns a symbolic vector of values `[x,x+1,...,x+r]`, where `x` is a symbolic variable and `r` is a rational number.

`x:d:x+r` returns a symbolic vector of values `[x,x+d,...,x+r]`, where `d` and `r` are rational numbers.

Examples

Create Numeric and Symbolic Arrays

Use the colon operator to create numeric and symbolic arrays. Because these inputs are not symbolic objects, you receive floating-point results.

```
1/2:7/2
```

```
ans =
    0.5000    1.5000    2.5000    3.5000
```

To obtain symbolic results, convert the inputs to symbolic objects.

```
sym(1/2):sym(7/2)

ans =
[ 1/2, 3/2, 5/2, 7/2]
```

Specify the increment used.

```
sym(1/2):2/3:sym(7/2)

ans =
[ 1/2, 7/6, 11/6, 5/2, 19/6]
```

Obtain Increments of Symbolic Variable

```
syms x
x:x+2

ans =
[ x, x + 1, x + 2]
```

Specify the increment used.

```
syms x
x:3/7:x+2

ans =
[ x, x + 3/7, x + 6/7, x + 9/7, x + 12/7]
```

Obtain increments between `x` and `2*x` in intervals of `x/3`.

```
syms x
x:x/3:2*x

ans =
[ x, (4*x)/3, (5*x)/3, 2*x]
```

Find Product of Harmonic Series

Find the product of the first four terms of the harmonic series.

```
syms x
p = sym(1);
for i = x:x+3
    p = p*1/i;
end
p

p =
1/(x*(x + 1)*(x + 2)*(x + 3))
```

Use `expand` to obtain the full polynomial.

```
expand(p)

ans =
1/(x^4 + 6*x^3 + 11*x^2 + 6*x)
```

Use `subs` to replace `x` with `1` and find the product in fractions.

```
p = subs(p,x,1)

p =
1/24
```

Use `vpa` to return the result as a floating-point value.

```
vpa(p)

ans =
0.041666666666666666666666666666667
```

You can also perform the described operations in a single line of code.

```
vpa(subs( expand(prod(1./(x:x+3))) ,x,1))

ans =
0.041666666666666666666666666666667
```

Input Arguments

`m` — Input

symbolic constant

Input, specified as a symbolic constant.

`n` — Input

symbolic constant

Input, specified as a symbolic constant.

`x` — Input

symbolic variable

Input, specified as a symbolic variable.

`r` — Upper bound on vector values

symbolic rational

Upper bound on vector values, specified as a symbolic rational. For example, `x:x+2` returns `[ x, x + 1, x + 2]`.

`d` — Increment in vector values

symbolic rational

Increment in vector values, specified as a symbolic rational. For example, `x:1/2:x+2` returns `[ x, x + 1/2, x + 1, x + 3/2, x + 2]`.

See Also

`reshape`

Introduced before R2006a

colspace

Column space of matrix

Syntax

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

Description

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

Examples

Find the basis for the column space of this symbolic matrix.

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

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

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

See Also

`null` | `size`

Introduced before R2006a

combine

Combine terms of identical algebraic structure

Syntax

```
Y = combine(S)
Y = combine(S,T)
Y = combine(___,Name,Value)
```

Description

`Y = combine(S)` rewrites products of powers in the expression `S` as a single power.

`Y = combine(S,T)` combines multiple calls to the target function `T` in the expression `S`. Use `combine` to implement the inverse functionality of `expand` with respect to the majority of the applied rules.

`Y = combine(___,Name,Value)` calls `combine` using additional options specified by one or more `Name,Value` pair arguments.

Examples

Powers of the Same Base

Combine powers of the same base.

```
syms x y z
combine(x^y*x^z)
```

```
ans =
x^(y + z)
```

Combine powers of numeric arguments. To prevent MATLAB from evaluating the expression, use `sym` to convert at least one numeric argument into a symbolic value.

```
syms x y
combine(x^(3)*x^y*x^exp(sym(1)))

ans =
x^(y + exp(1) + 3)
```

Here, `sym` converts `1` into a symbolic value, preventing MATLAB from evaluating the expression e^1.

Powers of the Same Exponent

Combine powers with the same exponents in certain cases.

```
combine(sqrt(sym(2))*sqrt(3))

ans =
6^(1/2)
```

`combine` does not usually combine the powers because the internal simplifier applies the same rules in the opposite direction to expand the result.

```
syms x y
combine(y^5*x^5)

ans =
x^5*y^5
```

Terms with Logarithms

Combine terms with logarithms by specifying the target argument as `log`. For real positive numbers, the logarithm of a product equals the sum of the logarithms of its factors.

```
S = log(sym(2)) + log(sym(3));
combine(S,'log')

ans =
log(6)
```

Try combining `log(a) + log(b)`. Because `a` and `b` are assumed to be complex numbers by default, the rule does not hold and `combine` does not combine the terms.

```
syms a b
S = log(a) + log(b);
combine(S,'log')
```

```
ans =
log(a) + log(b)
```

Apply the rule by setting assumptions such that `a` and `b` satisfy the conditions for the rule.

```
assume(a > 0)
assume(b > 0)
S = log(a) + log(b);
combine(S,'log')
```

```
ans =
log(a*b)
```

For future computations, clear the assumptions set on variables a and b.

```
syms a clear
syms b clear
```

Alternatively, apply the rule by ignoring analytic constraints using `IgnoreAnalyticConstraints`.

```
syms a b
S = log(a) + log(b);
combine(S,'log','IgnoreAnalyticConstraints',true)
```

```
ans =
 log(a*b)
```

Terms with Sine and Cosine Function Calls

Rewrite products of sine and cosine functions as a sum of the functions by setting the target argument to `sincos`.

```
syms a b
combine(sin(a)*cos(b) + sin(b)^2,'sincos')
```

```
ans =
sin(a + b)/2 - cos(2*b)/2 + sin(a - b)/2 + 1/2
```

Rewrite sums of sine and cosine functions by setting the target argument to `sincos`.

```
combine(cos(a) + sin(a),'sincos')

ans =
2^(1/2)*cos(a - pi/4)
```

`combine` does not rewrite powers of sine or cosine functions with negative integer exponents.

```
syms a b
combine(sin(b)^(-2)*cos(b)^(-2),'sincos')

ans =
1/(cos(b)^2*sin(b)^2)
```

Exponential Terms

Combine terms with exponents by specifying the target argument as `exp`.

```
combine(exp(sym(3))*exp(sym(2)),'exp')

ans =
exp(5)

syms a
combine(exp(a)^3, 'exp')

ans =
exp(3*a)
```

Terms with Integrals

Combine terms with integrals by specifying the target argument as `int`.

```
syms a f(x) g(x)
combine(int(f(x),x)+int(g(x),x),'int')
combine(a*int(f(x),x),'int')

ans =
int(f(x) + g(x), x)
ans =
int(a*f(x), x)
```

Combine integrals with the same limits.

```
syms a b h(z)
combine(int(f(x),x,a,b)+int(h(z),z,a,b),'int')

ans =
int(f(x) + h(x), x, a, b)
```

Terms with Inverse Tangent Function Calls

Combine two calls to the inverse tangent function by specifying the target argument as `atan`.

```
syms a b
assume(-1 < a < 1)
assume(-1 < b < 1)
combine(atan(a) + atan(b),'atan')

ans =
-atan((a + b)/(a*b - 1))
```

Combine two calls to the inverse tangent function. `combine` simplifies the expression to a symbolic value if possible.

```
assume(a > 0)
combine(atan(a) + atan(1/a),'atan')

ans =
pi/2
```

For further computations, clear the assumptions:

```
syms a clear
syms b clear
```

Terms with Calls to Gamma Function

Combine multiple gamma functions by specifying the target as `gamma`.

```
syms x
combine(gamma(x)*gamma(1-x),'gamma')

ans =
 -pi/sin(pi*(x - 1))
```

`combine` simplifies quotients of gamma functions to rational expressions.

Multiple Input Expressions in One Call

Evaluate multiple expressions in one function call by using a symbolic matrix as the input parameter.

```
S = [sqrt(sym(2))*sqrt(5), sqrt(2)*sqrt(sym(11))];
combine(S)
```

```
ans =
[ 10^(1/2), 22^(1/2)]
```

Input Arguments

S — Input expression

symbolic expression | symbolic vector | symbolic matrix | symbolic function

Input expression, specified as a symbolic expression, function, or as a vector or matrix of symbolic expressions or functions.

`combine` works recursively on subexpressions of `S`.

If `S` is a symbolic matrix, `combine` is applied to all elements of the matrix.

T — Target function

`'atan'` | `'exp'` | `'gamma'` | `'int'` | `'log'` | `'sincos'` | `'sinhcosh'`

Target function, specified as `'atan'`, `'exp'`, `'gamma'`, `'int'`, `'log'`, `'sincos'`, or `'sinhcosh'`. The rewriting rules apply only to calls to the target function.

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`.

Example: `combine(log (a) + log (b),'log','IgnoreAnalyticConstraints',true)`

`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 that generally are not correct, but can give simpler results. For example, `log(a) + log(b) = log(a*b)`. This option is most useful in simplifying expressions where direct use of the solver returns complicated results. Setting `IgnoreAnalyticConstraints` to `true` can lead to wrong or incomplete results.

Output Arguments

`Y` — Expression with combined functions

symbolic variable | symbolic number | symbolic expression | symbolic vector | symbolic matrix

Expression with the combined functions, returned as a symbolic variable, number, expression, or as a vector or matrix of symbolic variables, numbers, or expressions.

Algorithms

`combine` applies the following rewriting rules to the input expression `S`, depending on the value of the target argument `T`.

- When T = `'exp'`, `combine` applies these rewriting rules where valid,

$$e^a e^b = e^{a+b}$$

$$(e^a)^b = e^{ab}.$$

- When T = `'log'`,

  $$\log(a) + \log(b) = \log(ab).$$

  If $b < 1000$,

  $$b\log(a) = \log\left(a^b\right).$$

  When `b >= 1000`, `combine` does not apply this second rule.

  The rules applied to rewrite logarithms do not hold for arbitrary complex values of `a` and `b`. Specify appropriate properties for `a` or `b` to enable these rewriting rules.
- When T = `'int'`,

  $$a\int f(x)dx = \int af(x)dx$$

  $$\int f(x)dx + \int g(x)dx = \int f(x) + g(x)dx$$

  $$\int_a^b f(x)dx + \int_a^b g(x)dx = \int_a^b f(x) + g(x)dx$$

  $$\int_a^b f(x)dx + \int_a^b g(y)dy = \int_a^b f(y) + g(y)dy$$

  $$\int_a^b yf(x)dx + \int_a^b xg(y)dy = \int_a^b yf(c) + xf(c)dc.$$
- When T = `'sincos'`,

  $$\sin(x)\sin(y) = \frac{\cos(x-y)}{2} - \frac{\cos(x+y)}{2}.$$

  `combine` applies similar rules for `sin(x)cos(y)` and `cos(x)cos(y)`.

  $$A\cos(x) + B\sin(x) = A\sqrt{1+\frac{B^2}{A^2}}\cos\left(x+\tan^{-1}\left(\frac{-B}{A}\right)\right).$$

- When T = `'atan'` and -1 < x < 1, -1 < y < 1,

  $$\operatorname{atan}(x)+\operatorname{atan}(y)=\operatorname{atan}\left(\frac{x+y}{1-xy}\right).$$

- When T = `'sinhcosh'`,

  $$\sinh(x)\sinh(y)=\frac{\cosh(x+y)}{2}-\frac{\cosh(x-y)}{2}.$$

  `combine` applies similar rules for `sinh(x)cosh(y)` and `cosh(x)cosh(y)`.

  `combine` applies the previous rules recursively to powers of `sinh` and `cosh` with positive integral exponents.

- When T = `'gamma'`,

  $$a\Gamma(a)=\Gamma(a+1).$$

  and,

  $$\frac{\Gamma(a+1)}{\Gamma(a)}=a.$$

  For positive integers `n`,

  $$\Gamma(-a)\Gamma(a)=-\frac{\pi}{\sin(\pi a)}.$$

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

Introduced in R2014a

compose

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`

Introduced before R2006a

cond

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`.

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`

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-38i

condN1 =
748.0

condNf =
526.158821079719236517033364845

condNi =
748.0
```

Hilbert matrices are classic examples of ill-conditioned matrices.

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`.

Tips

- Calling `cond` for a numeric matrix that is not a symbolic object invokes the MATLAB `cond` function.

See Also

`equationsToMatrix` | `inv` | `linsolve` | `norm` | `rank`

Introduced in R2012b

conj

Symbolic complex conjugate

Syntax

```
conj(X)
```

Description

`conj(X)` is the complex conjugate of `X`. Because symbolic variables are complex by default, unresolved calls such as `conj(x)` can appear in the output of `norm`, `mtimes`, and other functions. For details, see “Use Assumptions on Symbolic Variables” on page 1-28.

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

See Also

`imag` | `real`

Introduced before R2006a

convertMuPADNotebook

Convert MuPAD notebook to MATLAB live script

Syntax

```
convertMuPADNotebook(MuPADfile,MATLABLiveScript)
convertMuPADNotebook(MuPADfile)
```

Description

`convertMuPADNotebook(MuPADfile,MATLABLiveScript)` converts a MuPAD notebook file `MuPADfile` (`.mn`) to a MATLAB live script file `MATLABLiveScript` (`.mlx`). Both `MuPADfile` and `MATLABLiveScript` must be full paths unless the files are in the current folder. For information on live scripts, see "Create Live Scripts in the Live Editor" (MATLAB).

`convertMuPADNotebook(MuPADfile)` uses the same name and path, `MuPADfile`, for the MATLAB live script file that contains converted code. The extension `.mn` changes to `.mlx` in the resulting MATLAB live script file.

Examples

Convert MuPAD Notebook to MATLAB Script

Using `convertMuPADNotebook`, convert a MuPAD notebook to a MATLAB live script. Alternatively, right-click the notebook in the Current Folder browser and select **Open as Live Script** from the context menu.

Suppose that your current folder contains a MuPAD notebook named `myNotebook.mn`. Convert this notebook to the MATLAB live script file named `myScript.mlx`.

```
convertMuPADNotebook('myNotebook.mn','myScript.mlx')
```

Open the resulting file.

```
edit('myScript.mlx')
```

Visually check the code for correctness and completeness. Then verify it by running it.

Use Same Name for Converted File

Convert a MuPAD notebook to a MATLAB live script file with the same name.

Suppose that your current folder contains a MuPAD notebook named `myFile.mn`. Convert this notebook to the MATLAB live script file named `myFile.mlx`.

```
convertMuPADNotebook('myFile.mn')
```

Open the resulting file.

```
edit('myFile.mlx')
```

Visually check the code for correctness and completeness. Then verify it by executing it.

Fix Translation Errors or Warnings

If `convertMuPADNotebook` reports that the converted code has translation errors or warnings, correct the resulting MATLAB code before using it.

Convert the MuPAD notebook, `myNotebook.mn`, to the MATLAB live script file, `myScript.mlx`. Because `myNotebook.mn` contains commands that cannot be directly translated to MATLAB code, `convertMuPADNotebook` flags these commands as translation errors and warnings.

```
convertMuPADNotebook('myNotebook.mn','myScript.mlx')
```

```
Created 'myScript.mlx': 4 translation errors, 1 warnings. For verifying...
 the document, see help.
ans =
c:\MATLABscripts\myScript.mlx
```

A translation error indicates that `convertMuPADNotebook` was unable to convert part of the MuPAD notebook, and that without this part the translated code will not run properly. A translation warning indicates that `convertMuPADNotebook` was unable to convert a part of the MuPAD notebook (for example, an empty input region) and ignored it. Converted code containing warnings is likely to run without any issues.

Open the resulting file.

```
edit('myScript.mlx');
```

Eliminate translation errors. First, search for “translation error”. Next to “translation error”, the converted code displays short comments explaining which MuPAD command did not translate properly. There is also a link to documentation that provides more details and suggestions for fixing the issue. After fixing the issue, remove the corresponding error message and any comments related to it.

Find translation warnings by searching for “translation warning”. The converted code displays a short comment and a link to documentation next to “translation warning”. Some warnings might require you to adapt the code so it runs properly. In most cases, you can ignore translation warnings. Whether you fixed the code or decided to ignore the warning, remove the warning message and any comments related to it.

Visually check the code for correctness and completeness.

Verify that the resulting MATLAB code runs properly by executing it.

Convert All Notebooks in a Folder

Convert all MuPAD notebooks in a folder by making it your current folder, and then using a loop to call the `convertMuPADNotebook` function on every notebook in the folder.

```
files = dir('*.mn');
for i = 1:numel(files)
    convertMuPADNotebook(files(i).name)
end
```

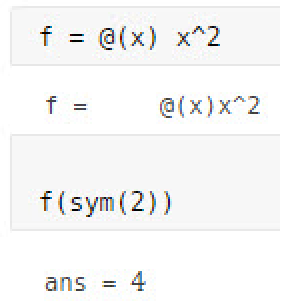
Convert MuPAD Procedure to MATLAB Function

`convertMuPADNotebook` converts MuPAD procedures to MATLAB functions. Not all MuPAD procedures can be converted.

Simple procedures are converted into anonymous functions. Convert a MuPAD notebook with the following code.

```
f := x -> x^2
f(2)
```

The output of `convertMuPADNotebook` is a live script with the anonymous function `f`.



```
f = @(x) x^2
```

```
f =     @(x)x^2
```

```
f(sym(2))
```

```
ans = 4
```

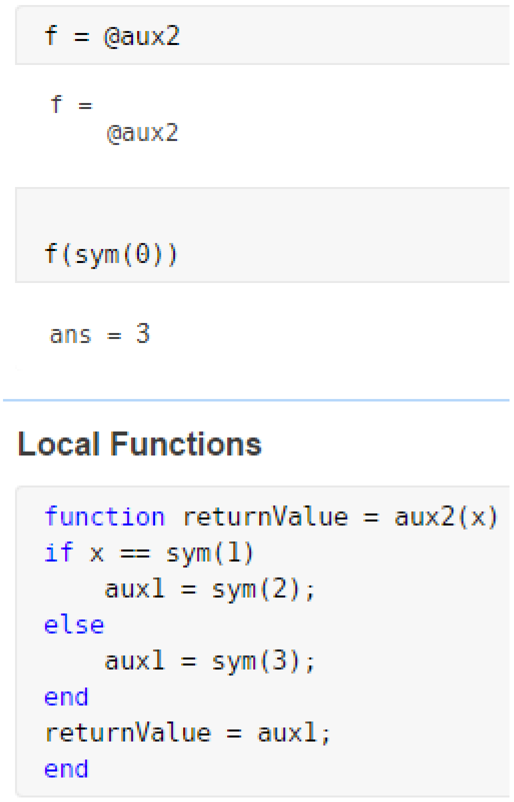
For details on anonymous functions, see “Anonymous Functions” (MATLAB).

When procedures are too complex to convert to anonymous functions, they are converted to local functions in the live script. Local functions are placed at the end of the live script.

Convert a MuPAD notebook with the following code.

```
x -> if x=1 then 2 else 3 end
f(0)
```

The procedure is too complex to convert to an anonymous function. The output of `convertMuPADNotebook` is a live script with the local function `aux2`.



```
f = @aux2
```

```
f =
    @aux2
```

```
f(sym(0))
```

```
ans = 3
```

Local Functions

```
function returnValue = aux2(x)
if x == sym(1)
    aux1 = sym(2);
else
    aux1 = sym(3);
end
returnValue = aux1;
end
```

For information on local functions in scripts, see “Add Functions to Scripts” (MATLAB).

When converting a notebook that reads a MuPAD program file (`.mu`), `convertMuPADNotebook` replaces the `read` command with the contents of the `.mu` file. The notebook and program files must be in the same directory.

Input Arguments

`MuPADfile` — Name of MuPAD notebook
character vector

Name of a MuPAD notebook, specified as a character vector. This character vector must specify the full path to the file, unless the file is in the current folder.

Example: `'C:\MuPAD_Notebooks\myFile.mn'`

`MATLABLiveScript` — Name of MATLAB live script file
character vector

Name of a MATLAB live script file, specified as a character vector. This character vector must specify the full path to the file, unless you intend to create a file in the current folder.

Example: `'C:\MATLAB_Scripts\myFile.mlx'`

See Also

`generate::MATLAB`

Topics

"Convert MuPAD Notebooks to MATLAB Live Scripts" on page 3-19
"Troubleshoot MuPAD to MATLAB Translation Errors" on page 3-25
"Troubleshoot MuPAD to MATLAB Translation Warnings" on page 3-35

Introduced in R2016a

cos

Symbolic cosine function

Syntax

```
cos(X)
```

Description

`cos(X)` returns the cosine function on page 4-259 of `X`.

Examples

Cosine Function for Numeric and Symbolic Arguments

Depending on its arguments, `cos` returns floating-point or exact symbolic results.

Compute the cosine function for these numbers. Because these numbers are not symbolic objects, `cos` returns floating-point results.

```
A = cos([-2, -pi, pi/6, 5*pi/7, 11])
```

```
A =
   -0.4161   -1.0000    0.8660   -0.6235    0.0044
```

Compute the cosine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `cos` returns unresolved symbolic calls.

```
symA = cos(sym([-2, -pi, pi/6, 5*pi/7, 11]))
```

```
symA =
[ cos(2), -1, 3^(1/2)/2, -cos((2*pi)/7), cos(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

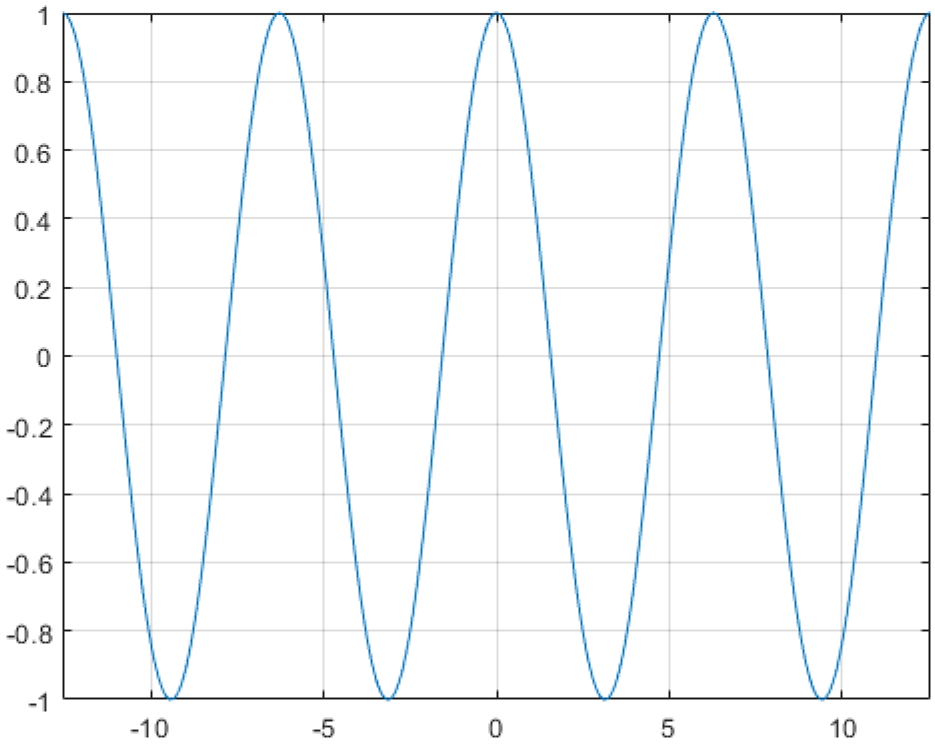
```
vpa(symA)
```

```
ans =
[ -0.41614683654714238699756822950076,...
-1.0,...
0.86602540378443864676372317075294,...
-0.62348980185873353052500488400424,...
0.0044256979880507857483550247239416]
```

Plot Cosine Function

Plot the cosine function on the interval from -4π to 4π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(cos(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Cosine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `cos`.

Find the first and second derivatives of the cosine function:

```
syms x
diff(cos(x), x)
diff(cos(x), x, x)
```

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

```
ans =
-cos(x)
```

Find the indefinite integral of the cosine function:

```
int(cos(x), x)
```

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

Find the Taylor series expansion of `cos(x)`:

```
taylor(cos(x), x)
```

```
ans =
x^4/24 - x^2/2 + 1
```

Rewrite the cosine function in terms of the exponential function:

```
rewrite(cos(x), 'exp')
```

```
ans =
exp(-x*1i)/2 + exp(x*1i)/2
```

Evaluate Units with `cos` Function

`cos` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the cosine of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
cosinf = cos(f)
```

```
cosinf =
[ cos((pi*x)/180), cos(2)]
```

You can calculate `cosinf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

`x` — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

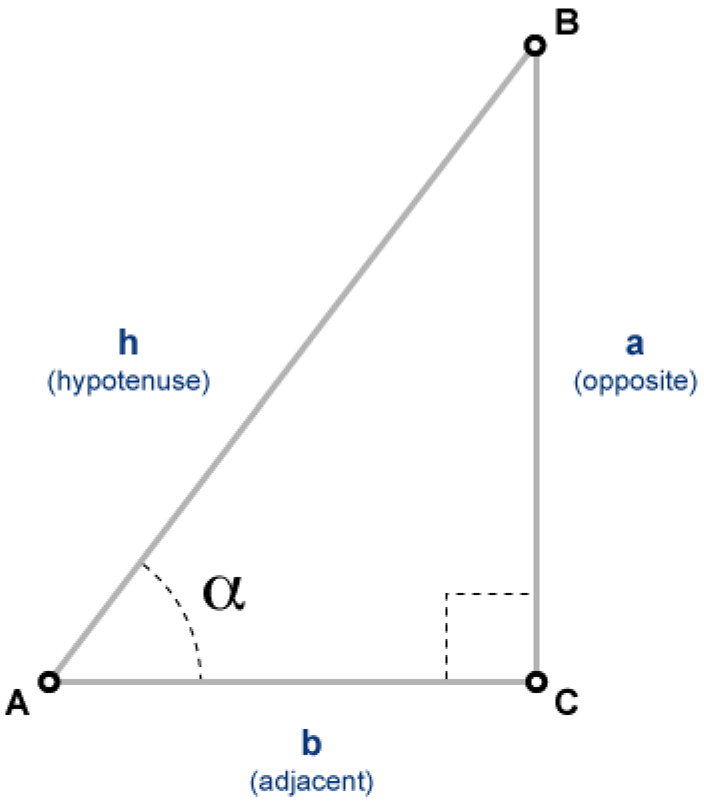
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Cosine Function

The cosine of an angle, α, defined with reference to a right angled triangle is

$$\cos(\alpha) = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{b}{h}.$$



The cosine of a complex angle, α, is

$$\cos(\alpha) = \frac{e^{i\alpha} + e^{-i\alpha}}{2}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cot` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

cosh

Symbolic hyperbolic cosine function

Syntax

```
cosh(X)
```

Description

`cosh(X)` returns the hyperbolic cosine function of `X`.

Examples

Hyperbolic Cosine Function for Numeric and Symbolic Arguments

Depending on its arguments, `cosh` returns floating-point or exact symbolic results.

Compute the hyperbolic cosine function for these numbers. Because these numbers are not symbolic objects, `cosh` returns floating-point results.

```
A = cosh([-2, -pi*i, pi*i/6, 5*pi*i/7, 3*pi*i/2])
```

```
A =
    3.7622   -1.0000    0.8660   -0.6235   -0.0000
```

Compute the hyperbolic cosine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `cosh` returns unresolved symbolic calls.

```
symA = cosh(sym([-2, -pi*i, pi*i/6, 5*pi*i/7, 3*pi*i/2]))
```

```
symA =
[ cosh(2), -1, 3^(1/2)/2, -cosh((pi*2i)/7), 0]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

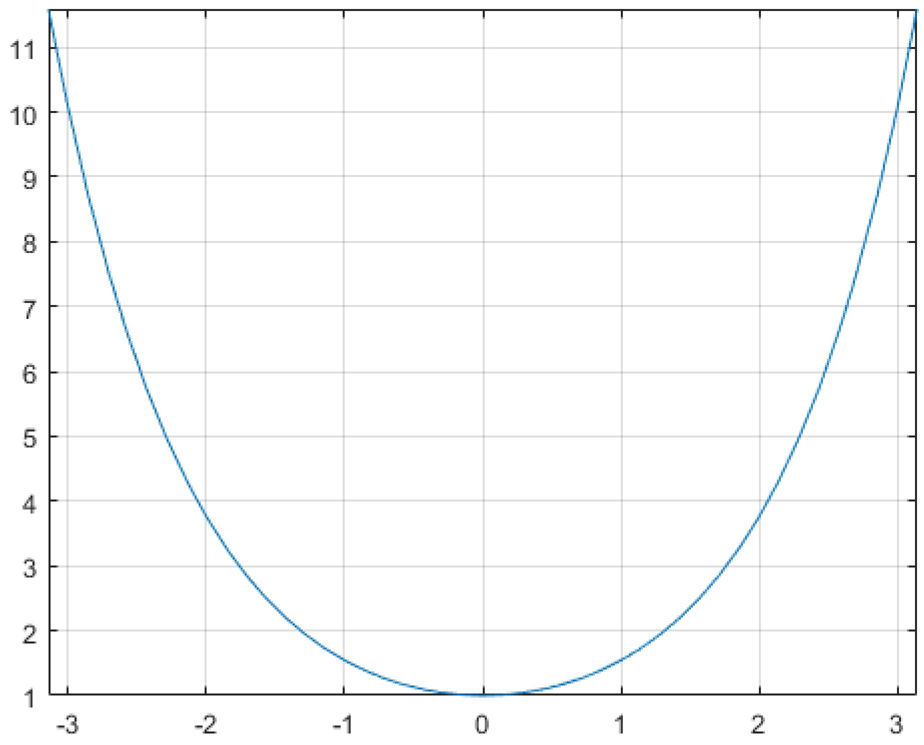
```
vpa(symA)
```

```
ans =
[ 3.7621956910836314595622134777737,...
-1.0,...
0.86602540378443864676372317075294,...
-0.62348980185873353052500488400424,...
0]
```

Plot Hyperbolic Cosine Function

Plot the hyperbolic cosine function on the interval from $-\pi$ to π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(cosh(x), [-pi, pi])
grid on
```



Handle Expressions Containing Hyperbolic Cosine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `cosh`.

Find the first and second derivatives of the hyperbolic cosine function:

```
syms x
diff(cosh(x), x)
diff(cosh(x), x, x)
```

```
ans =
sinh(x)
```

```
ans =
cosh(x)
```

Find the indefinite integral of the hyperbolic cosine function:

```
int(cosh(x), x)
```

```
ans =
sinh(x)
```

Find the Taylor series expansion of `cosh(x)`:

```
taylor(cosh(x), x)
```

```
ans =
x^4/24 + x^2/2 + 1
```

Rewrite the hyperbolic cosine function in terms of the exponential function:

```
rewrite(cosh(x), 'exp')
```

```
ans =
exp(-x)/2 + exp(x)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

coshint

Hyperbolic cosine integral function

Syntax

```
coshint(X)
```

Description

`coshint(X)` returns the hyperbolic cosine integral function on page 4-268 of `X`.

Examples

Hyperbolic Cosine Integral Function for Numeric and Symbolic Arguments

Depending on its arguments, `coshint` returns floating-point or exact symbolic results.

Compute the hyperbolic cosine integral function for these numbers. Because these numbers are not symbolic objects, `coshint` returns floating-point results.

```
A = coshint([-1, 0, 1/2, 1, pi/2, pi])
```

```
A =
   0.8379 + 3.1416i     -Inf + 0.0000i  -0.0528 + 0.0000i   0.8379...
 + 0.0000i   1.7127 + 0.0000i   5.4587 + 0.0000i
```

Compute the hyperbolic cosine integral function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `coshint` returns unresolved symbolic calls.

```
symA = coshint(sym([-1, 0, 1/2, 1, pi/2, pi]))
```

```
symA =
[ coshint(1) + pi*1i, -Inf, coshint(1/2), coshint(1), coshint(pi/2), coshint(pi)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

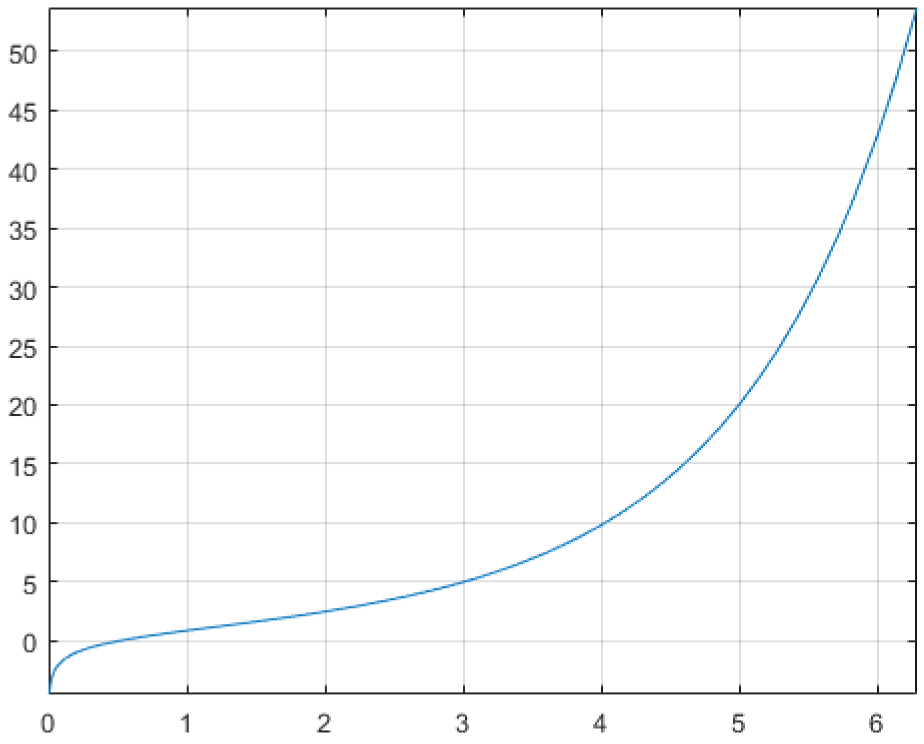
```
vpa(symA)
```

```
ans =
[ 0.83786694098020824089467857943576...
 + 3.1415926535897932384626433832795i,...
-Inf,...
-0.052776844956493615913136063326141,...
0.83786694098020824089467857943576,...
1.7126607364844281079951569897796,...
5.4587340442160681980014878977798]
```

Plot Hyperbolic Cosine Integral Function

Plot the hyperbolic cosine integral function on the interval from 0 to `2*pi` . Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(coshint(x), [0, 2*pi])
grid on
```



Handle Expressions Containing Hyperbolic Cosine Integral Function

Many functions, such as `diff` and `int`, can handle expressions containing `coshint`.

Find the first and second derivatives of the hyperbolic cosine integral function:

```
syms x
diff(coshint(x), x)
diff(coshint(x), x, x)
```

```
ans =
cosh(x)/x
```

```
ans =
sinh(x)/x - cosh(x)/x^2
```

Find the indefinite integral of the hyperbolic cosine integral function:

```
int(coshint(x), x)
```

```
ans =
x*coshint(x) - sinh(x)
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Hyperbolic Cosine Integral Function

The hyperbolic cosine integral function is defined as follows:

$$\mathrm{Chi}(x) = \gamma + \log(x) + \int_0^x \frac{\cosh(t) - 1}{t}\,dt$$

Here, γ is the Euler-Mascheroni constant:

$$\gamma = \lim_{n\to\infty}\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \ln(n)\right)$$

References

[1] Cautschi, W. and W. F. Cahill. “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

`cos` | `cosint` | `eulergamma` | `int` | `sinhint` | `sinint` | `ssinint`

Introduced in R2014a

cosint

Cosine integral function

Syntax

```
cosint(X)
```

Description

`cosint(X)` returns the cosine integral function on page 4-273 of `X`.

Examples

Cosine Integral Function for Numeric and Symbolic Arguments

Depending on its arguments, `cosint` returns floating-point or exact symbolic results.

Compute the cosine integral function for these numbers. Because these numbers are not symbolic objects, `cosint` returns floating-point results.

```
A = cosint([- 1, 0, pi/2, pi, 1])
```

```
A =
   0.3374 + 3.1416i     -Inf + 0.0000i   0.4720 + 0.0000i...
   0.0737 + 0.0000i   0.3374 + 0.0000i
```

Compute the cosine integral function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `cosint` returns unresolved symbolic calls.

```
symA = cosint(sym([- 1, 0, pi/2, pi, 1]))
```

```
symA =
[ cosint(1) + pi*1i, -Inf, cosint(pi/2), cosint(pi), cosint(1)]
```

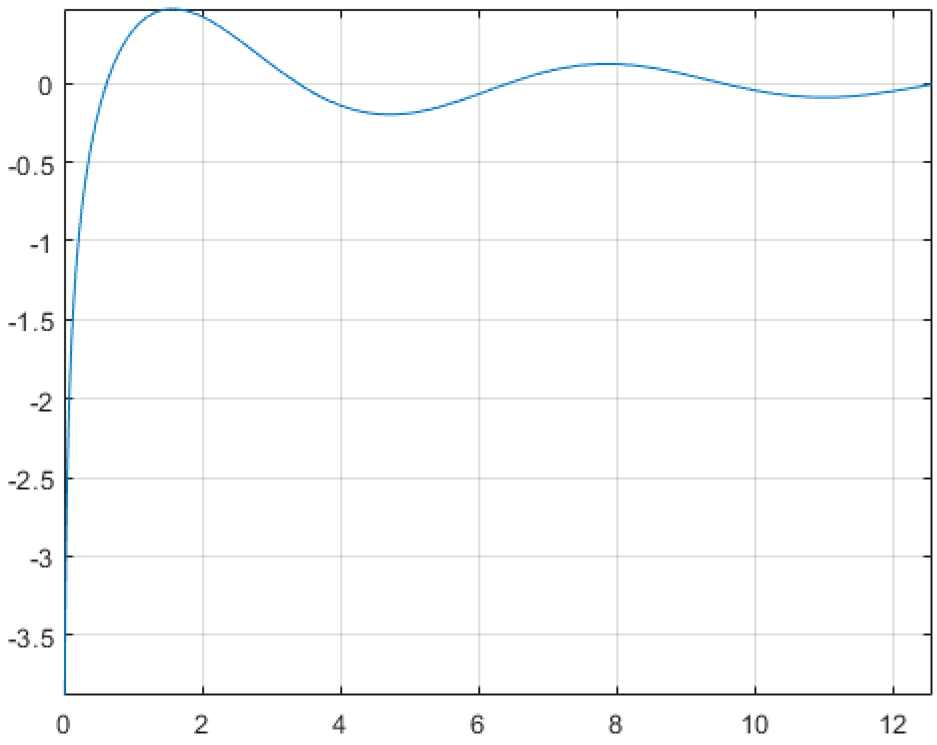
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.33740392290096813466264620388915...
 + 3.1415926535897932384626433832795i,...
-Inf,...
0.47200065143956865077760610761413,...
0.07366791204642548599010096523015,...
0.33740392290096813466264620388915]
```

Plot Cosine Integral Function

Plot the cosine integral function on the interval from 0 to `4*pi`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(cosint(x), [0, 4*pi])
grid on
```



Handle Expressions Containing Cosine Integral Function

Many functions, such as `diff` and `int`, can handle expressions containing `cosint`.

Find the first and second derivatives of the cosine integral function:

```
syms x
diff(cosint(x), x)
diff(cosint(x), x, x)
```

```
ans =
cos(x)/x
```

```
ans =
- cos(x)/x^2 - sin(x)/x
```

Find the indefinite integral of the cosine integral function:

```
int(cosint(x), x)

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

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Cosine Integral Function

The cosine integral function is defined as follows:

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

Here, γ is the Euler-Mascheroni constant:

$$\gamma = \lim_{n\to\infty}\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \ln(n)\right)$$

References

[1] Gautschi, W. and W. F. Cahill. “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

`cos` | `coshint` | `eulergamma` | `int` | `sinhint` | `sinint` | `ssinint`

Introduced before R2006a

cot

Symbolic cotangent function

Syntax

```
cot(X)
```

Description

`cot(X)` returns the cotangent function on page 4-279 of `X`.

Examples

Cotangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `cot` returns floating-point or exact symbolic results.

Compute the cotangent function for these numbers. Because these numbers are not symbolic objects, `cot` returns floating-point results.

```
A = cot([-2, -pi/2, pi/6, 5*pi/7, 11])
```

```
A =
    0.4577   -0.0000    1.7321   -0.7975   -0.0044
```

Compute the cotangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `cot` returns unresolved symbolic calls.

```
symA = cot(sym([-2, -pi/2, pi/6, 5*pi/7, 11]))
```

```
symA =
[ -cot(2), 0, 3^(1/2), -cot((2*pi)/7), cot(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

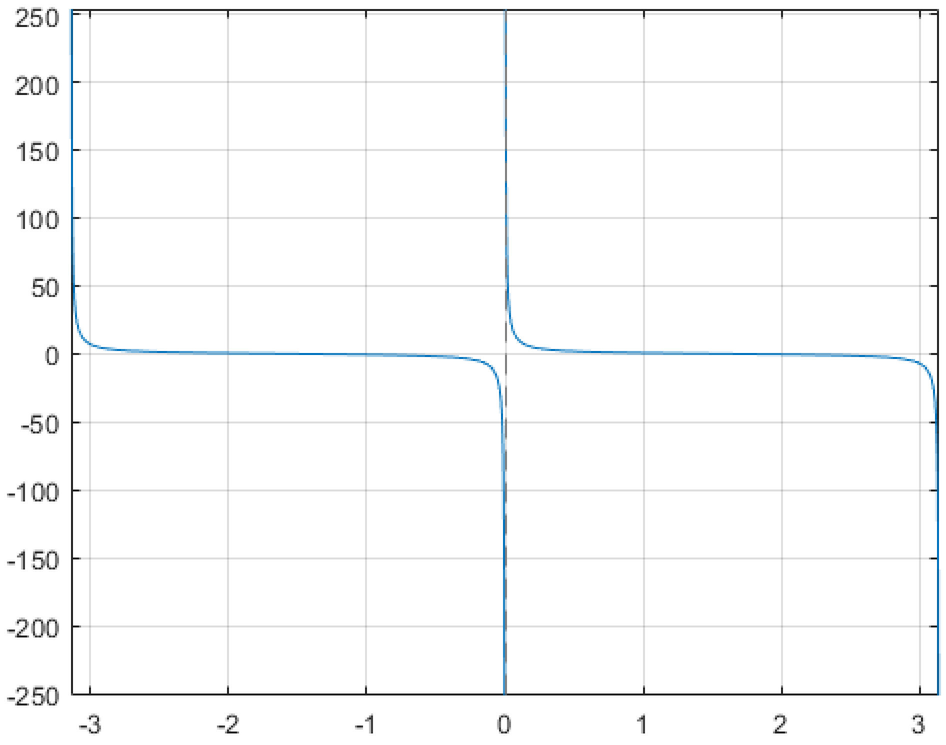
```
vpa(symA)
```

```
ans =
[ 0.45765755436028576375027741043205,...
0,...
1.7320508075688772935274463415059,...
-0.79747338888240396141568825421443,...
-0.0044257413313241136855482762848043]
```

Plot Cotangent Function

Plot the cotangent function on the interval from $-\pi$ to π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(cot(x), [-pi, pi])
grid on
```



Handle Expressions Containing Cotangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `cot`.

Find the first and second derivatives of the cotangent function:

```
syms x
diff(cot(x), x)
diff(cot(x), x, x)
```

```
ans =
- cot(x)^2 - 1
```

```
ans =
2*cot(x)*(cot(x)^2 + 1)
```

Find the indefinite integral of the cotangent function:

```
int(cot(x), x)

ans =
log(sin(x))
```

Find the Taylor series expansion of `cot(x)` around `x = pi/2`:

```
taylor(cot(x), x, pi/2)

ans =
pi/2 - x - (x - pi/2)^3/3 - (2*(x - pi/2)^5)/15
```

Rewrite the cotangent function in terms of the sine and cosine functions:

```
rewrite(cot(x), 'sincos')

ans =
 cos(x)/sin(x)
```

Rewrite the cotangent function in terms of the exponential function:

```
rewrite(cot(x), 'exp')

ans =
(exp(x*2i)*1i + 1i)/(exp(x*2i) - 1)
```

Evaluate Units with `cot` Function

`cot` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the cotangent of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
cotf = cot(f)
cotf =
[ cot((pi*x)/180), cot(2)]
```

You can calculate `cotf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

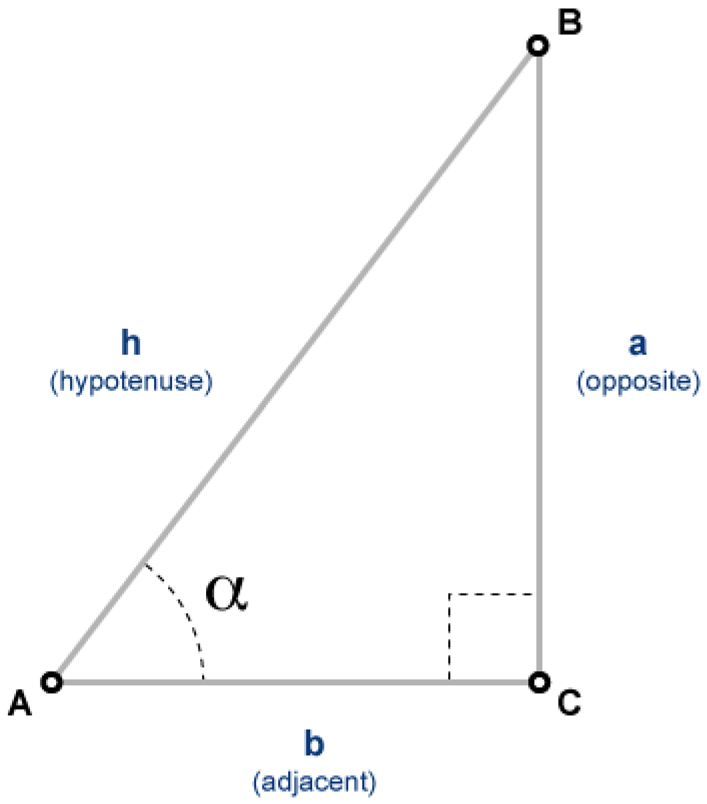
Definitions

Cotangent Function

The cotangent of an angle, α, defined with reference to a right angled triangle is

$$\cot(\alpha) = \frac{1}{\tan(\alpha)} = \frac{\text{adjacent side}}{\text{opposite side}} = \frac{b}{a}.$$

.



The cotangent of a complex angle α is

$$\cot(\alpha) = \frac{i\left(e^{i\alpha} + e^{-i\alpha}\right)}{\left(e^{i\alpha} - e^{-i\alpha}\right)}.$$

.

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `csc` | `sec` | `sin` | `tan`

Introduced before R2006a

coth

Symbolic hyperbolic cotangent function

Syntax

```
coth(X)
```

Description

`coth(X)` returns the hyperbolic cotangent function of `X`

Examples

Hyperbolic Cotangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `coth` returns floating-point or exact symbolic results.

Compute the hyperbolic cotangent function for these numbers. Because these numbers are not symbolic objects, `coth` returns floating-point results.

```
A = coth([-2, -pi*i/3, pi*i/6, 5*pi*i/7, 3*pi*i/2])
```

```
A =
  -1.0373 + 0.0000i   0.0000 + 0.5774i   0.0000 - 1.7321i...
   0.0000 + 0.7975i   0.0000 - 0.0000i
```

Compute the hyperbolic cotangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `coth` returns unresolved symbolic calls.

```
symA = coth(sym([-2, -pi*i/3, pi*i/6, 5*pi*i/7, 3*pi*i/2]))
```

```
symA =
[ -coth(2), (3^(1/2)*1i)/3, -3^(1/2)*1i, -coth((pi*2i)/7), 0]
```

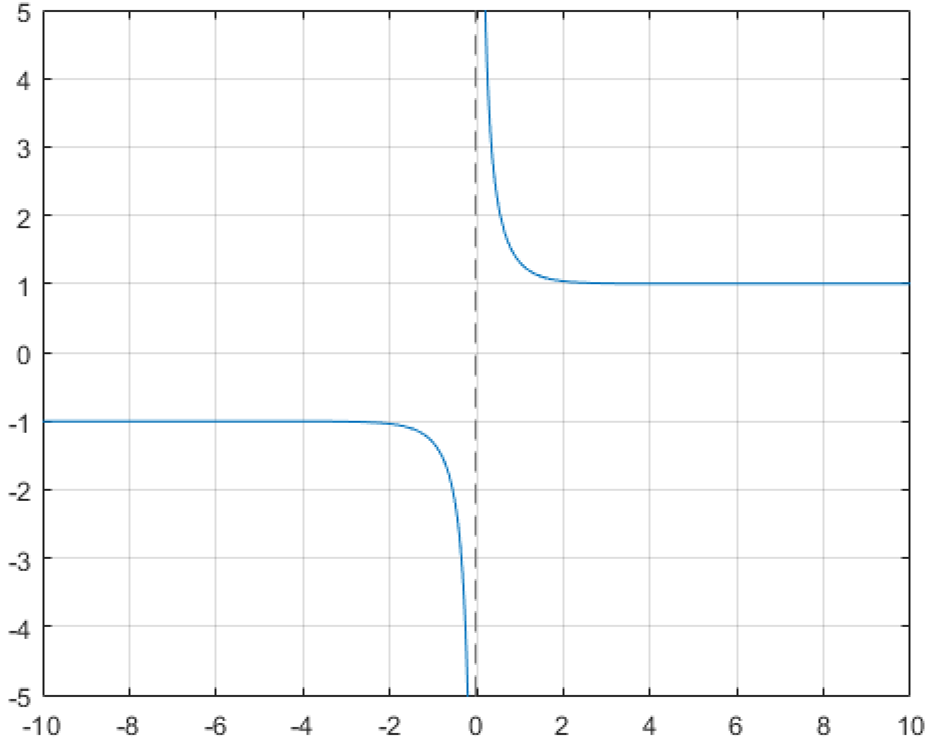
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.0373147207275480958778097647678,...
0.57735026918962576450914878050196i,...
-1.7320508075688772935274463415059i,...
0.79747338888240396141568825421443i,...
0]
```

Plot Hyperbolic Cotangent Function

Plot the hyperbolic cotangent function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(coth(x), [-10, 10])
grid on
```



Handle Expressions Containing Hyperbolic Cotangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `coth`.

Find the first and second derivatives of the hyperbolic cotangent function:

```
syms x
diff(coth(x), x)
diff(coth(x), x, x)
```

```
ans =
1 - coth(x)^2
```

```
ans =
2*coth(x)*(coth(x)^2 - 1)
```

Find the indefinite integral of the hyperbolic cotangent function:

```
int(coth(x), x)
```

```
ans =
log(sinh(x))
```

Find the Taylor series expansion of `coth(x)` around `x = pi*i/2`:

```
taylor(coth(x), x, pi*i/2)
```

```
ans =
x - (pi*1i)/2 - (x - (pi*1i)/2)^3/3 + (2*(x - (pi*1i)/2)^5)/15
```

Rewrite the hyperbolic cotangent function in terms of the exponential function:

```
rewrite(coth(x), 'exp')
```

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

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `csch` | `sech` | `sinh` | `tanh`

Introduced before R2006a

csc

Symbolic cosecant function

Syntax

```
csc(X)
```

Description

`csc(X)` returns the cosecant function on page 4-290 of `X`.

Examples

Cosecant Function for Numeric and Symbolic Arguments

Depending on its arguments, `csc` returns floating-point or exact symbolic results.

Compute the cosecant function for these numbers. Because these numbers are not symbolic objects, `csc` returns floating-point results.

```
A = csc([-2, -pi/2, pi/6, 5*pi/7, 11])
```

```
A =
   -1.0998   -1.0000    2.0000    1.2790   -1.0000
```

Compute the cosecant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `csc` returns unresolved symbolic calls.

```
symA = csc(sym([-2, -pi/2, pi/6, 5*pi/7, 11]))
```

```
symA =
[ -1/sin(2), -1, 2, 1/sin((2*pi)/7), 1/sin(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

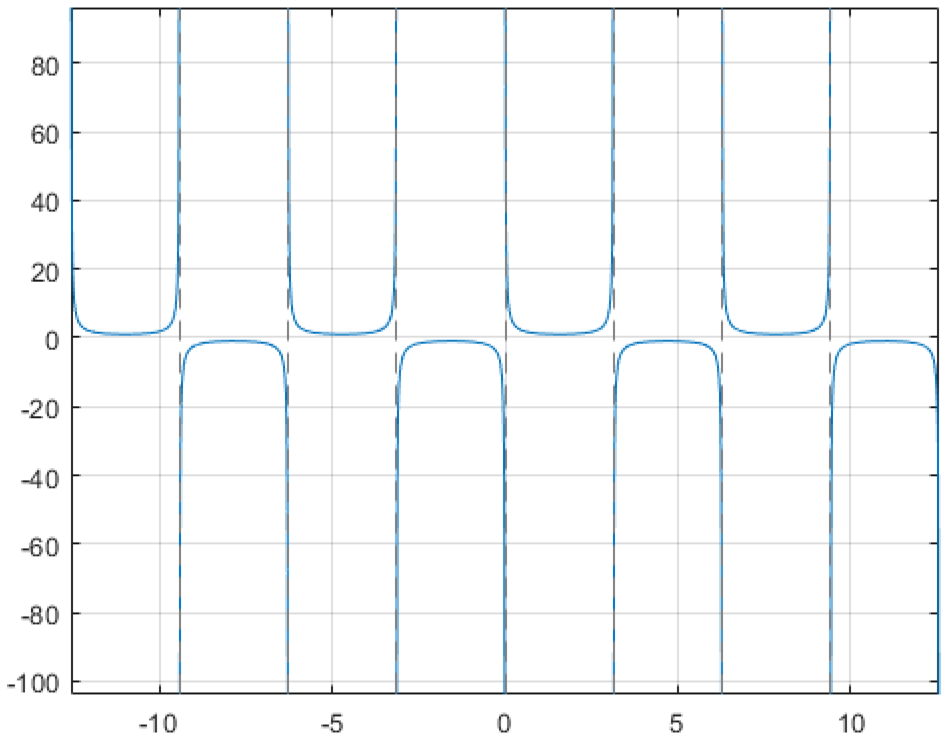
```
vpa(symA)
```

```
ans =
[ -1.0997501702946164667566973970263,...
-1.0,...
2.0,...
1.2790480076899326057478506072714,...
-1.0000097935452091313874644503551]
```

Plot Cosecant Function

Plot the cosecant function on the interval from -4π to 4π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(csc(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Cosecant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `csc`.

Find the first and second derivatives of the cosecant function:

```
syms x
diff(csc(x), x)
diff(csc(x), x, x)
```

```
ans =
-cos(x)/sin(x)^2
```

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

Find the indefinite integral of the cosecant function:

```
int(csc(x), x)
```

```
ans =
log(tan(x/2))
```

Find the Taylor series expansion of `csc(x)` around `x = pi/2`:

```
taylor(csc(x), x, pi/2)
```

```
ans =
(x - pi/2)^2/2 + (5*(x - pi/2)^4)/24 + 1
```

Rewrite the cosecant function in terms of the exponential function:

```
rewrite(csc(x), 'exp')
```

```
ans =
1/((exp(-x*1i)*1i)/2 - (exp(x*1i)*1i)/2)
```

Evaluate Units with `csc` Function

`csc` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the cosecant of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
cosecf = csc(f)
```

```
cosecf =
[ 1/sin((pi*x)/180), 1/sin(2)]
```

You can calculate `cosecf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

x — Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

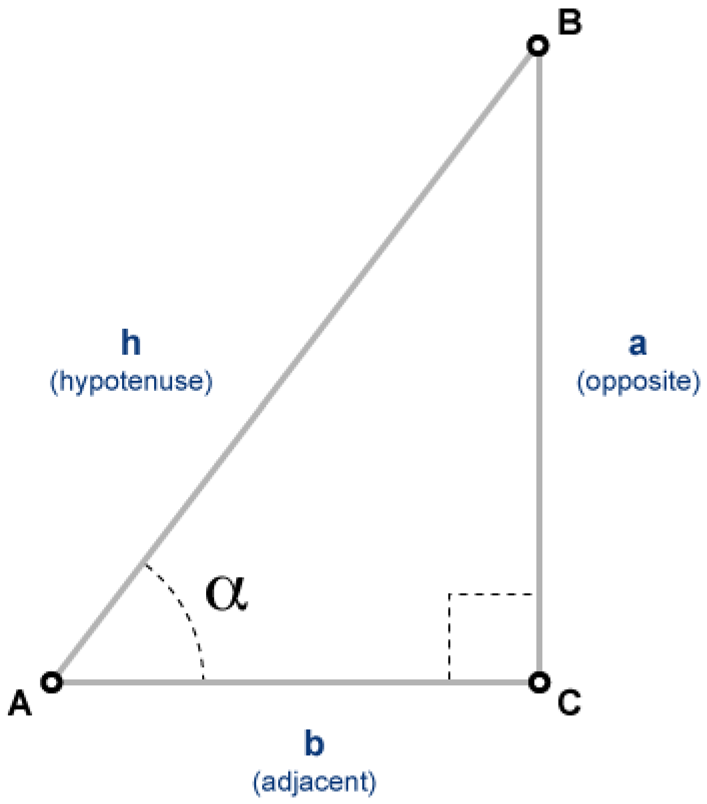
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Cosecant Function

The cosecant of an angle, α, defined with reference to a right angled triangle is

$$\csc(\alpha) = \frac{1}{\sin(\alpha)} = \frac{\text{hypotenuse}}{\text{opposite side}} = \frac{h}{a}.$$



The cosecant of a complex angle, α, is

$$\csc(\alpha) = \frac{2i}{e^{i\alpha} - e^{-i\alpha}}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sin` | `tan`

Introduced before R2006a

csch

Symbolic hyperbolic cosecant function

Syntax

```
csch(X)
```

Description

`csch(X)` returns the hyperbolic cosecant function of `X`.

Examples

Hyperbolic Cosecant Function for Numeric and Symbolic Arguments

Depending on its arguments, `csch` returns floating-point or exact symbolic results.

Compute the hyperbolic cosecant function for these numbers. Because these numbers are not symbolic objects, `csch` returns floating-point results.

```
A = csch([-2, -pi*i/2, 0, pi*i/3, 5*pi*i/7, pi*i/2])
```

```
A =
  -0.2757 + 0.0000i   0.0000 + 1.0000i      Inf + 0.0000i...
   0.0000 - 1.1547i   0.0000 - 1.2790i   0.0000 - 1.0000i
```

Compute the hyperbolic cosecant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `csch` returns unresolved symbolic calls.

```
symA = csch(sym([-2, -pi*i/2, 0, pi*i/3, 5*pi*i/7, pi*i/2]))
```

```
symA =
[ -1/sinh(2), 1i, Inf, -(3^(1/2)*2i)/3, 1/sinh((pi*2i)/7), -1i]
```

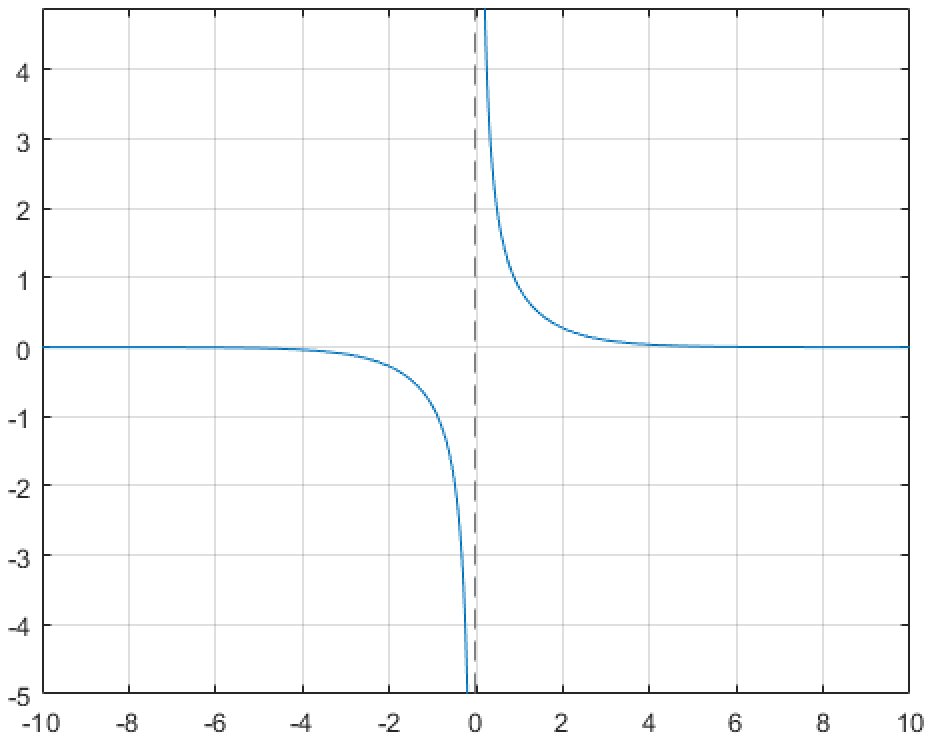
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.27572056477178320775835148216303,...
1.0i,...
Inf,...
-1.1547005383792515290182975610039i,...
-1.2790480076899326057478506072714i,...
-1.0i]
```

Plot Hyperbolic Cosecant Function

Plot the hyperbolic cosecant function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(csch(x), [-10, 10])
grid on
```



Handle Expressions Containing Hyperbolic Cosecant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `csch`.

Find the first and second derivatives of the hyperbolic cosecant function:

```
syms x
diff(csch(x), x)
diff(csch(x), x, x)
```

```
ans =
-cosh(x)/sinh(x)^2
```

```
ans =
(2*cosh(x)^2)/sinh(x)^3 - 1/sinh(x)
```

Find the indefinite integral of the hyperbolic cosecant function:

```
int(csch(x), x)
```

```
ans =
log(tanh(x/2))
```

Find the Taylor series expansion of `csch(x)` around `x = pi*i/2`:

```
taylor(csch(x), x, pi*i/2)
```

```
ans =
((x - (pi*1i)/2)^2*1i)/2 - ((x - (pi*1i)/2)^4*5i)/24 - 1i
```

Rewrite the hyperbolic cosecant function in terms of the exponential function:

```
rewrite(csch(x), 'exp')
```

```
ans =
-1/(exp(-x)/2 - exp(x)/2)
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `sech` | `sinh` | `tanh`

Introduced before R2006a

ctranspose'

Symbolic matrix complex conjugate transpose

Syntax

```
A'
ctranspose(A)
```

Description

`A'` computes the complex conjugate transpose on page 4-298 of `A`.

`ctranspose(A)` is equivalent to `A'`.

Examples

Conjugate Transpose of Real Matrix

Create a `2`-by-`3` matrix, the elements of which represent real numbers.

```
syms x y real
A = [x x x; y y y]
```

```
A =
[ x, x, x]
[ y, y, y]
```

Find the complex conjugate transpose of this matrix.

```
A'
```

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

If all elements of a matrix represent real numbers, then its complex conjugate transform equals to its nonconjugate transform.

```
isAlways(A' == A.')

ans =
  3×2 logical array
     1     1
     1     1
     1     1
```

Conjugate Transpose of Complex Matrix

Create a `2`-by-`2` matrix, the elements of which represent complex numbers.

```
syms x y real
A = [x + y*i x - y*i; y + x*i y - x*i]

A =
[ x + y*1i, x - y*1i]
[ y + x*1i, y - x*1i]
```

Find the conjugate transpose of this matrix. The complex conjugate transpose operator, `A'`, performs a transpose and negates the sign of the imaginary portion of the complex elements in `A`.

```
A'

ans =
[ x - y*1i, y - x*1i]
[ x + y*1i, y + x*1i]
```

For a matrix of complex numbers with nonzero imaginary parts, the complex conjugate transform is not equal to the nonconjugate transform.

```
isAlways(A' == A.','Unknown','false')

ans =
  2×2 logical array
     0     0
     0     0
```

Input Arguments

`A` — Input

number | symbolic number | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic multidimensional array

Input, specified as a number or a symbolic number, variable, expression, vector, matrix, multidimensional array.

Definitions

Complex Conjugate Transpose

The complex conjugate transpose of a matrix interchanges the row and column index for each element, reflecting the elements across the main diagonal. The operation also negates the imaginary part of any complex numbers.

For example, if `B = A'` and `A(1,2)` is `1+1i`, then the element `B(2,1)` is `1-1i`.

See Also

`ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

cumprod

Symbolic cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
B = cumprod(___,direction)
```

Description

`B = cumprod(A)` returns an array the same size as `A` containing the cumulative product.

- If `A` is a vector, then `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.
- If `A` is a matrix, then `cumprod(A)` returns a matrix containing the cumulative products of each column of `A`.

`B = cumprod(A,dim)` returns the cumulative product along dimension `dim`. For example, if `A` is a matrix, then `cumprod(A,2)` returns the cumulative product of each row.

`B = cumprod(___,direction)` specifies the direction using any of the previous syntaxes. For instance, `cumprod(A,2,'reverse')` returns the cumulative product within the rows of `A` by working from end to beginning of the second dimension.

Examples

Cumulative Product of Vector

Create a vector and find the cumulative product of its elements.

```
V = 1./factorial(sym([1:5]))
prod_V = cumprod(V)
```

```
V =
[ 1, 1/2, 1/6, 1/24, 1/120]

prod_V =
[ 1, 1/2, 1/12, 1/288, 1/34560]
```

Cumulative Product of Each Column in Symbolic Matrix

Create matrix a 4-by-4 symbolic matrix `X` all elements of which equal `x`.

```
syms x
X = x*ones(4,4)
```

```
X =
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
```

Compute the cumulative product of the elements of `X`. By default, `cumprod` returns the cumulative product of each column.

```
productX = cumprod(X)
```

```
productX =
[   x,   x,   x,   x]
[ x^2, x^2, x^2, x^2]
[ x^3, x^3, x^3, x^3]
[ x^4, x^4, x^4, x^4]
```

Cumulative Product of Each Row in Symbolic Matrix

Create matrix a 4-by-4 symbolic matrix, all elements of which equal `x`.

```
syms x
X = x*ones(4,4)
```

```
X =
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
```

Compute the cumulative product of each row of the matrix `X`.

```
productX = cumprod(X,2)

productX =
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]
```

Reverse Cumulative Product

Create matrix a 4-by-4 symbolic matrix `X` all elements of which equal `x`.

```
syms x
X = x*ones(4,4)

X =
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
[ x, x, x, x]
```

Calculate the cumulative product along the columns in both directions. Specify the `'reverse'` option to work from right to left in each row.

```
columnsDirect = cumprod(X)
columnsReverse = cumprod(X,'reverse')

columnsDirect =
[   x,   x,   x,   x]
[ x^2, x^2, x^2, x^2]
[ x^3, x^3, x^3, x^3]
[ x^4, x^4, x^4, x^4]

columnsReverse =
[ x^4, x^4, x^4, x^4]
[ x^3, x^3, x^3, x^3]
[ x^2, x^2, x^2, x^2]
[   x,   x,   x,   x]
```

Calculate the cumulative product along the rows in both directions. Specify the `'reverse'` option to work from right to left in each row.

```
rowsDirect = cumprod(X,2)
rowsReverse = cumprod(X,2,'reverse')
```

```
rowsDirect =
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]
[ x, x^2, x^3, x^4]

rowsReverse =
[ x^4, x^3, x^2, x]
[ x^4, x^3, x^2, x]
[ x^4, x^3, x^2, x]
[ x^4, x^3, x^2, x]
```

Input Arguments

`A` — Input array

symbolic vector | symbolic matrix

Input array, specified as a vector or matrix.

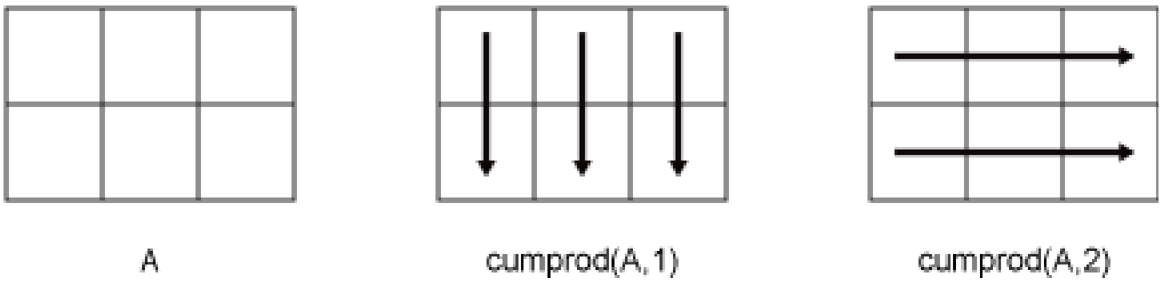
`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. The default value is 1.

Consider a two-dimensional input array, `A`.

- `cumprod(A,1)`) works on successive elements in the columns of `A` and returns the cumulative product of each column.
- `cumprod(A,2)` works on successive elements in the rows of `A` and returns the cumulative product of each row.



`cumprod` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Output Arguments

`B` — Cumulative product array
vector | matrix

Cumulative product array, returned as a vector or matrix of the same size as the input `A`.

See Also

`cumsum` | `fold` | `int` | `symprod` | `symsum`

Introduced in R2013b

cumsum

Symbolic cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
B = cumsum(___,direction)
```

Description

`B = cumsum(A)` returns an array the same size as `A` containing the cumulative sum.

- If `A` is a vector, then `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.
- If `A` is a matrix, then `cumsum(A)` returns a matrix containing the cumulative sums of each column of `A`.

`B = cumsum(A,dim)` returns the cumulative sum along dimension `dim`. For example, if `A` is a matrix, then `cumsum(A,2)` returns the cumulative sum of each row.

`B = cumsum(___,direction)` specifies the direction using any of the previous syntaxes. For instance, `cumsum(A,2,'reverse')` returns the cumulative sum within the rows of `A` by working from end to beginning of the second dimension.

Examples

Cumulative Sum of Vector

Create a vector and find the cumulative sum of its elements.

```
V = 1./factorial(sym([1:5]))
sum_V = cumsum(V)
```

```
V =
[ 1, 1/2, 1/6, 1/24, 1/120]

sum_V =
[ 1, 3/2, 5/3, 41/24, 103/60]
```

Cumulative Sum of Each Column in Symbolic Matrix

Create matrix a 4-by-4 symbolic matrix `A` all elements of which equal 1.

```
A = sym(ones(4,4))
```

```
A =
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
```

Compute the cumulative sum of elements of `A`. By default, `cumsum` returns the cumulative sum of each column.

```
sumA = cumsum(A)
```

```
sumA =
[ 1, 1, 1, 1]
[ 2, 2, 2, 2]
[ 3, 3, 3, 3]
[ 4, 4, 4, 4]
```

Cumulative Sum of Each Row in Symbolic Matrix

Create matrix a 4-by-4 symbolic matrix `A` all elements of which equal 1.

```
A = sym(ones(4,4))
```

```
A =
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
```

Compute the cumulative sum of each row of the matrix `A`.

```
sumA = cumsum(A,2)
sumA =
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]
```

Reverse Cumulative Sum

Create matrix a 4-by-4 symbolic matrix, all elements of which equal 1.

```
A = sym(ones(4,4))
```

```
A =
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
[ 1, 1, 1, 1]
```

Calculate the cumulative sum along the columns in both directions. Specify the `'reverse'` option to work from right to left in each row.

```
columnsDirect = cumsum(A)
columnsReverse = cumsum(A,'reverse')
```

```
columnsDirect =
[ 1, 1, 1, 1]
[ 2, 2, 2, 2]
[ 3, 3, 3, 3]
[ 4, 4, 4, 4]

columnsReverse =
[ 4, 4, 4, 4]
[ 3, 3, 3, 3]
[ 2, 2, 2, 2]
[ 1, 1, 1, 1]
```

Calculate the cumulative sum along the rows in both directions. Specify the `'reverse'` option to work from right to left in each row.

```
rowsDirect = cumsum(A,2)
rowsReverse = cumsum(A,2,'reverse')
```

```
rowsDirect =
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]
[ 1, 2, 3, 4]

rowsReverse =
[ 4, 3, 2, 1]
[ 4, 3, 2, 1]
[ 4, 3, 2, 1]
[ 4, 3, 2, 1]
```

Input Arguments

`A` — Input array

symbolic vector | symbolic matrix

Input array, specified as a vector or matrix.

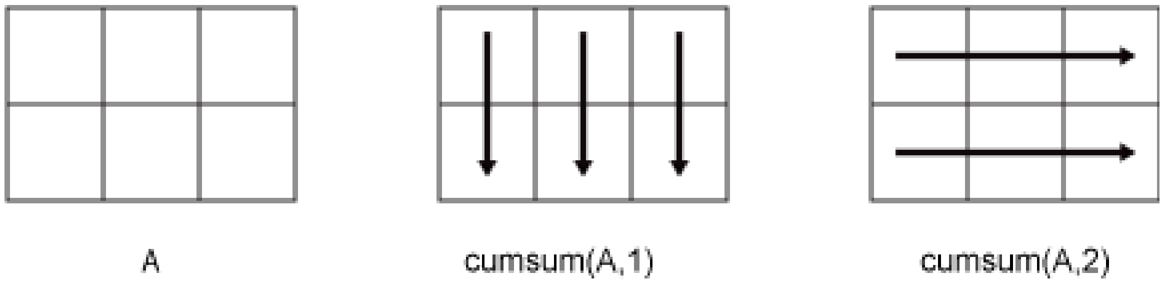
`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. The default value is 1.

Consider a two-dimensional input array, `A`:

- `cumsum(A,1)` works on successive elements in the columns of `A` and returns the cumulative sum of each column.
- `cumsum(A,2)` works on successive elements in the rows of `A` and returns the cumulative sum of each row.



`cumsum` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as the `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Output Arguments

`B` — Cumulative sum array
vector | matrix

Cumulative sum array, returned as a vector or matrix of the same size as the input `A`.

See Also

`cumprod` | `fold` | `int` | `symprod` | `symsum`

Introduced in R2013b

curl

Curl of vector field

Syntax

```
curl(V,X)
curl(V)
```

Description

`curl(V,X)` returns the curl of the vector field on page 4-310 `V` with respect to the vector `X`. The vector field `V` and the vector `X` are both three-dimensional.

`curl(V)` returns the curl of the vector field `V` with respect to the vector of variables returned by `symvar(V,3)`.

Input Arguments

V

Three-dimensional vector of symbolic expressions or symbolic functions.

X

Three-dimensional vector with respect to which you compute the curl.

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
```

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

$$curl(V) = \nabla \times V = \begin{pmatrix} \dfrac{\partial V_3}{\partial X_2} - \dfrac{\partial V_2}{\partial X_3} \\ \dfrac{\partial V_1}{\partial X_3} - \dfrac{\partial V_3}{\partial X_1} \\ \dfrac{\partial V_2}{\partial X_1} - \dfrac{\partial V_1}{\partial X_2} \end{pmatrix}$$

See Also

`diff` | `divergence` | `gradient` | `hessian` | `jacobian` | `laplacian` | `potential` | `vectorPotential`

Introduced in R2012a

daeFunction

Convert system of differential algebraic equations to MATLAB function handle suitable for `ode15i`

Syntax

```
f = daeFunction(eqs,vars)
f = daeFunction(eqs,vars,p1,...,pN)
f = daeFunction(___,Name,Value)
```

Description

`f = daeFunction(eqs,vars)` converts a system of symbolic first-order differential algebraic equations (DAEs) to a MATLAB function handle acceptable as an input argument to the numerical MATLAB DAE solver `ode15i`.

`f = daeFunction(eqs,vars,p1,...,pN)` lets you specify the symbolic parameters of the system as `p1,...,pN`.

`f = daeFunction(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Convert DAE System to Function Handle

Create the system of differential algebraic equations. Here, the symbolic functions `x1(t)` and `x2(t)` represent the state variables of the system. The system also contains constant symbolic parameters `a`, `b`, and the parameter function `r(t)`. These parameters do not represent state variables. Specify the equations and state variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x1(t) x2(t) a b r(t)
eqs = [diff(x1(t),t) == a*x1(t) + b*x2(t)^2,...
       x1(t)^2 + x2(t)^2 == r(t)^2];
vars = [x1(t), x2(t)];
```

Use `daeFunction` to generate a MATLAB function handle `f` depending on the variables `x1(t)`, `x2(t)` and on the parameters `a`, `b`, `r(t)`.

```
f = daeFunction(eqs, vars, a, b, r(t))
```

```
f =
  function_handle with value:
    @(t,in2,in3,param1,param2,param3)[in3(1,:)-param1.*in2(1,:)...
-param2.*in2(2,:).^2;-param3.^2+in2(1,:).^2+in2(2,:).^2]
```

Specify the parameter values, and create the reduced function handle `F` as follows.

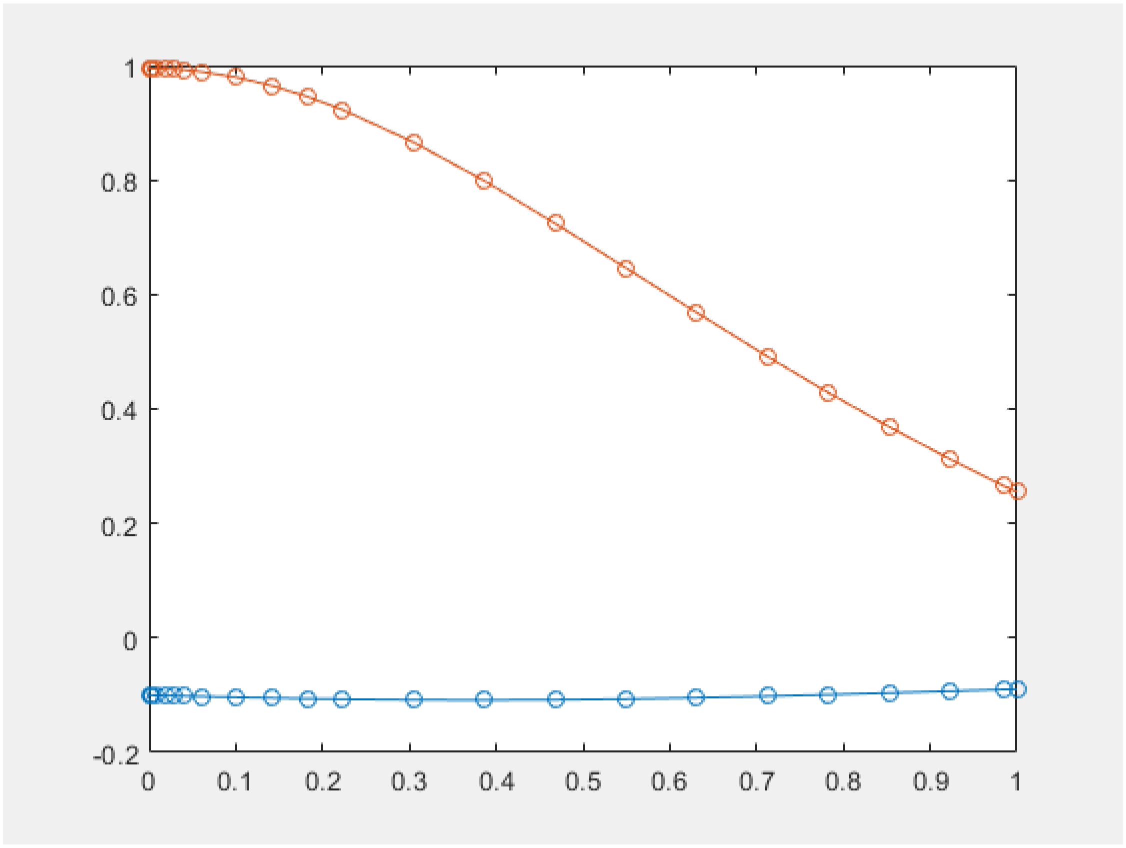
```
a = -0.6;
b = -0.1;
r = @(t) cos(t)/(1 + t^2);
F = @(t, Y, YP) f(t,Y,YP,a,b,r(t));
```

Specify consistent initial conditions for the DAE system.

```
t0 = 0;
y0 = [-r(t0)*sin(0.1); r(t0)*cos(0.1)];
yp0= [a*y0(1) + b*y0(2)^2; 1.234];
```

Now, use `ode15i` to solve the system of equations.

```
ode15i(F, [t0, 1], y0, yp0)
```



Write Function to File with Comments

Write the generated function handle to a file by specifying the `File` option. When writing to a file, `daeFunction` optimizes the code using intermediate variables named `t0`, `t1`, .… Include comments in the file using the `Comments` option.

Write the generated function handle to the file `myfile`.

```
syms x1(t) x2(t) a b r(t)
eqs = [diff(x1(t),t) == a*x1(t) + b*x2(t)^2,...
       x1(t)^2 + x2(t)^2 == r(t)^2];
vars = [x1(t), x2(t)];
daeFunction(eqs, vars, a, b, r(t), 'File', 'myfile')
```

```
function eqs = myfile(t,in2,in3,param1,param2,param3)
%MYFILE
%    EQS = MYFILE(T,IN2,IN3,PARAM1,PARAM2,PARAM3)

%    This function was generated by the Symbolic Math Toolbox version 7.3.
%    01-Jan-2017 00:00:00

YP1 = in3(1,:);
x1 = in2(1,:);
x2 = in2(2,:);
t2 = x2.^2;
eqs = [YP1-param2.*t2-param1.*x1;t2-param3.^2+x1.^2];
```

Include the comment `Version: 1.1`.

```
daeFunction(eqs, vars, a, b, r(t), 'File', 'myfile',...
                    'Comments','Version: 1.1');
```

```
function eqs = myfile(t,in2,in3,param4,param5,param6)
...
%Version: 1.1
YP3 = in3(1,:);
...
```

- "Solve Differential Algebraic Equations (DAEs)" on page 2-193

Input Arguments

`eqs` — System of first-order DAEs

vector of symbolic equations | vector of symbolic expressions

System of first-order DAEs, specified as a vector of symbolic equations or expressions. Here, expressions represent equations with zero right side.

`vars` — State variables

vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

`p1,...,pN` — Parameters of system
symbolic variables | symbolic functions | symbolic function calls | symbolic vector | symbolic matrix

Parameters of the system, specified as symbolic variables, functions, or function calls, such as `f(t)`. You can also specify parameters of the system as a vector or matrix of symbolic variables, functions, or function calls. If `eqs` contains symbolic parameters other than the variables specified in `vars`, you must specify these additional parameters as `p1,...,pN`.

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`.

Example: `daeFunction(eqns,vars,'File','myfile')`

`Comments` — Comments to include in file header
character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector.

`File` — Path to file containing generated code
character vector

Path to the file containing generated code, specified as a character vector. The generated file accepts arguments of type `double`, and can be used without Symbolic Math Toolbox. If the value is an empty character vector, `odeFunction` generates an anonymous function. If the character vector does not end in `.m`, the function appends `.m`.

By default, `daeFunction` with the `File` argument generates a file containing optimized code. 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`. To disable code optimization, use the `Optimize` argument.

`Optimize` — Flag preventing optimization of code written to function file
`true` (default) | `false`

Flag preventing optimization of code written to a function file, specified as `false` or `true`.

By default, `daeFunction` with the `File` argument generates a file containing optimized code. 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`.

`daeFunction` without the `File` argument (or with a file path specified by an empty character vector) creates a function handle. In this case, the code is not optimized. If you try to enforce code optimization by setting `Optimize` to `true`, then `daeFunction` throws an error.

`Sparse` — Flag that switches between sparse and dense matrix generation

`false` (default) | `true`

Flag that switches between sparse and dense matrix generation, specified as `true` or `false`. When you specify `'Sparse',true`, the generated function represents symbolic matrices by sparse numeric matrices. Use `'Sparse',true` when you convert symbolic matrices containing many zero elements. Often, operations on sparse matrices are more efficient than the same operations on dense matrices.

Output Arguments

`f` — Function handle that can serve as input argument to **`ode15i`**

MATLAB function handle

Function handle that can serve as input argument to `ode15i`, returned as a MATLAB function handle.

See Also

`decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `matlabFunction` | `ode15i` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

"Solve Differential Algebraic Equations (DAEs)" on page 2-193

Introduced in R2014b

dawson

Dawson integral

Syntax

```
dawson(X)
```

Description

`dawson(X)` represents the Dawson integral on page 4-322.

Examples

Dawson Integral for Numeric and Symbolic Arguments

Depending on its arguments, `dawson` returns floating-point or exact symbolic results.

Compute the Dawson integrals for these numbers. Because these numbers are not symbolic objects, `dawson` returns floating-point results.

```
A = dawson([-Inf, -3/2, -1, 0, 2, Inf])
```

```
A =
         0   -0.4282   -0.5381         0    0.3013         0
```

Compute the Dawson integrals for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `dawson` returns unresolved symbolic calls.

```
symA = dawson(sym([-Inf, -3/2, -1, 0, 2, Inf]))
```

```
symA =
[ 0, -dawson(3/2), -dawson(1), 0, dawson(2), 0]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

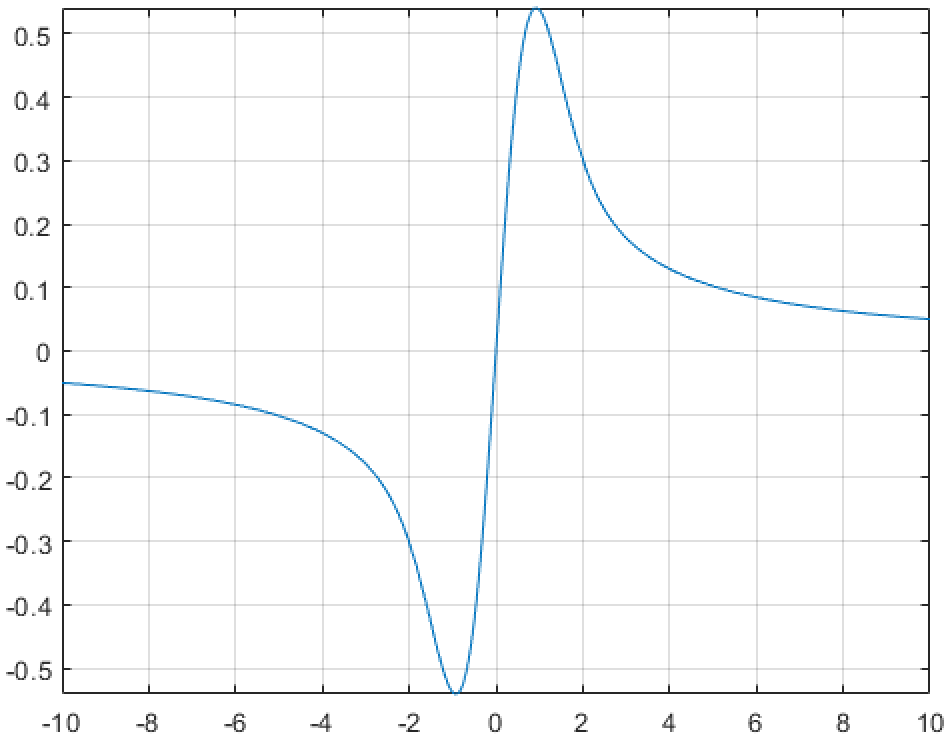
```
vpa(symA)
```

```
ans =
[ 0,...
-0.42824907108539862547719010515175,...
-0.53807950691276841913638742040756,...
0,...
0.30134038892379196603466443928642,...
0]
```

Plot the Dawson Integral

Plot the Dawson integral on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(dawson(x), [-10, 10])
grid on
```



Handle Expressions Containing Dawson Integral

Many functions, such as `diff` and `limit`, can handle expressions containing `dawson`.

Find the first and second derivatives of the Dawson integral:

```
syms x
diff(dawson(x), x)
diff(dawson(x), x, x)
```

```
ans =
1 - 2*x*dawson(x)
```

```
ans =
2*x*(2*x*dawson(x) - 1) - 2*dawson(x)
```

Find the limit of this expression involving `dawson`:

```
limit(x*dawson(x), Inf)

ans =
1/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Dawson Integral

The Dawson integral, also called the Dawson function, is defined as follows:

$$\operatorname{dawson}(x) = D(x) = e^{-x^2}\int_0^x e^{t^2}\,dt$$

Symbolic Math Toolbox uses this definition to implement `dawson`.

The alternative definition of the Dawson integral is

$$D(x) = e^{x^2}\int_0^x e^{-t^2}\,dt$$

Tips

- `dawson(0)` returns `0`.
- `dawson(Inf)` returns `0`.
- `dawson(-Inf)` returns `0`.

See Also

`erf` | `erfc`

Introduced in R2014a

decic

Find consistent initial conditions for first-order implicit ODE system with algebraic constraints

Syntax

```
[y0,yp0] = decic(eqs,vars,constraintEqs,t0,y0_est,fixedVars,yp0_est,
options)
```

Description

`[y0,yp0] = decic(eqs,vars,constraintEqs,t0,y0_est,fixedVars,yp0_est, options)` finds consistent initial conditions for the system of first-order implicit ordinary differential equations with algebraic constraints returned by the `reduceDAEToODE` function.

The call `[eqs,constraintEqs] = reduceDAEToODE(DA_eqs,vars)` reduces the system of differential algebraic equations `DA_eqs` to the system of implicit ODEs `eqs`. It also returns constraint equations encountered during system reduction. For the variables of this ODE system and their derivatives, `decic` finds consistent initial conditions `y0`, `yp0` at the time `t0`.

Substituting the numerical values `y0`, `yp0` into the differential equations `subs(eqs, [t; vars(t); diff(vars(t))], [t0; y0; yp0])` and the constraint equations `subs(constr, [t; vars(t); diff(vars(t))], [t0; y0; yp0])` produces zero vectors. Here, `vars` must be a column vector.

`y0_est` specifies numerical estimates for the values of the variables `vars` at the time `t0`, and `fixedVars` indicates the values in `y0_est` that must not change during the numerical search. The optional argument `yp0_est` lets you specify numerical estimates for the values of the derivatives of the variables `vars` at the time `t0`.

Examples

Find Consistent Initial Conditions for ODE System

Reduce the DAE system to a system of implicit ODEs. Then, find consistent initial conditions for the variables of the resulting ODE system and their first derivatives.

Create the following differential algebraic system.

```
syms x(t) y(t)
DA_eqs = [diff(x(t),t) == cos(t) + y(t),...
          x(t)^2 + y(t)^2 == 1];
vars = [x(t); y(t)];
```

Use `reduceDAEToODE` to convert this system to a system of implicit ODEs.

```
[eqs, constraintEqs] = reduceDAEToODE(DA_eqs, vars)
```

```
eqs =
                 diff(x(t), t) - y(t) - cos(t)
 - 2*x(t)*diff(x(t), t) - 2*y(t)*diff(y(t), t)

constraintEqs =
1 - y(t)^2 - x(t)^2
```

Create an option set that specifies numerical tolerances for the numerical search.

```
options = odeset('RelTol', 10.0^(-7), 'AbsTol', 10.0^(-7));
```

Fix values `t0 = 0` for the time and numerical estimates for consistent values of the variables and their derivatives.

```
t0 = 0;
y0_est = [0.1, 0.9];
yp0_est = [0.0, 0.0];
```

You can treat the constraint as an algebraic equation for the variable `x` with the fixed parameter `y`. For this, set `fixedVars = [0 1]`. Alternatively, you can treat it as an algebraic equation for the variable `y` with the fixed parameter `x`. For this, set `fixedVars = [1 0]`.

First, set the initial value `x(t0) = y0_est(1) = 0.1`.

```
fixedVars = [1 0];
[y0,yp0] = decic(eqs,vars,constraintEqs,t0,y0_est,fixedVars,yp0_est,options)

y0 =
    0.1000
    0.9950

yp0 =
    1.9950
   -0.2005
```

Now, change `fixedVars` to `[0 1]`. This fixes `y(t0) = y0_est(2) = 0.9`.

```
fixedVars = [0 1];
[y0,yp0] = decic(eqs,vars,constraintEqs,t0,y0_est,fixedVars,yp0_est,options)

y0 =
   -0.4359
    0.9000

yp0 =
    1.9000
    0.9202
```

Verify that these initial values are consistent initial values satisfying the equations and the constraints.

```
subs(eqs, [t; vars; diff(vars,t)], [t0; y0; yp0])

ans =
 0
 0

subs(constraintEqs, [t; vars; diff(vars,t)], [t0; y0; yp0])

ans =
0
```

Input Arguments

`eqs` — System of implicit ordinary differential equations

vector of symbolic equations | vector of symbolic expressions

System of implicit ordinary differential equations, specified as a vector of symbolic equations or expressions. Here, expressions represent equations with zero right side.

Typically, you use expressions returned by `reduceDAEToODE`.

`vars` — State variables of original DAE system

vector of symbolic functions | vector of symbolic function calls

State variables of original DAE system, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

`constraintEqs` — Constraint equations found by `reduceDAEToODE` during system reduction

vector of symbolic equations | vector of symbolic expressions

Constraint equations encountered during system reduction, specified as a vector of symbolic equations or expressions. These expressions or equations depend on the variables `vars`, but not on their derivatives.

Typically, you use constraint equations returned by `reduceDAEToODE`.

`t0` — Initial time

number

Initial time, specified as a number.

`y0_est` — Estimates for values of variables `vars` at initial time `t0`

numeric vector

Estimates for the values of the variables `vars` at the initial time `t0`, specified as a numeric vector.

`fixedVars` — Input vector indicating which elements of `y0_est` are fixed values

vector with elements `0` or `1`

Input vector indicating which elements of `y0_est` are fixed values, specified as a vector with `0`s or `1`s. Fixed values of `y0_est` correspond to values `1` in `fixedVars`. These values are not modified during the numerical search. The zero entries in `fixedVars` correspond to those variables in `y0_est` for which `decic` solves the constraint equations. The number of `0`s must coincide with the number of constraint equations. The Jacobian matrix of the constraints with respect to the variables `vars(fixedVars == 0)` must be invertible.

`yp0_est` — Estimates for values of first derivatives of variables `vars` at initial time `t0`
numeric vector

Estimates for the values of the first derivatives of the variables `vars` at the initial time `t0`, specified as a numeric vector.

`options` — Options for numerical search
options structure, returned by `odeset`

Options for numerical search, specified as an options structure, returned by `odeset`. For example, you can specify tolerances for the numerical search here.

Output Arguments

`y0` — Consistent initial values for variables
numeric column vector

Consistent initial values for variables, returned as a numeric column vector.

`yp0` — Consistent initial values for first derivatives of variables
numeric column vector

Consistent initial values for first derivatives of variables, returned as a numeric column vector.

See Also

`daeFunction` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

"Solve Differential Algebraic Equations (DAEs)" on page 2-193

Introduced in R2014b

derivedUnits

Derived units of unit system

Syntax

```
derivedUnits(unitSystem)
```

Description

`derivedUnits(unitSystem)` returns the derived units of the unit system `unitSystem` as a vector of symbolic units. You can use the returned units to create new unit systems by using `newUnitSystem`.

Examples

Derived Units of Unit System

Get the derived units of a unit system by using `derivedUnits`. By default, the available unit systems are `SI`, `CGS`, and `US`. Then, modify the derived units and create a new unit system using the modified derived units.

Get the derived units of the `SI` unit system.

```
dunits = derivedUnits('SI')
```

```
dunits =
[ [F], [C], [S], [H], [V], [J], [N], [lx], [lm], [Wb], [W], [Pa],...
 [Ohm], [T], [Gy], [Bq], [Sv], [Hz], [kat], [rad], [sr], [Celsius]]
```

Note Do not define a variable called `derivedUnits` because the variable will prevent access to the `derivedUnits` function.

Define derived units that use kilonewton for force and millibar for pressure by modifying `dunits` using `subs`.

```
u = symunit;
newUnits = subs(dunits,[u.N u.Pa],[u.kN u.mbar])
```

```
newUnits =
[ [F], [C], [S], [H], [V], [J], [kN], [lx], [lm], [Wb], [W], [mbar],...
 [Ohm], [T], [Gy], [Bq], [Sv], [Hz], [kat], [rad], [sr], [Celsius]]
```

Define the new unit system by using `newUnitSystem`. Keep the `SI` base units.

```
bunits = baseUnits('SI');
newUnitSystem('SI_kN_mbar',bunits,newUnits)
```

```
ans =
    "SI_kN_mbar"
```

To convert between unit systems, see “Unit Conversions and Unit Systems” on page 2-30.

- “Units of Measurement Tutorial” on page 2-5
- “Unit Conversions and Unit Systems” on page 2-30
- “Units List” on page 2-13

Input Arguments

`unitSystem` — Name of unit system

string | character vector

Name of the unit system, specified as a string or character vector.

See Also

`baseUnits` | `newUnitSystem` | `removeUnitSystem` | `rewrite` | `symunit` | `unitSystems`

Topics

“Units of Measurement Tutorial” on page 2-5
“Unit Conversions and Unit Systems” on page 2-30
“Units List” on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

det

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])

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)

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

r =
1/3
```

See Also

`eig` | `rank`

Introduced before R2006a

diag

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)
```

```
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)

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)

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)

ans =
 b
 3
```

See Also

`tril` | `triu`

Introduced before R2006a

diff

Differentiate symbolic expression or function

Syntax

```
diff(F)
diff(F,var)
diff(F,n)
diff(F,var,n)
diff(F,var1,...,varN)
```

Description

`diff(F)` differentiates `F` with respect to the variable determined by `symvar(F,1)`.

`diff(F,var)` differentiates `F` with respect to the variable `var`.

`diff(F,n)` computes the `n`th derivative of `F` with respect to the variable determined by `symvar`.

`diff(F,var,n)` computes the `n`th derivative of `F` with respect to the variable `var`.

`diff(F,var1,...,varN)` differentiates `F` with respect to the variables `var1,...,varN`.

Examples

Differentiate Function

Find the derivative of the function `sin(x^2)`.

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

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

Find the value of the derivative at `x = 2`. Convert the value to double.

```
df2 = df(2)

df2 =
4*cos(4)

double(df2)

ans =
   -2.6146
```

Differentiation with Respect to Particular Variable

Find the first derivative of this expression:

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

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

Because you did not specify the differentiation variable, `diff` uses the default variable defined by `symvar`. For this expression, the default variable is `x`:

```
symvar(sin(x*t^2),1)

ans =
x
```

Now, find the derivative of this expression with respect to the variable `t`:

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

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

Higher-Order Derivatives of Univariate Expression

Find the 4th, 5th, and 6th derivatives of this expression:

```
syms t
d4 = diff(t^6,4)
d5 = diff(t^6,5)
d6 = diff(t^6,6)
```

```
d4 =
360*t^2

d5 =
720*t

d6 =
720
```

Higher-Order Derivatives of Multivariate Expression with Respect to Particular Variable

Find the second derivative of this expression with respect to the variable `y`:

```
syms x y
diff(x*cos(x*y), y, 2)
```

```
ans =
-x^3*cos(x*y)
```

Higher-Order Derivatives of Multivariate Expression with Respect to Default Variable

Compute the second derivative of the expression `x*y`. If you do not specify the differentiation variable, `diff` uses the variable determined by `symvar`. For this expression, `symvar(x*y,1)` returns `x`. Therefore, `diff` computes the second derivative of `x*y` with respect to `x`.

```
syms x y
diff(x*y, 2)
```

```
ans =
0
```

If you use nested `diff` calls and do not specify the differentiation variable, `diff` determines the differentiation variable for each call. For example, differentiate the expression `x*y` by calling the `diff` function twice:

```
diff(diff(x*y))
```

```
ans =
1
```

In the first call, `diff` differentiate `x*y` with respect to `x`, and returns `y`. In the second call, `diff` differentiates `y` with respect to `y`, and returns `1`.

Thus, `diff(x*y, 2)` is equivalent to `diff(x*y, x, x)`, and `diff(diff(x*y))` is equivalent to `diff(x*y, x, y)`.

Mixed Derivatives

Differentiate this expression with respect to the variables `x` and `y`:

```
syms x y
diff(x*sin(x*y), x, y)
```

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

You also can compute mixed higher-order derivatives by providing all differentiation variables:

```
syms x y
diff(x*sin(x*y), x, x, x, y)
```

```
ans =
x^2*y^3*sin(x*y) - 6*x*y^2*cos(x*y) - 6*y*sin(x*y)
```

Input Arguments

`F` — Expression or function to differentiate

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Expression or function to differentiate, specified as a symbolic expression or function or as a vector or matrix of symbolic expressions or functions. If `F` is a vector or a matrix, `diff` differentiates each element of `F` and returns a vector or a matrix of the same size as `F`.

`var` — Differentiation variable

symbolic variable

Differentiation variable, specified as a symbolic variable.

`var1,...,varN` — **Differentiation variables**
symbolic variables

Differentiation variables, specified as symbolic variables.

`n` — **Differentiation order**
nonnegative integer

Differentiation order, specified as a nonnegative integer.

Tips

- When computing mixed higher-order derivatives, do not use `n` to specify the differentiation order. Instead, specify all differentiation variables explicitly.
- To improve performance, `diff` assumes that all mixed derivatives commute. For example,

  $$\frac{\partial}{\partial x}\frac{\partial}{\partial y}f(x,y)=\frac{\partial}{\partial y}\frac{\partial}{\partial x}f(x,y)$$

  This assumption suffices for most engineering and scientific problems.
- If you differentiate a multivariate expression or function `F` without specifying the differentiation variable, then a nested call to `diff` and `diff(F,n)` can return different results. This is because in a nested call, each differentiation step determines and uses its own differentiation variable. In calls like `diff(F,n)`, the differentiation variable is determined once by `symvar(F,1)` and used for all differentiation steps.
- If you differentiate an expression or function containing `abs` or `sign`, ensure that the arguments are real values. For complex arguments of `abs` and `sign`, the `diff` function formally computes the derivative, but this result is not generally valid because `abs` and `sign` are not differentiable over complex numbers.

See Also

`curl` | `divergence` | `functionalDerivative` | `gradient` | `hessian` | `int` | `jacobian` | `laplacian` | `symvar`

Topics

“Differentiation” on page 2-39
“Find Asymptotes, Critical and Inflection Points” on page 2-79

Introduced before R2006a

digits

Change variable precision used

Syntax

```
digits(d)
d1 = digits
d1 = digits(d)
```

Description

`digits(d)` sets the precision used by `vpa` to `d` significant decimal digits. The default is 32 digits.

`d1 = digits` returns the current precision used by `vpa`.

`d1 = digits(d)` sets the new precision `d` and returns the old precision in `d1`.

Examples

Increase Precision of Results

By default, MATLAB uses 16 digits of precision. For higher precision, use `vpa`. The default precision for `vpa` is 32 digits. Increase precision beyond 32 digits by using `digits`.

Find `pi` using `vpa`, which uses the default 32 digits of precision. Confirm that the current precision is 32 by using `digits`.

```
pi32 = vpa(pi)
```

```
pi32 =
3.1415926535897932384626433832795
```

```
currentPrecision = digits
currentPrecision =
    32
```

Save the current value of `digits` in `digitsOld` and set the new precision to `100` digits. Find `pi` using `vpa`. The result has 100 digits.

```
digitsOld = digits(100);
pi100 = vpa(pi)
```

```
pi100 =
3.1415926535897932384626433832795028841971693993751058209...
749445923078164062862089986280348253421170068
```

Note `vpa` output is symbolic. To use symbolic output with a MATLAB function that does not accept symbolic values, convert symbolic values to double precision by using `double`.

Lastly, restore the old value of `digits` for further calculations.

```
digits(digitsOld)
```

For more information, see “Increase Precision of Numeric Calculations” on page 2-116.

Increase Speed by Decreasing Precision

Increase the speed of MATLAB calculations by using `vpa` with a lower precision. Set the lower precision by using `digits`.

First, find the time taken to perform an operation on a large input.

```
input = 1:0.01:500;
tic
zeta(input);
toc
```

```
Elapsed time is 48.968983 seconds.
```

Now, repeat the operation with a lower precision by using `vpa`. Lower the precision to `10` digits by using `digits`. Then, use `vpa` to reduce the precision of `input` and perform the same operation. The time taken decreases significantly.

```
digitsOld = digits(10);
vpaInput = vpa(input);
```

```
tic
zeta(vpaInput);
toc

Elapsed time is 31.450342 seconds.
```

Note `vpa` output is symbolic. To use symbolic output with a MATLAB function that does not accept symbolic values, convert symbolic values to double precision by using `double`.

Lastly, restore the old value of `digits` for further calculations.

```
digits(digitsOld)
```

For more information, see "Increase Speed by Reducing Precision" on page 2-123.

Guard Digits

The number of digits that you specify using the `vpa` function or the `digits` function is the guaranteed number of digits. Internally, the toolbox can use a few 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 four digits:

```
old = 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 four 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

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

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

```
a =
0.1

b =
0.1
```

Now, compute the difference `a - b`. The result is not 0:

```
a - b
```

```
ans =
0.000000000000000000086736173798840354720600815844403
```

The difference `a - b` is not equal to zero because the toolbox internally boosts the 10-digit number `b = 0.1` to 32-digit accuracy. This process implies round-off errors. The toolbox actually computes the difference `a - b` as follows:

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

```
b =
0.09999999999999999999132638262011

ans =
0.000000000000000000086736173798840354720600815844403
```

Techniques Used to Convert Floating-Point Numbers to Symbolic Objects

Suppose you convert a double 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 $\pi = 3.141592653589793...$ 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
```

Although the toolbox displays these numbers differently on the screen, they are rational approximations of `pi`. Use `vpa` to convert these rational approximations of `pi` back to floating-point values.

Set the number of digits to 4. Three of the four 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 differences between the symbolic approximations of `pi` become more visible.

```
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
```

Input Arguments

`d` — New accuracy setting

number | symbolic number

New accuracy setting, specified as a number or symbolic number. The setting specifies the number of significant decimal digits to be used for variable-precision calculations. If the value `d` is not an integer, `digits` rounds it to the nearest integer.

Output Arguments

`d1` — Current accuracy setting

double-precision number

Current accuracy setting, returned as a double-precision number. The setting specifies the number of significant decimal digits currently used for variable-precision calculations.

See Also

`double` | `vpa`

Topics

"Increase Precision of Numeric Calculations" on page 2-116
"Recognize and Avoid Round-Off Errors" on page 2-118
"Increase Speed by Reducing Precision" on page 2-123

Introduced before R2006a

dilog

Dilogarithm function

Syntax

```
dilog(X)
```

Description

`dilog(X)` returns the dilogarithm function.

Examples

Dilogarithm Function for Numeric and Symbolic Arguments

Depending on its arguments, `dilog` returns floating-point or exact symbolic results.

Compute the dilogarithm function for these numbers. Because these numbers are not symbolic objects, `dilog` returns floating-point results.

```
A = dilog([-1, 0, 1/4, 1/2, 1, 2])
```

```
A =
   2.4674 - 2.1776i   1.6449 + 0.0000i   0.9785 + 0.0000i...
   0.5822 + 0.0000i   0.0000 + 0.0000i  -0.8225 + 0.0000i
```

Compute the dilogarithm function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `dilog` returns unresolved symbolic calls.

```
symA = dilog(sym([-1, 0, 1/4, 1/2, 1, 2]))
```

```
symA =
[ pi^2/4 - pi*log(2)*1i, pi^2/6, dilog(1/4), pi^2/12 - log(2)^2/2, 0, -pi^2/12]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

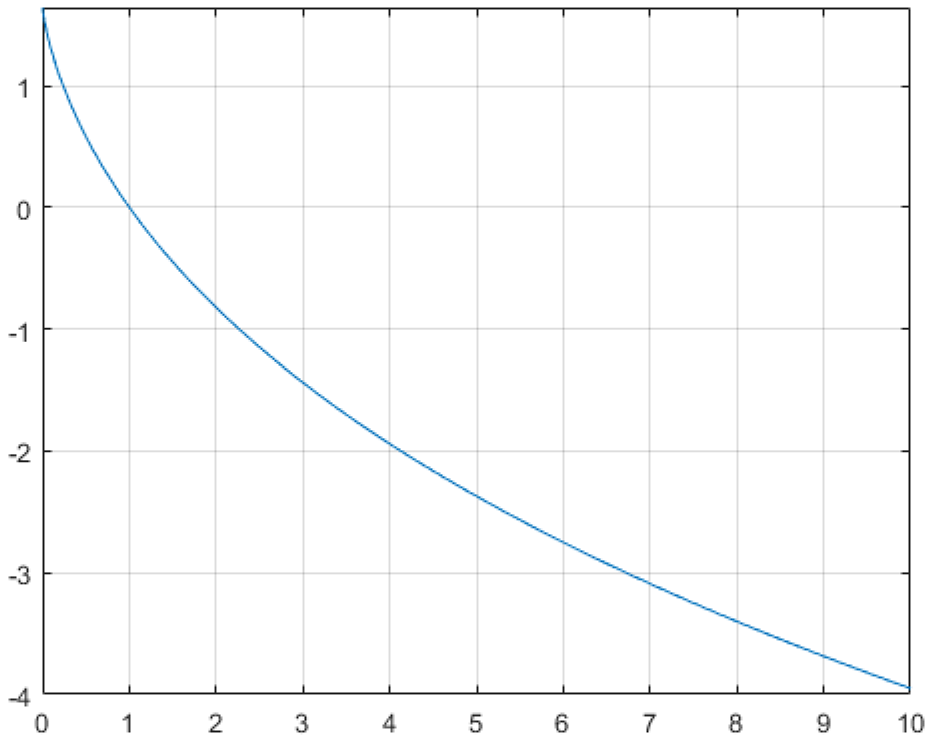
```
vpa(symA)
```

```
ans =
[ 2.467401100272339654708622749969 - 2.1775860903036021305006888982376i,...
1.644934066848226436472415166646,...
0.9784693929303061037430666652456,...
0.5822405264650125059026563201596 8,...
0,...
-0.8224670334241132182362075833230 1]
```

Plot Dilogarithm Function

Plot the dilogarithm function on the interval from 0 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(dilog(x), [0, 10])
grid on
```



Handle Expressions Containing Dilogarithm Function

Many functions, such as `diff`, `int`, and `limit`, can handle expressions containing `dilog`.

Find the first and second derivatives of the dilogarithm function:

```
syms x
diff(dilog(x), x)
diff(dilog(x), x, x)
```

```
ans =
-log(x)/(x - 1)
```

```
ans =
log(x)/(x - 1)^2 - 1/(x*(x - 1))
```

Find the indefinite integral of the dilogarithm function:

```
int(dilog(x), x)

ans =
x*(dilog(x) + log(x) - 1) - dilog(x)
```

Find the limit of this expression involving `dilog`:

```
limit(dilog(x)/x, Inf)

ans =
0
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Dilogarithm Function

There are two common definitions of the dilogarithm function.

The implementation of the `dilog` function uses the following definition:

$$\operatorname{dilog}(x) = \int_1^x \frac{\ln(t)}{1-t}\,dt$$

Another common definition of the dilogarithm function is

$$\mathrm{Li}_2(x) = \int_x^0 \frac{\ln(1-t)}{t}\,dt$$

Thus, dilog(x) = $\mathrm{Li}_2(1-x)$.

Tips

- `dilog(sym(-1))` returns `pi^2/4 - pi*log(2)*i`.
- `dilog(sym(0))` returns `pi^2/6`.
- `dilog(sym(1/2))` returns `pi^2/12 - log(2)^2/2`.
- `dilog(sym(1))` returns `0`.
- `dilog(sym(2))` returns `-pi^2/12`.
- `dilog(sym(i))` returns `pi^2/16 - (pi*log(2)*i)/4 - catalan*i`.
- `dilog(sym(-i))` returns `catalan*i + (pi*log(2)*i)/4 + pi^2/16`.
- `dilog(sym(1 + i))` returns `- catalan*i - pi^2/48`.
- `dilog(sym(1 - i))` returns `catalan*i - pi^2/48`.
- `dilog(sym(Inf))` returns `-Inf`.

References

[1] Stegun, I. A. “Miscellaneous Functions.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`log` | `zeta`

Introduced in R2014a

dirac

Dirac delta function

Syntax

```
dirac(x)
dirac(n,x)
```

Description

`dirac(x)` represents the Dirac delta function on page 4-356 of `x`.

`dirac(n,x)` represents the `n`th derivative of the Dirac delta function at `x`.

Examples

Handle Expressions Involving Dirac and Heaviside Functions

Compute derivatives and integrals of expressions involving the Dirac delta and Heaviside functions.

Find the first and second derivatives of the Heaviside function. The result is the Dirac delta function and its first derivative.

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

```
ans =
dirac(x)

ans =
dirac(1, x)
```

Find the indefinite integral of the Dirac delta function. The results returned by `int` do not include integration constants.

```
int(dirac(x), x)
```

```
ans =
sign(x)/2
```

Find the integral of this expression involving the Dirac delta function.

```
syms a
int(dirac(x - a)*sin(x), x, -Inf, Inf)
```

```
ans =
sin(a)
```

Use Assumptions on Variables

`dirac` takes into account assumptions on variables.

```
syms x real
assumeAlso(x ~= 0)
dirac(x)
```

```
ans =
0
```

For further computations, clear the assumptions.

```
syms x clear
```

Evaluate Dirac delta Function for Symbolic Matrix

Compute the Dirac delta function of `x` and its first three derivatives.

Use a vector `n = [0, 1, 2, 3]` to specify the order of derivatives. The `dirac` function expands the scalar into a vector of the same size as `n` and computes the result.

```
n = [0, 1, 2, 3];
d = dirac(n, x)
```

```
d =
[ dirac(x), dirac(1, x), dirac(2, x), dirac(3, x)]
```

Substitute `x` with `0`.

```
subs(d, x, 0)
```

```
ans =
[ Inf, -Inf, Inf, -Inf]
```

Input Arguments

x — Input

number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix | multidimensional array

Input, specified as a number, symbolic number, variable, expression, or function, representing a real number. This input can also be a vector, matrix, or multidimensional array of numbers, symbolic numbers, variables, expressions, or functions.

n — Order of derivative

nonnegative number | symbolic variable | symbolic expression | symbolic function | vector | matrix | multidimensional array

Order of derivative, specified as a nonnegative number, or symbolic variable, expression, or function representing a nonnegative number. This input can also be a vector, matrix, or multidimensional array of nonnegative numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Dirac delta Function

The Dirac delta function, $\delta(x)$, has the value 0 for all $x \neq 0$, and ∞ for $x = 0$.

For any smooth function f and a real number a,

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

Tips

- For complex values `x` with nonzero imaginary parts, `dirac` returns `NaN`.
- `dirac` returns floating-point results for numeric arguments that are not symbolic objects.
- `dirac` acts element-wise on nonscalar inputs.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, then `dirac` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

See Also

`heaviside` | `kroneckerDelta`

Introduced before R2006a

disp

Display symbolic input

Syntax

```
disp(X)
```

Description

`disp(X)` displays the symbolic input `X`. `disp` does not display the argument's name.

Examples

Display Symbolic Scalar

```
syms x
y = x^3 - exp(x);
disp(y)
```

```
x^3 - exp(x)
```

Display Symbolic Matrix

```
A = sym('a%d%d',[3 3]);
disp(A)
```

```
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
```

Display Symbolic Function

```
syms f(x)
f(x) = x+1;
disp(f)
```

```
x + 1
symbolic function inputs: x
```

Display Sentence with Text and Symbolic Expressions

Display the sentence “Euler’s formula is $e^{ix} = \cos(x) + i\sin(x)$”.

To concatenate character vectors with symbolic expressions, convert the symbolic expressions to character vectors using `char`.

```
syms x
disp(['Euler''s formula is ',char(exp(i*x)),' = ',char(cos(x)+i*sin(x)),'.'])
```

```
Euler's formula is exp(x*1i) = cos(x) + sin(x)*1i.
```

Because `'` terminates the character vector, repeat it in `Euler''s` for MATLAB to interpret it as an apostrophe and not a character vector terminator.

Input Arguments

x — Symbolic input to display

symbolic variable | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array | symbolic expression

Symbolic input to display, specified as a symbolic variable, vector, matrix, function, multidimensional array, or expression.

See Also

`char` | `disp` | `display` | `pretty`

Introduced before R2006a

display

Display symbolic input

Syntax

```
display(X)
```

Description

`display(X)` displays the symbolic input `X`.

Examples

Display Symbolic Scalar

```
syms x
y = x^3 - exp(x);
display(y)

y =
x^3 - exp(x)
```

Display Symbolic Matrix

```
A = sym('a%d%d',[3 3]);
display(A)

A =
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
```

Display Symbolic Function

```
syms f(x)
f(x) = x+1;
display(f)
```

```
f(x) =
x + 1
```

Display Sentence with Text and Symbolic Expressions

Display the sentence “Euler’s formula is $e^{ix} = \cos(x) + i\sin(x)$”.

To concatenate character vectors with symbolic expressions, convert the symbolic expressions to character vectors using `char`.

```
syms x
display(['Euler''s formula is ',char(exp(i*x)),' = ',char(cos(x)+i*sin(x)),'.'])
```

```
Euler's formula is exp(x*1i) = cos(x) + sin(x)*1i.
```

Because `'` terminates the character vector, you need to repeat it in `Euler''s` for MATLAB to interpret it as an apostrophe and not a character vector terminator.

Input Arguments

x — Symbolic input to display
symbolic variable | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array | symbolic expression

Symbolic input to display, specified as a symbolic variable, vector, matrix, function, multidimensional array, or expression.

See Also

`char` | `disp` | `display` | `pretty`

Introduced before R2006a

divergence

Divergence of vector field

Syntax

```
divergence(V,X)
```

Description

`divergence(V,X)` returns the divergence of vector field on page 4-362 `V` with respect to the vector `X` in Cartesian coordinates. Vectors `V` and `X` must have the same length.

Examples

Find Divergence of Vector Field

Find 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
```

Find 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
```

Find 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
```

Find Electric Charge Density from Electric Field

Gauss' Law in differential form states that the divergence of electric field is proportional to the electric charge density as

$$\vec{\nabla}.\vec{E}(\vec{r}) = \frac{\rho(\vec{r})}{\varepsilon_0}.$$

Find the electric charge density for the electric field $\overrightarrow{E} = x^2\vec{i} + y^2\vec{j}$.

```
syms x y ep0
E = [x^2 y^2];
rho = divergence(E,[x y])*ep0
```

```
rho =
ep0*(2*x + 2*y)
```

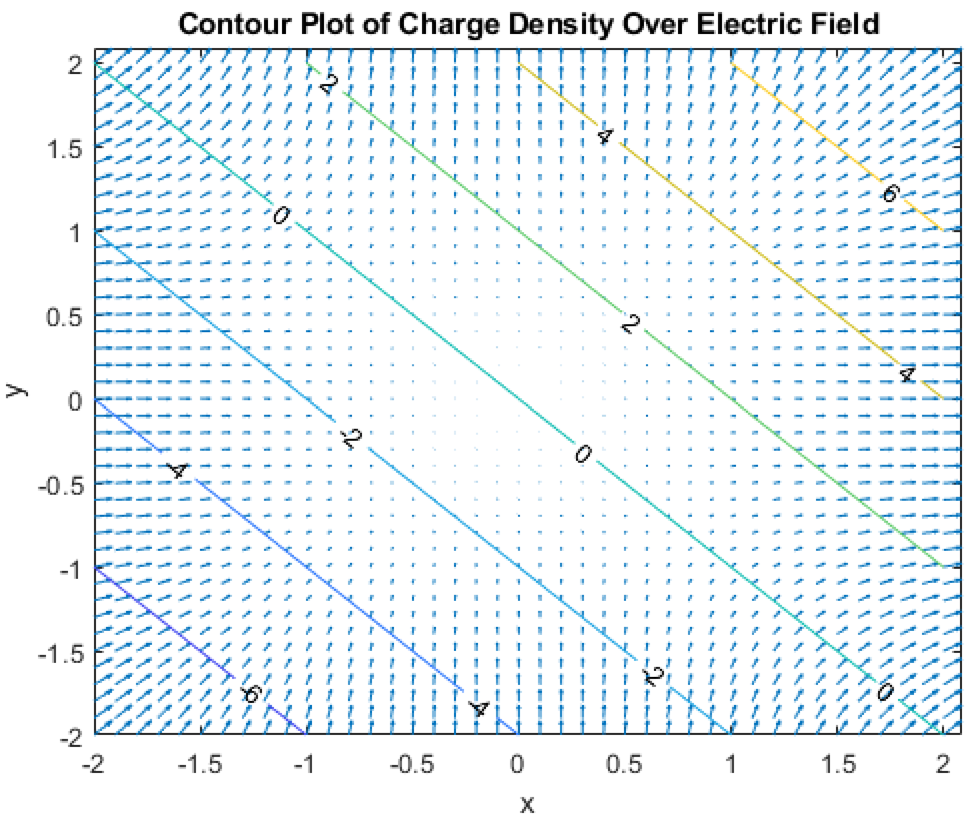
Visualize the electric field and electric charge density for `-2 < x < 2` and `-2 < y < 2` with `ep0 = 1`. Create a grid of values of `x` and `y` using `meshgrid`. Find the values of electric field and charge density by substituting grid values using `subs`. To simultaneously substitute the grid values `xPlot` and `yPlot` into the charge density `rho`, use cells arrays as inputs to `subs`.

```
rho = subs(rho,ep0,1);
v = -2:0.1:2;
[xPlot,yPlot] = meshgrid(v);
Ex = subs(E(1),x,xPlot);
Ey = subs(E(2),y,yPlot);
rhoPlot = double(subs(rho,{x,y},{xPlot,yPlot}));
```

Plot the electric field using `quiver`. Overlay the charge density using `contour`. The contour lines indicate the values of the charge density.

```
quiver(xPlot,yPlot,Ex,Ey)
hold on
```

```
contour(xPlot,yPlot,rhoPlot,'ShowText','on')
title('Contour Plot of Charge Density Over Electric Field')
xlabel('x')
ylabel('y')
```



Input Arguments

v — Vector field

symbolic expression | symbolic function | vector of symbolic expressions | vector of symbolic functions

Vector field to find divergence of, specified as a symbolic expression or function, or as a vector of symbolic expressions or functions. `V` must be the same length as `X`.

x — Variables with respect to which you find the divergence
symbolic variable | vector of symbolic variables

Variables with respect to which you find the divergence, specified as a symbolic variable or a vector of symbolic variables. `X` must be the same length as `V`.

Definitions

Divergence of Vector Field

The divergence of the vector field $V = (V_1,\dots,V_n)$ with respect to the vector $X = (X_1,\dots,X_n)$ in Cartesian coordinates is the sum of partial derivatives of V with respect to $X_1,\dots,X_n$

$$div(\vec{V}) = \nabla \cdot \vec{V} = \sum_{i=1}^{n} \frac{\partial V_i}{\partial x_i}.$$

See Also

`curl` | `diff` | `gradient` | `hessian` | `jacobian` | `laplacian` | `potential` | `vectorPotential`

Introduced in R2012a

divisors

Divisors of integer or expression

Syntax

```
divisors(n)
divisors(expr,vars)
```

Description

`divisors(n)` finds all nonnegative divisors of an integer `n`.

`divisors(expr,vars)` finds the divisors of a polynomial expression `expr`. Here, `vars` are polynomial variables.

Examples

Divisors of Integers

Find all nonnegative divisors of these integers.

Find the divisors of integers. You can use double precision numbers or numbers converted to symbolic objects. If you call `divisors` for a double-precision number, then it returns a vector of double-precision numbers.

```
divisors(42)
```

```
ans =
     1     2     3     6     7    14    21    42
```

Find the divisors of negative integers. `divisors` returns nonnegative divisors for negative integers.

```
divisors(-42)
```

```
ans =
     1     2     3     6     7    14    21    42
```

If you call `divisors` for a symbolic number, it returns a symbolic vector.

```
divisors(sym(42))
```

```
ans =
[ 1, 2, 3, 6, 7, 14, 21, 42]
```

The only divisor of `0` is `0`.

```
divisors(0)
```

```
ans =
     0
```

Divisors of Univariate Polynomials

Find the divisors of univariate polynomial expressions.

Find the divisors of this univariate polynomial. You can specify the polynomial as a symbolic expression.

```
syms x
divisors(x^4 - 1, x)
```

```
ans =
[ 1, x - 1, x + 1, (x - 1)*(x + 1), x^2 + 1, (x^2 + 1)*(x - 1),...
(x^2 + 1)*(x + 1), (x^2 + 1)*(x - 1)*(x + 1)]
```

You also can use a symbolic function to specify the polynomial.

```
syms f(t)
f(t) = t^5;
divisors(f,t)
```

```
ans(t) =
[ 1, t, t^2, t^3, t^4, t^5]
```

When finding the divisors of a polynomial, `divisors` does not return the divisors of the constant factor.

```
f(t) = 9*t^5;
divisors(f,t)
```

```
ans(t) =
[ 1, t, t^2, t^3, t^4, t^5]
```

Divisors of Multivariate Polynomials

Find the divisors of multivariate polynomial expressions.

Find the divisors of the multivariate polynomial expression. Suppose that `u` and `v` are variables, and `a` is a symbolic parameter. Specify the variables as a symbolic vector.

```
syms a u v
divisors(a*u^2*v^3, [u,v])
```

```
ans =
[ 1, u, u^2, v, u*v, u^2*v, v^2, u*v^2, u^2*v^2, v^3, u*v^3, u^2*v^3]
```

Now, suppose that this expression contains only one variable (for example, `v`), while `a` and `u` are symbolic parameters. Here, `divisors` treats the expression `a*u^2` as a constant and ignores it, returning only the divisors of `v^3`.

```
divisors(a*u^2*v^3, v)
```

```
ans =
[ 1, v, v^2, v^3]
```

Input Arguments

`n` — Number for which to find divisors
number | symbolic number

Number for which to find the divisors, specified as a number or symbolic number.

`expr` — Polynomial expression for which to find divisors
symbolic expression | symbolic function

Polynomial expression for which to find divisors, specified as a symbolic expression or symbolic function.

`vars` — Polynomial variables
symbolic variable | vector of symbolic variables

Polynomial variables, specified as a symbolic variable or a vector of symbolic variables.

Tips

- `divisors(0)` returns `0`.
- `divisors(expr,vars)` does not return the divisors of the constant factor when finding the divisors of a polynomial.
- If you do not specify polynomial variables, `divisors` returns as many divisors as it can find, including the divisors of constant symbolic expressions. For example, `divisors(sym(pi)^2*x^2)` returns `[ 1, pi, pi^2, x, pi*x, pi^2*x, x^2, pi*x^2, pi^2*x^2]` while `divisors(sym(pi)^2*x^2, x)` returns `[ 1, x, x^2]`.
- For rational numbers, `divisors` returns all divisors of the numerator divided by all divisors of the denominator. For example, `divisors(sym(9/8))` returns `[ 1, 3, 9, 1/2, 3/2, 9/2, 1/4, 3/4, 9/4, 1/8, 3/8, 9/8]`.

See Also

`coeffs` | `factor` | `numden`

Introduced in R2014b

doc

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.

Introduced in R2008b

double

Convert symbolic values to MATLAB double precision

Syntax

```
double(s)
```

Description

`double(s)` converts the symbolic value `s` to double precision. Converting symbolic values to double precision is useful when a MATLAB function does not accept symbolic values. For differences between symbolic and double-precision numbers, see “Choose Symbolic or Numeric Arithmetic” on page 2-114.

Examples

Convert Symbolic Number to Double Precision

Convert symbolic numbers to double precision by using `double`. Symbolic numbers are exact while double-precision numbers have round-off errors.

Convert `pi` and `1/3` from symbolic form to double precision.

```
symN = sym([pi 1/3])
```

```
symN =
[ pi, 1/3]
```

```
doubleN = double(symN)
```

```
doubleN =
    3.1416    0.3333
```

For information on round-off errors, see “Recognize and Avoid Round-Off Errors” on page 2-118.

Convert Variable Precision to Double Precision

Variable-precision numbers created by `vpa` are symbolic values. When a MATLAB function does not accept symbolic values, convert variable precision to double precision by using `double`.

Convert `pi` and `1/3` from variable-precision form to double precision.

```
vpaN = vpa([pi 1/3])

vpaN =
[ 3.1415926535897932384626433832795, 0.33333333333333333333333333333333]

doubleN = double(vpaN)

doubleN =
    3.1416    0.3333
```

Convert Symbolic Matrix to Double-Precision Matrix

Convert the symbolic numbers in matrix `symM` to double-precision numbers by using `double`.

```
a = sym(sqrt(2));
b = sym(2/3);
symM = [a b; a*b b/a]

symM =
[        2^(1/2),        2/3]
[ (2*2^(1/2))/3, 2^(1/2)/3]

doubleM = double(symM)

doubleM =
    1.4142    0.6667
    0.9428    0.4714
```

High-Precision Conversion

When converting symbolic expressions that suffer from internal cancelation or round-off errors, increase the working precision by using `digits` before converting the number.

Convert a numerically unstable expression `Y` with `double`. Then, increase precision to `100` digits by using `digits` and convert `Y` again. This high-precision conversion is accurate while the low-precision conversion is not.

```
Y = ((exp(sym(200)) + 1)/(exp(sym(200)) - 1)) - 1;
lowPrecisionY = double(Y)

lowPrecisionY =
     0

digitsOld = digits(100);
highPrecisionY = double(Y)

highPrecisionY =
   2.7678e-87
```

Restore the old precision used by `digits` for further calculations.

```
digits(digitsOld)
```

Input Arguments

s — Symbolic input
symbolic number | vector of symbolic numbers | matrix of symbolic numbers | multidimensional array of symbolic numbers

Symbolic input, specified as a symbolic number, or a vector, matrix, or multidimensional array of symbolic numbers.

See Also

`sym` | `vpa`

Topics

“Choose Symbolic or Numeric Arithmetic” on page 2-114
“Increase Precision of Numeric Calculations” on page 2-116
“Recognize and Avoid Round-Off Errors” on page 2-118
“Increase Speed by Reducing Precision” on page 2-123

Introduced before R2006a

dsolve

Differential equations and systems solver

Note Character vector inputs will be removed in a future release. Instead, use `syms` to declare variables and replace inputs such as `dsolve('Dy = y')` with `syms y(t); dsolve(diff(y,t) == y)`.

Syntax

```
S = dsolve(eqn)
S = dsolve(eqn,cond)
S = dsolve(eqn,cond,Name,Value)

[y1,...,yN] = dsolve(___)
```

Description

`S = dsolve(eqn)` solves the differential equation `eqn`, where `eqn` is a symbolic equation. Use `diff` and == to represent differential equations. For example, `diff(y,x) == y` represents the equation `dy/dx=y`. Solve a system of differential equations by specifying `eqn` as a vector of those equations.

`S = dsolve(eqn,cond)` solves `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.

`[y1,...,yN] = dsolve(___)` assigns the solutions to the variables `y1,...,yN`.

Examples

Solve Differential Equation

Specify a differential equation by using == and represent differentiation by using the `diff` function. Then, solve the equation by using `dsolve`.

Solve the equation $\frac{dy}{dt} = ay$.

```
syms a y(t)
eqn = diff(y,t) == a*y;
dsolve(eqn)
```

```
ans =
C2*exp(a*t)
```

`C2` is a constant. You can eliminate constants by specifying conditions. See “Solve Differential Equation with Condition” on page 4-377.

For more examples, see “Solve Differential Equation” on page 2-183. For a complex workflow involving differential equations, see “Solving Partial Differential Equations”.

Solve Higher-Order Differential Equation

Specify the second-order derivative of a function `y` by using `diff(y,t,2)` or `diff(y,t,t)`. Similarly, specify the n-th order derivative by using `diff(y,t,n)`.

Solve the equation $\frac{d^2y}{dt^2} = ay$.

```
syms y(t) a
eqn = diff(y,t,2) == a*y;
ySol(t) = dsolve(eqn)
```

```
ySol(t) =
C5*exp(a^(1/2)*t) + C6*exp(-a^(1/2)*t)
```

`C5` and `C6` are constants. You can eliminate constants by specifying conditions. See “Solve Differential Equation with Condition” on page 4-377.

Solve Differential Equation with Condition

Specify conditions as the second input to `dsolve` by using the == operator. Specifying conditions eliminates arbitrary constants, such as `C1`, `C2`,`...` from the solution.

Solve the equation $\frac{dy}{dt} = ay$ with the condition $y(0) = 5$.

```
syms y(t) a
eqn = diff(y,t) == a*y;
cond = y(0) == 5;
ySol(t) = dsolve(eqn,cond)
```

```
ySol(t) =
5*exp(a*t)
```

Solve the second-order differential equation $\frac{d^2y}{dt^2} = a^2y$ with two conditions, $y(0) = b$ and $y'(0) = 1$. Create the second condition by assigning `diff(y,t)` to `Dy` and then using `Dy(0) == 1`.

```
syms y(t) a b
eqn = diff(y,t,2) == a^2*y;
Dy = diff(y,t);
cond = [y(0)==b, Dy(0)==1];
ySol(t) = dsolve(eqn,cond)
```

```
ySol(t) =
(exp(a*t)*(a*b + 1))/(2*a) + (exp(-a*t)*(a*b - 1))/(2*a)
```

Since two conditions are specified here, constants are eliminated from the solution of the second-order equation. In general, to eliminate constants from the solution, the number of conditions must equal the order of the equation.

Solve System of Differential Equations

Solve a system of differential equations by specifying the equations as a vector. `dsolve` returns a structure containing the solutions.

Solve the system of equations

$$\frac{dy}{dt} = z$$
$$\frac{dz}{dt} = -y.$$

```
syms y(t) z(t)
eqns = [diff(y,t)==z, diff(z,t)==-y];
sol = dsolve(eqns)
```

```
sol = 
  struct with fields:

    z: [1×1 sym]
    y: [1×1 sym]
```

Access the solutions by addressing the elements of the structure.

```
soly(t) = sol.y
```

```
soly(t) =
C2*cos(t) + C1*sin(t)
```

```
solz(t) = sol.z
```

```
solz(t) =
C1*cos(t) - C2*sin(t)
```

Assign Outputs to Functions or Variables

When solving for multiple functions, `dsolve` returns a structure by default. Alternatively, you can directly assign solutions to functions or variables by specifying the outputs explicitly as a vector. `dsolve` sorts outputs in alphabetical order using `symvar`.

Solve a system of differential equations and assign the outputs to functions.

```
syms y(t) z(t)
eqns = [diff(y,t)==z, diff(z,t)==-y];
[ySol(t) zSol(t)] = dsolve(eqns)
```

```
ySol(t) =
C2*cos(t) + C1*sin(t)
zSol(t) =
C1*cos(t) - C2*sin(t)
```

When No Solutions Are Found

If `dsolve` cannot solve the input symbolically, then `dsolve` issues a warning and returns an empty result. This does not mean that no solutions exist. Instead, solve the equation numerically using a function such as `ode45`.

```
syms y(x)
eqn = diff(y, 2) == (1 - y^2)*diff(y) - y;
dsolve(eqn)
```

```
 Warning: Unable to find explicit solution.
> In dsolve (line 201)
ans =
[ empty sym ]
```

Turn Off Internal Simplifications for Complete Results

By default, `dsolve` applies simplifications that are not generally correct, but produce simpler solutions. For details, see “Algorithms” on page 4-382. Instead, obtain complete results by turning off these simplifications.

Solve $\frac{dy}{dt} = \frac{a}{\sqrt{y}} + y$ where $y(a) = 1$ with and without simplifications. Turn off simplifications by setting ‘`IgnoreAnalyticConstraints`’ to `false`.

```
syms a y(t)
eqn = diff(y) == a/sqrt(y) + y;
cond = y(a) == 1;
withSimplifications = dsolve(eqn, cond)
```

```
withSimplifications =
(exp((3*t)/2 - (3*a)/2 + log(a + 1)) - a)^(2/3)
```

```
withoutSimplifications = dsolve(eqn, cond, 'IgnoreAnalyticConstraints', false)
```

```
withoutSimplifications =
piecewise(pi/2 < angle(-a), {piecewise(in(C11, 'integer'),...
 (- a + exp((3*t)/2 - (3*a)/2 + log(a + 1) + pi*C11*2i))^(2/3))},...
 angle(-a) <= -pi/2, {piecewise(in(C12, 'integer'),...
 (- a + exp((3*t)/2 - (3*a)/2 + log(a + 1) + pi*C12*2i))^(2/3))},...
 angle(-a) in Dom::Interval(-pi/2, [pi/2]), {piecewise(in(C13, 'integer'),...
 (- a + exp((3*t)/2 - (3*a)/2 + log(a + 1) + pi*C13*2i))^(2/3))})
```

`withSimplifications` is easy to use but incomplete, while `withoutSimplifications` includes special cases but is not easy to use.

Further, for certain equations, `dsolve` cannot find an explicit solution if you set ‘`IgnoreAnalyticConstraints`’ to `false`.

Input Arguments

`eqn` — Differential equation or system of equations

symbolic equation | vector of symbolic equations

Differential equation or system of equations, specified as a symbolic equation or a vector of symbolic equations.

Specify a differential equation by using the == operator. In the equation, represent differentiation by using `diff`. For example, `diff(y,x)` differentiates the symbolic function `y(x)` with respect to `x`. Create the symbolic function `y(x)` by using `syms` as `syms y(x)`. Thus, to solve $d^2y(x)/dx^2 = x*y(x)$, enter:

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

Specify a system of differential equations by using a vector of equations, such as `dsolve([diff(y,t) == z, diff(z,t) == -y])`.

`cond` — Initial or boundary condition

symbolic equation | vector of symbolic equations

Initial or boundary condition, specified as a symbolic equation or vector of symbolic equations.

When a condition contains a derivative, represent the derivative with `diff` and assign the `diff` call to a variable. Then create conditions by using that variable. For an example, see “Solve Differential Equation with Condition” on page 4-377.

Specify multiple conditions by using a vector of equations. If the number of conditions is less than the number of dependent variables, the 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`.

Example: `'IgnoreAnalyticConstraints',false` does not apply internal simplifications.

`IgnoreAnalyticConstraints` — Use internal simplifications

`true` (default) | false

Use internal simplifications, specified as `true` or `false`.

By default, the solver applies simplifications while solving the differential equation. These simplifications might not be generally valid. Therefore, by default, the solver does not guarantee the completeness of results. For details, see “Algorithms” on page 4-382. To solve ordinary differential equations without these simplifications, set ‘`IgnoreAnalyticConstraints`’ to `false`. 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.

`MaxDegree` — Maximum degree of polynomial equation for which solver uses explicit formulas

2 (default) | positive integer smaller than 5

Maximum degree of polynomial equations for which solver uses explicit formulas, specified as a positive integer smaller than 5. `dsolve` does not use explicit formulas when solving polynomial equations of degrees larger than `MaxDegree`.

Output Arguments

`S` — Solutions of differential equation

symbolic expression | vector of symbolic expressions

Solutions of differential equation, returned as a symbolic expression or a vector of symbolic expressions. The size of `S` is the number of solutions.

`y1,...,yN` — Variables storing solutions of differential equation
vector of symbolic variables

Variables storing solutions of differential equation, returned as a vector of symbolic variables. The number of output variables must equal the number of dependent variables in a system. `dsolve` sorts the dependent variables alphabetically, and then assigns the solutions for the variables to output variables or symbolic arrays.

Tips

- 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` cannot find an explicit or 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. Sometimes, the output is an equivalent lower-order differential equation or an integral.

Algorithms

If you do not set '`IgnoreAnalyticConstraints`' to `false`, then `dsolve` applies these rules while solving the equation:

- $\log(a) + \log(b) = \log(a \cdot b)$ for all values of a and b. In particular, the following equality is applied 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 applied 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:

  - $\log(e^x) = x$
  - $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
  - $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ 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

`functionalDerivative` | `linsolve` | `ode23` | `ode45` | `odeToVectorField` | `solve` | `syms` | `vpasolve`

Topics

“Solve Differential Equation” on page 2-183
“Solve a System of Differential Equations” on page 2-187

Introduced before R2006a

ei

One-argument exponential integral function

Syntax

```
ei(x)
```

Description

`ei(x)` returns the one-argument exponential integral defined as

$$\mathrm{ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt.$$

Examples

Exponential Integral for Floating-Point and Symbolic Numbers

Compute exponential integrals for numeric inputs. 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 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 10-digit accuracy.

```
vpa(s, 10)

ans =
[ -0.04890051071, -0.5597735948, 1.895117816, 3.048462479]
```

Branch Cut at Negative Real Axis

The negative real axis is a branch cut. The exponential integral has a jump of height $2\pi i$ when crossing this cut. Compute the exponential integrals at `-1`, above `-1`, and below `-1` to demonstrate this.

```
[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 Exponential Integral

Compute the first, second, and third derivatives of a 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 Exponential Integral

Compute the limits of a 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
```

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 number or symbolic number, variable, expression, function, vector, or matrix.

Tips

- The one-argument exponential integral is singular at `x = 0`. The toolbox uses this special value: `ei(0) = -Inf`.

Algorithms

The relation between `ei` and `expint` is

```
ei(x) = -expint(1,-x) + (ln(x)-ln(1/x))/2 - ln(-x)
```

Both functions `ei(x)` and `expint(1,x)` have a logarithmic singularity at the origin and a branch cut along the negative real axis. The `ei` function is not continuous when approached from above or below this branch cut.

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` | `expint` | `int` | `vpa`

Introduced in R2013a

eig

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)

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))

ans =
                                 65.0
 21.276765471473795530626426697974 23
 13.126280930709218802525643085949 14
  -13.126280930709218802525643085949
  -21.276765471473795530626426697974
```

Compute the eigenvalues and eigenvectors for one of the MATLAB test matrices:

```
A = sym(gallery(5))
[v, lambda] = eig(A)

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`

Topics

“Eigenvalues” on page 2-136

Introduced before R2006a

ellipke

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 on page 4-393 and second kinds on page 4-394.

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.

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]
```

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 $m = k^2 = \sin^2\alpha$.

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`.

Alternatives

You can use `ellipticK` and `ellipticE` to compute elliptic integrals of the first and second kinds separately.

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` | `ellipticE` | `ellipticK` | `vpa`

Introduced in R2013a

ellipticCE

Complementary complete elliptic integral of the second kind

Syntax

```
ellipticCE(m)
```

Description

`ellipticCE(m)` returns the complementary complete elliptic integral of the second kind on page 4-399.

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.

Examples

Find Complementary Complete Elliptic Integral of the Second Kind

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]
```

Find Elliptic Integral for Matrix Input

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)]
```

Differentiate Complementary Complete Elliptic Integral of the Second Kind

Differentiate these expressions involving the complementary complete elliptic integral of the second kind:

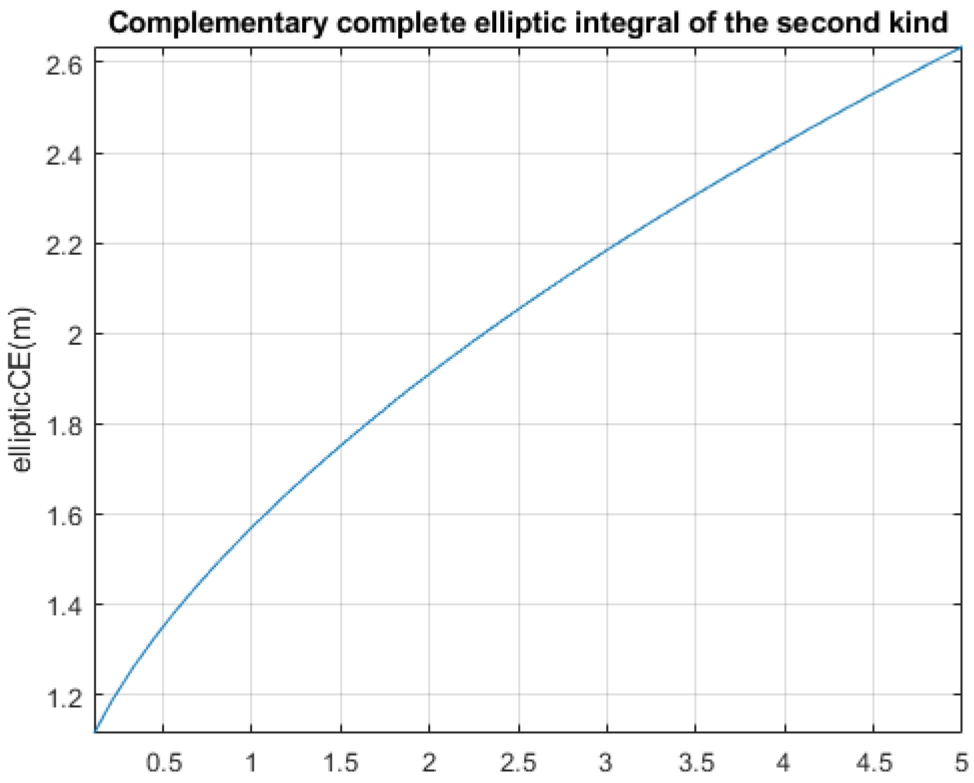
```
syms m
diff(ellipticCE(m))
diff(ellipticCE(m^2), m, 2)

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 Complementary Complete Elliptic Integral of Second Kind

Plot the complementary complete elliptic integral of the second kind. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms m
fplot(ellipticCE(m))
title('Complementary complete elliptic integral of the second kind')
ylabel('ellipticCE(m)')
grid on
```



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:

$$E(m) = E\left(\frac{\pi}{2} \,|\, m\right) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2 \alpha$.

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`.

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` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticCK

Complementary complete elliptic integral of the first kind

Syntax

```
ellipticCK(m)
```

Description

`ellipticCK(m)` returns the complementary complete elliptic integral of the first kind on page 4-404.

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.

Examples

Find Complementary Complete Elliptic Integral of First Kind

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 Complementary Complete Elliptic Integral of First Kind

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.

Find Elliptic Integral for Matrix Input

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.

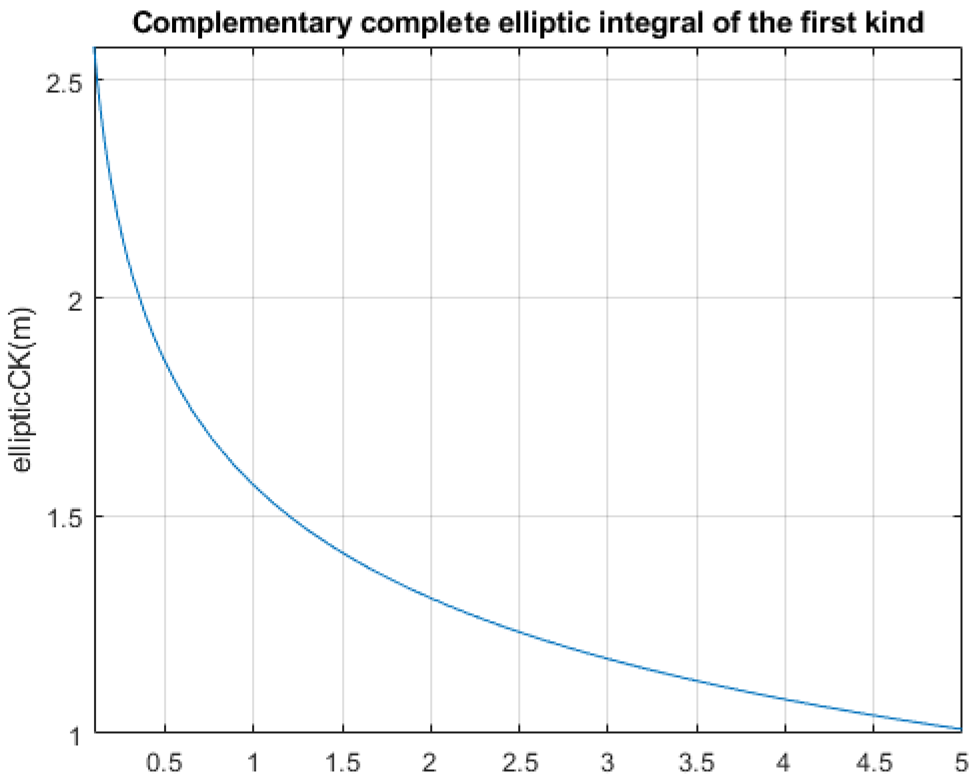
```
ellipticCK(sym([pi/6 pi/4; pi/3 pi/2]))
```

```
ans =
[ ellipticCK(pi/6), ellipticCK(pi/4)]
[ ellipticCK(pi/3), ellipticCK(pi/2)]
```

Plot Complementary Complete Elliptic Integral of First Kind

Plot complementary complete elliptic integral of first kind. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms m
fplot(ellipticCK(m), [0.1, 5])
title('Complementary complete elliptic integral of the first kind')
ylabel('ellipticCK(m)')
grid on
hold off
```



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:

$$K(m) = F\left(\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - m\sin^2\theta}}\,d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

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`.

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` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticCPi

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 on page 4-408.

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.

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.

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 $m = k^2 = \sin^2\alpha$.

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`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. 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.

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` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticE

Complete and incomplete elliptic integrals of the second kind

Syntax

```
ellipticE(m)
ellipticE(phi,m)
```

Description

`ellipticE(m)` returns the complete elliptic integral of the second kind on page 4-414.

`ellipticE(phi,m)` returns the incomplete elliptic integral of the second kind on page 4-413.

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.

Examples

Find Complete Elliptic Integrals of Second Kind

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 Elliptic Integrals of Second Kind

Differentiate these expressions involving elliptic integrals of the second kind. `ellipticK` and `ellipticF` represent the complete and incomplete elliptic integrals of the first kind, respectively.

```
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
```

Elliptic Integral for Matrix Input

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.

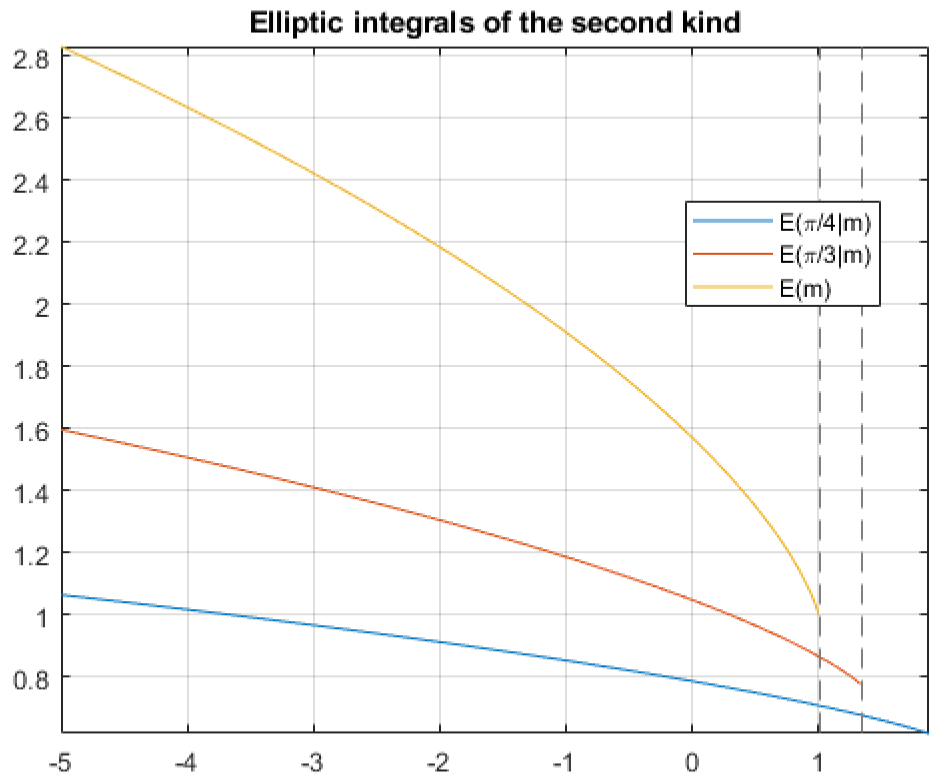
```
ellipticE(sym([1/3 1; 1/2 0]))
```

```
ans =
[ ellipticE(1/3),    1]
[ ellipticE(1/2), pi/2]
```

Plot Complete and Incomplete Elliptic Integrals of Second Kind

Plot the incomplete elliptic integrals `ellipticE(phi,m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticE(m)`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms m
fplot([ellipticE(pi/4, m) ellipticE(pi/3, m) ellipticE(m)])

title('Elliptic integrals of the second kind')
legend('E(\pi/4|m)', 'E(\pi/3|m)', 'E(m)', 'Location', 'Best')
grid on
```



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 $m = k^2 = \sin^2\alpha$.

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`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. 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).`

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

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` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticF

Incomplete elliptic integral of the first kind

Description

`ellipticF(phi,m)` returns the incomplete elliptic integral of the first kind on page 4-418.

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.

Examples

Find Incomplete Elliptic Integrals of First Kind

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 Incomplete Elliptic Integrals of First Kind

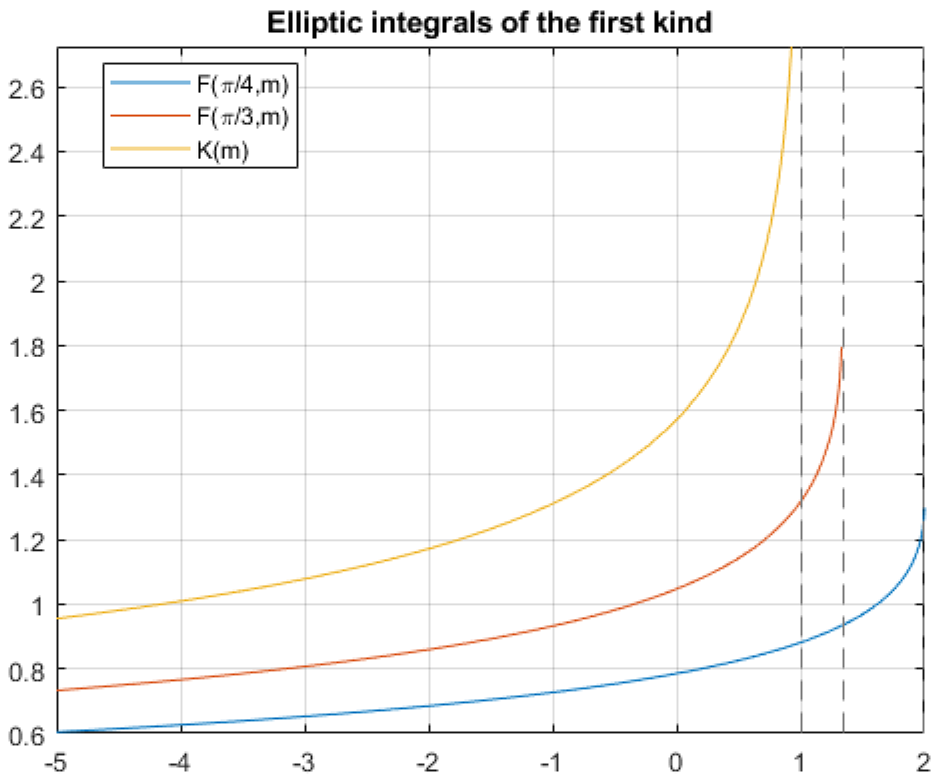
Differentiate this expression involving the incomplete elliptic integral of the first kind. `ellipticE` represents the incomplete elliptic integral of the second kind.

```
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))
```

Plot Incomplete and Complete Elliptic Integrals

Plot the incomplete elliptic integrals `ellipticF(phi,m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticK(m)`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms m
fplot([ellipticF(pi/4, m) ellipticF(pi/3, m) ellipticK(m)])
grid on

title('Elliptic integrals of the first kind')
legend('F(\pi/4,m)', 'F(\pi/3,m)', 'K(m)', 'Location', 'Best')
```



Definitions

Incomplete Elliptic Integral of the First Kind

The complete elliptic integral of the first kind is defined as follows:

$$F(\varphi\,|\,m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - m\sin^2\theta}}\,d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

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`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. 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).`

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` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticK

Complete elliptic integral of the first kind

Syntax

```
ellipticK(m)
```

Description

`ellipticK(m)` returns the complete elliptic integral of the first kind on page 4-423.

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.

Examples

Find Complete Elliptic Integrals of First Kind

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 Complete Elliptic Integral of First Kind

Differentiate these expressions involving the complete elliptic integral of the first kind. `ellipticE` represents the complete elliptic integral of the second 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))
```

Elliptic Integral for Matrix Input

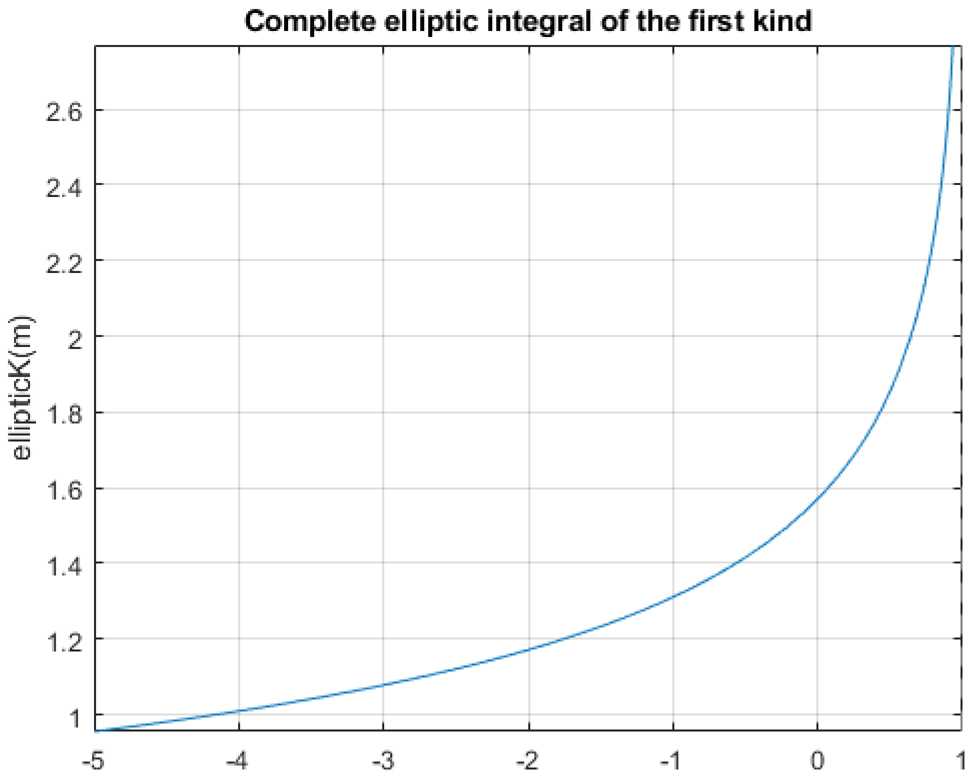
Call `ellipticK` for this symbolic matrix. When the input argument is a matrix, `ellipticK` computes the complete elliptic integral of the first kind for each element.

```
ellipticK(sym([-2*pi -4; 0 1]))

ans =
[ ellipticK(-2*pi), ellipticK(-4)]
[             pi/2,           Inf]
```

Plot Complete Elliptic Integral of First Kind

Plot the complete elliptic integral of the first kind. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms m
fplot(ellipticK(m))
title('Complete elliptic integral of the first kind')
ylabel('ellipticK(m)')
grid on
```



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 $m = k^2 = \sin^2\alpha$.

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`.

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

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` | `ellipticPi` | `vpa`

Introduced in R2013a

ellipticNome

Elliptic nome function

Syntax

```
ellipticNome(m)
```

Description

`ellipticNome(m)` returns the "Elliptic Nome" on page 4-429 of `m`. If `m` is an array, then `ellipticNome` acts element-wise.

Examples

Calculate Elliptic Nome for Numeric Inputs

```
ellipticNome(1.3)
```

```
ans =
   0.0881 - 0.2012i
```

Call `ellipticNome` on array inputs. `ellipticNome` acts element-wise when `m` is an array.

```
ellipticNome([2 1 -3/2])
```

```
ans =
   0.0000 - 0.2079i   1.0000 + 0.0000i  -0.0570 + 0.0000i
```

Calculate Elliptic Nome for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the elliptic nome. For symbolic input where `m = 0`, `1/2`, or `1`, `ellipticNome` returns exact symbolic output.

```
ellipticNome([0 1/2 1])

ans =
         0    0.0432    1.0000
```

Show that for any other symbolic values of `m`, `ellipticNome` returns an unevaluated function call.

```
ellipticNome(sym(2))

ans =
ellipticNome(2)
```

Find Elliptic Nome for Symbolic Variables or Expressions

For symbolic variables or expressions, `ellipticNome` returns the unevaluated function call.

```
syms x
f = ellipticNome(x)

f =
ellipticNome(x)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, x, 5)

f =
ellipticNome(5)

fVal = double(f)

fVal =
  -0.1008 - 0.1992i
```

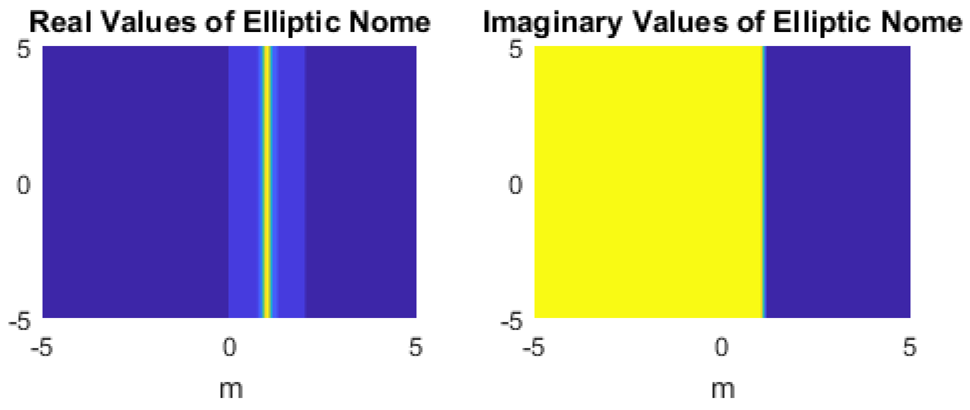
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
- 0.10080189716733475056506021415746 - 0.19922973618609837873340100821425i
```

Plot Elliptic Nome

Plot the real and imaginary values of the elliptic nome using `fcontour`. Fill plot contours by setting `Fill` to `on`.

```
syms m
f = ellipticNome(m);

subplot(2,2,1)
fcontour(real(f),'Fill','on')
title('Real Values of Elliptic Nome')
xlabel('m')

subplot(2,2,2)
fcontour(imag(f),'Fill','on')
title('Imaginary Values of Elliptic Nome')
xlabel('m')
```



Input Arguments

m — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Elliptic Nome

The elliptic nome is

$$q(m) = e^{-\frac{\pi K'(m)}{K(m)}}$$

where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

$\left|q(m)\right| \leq 1$ holds for all $m \in \mathbb{C}$.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

Introduced in R2017b

ellipticPi

Complete and incomplete elliptic integrals 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 on page 4-432.

`ellipticPi(n,phi,m)` returns the incomplete elliptic integral of the third kind on page 4-432.

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.

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.

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 $m = k^2 = \sin^2\alpha$.

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`.
- All non-scalar arguments must have the same size. If one or two input arguments are non-scalar, then `ellipticPi` expands the scalars into vectors or matrices of the same size as the non-scalar arguments, with all elements equal to the corresponding scalar.
- `ellipticPi(n, pi/2, m) = ellipticPi(n, m).`

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` | `ellipticK` | `vpa`

Introduced in R2013a

eq

Define equation

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` creates a symbolic equation. You can use that equation as an argument for such functions as `solve`, `assume`, `ezplot`, and `subs`.

`eq(A,B)` is equivalent to `A == B`.

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

Define and Solve Equation

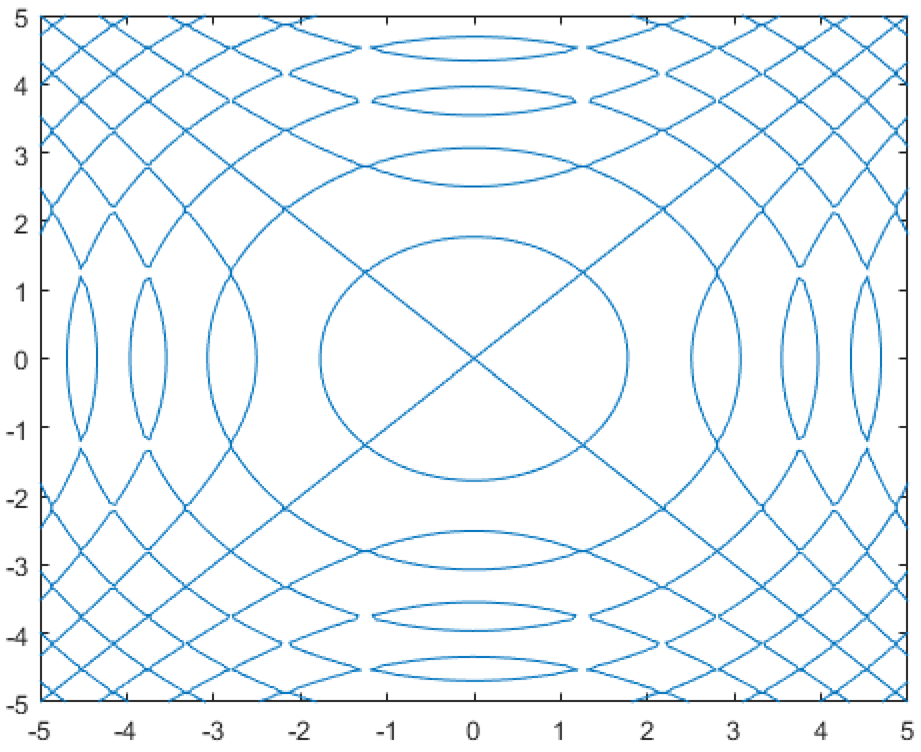
Solve this trigonometric equation. Define the equation by using the relational operator ==.

```
syms x
solve(sin(x) == cos(x), x)

ans =
pi/4
```

Plot Symbolic Equation

Plot the equation $\sin(x^2) - \sin(y^2)$. Define the equation by using the == operator.

```
syms x y
fimplicit(sin(x^2) == sin(y^2))
```



Test Equality of Symbolic Expressions

Test the equality of two symbolic expressions by using `isAlways`.

```
syms x
isAlways(x + 1 == x + 1)

ans =
  logical
   1

isAlways(sin(x)/cos(x) == tan(x))

ans =
  logical
   1
```

Test Equality of Symbolic Matrices

Check the equality of two symbolic matrices.

```
A = sym(hilb(3));
B = sym([1, 1/2, 5; 1/2, 2, 1/4; 1/3, 1/8, 1/5]);
isAlways(A == B)

ans =
  3×3 logical array
     1     1     0
     1     0     1
     1     0     1
```

If you 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);
isAlways(A == B)

ans =
  3×3 logical array
     0     1     0
     1     0     0
     0     0     0
```

Tips

- Calling == or `eq` for non-symbolic `A` and `B` invokes the MATLAB `eq` function. This function returns a logical array with elements set to logical `1 (true)` where `A` and `B` are equal; otherwise, it returns logical `0 (false)`.
- 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 is 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`.

See Also

`ge` | `gt` | `isAlways` | `le` | `lt` | `ne` | `solve`

Topics

“Solve Equations” on page 1-15
“Set Assumptions” on page 1-28

Introduced in R2012a

equationsToMatrix

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.

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.

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
```

Find the vector of variables determined for this system by `symvar`:

```
X = symvar([s - 2*t + 1 == 0, 3*s - t == 10])
```

```
X =
[ s, t]
```

Convert `X` to a column vector:

```
X = X.'
```

```
X =
 s
 t
```

Verify that `A`, `b`, and `X` form the original equations:

```
A*X == b
```

```
ans =
 s - 2*t == -1
 3*s - t == 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.
```

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:

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

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.

See Also

`linsolve` | `odeToVectorField` | `solve` | `symvar`

Topics

"Solve System of Linear Equations" on page 2-169

Introduced in R2012b

erf

Error function

Syntax

```
erf(X)
```

Description

`erf(X)` represents the error function on page 4-448 of `X`. If `X` is a vector or a matrix, `erf(X)` computes the error function of each element of `X`.

Examples

Error Function for Floating-Point and Symbolic Numbers

Depending on its arguments, `erf` can return floating-point or exact symbolic results.

Compute the error function for these numbers. Because these numbers are not symbolic objects, you get the floating-point results:

```
A = [erf(1/2), erf(1.41), erf(sqrt(2))]
```

```
A =
    0.5205    0.9539    0.9545
```

Compute the error function for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `erf` returns unresolved symbolic calls:

```
symA = [erf(sym(1/2)), erf(sym(1.41)), erf(sqrt(sym(2)))]
```

```
symA =
[ erf(1/2), erf(141/100), erf(2^(1/2))]
```

Use `vpa` to approximate symbolic results with the required number of digits:

```
d = digits(10);
vpa(symA)
digits(d)

ans =
[ 0.5204998778, 0.9538524394, 0.9544997361]
```

Error Function for Variables and Expressions

For most symbolic variables and expressions, `erf` returns unresolved symbolic calls.

Compute the error function for `x` and `sin(x) + x*exp(x)`:

```
syms x
f = sin(x) + x*exp(x);
erf(x)
erf(f)

ans =
erf(x)

ans =
erf(sin(x) + x*exp(x))
```

Error Function for Vectors and Matrices

If the input argument is a vector or a matrix, `erf` returns the error function for each element of that vector or matrix.

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*1i
```

Special Values of Error Function

`erf` returns special values for particular parameters.

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*1i, -Inf*1i]
```

Handling Expressions That Contain Error Function

Many functions, such as `diff` and `int`, can handle expressions containing `erf`.

Compute the first and second derivatives of the error function:

```
syms x
diff(erf(x), x)
diff(erf(x), x, 2)

ans =
(2*exp(-x^2))/pi^(1/2)

ans =
-(4*x*exp(-x^2))/pi^(1/2)
```

Compute the integrals of these expressions:

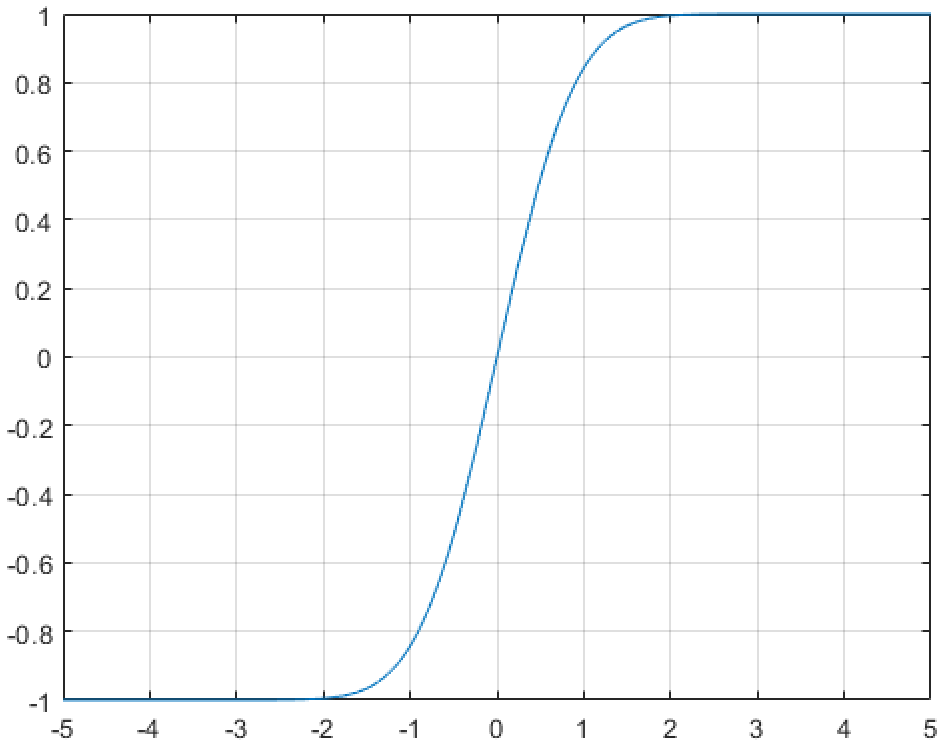
```
int(erf(x), x)
int(erf(log(x)), x)

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

```
ans =
x*erf(log(x)) - int((2*exp(-log(x)^2))/pi^(1/2), x)
```

Plot Error Function

Plot the error function on the interval from -5 to 5. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(erf(x),[-5,5])
grid on
```



Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Error Function

The following integral defines the error function:

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

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.
- For most symbolic (exact) numbers, `erf` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.

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

[1] 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`

Introduced before R2006a

erfc

Complementary error function

Syntax

```
erfc(X)
erfc(K,X)
```

Description

`erfc(X)` represents the complementary error function on page 4-455 of `X`, that is,`erfc(X) = 1 - erf(X).`

`erfc(K,X)` represents the iterated integral of the complementary error function on page 4-455 of `X`, that is, `erfc(K, X) = int(erfc(K - 1, y), y, X, inf).`

Examples

Complementary Error Function for Floating-Point and Symbolic Numbers

Depending on its arguments, `erfc` can return floating-point or exact symbolic results.

Compute the complementary error function for these numbers. Because these numbers are not symbolic objects, you get the floating-point results:

```
A = [erfc(1/2), erfc(1.41), erfc(sqrt(2))]
```

```
A =
    0.4795    0.0461    0.0455
```

Compute the complementary error function for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfc` returns unresolved symbolic calls:

```
symA = [erfc(sym(1/2)), erfc(sym(1.41)), erfc(sqrt(sym(2)))]

symA =
[ erfc(1/2), erfc(141/100), erfc(2^(1/2))]
```

Use `vpa` to approximate symbolic results with the required number of digits:

```
d = digits(10);
vpa(symA)
digits(d)

ans =
[ 0.4795001222, 0.04614756064, 0.0455002639]
```

Error Function for Variables and Expressions

For most symbolic variables and expressions, `erfc` returns unresolved symbolic calls.

Compute the complementary error function for `x` and `sin(x) + x*exp(x)`:

```
syms x
f = sin(x) + x*exp(x);
erfc(x)
erfc(f)

ans =
erfc(x)

ans =
erfc(sin(x) + x*exp(x))
```

Complementary Error Function for Vectors and Matrices

If the input argument is a vector or a matrix, `erfc` returns the complementary error function for each element of that vector or matrix.

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)
 1 + Inf*1i
```

Compute the iterated integral of the complementary error function for the elements of `V` and `M`, and the integer `-1`:

```
erfc(-1, M)
erfc(-1, V)

ans =
[              2/pi^(1/2), 0]
[ (2*exp(-1/9))/pi^(1/2), 0]

ans =
 (2*exp(-1))/pi^(1/2)
                  Inf
```

Special Values of Complementary Error Function

`erfc` returns special values for particular parameters.

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))]

ans =
[ 1 - Inf*1i, 1 + Inf*1i]
```

Handling Expressions That Contain Complementary Error Function

Many functions, such as `diff` and `int`, can handle expressions containing `erfc`.

Compute the first and second derivatives of the complementary error function:

```
syms x
diff(erfc(x), x)
diff(erfc(x), x, 2)

ans =
-(2*exp(-x^2))/pi^(1/2)

ans =
(4*x*exp(-x^2))/pi^(1/2)
```

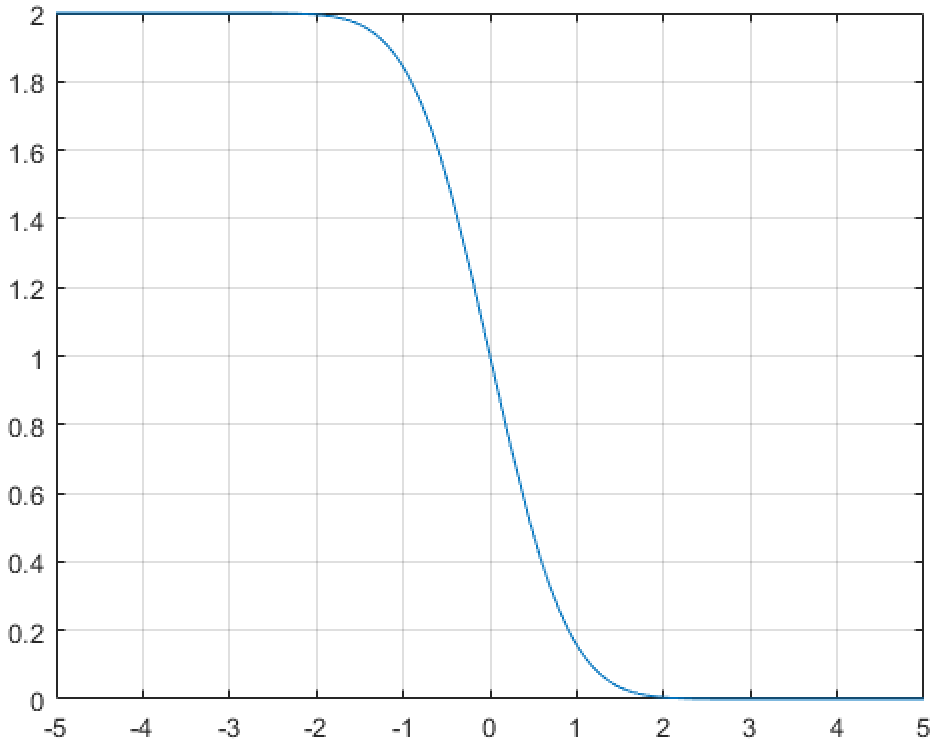
Compute the integrals of these expressions:

```
syms x
int(erfc(-1, x), x)

ans =
erf(x)

int(erfc(x), x)

ans =
x*erfc(x) - exp(-x^2)/pi^(1/2)

int(erfc(2, x), x)

ans =
(x^3*erfc(x))/6 - exp(-x^2)/(6*pi^(1/2)) +...
(x*erfc(x))/4 - (x^2*exp(-x^2))/(6*pi^(1/2))
```

Plot Complementary Error Function

Plot the complementary error function on the interval from -5 to 5. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(erfc(x),[-5,5])
grid on
```



Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

`K` — Input representing an integer larger than `-2`
number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input representing an integer larger than `-2`, specified as a number, symbolic number, variable, expression, or function. This arguments can also be a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

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.

Iterated Integral of Complementary Error Function

The following integral is the iterated integral of the complementary error function:

$$erfc(k,x) = \int_x^\infty erfc(k-1,y)dy$$

Here, $erfc(0,x) = erfc(x)$.

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.

- For most symbolic (exact) numbers, `erfc` returns unresolved symbolic calls. You can approximate such results with floating-point numbers using `vpa`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, then `erfc` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

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

[1] 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`

Introduced in R2011b

erfcinv

Inverse complementary error function

Syntax

```
erfcinv(X)
```

Description

`erfcinv(X)` computes the inverse complementary error function on page 4-462 of `X`. If `X` is a vector or a matrix, `erfcinv(X)` computes the inverse complementary error function of each element of `X`.

Examples

Inverse Complementary Error Function for Floating-Point and Symbolic Numbers

Depending on its arguments, `erfcinv` can return floating-point or exact symbolic results.

Compute the inverse complementary error function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
A = [erfcinv(1/2), erfcinv(1.33), erfcinv(3/2)]
```

```
A =
    0.4769   -0.3013   -0.4769
```

Compute the inverse complementary error function for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfcinv` returns unresolved symbolic calls:

```
symA = [erfcinv(sym(1/2)), erfcinv(sym(1.33)), erfcinv(sym(3/2))]
symA =
[ -erfcinv(3/2), erfcinv(133/100), erfcinv(3/2)]
```

Use `vpa` to approximate symbolic results with the required number of digits:

```
d = digits(10);
vpa(symA)
digits(d)

ans =
[ 0.4769362762, -0.3013321461, -0.4769362762]
```

Inverse Complementary Error Function for Variables and Expressions

For most symbolic variables and expressions, `erfcinv` returns unresolved symbolic calls.

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))
```

Inverse Complementary Error Function for Vectors and Matrices

If the input argument is a vector or a matrix, `erfcinv` returns the inverse complementary error function for each element of that vector or matrix.

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, NaN]
[ -erfcinv(5/3),   0]

ans =
 -Inf
  NaN
```

Special Values of Inverse Complementary Error Function

`erfcinv` returns special values for particular parameters.

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
```

Handling Expressions That Contain Inverse Complementary Error Function

Many functions, such as `diff` and `int`, can handle expressions containing `erfcinv`.

Compute the first and second derivatives of the inverse complementary error function:

```
syms x
diff(erfcinv(x), x)
diff(erfcinv(x), x, 2)
```

```
ans =
-(pi^(1/2)*exp(erfcinv(x)^2))/2

ans =
(pi*exp(2*erfcinv(x)^2)*erfcinv(x))/2
```

Compute the integral of the inverse complementary error function:

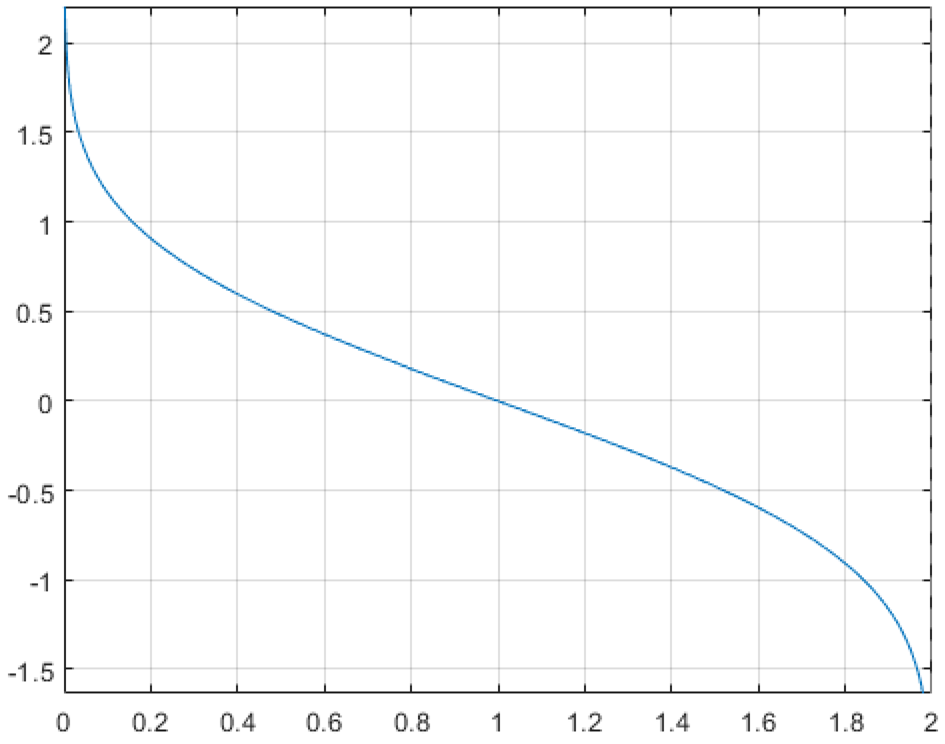
```
int(erfcinv(x), x)
```

```
ans =
exp(-erfcinv(x)^2)/pi^(1/2)
```

Plot Inverse Complementary Error Function

Plot the inverse complementary error function on the interval from 0 to 2. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(erfcinv(x),[0,2])
grid on
```



Input Arguments

x — Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

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.

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 $x > 2$, or if x is complex, then `erfcinv(x)` returns `NaN`.

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

[1] 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`

Introduced in R2012a

erfi

Imaginary error function

Syntax

```
erfi(x)
```

Description

`erfi(x)` returns the imaginary error function on page 4-468 of `x`. If `x` is a vector or a matrix, `erfi(x)` returns the imaginary error function of each element of `x`.

Examples

Imaginary Error Function for Floating-Point and Symbolic Numbers

Depending on its arguments, `erfi` can return floating-point or exact symbolic results.

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]
```

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))
```

Imaginary Error Function for Vectors and Matrices

If the input argument is a vector or a matrix, `erfi` returns the imaginary error function for each element of that vector or matrix.

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)
      -1i
```

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 =
[ 1i, -1i]
```

Handling Expressions That Contain Imaginary Error Function

Many functions, such as `diff` and `int`, can handle expressions containing `erfi`.

Compute the first and second derivatives of the imaginary error function:

```
syms x
diff(erfi(x), x)
diff(erfi(x), x, 2)

ans =
(2*exp(x^2))/pi^(1/2)

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

Compute the integrals of these expressions:

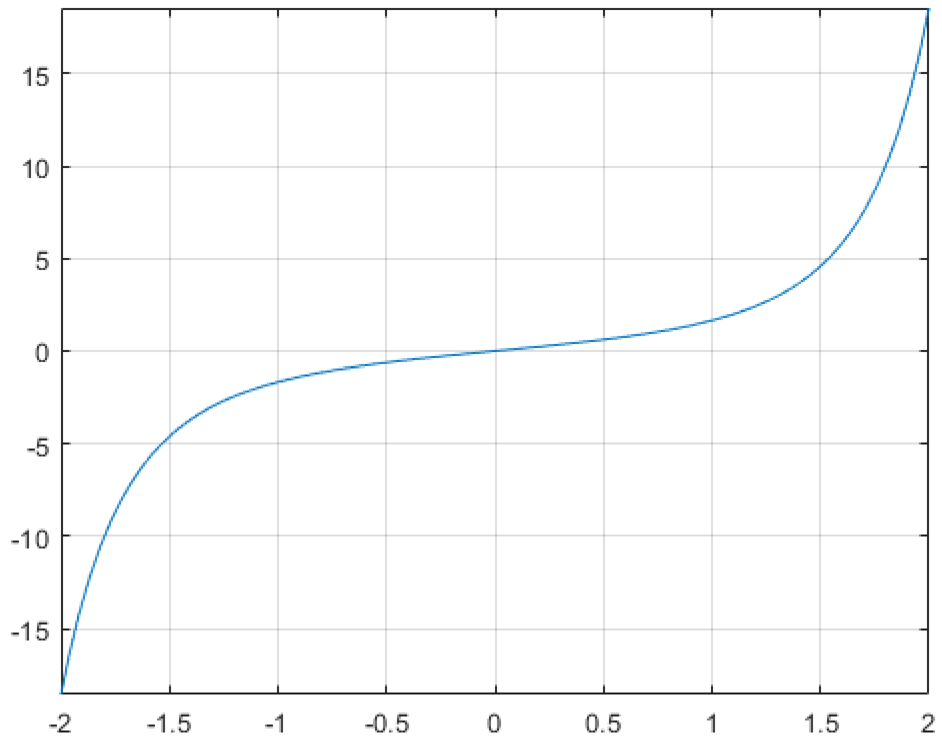
```
int(erfi(x), x)
int(erfi(log(x)), x)

ans =
x*erfi(x) - exp(x^2)/pi^(1/2)
```

```
ans =
x*erfi(log(x)) - int((2*exp(log(x)^2))/pi^(1/2), x)
```

Plot Imaginary Error Function

Plot the imaginary error function on the interval from -2 to 2. Before R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(erfi(x),[-2,2])
grid on
```



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:

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

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`

See Also

`erf` | `erfc` | `erfcinv` | `erfinv` | `vpa`

Introduced in R2013a

erfinv

Inverse error function

Syntax

```
erfinv(X)
```

Description

`erfinv(X)` computes the inverse error function on page 4-473 of `X`. If `X` is a vector or a matrix, `erfinv(X)` computes the inverse error function of each element of `X`.

Examples

Inverse Error Function for Floating-Point and Symbolic Numbers

Depending on its arguments, `erfinv` can return floating-point or exact symbolic results.

Compute the inverse error function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
A = [erfinv(1/2), erfinv(0.33), erfinv(-1/3)]
```

```
A =
    0.4769    0.3013   -0.3046
```

Compute the inverse error function for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `erfinv` returns unresolved symbolic calls:

```
symA = [erfinv(sym(1)/2), erfinv(sym(0.33)), erfinv(sym(-1)/3)]
```

```
symA =
[ erfinv(1/2), erfinv(33/100), -erfinv(1/3)]
```

Use `vpa` to approximate symbolic results with the required number of digits:

```
d = digits(10);
vpa(symA)
digits(d)

ans =
[ 0.4769362762, 0.3013321461, -0.3045701942]
```

Inverse Error Function for Variables and Expressions

For most symbolic variables and expressions, `erfinv` returns unresolved symbolic calls.

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))
```

Inverse Error Function for Vectors and Matrices

If the input argument is a vector or a matrix, `erfinv` returns the inverse error function for each element of that vector or matrix.

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, NaN]
[ erfinv(1/3), Inf]

ans =
 -Inf
  NaN
```

Special Values of Inverse Complementary Error Function

`erfinv` returns special values for particular parameters.

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
```

Handling Expressions That Contain Inverse Complementary Error Function

Many functions, such as `diff` and `int`, can handle expressions containing `erfinv`.

Compute the first and second derivatives of the inverse error function:

```
syms x
diff(erfinv(x), x)
diff(erfinv(x), x, 2)
```

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

ans =
(pi*exp(2*erfinv(x)^2)*erfinv(x))/2
```

Compute the integral of the inverse error function:

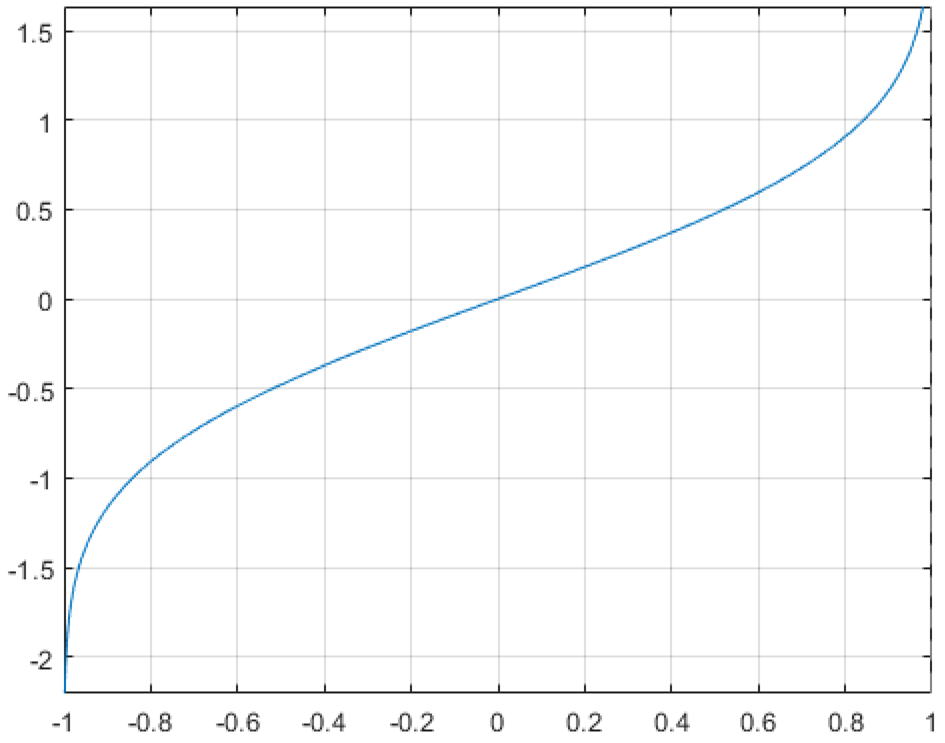
```
int(erfinv(x), x)
```

```
ans =
-exp(-erfinv(x)^2)/pi^(1/2)
```

Plot Inverse Error Function

Plot the inverse error function on the interval from -1 to 1. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(erfinv(x),[-1,1])
grid on
```



Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

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.

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 $x > 1$, or if x is complex, then `erfinv(x)` returns `NaN`.

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

[1] 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`

Introduced in R2012a

euler

Euler numbers and polynomials

Syntax

```
euler(n)
euler(n,x)
```

Description

`euler(n)` returns the `n`th Euler number on page 4-479.

`euler(n,x)` returns the `n`th Euler polynomial on page 4-479.

Examples

Euler Numbers with Odd and Even Indices

The Euler numbers with even indices alternate the signs. Any Euler number with an odd index is `0`.

Compute the even-indexed Euler numbers with the indices from `0` to `10`:

```
euler(0:2:10)
```

```
ans =
           1           -1            5          -61...
        1385       -50521
```

Compute the odd-indexed Euler numbers with the indices from `1` to `11`:

```
euler(1:2:11)
```

```
ans =
     0     0     0     0     0     0
```

Euler Polynomials

For the Euler polynomials, use `euler` with two input arguments.

Compute the first, second, and third Euler polynomials in variables `x`, `y`, and `z`, respectively:

```
syms x y z
euler(1, x)
euler(2, y)
euler(3, z)

ans =
x - 1/2

ans =
y^2 - y

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

If the second argument is a number, `euler` evaluates the polynomial at that number. Here, the result is a floating-point number because the input arguments are not symbolic numbers:

```
euler(2, 1/3)

ans =
   -0.2222
```

To get the exact symbolic result, convert at least one number to a symbolic object:

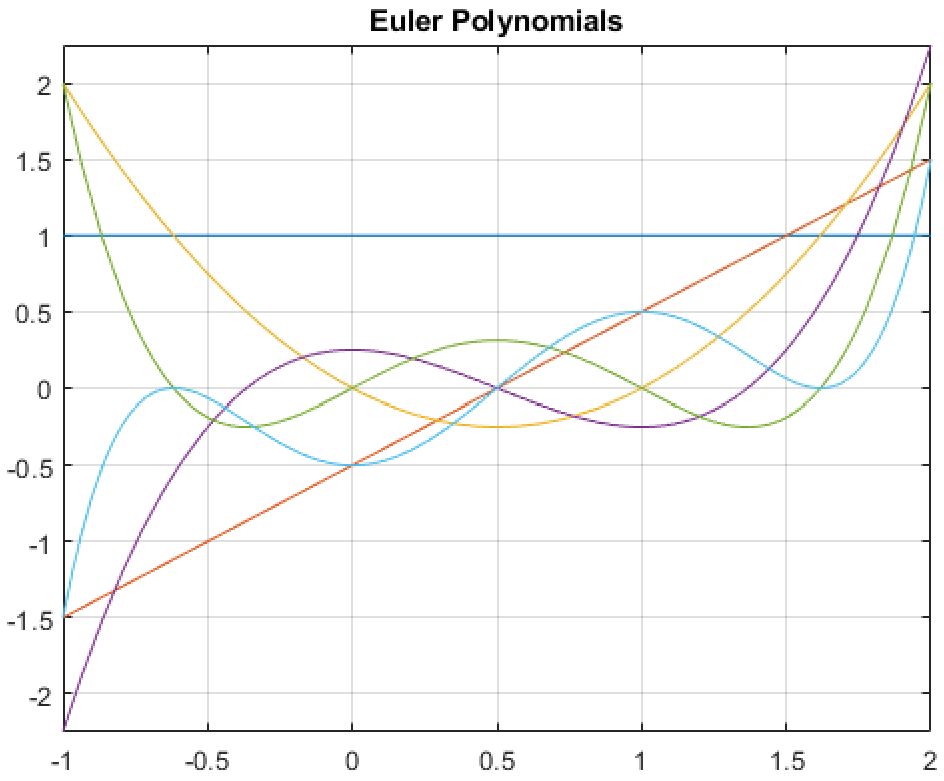
```
euler(2, sym(1/3))

ans =
-2/9
```

Plot Euler Polynomials

Plot the first six Euler polynomials. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(euler(0:5, x), [-1 2])
```

```
title('Euler Polynomials')
grid on
```



Handle Expressions Containing Euler Polynomials

Many functions, such as `diff` and `expand`, can handle expressions containing `euler`.

Find the first and second derivatives of the Euler polynomial:

```
syms n x
diff(euler(n,x^2), x)
```

```
ans =
2*n*x*euler(n - 1, x^2)

diff(euler(n,x^2), x, x)

ans =
2*n*euler(n - 1, x^2) + 4*n*x^2*euler(n - 2, x^2)*(n - 1)
```

Expand these expressions containing the Euler polynomials:

```
expand(euler(n, 2 - x))

ans =
2*(1 - x)^n - (-1)^n*euler(n, x)

expand(euler(n, 2*x))

ans =
(2*2^n*bernoulli(n + 1, x + 1/2))/(n + 1) -...
(2*2^n*bernoulli(n + 1, x))/(n + 1)
```

Input Arguments

`n` — Index of the Euler number or polynomial

nonnegative integer | symbolic nonnegative integer | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Index of the Euler number or polynomial, specified as a nonnegative integer, symbolic nonnegative integer, variable, expression, function, vector, or matrix. If `n` is a vector or matrix, `euler` returns Euler numbers or polynomials for each element of `n`. If one input argument is a scalar and the other one is a vector or a matrix, `euler(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.

`x` — Polynomial variable

symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Polynomial variable, specified as a symbolic variable, expression, function, vector, or matrix. If `x` is a vector or matrix, `euler` returns Euler numbers or polynomials for each element of `x`. When you use the `euler` function to find Euler polynomials, at least one argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `euler(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.

Definitions

Euler Polynomials

The Euler polynomials are defined as follows:

$$\frac{2e^{xt}}{e^t+1} = \sum_{n=0}^{\infty} \operatorname{euler}(n,x)\frac{t^n}{n!}$$

Euler Numbers

The Euler numbers are defined in terms of Euler polynomials as follows:

$$\operatorname{euler}(n) = 2^n \operatorname{euler}\left(n,\frac{1}{2}\right)$$

Tips

- For the other meaning of Euler's number, $e = 2.71828\ldots$, call `exp(1)` to return the double-precision representation. For the exact representation of Euler's number e, call `exp(sym(1))`.
- For the Euler-Mascheroni constant, see `eulergamma`.

See Also

`bernoulli` | `eulergamma`

Introduced in R2014a

eulergamma

Euler-Mascheroni constant

Syntax

```
eulergamma
```

Description

`eulergamma` represents the Euler-Mascheroni constant on page 4-481. To get a floating-point approximation with the current precision set by `digits`, use `vpa(eulergamma)`.

Examples

Represent and Numerically Approximate the Euler-Mascheroni Constant

Represent the Euler-Mascheroni constant using `eulergamma`, which returns the symbolic form `eulergamma`.

```
eulergamma
```

```
ans =
eulergamma
```

Use `eulergamma` in symbolic calculations. Numerically approximate your result with `vpa`.

```
a = eulergamma;
g = a^2 + log(a)
gVpa = vpa(g)
```

```
g =
log(eulergamma) + eulergamma^2
```

```
gVpa =
-0.21636138917392614801928563244766
```

Find the double-precision approximation of the Euler-Mascheroni constant using `double`.

```
double(eulergamma)
```

```
ans =
    0.5772
```

Show Relation of Euler-Mascheroni Constant to Gamma Functions

Show the relations between the Euler-Mascheroni constant γ, digamma function Ψ, and gamma function Γ.

Show that $\gamma = -\Psi(1)$.

```
-psi(sym(1))
```

```
ans =
eulergamma
```

Show that $\gamma = -\Gamma'(x)\big|_{x=1}$.

```
syms x
-subs(diff(gamma(x)),x,1)
```

```
ans =
eulergamma
```

Definitions

Euler-Mascheroni Constant

The Euler-Mascheroni constant is defined as follows:

$$\gamma = \lim_{n\to\infty}\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \ln(n)\right)$$

Tips

- For the value *e* = 2.71828…, called Euler's number, use `exp(1)` to return the double-precision representation. For the exact representation of Euler's number *e*, call `exp(sym(1))`.
- For the other meaning of Euler's numbers and for Euler's polynomials, see `euler`.

See Also

`coshint` | `euler`

Introduced in R2014a

evalin

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 character vector.

Input Arguments

`MuPAD_expression`

Character vector containing a MuPAD expression.

Output Arguments

`result`

Symbolic object or character vector 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 =
    'Arithmetical expression expected.'

status =
     2
```

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.
- `evalin` does not open a MuPAD notebook, and therefore, you cannot use this function to access MuPAD graphics capabilities.

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`

Topics

“Call Built-In MuPAD Functions from MATLAB” on page 3-58
“Evaluations in Symbolic Computations”
“Level of Evaluation”

Introduced in R2008b

evaluateMuPADNotebook

Evaluate MuPAD notebook

Syntax

```
evaluateMuPADNotebook(nb)
evaluateMuPADNotebook(nb,'IgnoreErrors',true)
```

Description

`evaluateMuPADNotebook(nb)` evaluates the MuPAD notebook with the handle `nb` and returns logical `1` (`true`) if evaluation runs without errors. If `nb` is a vector of notebook handles, then this syntax returns a vector of logical `1`s.

`evaluateMuPADNotebook(nb,'IgnoreErrors',true)` does not stop evaluating the notebook when it encounters an error. This syntax skips any input region of a MuPAD notebook that causes errors, and proceeds to the next one. If the evaluation runs without errors, this syntax returns logical `1` (`true`). Otherwise, it returns logical `0` (`false`). The error messages appear in the MuPAD notebook only.

By default, `evaluateMuPADNotebook` uses `'IgnoreErrors',false`, and therefore, `evaluateMuPADNotebook` stops when it encounters an error in a notebook. The error messages appear in the MATLAB Command Window and in the MuPAD notebook.

Examples

Evaluate Particular Notebook

Execute commands in all input regions of a MuPAD notebook. Results of the evaluation appear in the output regions of the notebook.

Suppose that your current folder contains a MuPAD notebook named `myFile1.mn`. Open this notebook keeping its handle in the variable `nb1`:

```
nb1 = mupad('myFile1.mn');
```

Evaluate all input regions in this notebook. If all calculations run without an error, then `evaluateMuPADNotebook` returns logical `1` (`true`):

```
evaluateMuPADNotebook(nb1)
```

```
ans =
     1
```

Evaluate Several Notebooks

Use a vector of notebook handles to evaluate several notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Evaluate `myFile1.mn` and `myFile2.mn`:

```
evaluateMuPADNotebook([nb1, nb2])
```

```
ans =
     1
```

Evaluate All Open Notebooks

Identify and evaluate all open MuPAD notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Get a list of all currently open notebooks:

```
allNBs = allMuPADNotebooks;
```

Evaluate all notebooks. If all calculations run without an error, then `evaluateMuPADNotebook` returns an array of logical `1`s (`true`):

```
evaluateMuPADNotebook(allNBs)
```

```
ans =
     1
     1
     1
```

Evaluate All Open Notebooks Ignoring Errors

Identify and evaluate all open MuPAD notebooks skipping evaluations that cause errors.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Get a list of all currently open notebooks:

```
allNBs = allMuPADNotebooks;
```

Evaluate all notebooks using `'IgnoreErrors',true` to skip any calculations that cause errors. If all calculations run without an error, then `evaluateMuPADNotebook` returns an array of logical `1`s (`true`):

```
evaluateMuPADNotebook(allNBs,'IgnoreErrors',true)
```

```
ans =
     1
     1
     1
```

Otherwise, it returns logical `0`s for notebooks that cause errors (`false`):

```
ans =
     0
     1
     1
```

- “Create MuPAD Notebooks” on page 3-3
- “Open MuPAD Notebooks” on page 3-6
- “Save MuPAD Notebooks” on page 3-12
- “Evaluate MuPAD Notebooks from MATLAB” on page 3-13
- “Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52
- “Close MuPAD Notebooks from MATLAB” on page 3-17

Input Arguments

nb — Pointer to MuPAD notebook

handle to notebook | vector of handles to notebooks

Pointer to MuPAD notebook, specified as a MuPAD notebook handle or a vector of handles. You create the notebook handle when opening a notebook with the `mupad` or `openmn` function.

You can get the list of all open notebooks using the `allMuPADNotebooks` function. `evaluateMuPADNotebook` accepts a vector of handles returned by `allMuPADNotebooks`.

See Also

`allMuPADNotebooks` | `close` | `getVar` | `mupad` | `mupadNotebookTitle` | `openmn` | `setVar`

Topics

"Create MuPAD Notebooks" on page 3-3
"Open MuPAD Notebooks" on page 3-6
"Save MuPAD Notebooks" on page 3-12
"Evaluate MuPAD Notebooks from MATLAB" on page 3-13
"Copy Variables and Expressions Between MATLAB and MuPAD" on page 3-52
"Close MuPAD Notebooks from MATLAB" on page 3-17

Introduced in R2013b

expand

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))

ans =
x^2 - 6*x + 8
```

Expand the trigonometric expression:

```
syms x y
expand(cos(x + y))

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

Expand the exponent:

```
syms a b
expand(exp((a + b)^2))

ans =
exp(a^2)*exp(b^2)*exp(2*a*b)
```

Expand the expressions that form a vector:

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

```
ans =
[ 2*cos(t)*sin(t), 2*cos(t)^2 - 1]
```

Expand this expression. By default, `expand` works on all subexpressions including trigonometric subexpressions:

```
syms x
expand((sin(3*x) - 1)^2)
```

```
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)
```

```
ans =
sin(3*x)^2 - 2*sin(3*x) + 1
```

Expand this logarithm. By default, the `expand` function does not expand logarithms because expanding logarithms is not valid for generic complex values:

```
syms a b c
expand(log((a*b/c)^2))
```

```
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)
```

```
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` | `combine` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

Topics

"Choose Function to Rearrange Expression" on page 2-94

Introduced before R2006a

expint

Exponential integral function

Syntax

```
expint(x)
expint(n,x)
```

Description

`expint(x)` returns the one-argument exponential integral function defined as

$$\operatorname{expint}(x) = \int_1^\infty \frac{e^{-xt}}{t}\,dt.$$

`expint(n,x)` returns the two-argument exponential integral function defined as

$$\operatorname{expint}(n,x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt.$$

Examples

One-Argument Exponential Integral for Floating-Point and Symbolic Numbers

Compute the exponential integrals for floating-point 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*1i - ei(2)]
```

Use vpa to approximate this result with 10-digit accuracy.

```
vpa(s, 10)
```

```
ans =
[ 0.8288877453, 0.2193839344, - 4.954234356 - 3.141592654i]
```

Two-Argument Exponential Integral for Floating-Point and Symbolic Numbers

When computing two-argument exponential integrals, convert the 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 25-digit accuracy.

```
vpa(s, 25)
```

```
ans =
[ 0.4402353954575937050522018, 0, -1.847264024732662556807607]
```

Two-Argument Exponential Integral with Nonpositive First Argument

Compute 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 Exponential Integral

Compute the first, second, and third derivatives of a 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 a 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 - (x^(n - 1)*pi*(psi(n) - ...
             log(x) + pi*cot(pi*n)))/(sin(pi*n)*gamma(n))
```

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.

Tips

- 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 argument is a vector or a matrix, then `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.

Algorithms

The relation between `expint` and `ei` is

```
expint(1,-x) = ei(x) + (ln(x)-ln(1/x))/2 - ln(-x)
```

Both functions `ei(x)` and `expint(1,x)` have a logarithmic singularity at the origin and a branch cut along the negative real axis. The `ei` function is not continuous when approached from above or below this branch cut.

The `expint` function is related to the upper incomplete gamma function `igamma` as

```
expint(n,x) = (x^(n-1))*igamma(1-n,x)
```

See Also

`ei` | `expint` | `vpa`

Introduced in R2013a

expm

Matrix exponential

Syntax

```
R = expm(A)
```

Description

`R = expm(A)` computes the matrix exponential on page 4-501 of the square matrix `A`.

Examples

Matrix Exponential

Compute the matrix exponential for the 2-by-2 matrix and simplify the result.

```
syms x
A = [0 x; -x 0];
simplify(expm(A))

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

Input Arguments

`A` — Input matrix
square matrix

Input matrix, specified as a square symbolic matrix.

Output Arguments

R — Resulting matrix
symbolic matrix

Resulting function, returned as a symbolic matrix.

Definitions

Matrix Exponential

The matrix exponential e_A of matrix A is

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} A^k = 1 + A + \frac{A^2}{2} + \ldots$$

See Also

`eig` | `funm` | `jordan` | `logm` | `sqrtm`

Introduced before R2006a

ezcontour

Contour plotter

Note `ezcontour` is not recommended. Use `fcontour` instead.

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

Plot Contour Lines of Symbolic Expression

The following mathematical expression defines a function of two variables, *x* and *y*.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

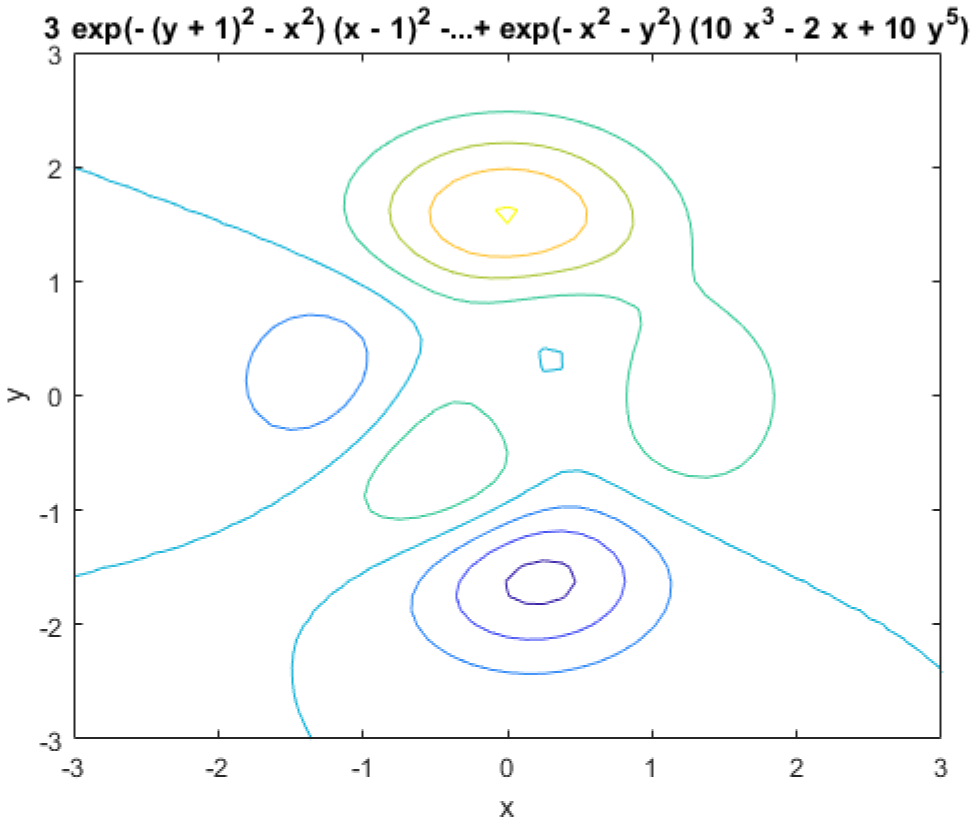
`ezcontour` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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 title.

See Also

`contour` | `fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Introduced before R2006a

ezcontourf

Filled contour plotter

Note `ezcontourf` is not recommended. Use `fcontour` instead.

Syntax

```
ezcontourf(f)
ezcontourf(f,domain)
ezcontourf(...,n)
```

Description

`ezcontourf(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.

`ezcontourf(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

Plot Filled Contours

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

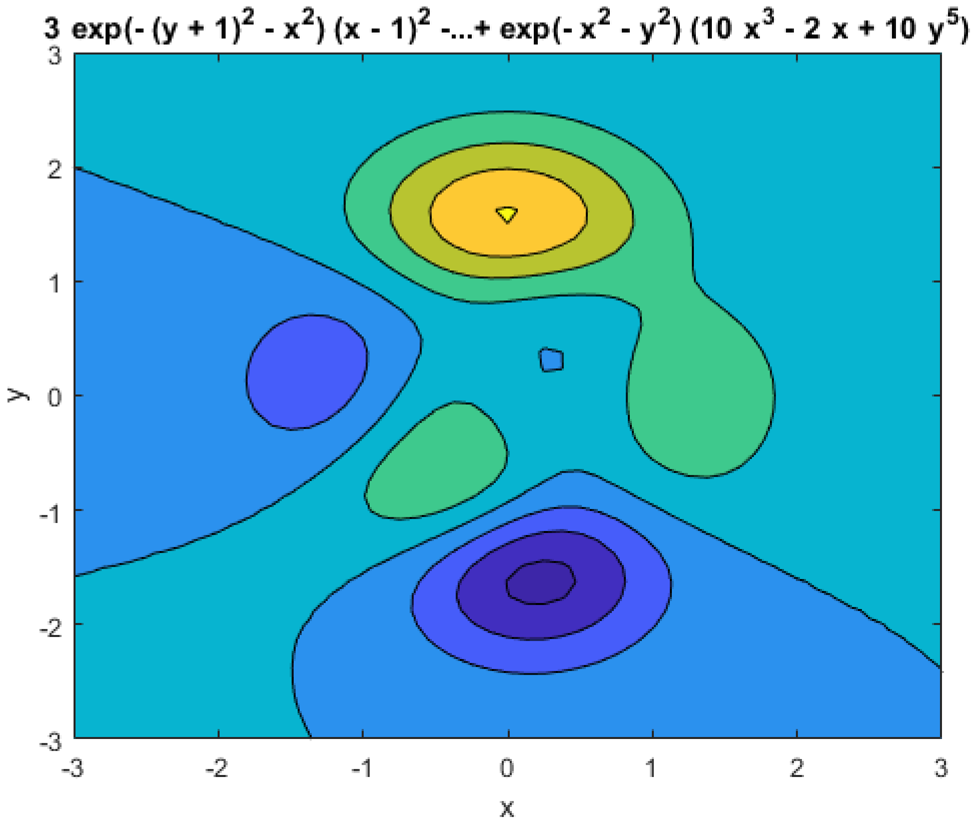
`ezcontourf` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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 title.

See Also

`contourf` | `fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Introduced before R2006a

ezmesh

3-D mesh plotter

Note `ezmesh` is not recommended. Use `fmesh` instead.

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

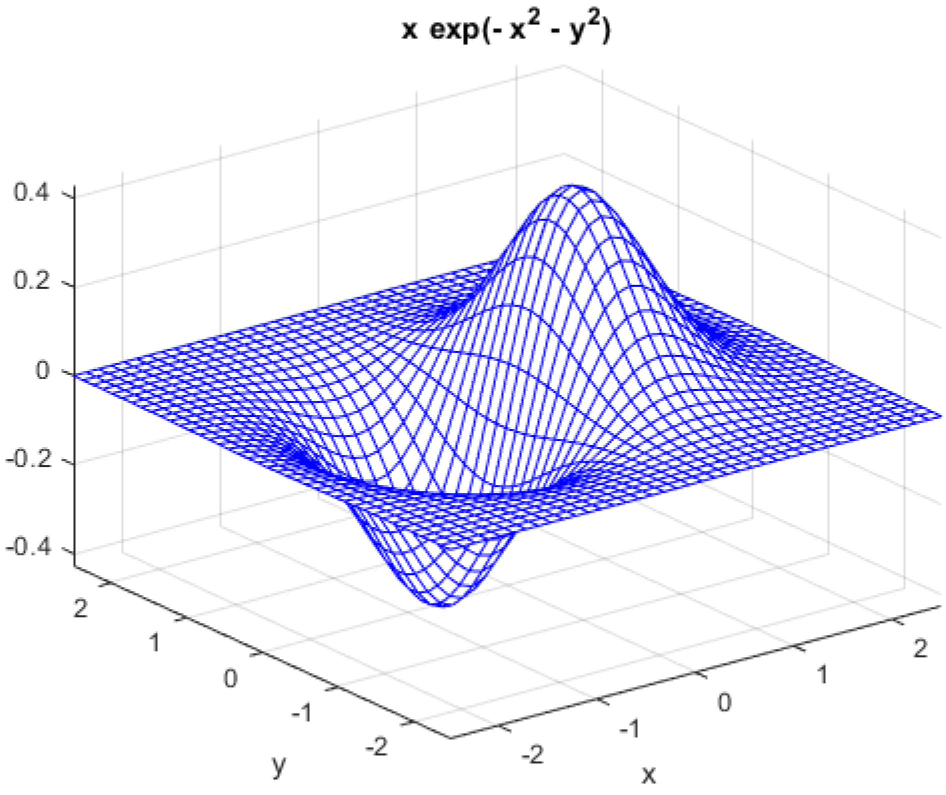
3-D Mesh Plot of Symbolic Expression

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

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `mesh`

Introduced before R2006a

ezmeshc

Combined mesh and contour plotter

Note `ezmeshc` is not recommended. Use `fmesh` instead.

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

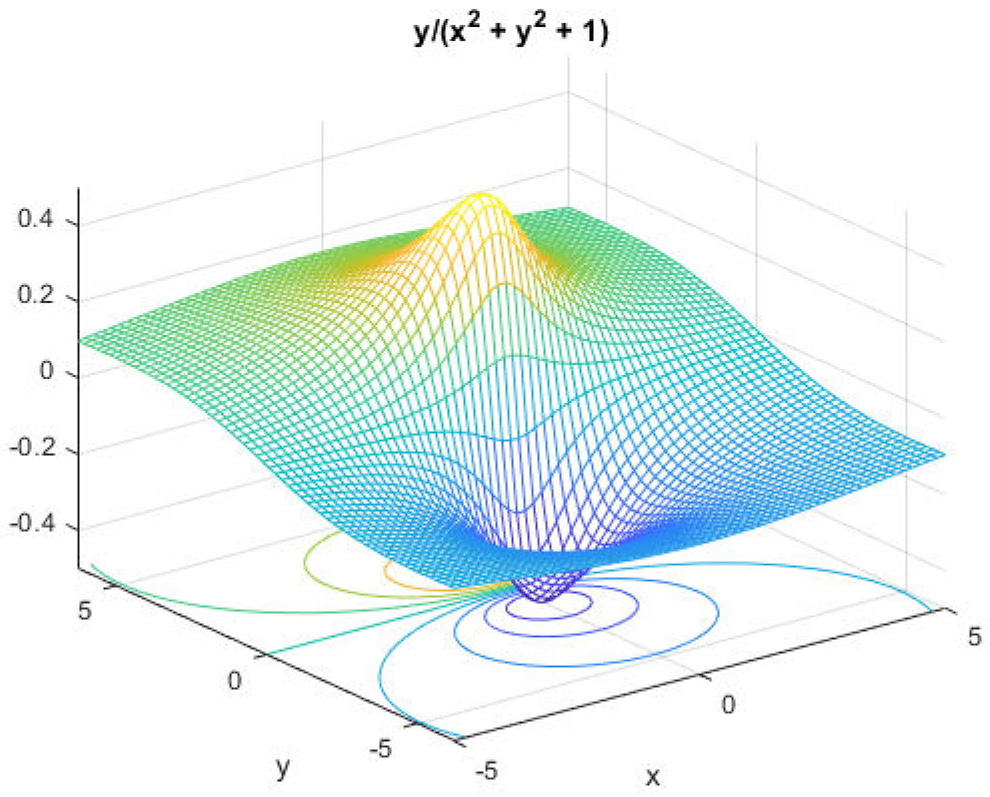
3-D Mesh Plot with Contours

Create a mesh/contour graph of the expression,

$$f(x,y)=\frac{y}{1+x^2+y^2},$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$. Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = –65 and elevation = 26).

```
syms x y
ezmeshc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi])
```



See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `meshc`

Introduced before R2006a

ezplot

Plot symbolic expression, equation, or function

Note `ezplot` is not recommended. Use `fplot` instead. For implicit plots, use `fimplicit`.

Syntax

```
ezplot(f)
ezplot(f,[min,max])
ezplot(f,[xmin,xmax,ymin,ymax])

ezplot(x,y)
ezplot(x,y,[tmin,tmax])

ezplot(f,[min,max],fig)
ezplot(f,[xmin,xmax,ymin,ymax],fig)
ezplot(x,y,[tmin,tmax],fig)

h = ezplot(___)
```

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.

`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,[min,max],fig)` plots `f` over the specified range in the figure with the figure number or figure handle `fig`. The title of each plot window contains the word Figure and the number, for example, **Figure 1**, **Figure 2**, and so on. If `fig` is already open, `ezplot` overwrites the content of that figure with the new plot.

`ezplot(f,[xmin,xmax,ymin,ymax],fig)` plots `f` over the specified ranges along the abscissa and the ordinate in `fig`.

`ezplot(x,y,[tmin,tmax],fig)` plots $x = x(t)$ and $y = y(t)$ over the specified range in `fig`.

`h = ezplot(___)` returns the plot handle as either a chart line or contour object.

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.

`fig`

Figure handle or number of the figure window where you want to display a plot.

Default: For figure handle, the current figure handle returned by `gcf`. For figure number, 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.

Output Arguments

`h` — Chart line or contour line object
scalar

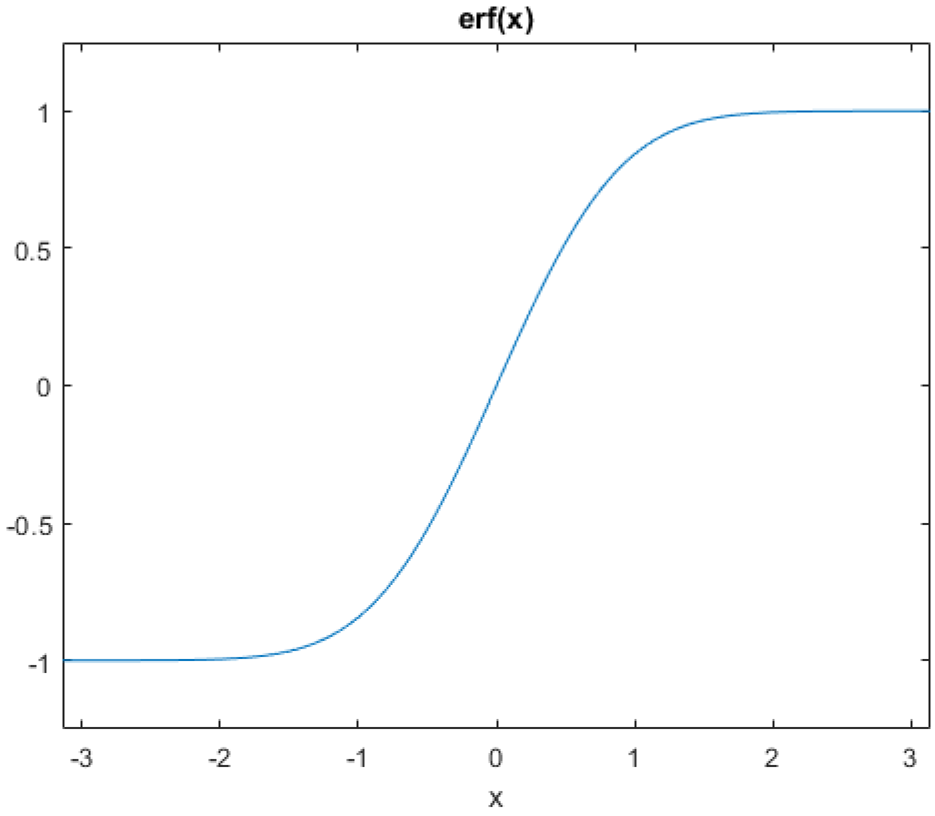
Chart line or contour line object, returned as a scalar. For details, see Chart Line and Contour.

Examples

Plot Over Particular Range

Plot the expression `erf(x)*sin(x)` over the range $[-\pi, \pi]$:

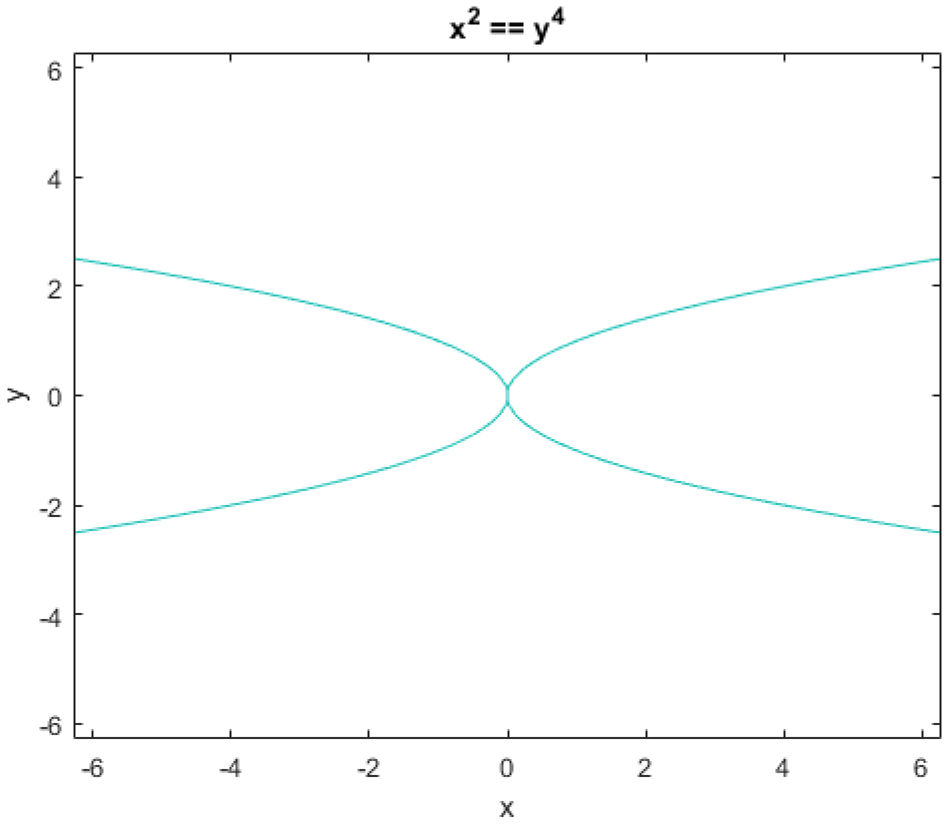
```
syms x
ezplot(erf(x), [-pi, pi])
```



Plot Over Default Range

Plot this equation over the default range.

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



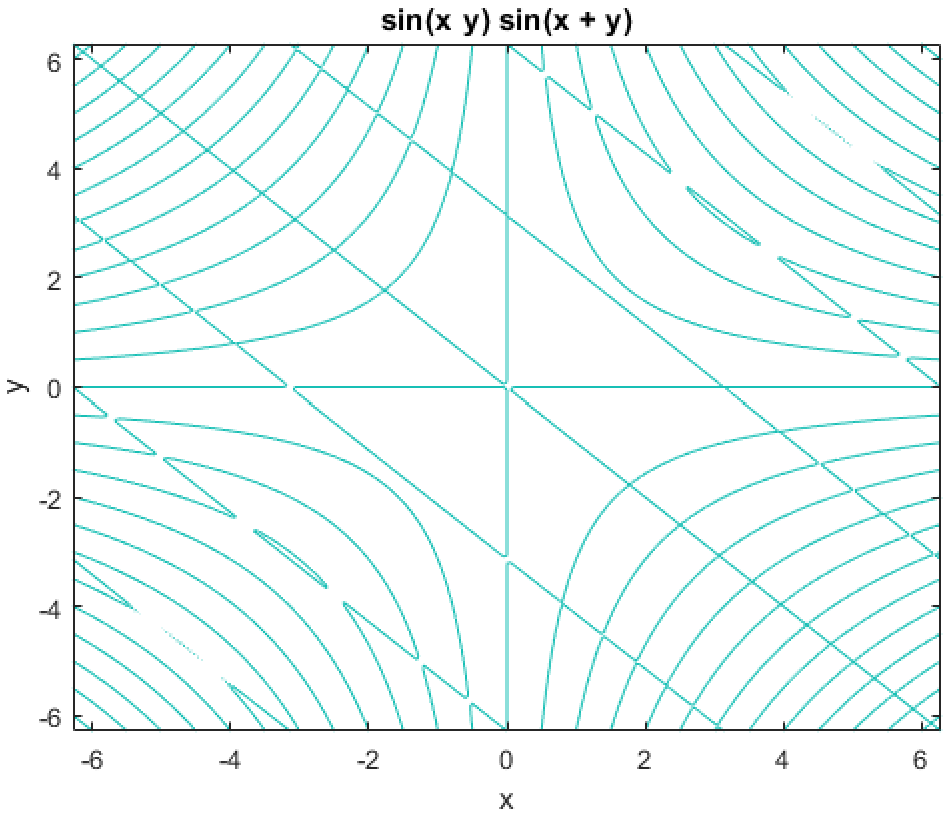
Plot Symbolic Function

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:

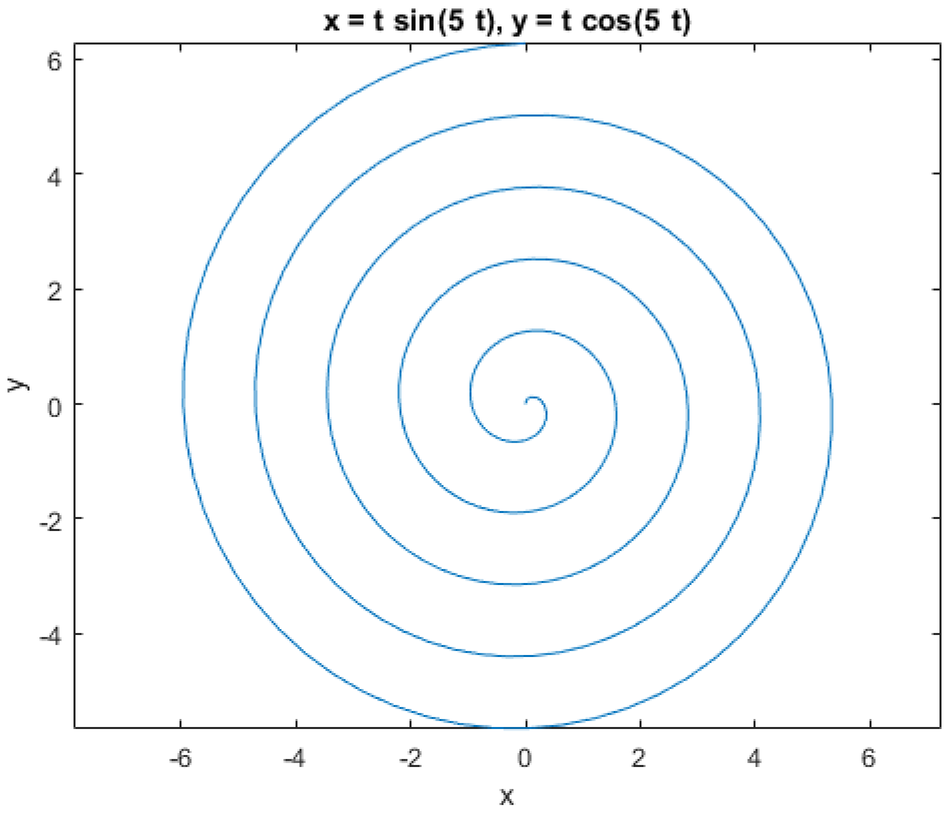
```
ezplot(f)
```



Plot Parametric Curve

Plot this parametric curve:

```
syms t
x = t*sin(5*t);
y = t*cos(5*t);
ezplot(x, y)
```



Tips

- If you do not specify a plot range, `ezplot` uses the interval [–2*π* 2*π*] as a starting point. Then it can choose to display a part of the plot over a subinterval of [–2*π* 2*π*] 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 `fig`.

- 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.

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `plot`

Topics

"Create Plots" on page 2-240

Introduced before R2006a

ezplot3

3-D parametric curve plotter

Note `ezplot3` is not recommended. Use `fplot3` instead.

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 $0 < \mathrm{t} < 2\pi$.

`ezplot3(x,y,z,[tmin,tmax])` plots the curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the domain $tmin < t < tmax$.

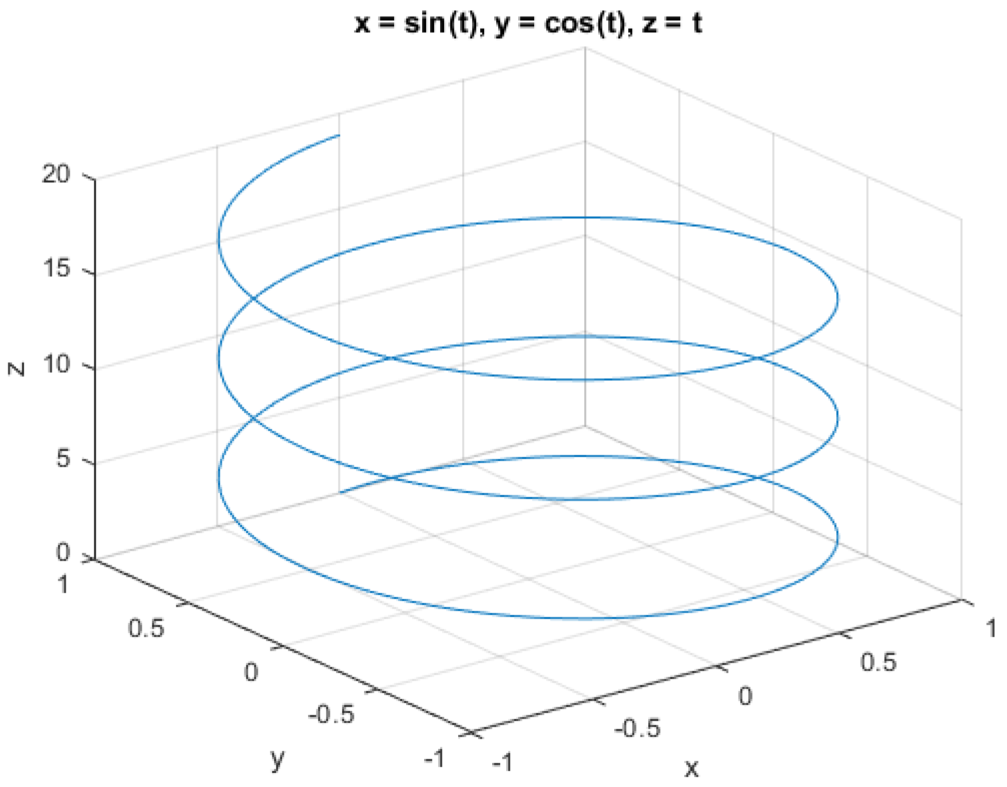
`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

Examples

3-D Parametric Curve

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

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `plot3`

Introduced before R2006a

ezpolar

Polar coordinate plotter

Syntax

```
ezpolar(f)
ezpolar(f, [a, b])
```

Description

`ezpolar(f)` plots the polar curve $r = f(\theta)$ over the default domain $0 < \theta < 2\pi$.

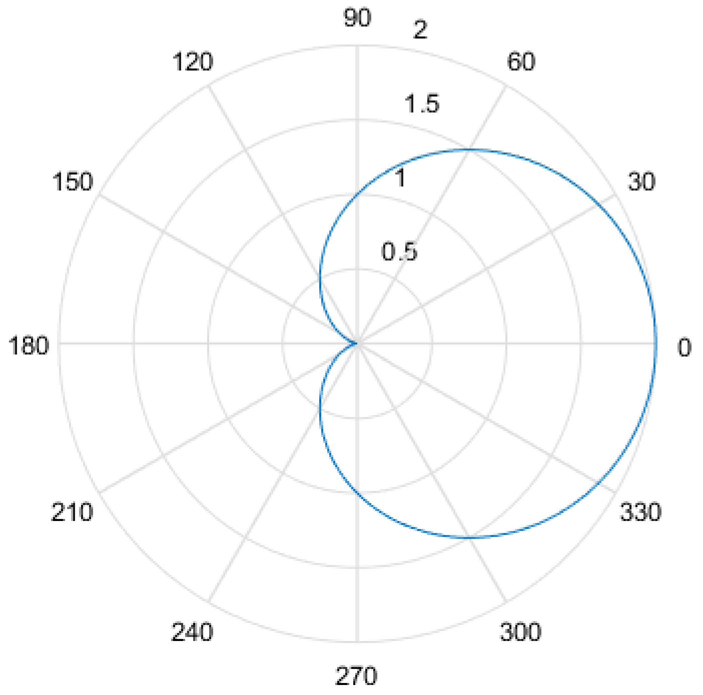
`ezpolar(f, [a, b])` plots f for `a` $< \theta <$ `b`.

Examples

Polar Plot of Symbolic Expression

This example creates a polar plot of the function, `1 + cos(t)`, over the domain $[0, 2\pi]$.

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



r = cos(t) + 1

Introduced before R2006a

ezsurf

Plot 3-D surface

Note `ezsurf` is not recommended. Use `fsurf` instead.

Syntax

```
ezsurf(f)
ezsurf(f,[xmin,xmax])
ezsurf(f,[xmin,xmax,ymin,ymax])

ezsurf(x,y,z)
ezsurf(x,y,z,[smin,smax])
ezsurf(x,y,z,[smin,smax,tmin,tmax])

ezsurf(___,n)
ezsurf(___,'circ')

h = ezsurf(___)
```

Description

`ezsurf(f)` plots a two-variable symbolic expression or function `f(x,y)` over the range `-2*pi < x < 2*pi`, `-2*pi < y < 2*pi`.

`ezsurf(f,[xmin,xmax])` plots `f(x,y)` over the specified range `xmin < x < xmax`. This is the range along the abscissa (horizontal axis).

`ezsurf(f,[xmin,xmax,ymin,ymax])` plots `f(x,y)` over the specified ranges along the abscissa, `xmin < x < xmax`, and the ordinate, `ymin < y < ymax`.

When determining the range values, `ezsurf` sorts variables alphabetically. For example, `ezsurf(x^2 - a^3, [0,1,3,6])` plots `x^2 - a^3` over `0 < a < 1`, `3 < x < 6`.

`ezsurf(x,y,z)` plots the parametric surface `x = x(s,t)`, `y = y(s,t)`, `z = z(s,t)` over the range `-2*pi < s < 2*pi`, `-2*pi < t < 2*pi`.

`ezsurf(x,y,z,[smin,smax])` plots the parametric surface `x = x(s,t)`, `y = y(s,t)`, `z = z(s,t)` over the specified range `smin < s < smax`.

`ezsurf(x,y,z,[smin,smax,tmin,tmax])` plots the parametric surface `x = x(s,t)`, `y = y(s,t)`, `z = z(s,t)` over the specified ranges `smin < s < smax` and `tmin < t < tmax`.

`ezsurf(___,n)` specifies the grid. You can specify `n` after the input arguments in any of the previous syntaxes. By default, `n = 60`.

`ezsurf(___,'circ')` creates the surface plot over a disk centered on the range. You can specify`'circ'` after the input arguments in any of the previous syntaxes.

`h = ezsurf(___)` returns a handle `h` to the surface plot object. You can use the output argument `h` with any of the previous syntaxes.

Examples

Plot Function Over Default Range

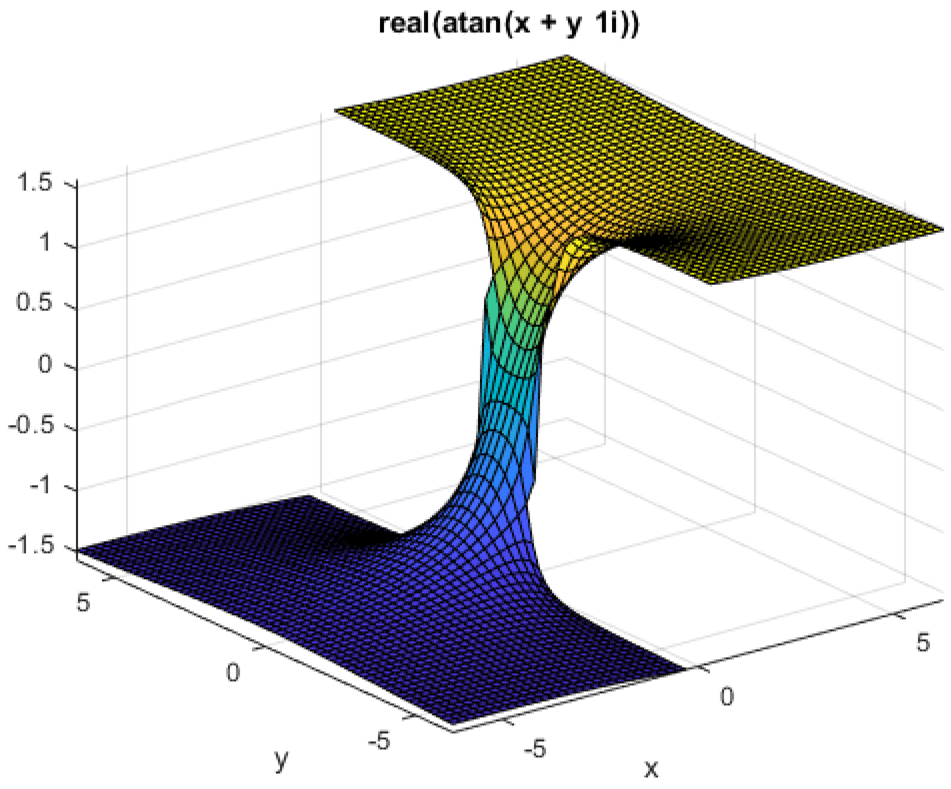
Plot the symbolic function `f(x,y) = real(atan(x + i*y))` over the default range `-2*pi < x < 2*pi`, `-2*pi < y < 2*pi`.

Create the symbolic function.

```
syms f(x,y)
f(x,y) = real(atan(x + i*y));
```

Plot this function using `ezsurf`.

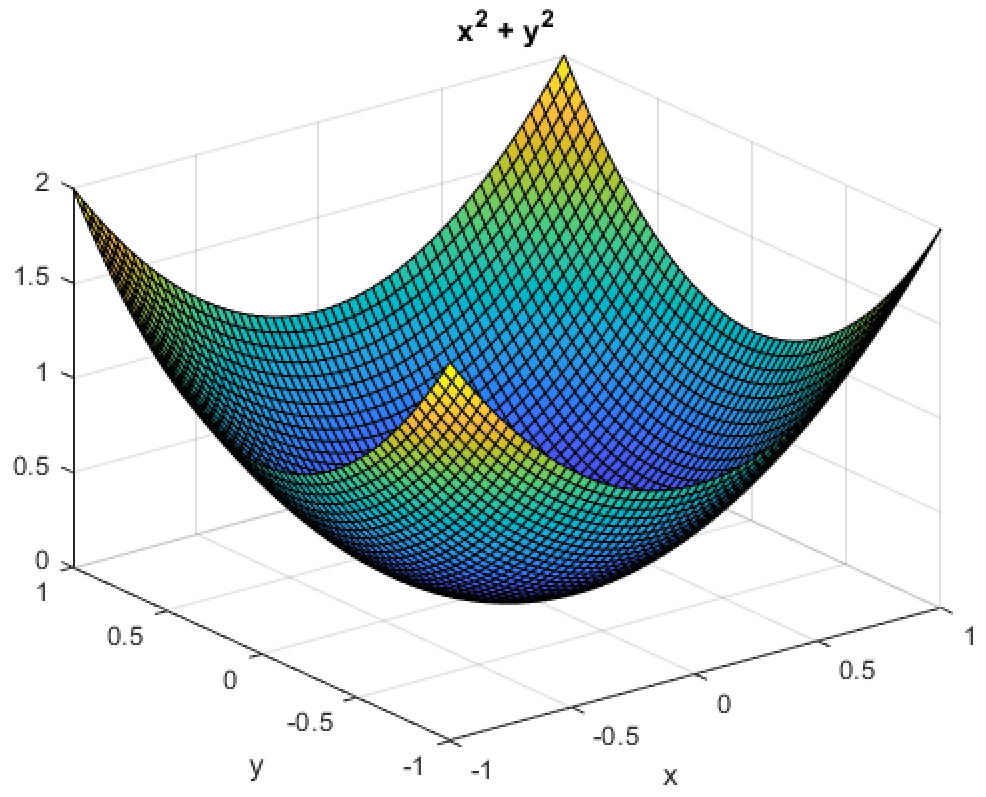
```
ezsurf(f)
```



Specify Plotting Ranges

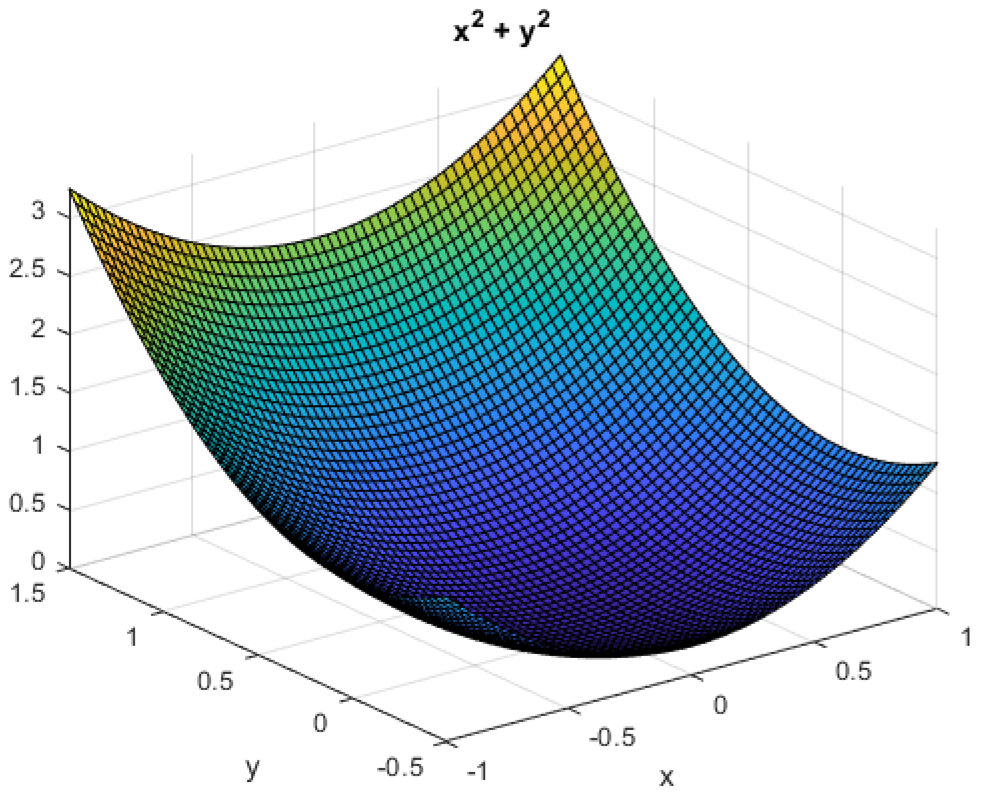
Plot the symbolic expression `x^2 + y^2` over the range `-1 < x < 1`. Because you do not specify the range for the `y`-axis, `ezsurf` chooses it automatically.

```
syms x y
ezsurf(x^2 + y^2, [-1, 1])
```



Specify the range for both axes.

```
ezsurf(x^2 + y^2, [-1, 1, -0.5, 1.5])
```



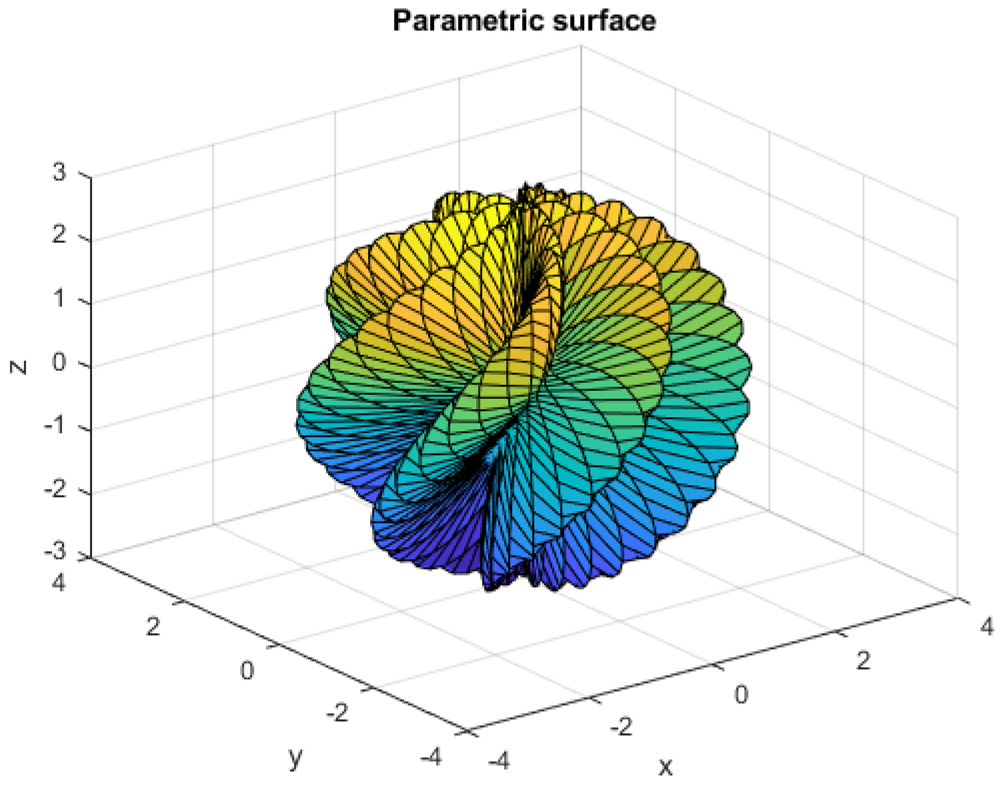
Plot Parameterized Surface

Define the parametric surface `x(s,t), y(s,t), z(s,t)` as follows.

```
syms s t
r = 2 + sin(7*s + 5*t);
x = r*cos(s)*sin(t);
y = r*sin(s)*sin(t);
z = r*cos(t);
```

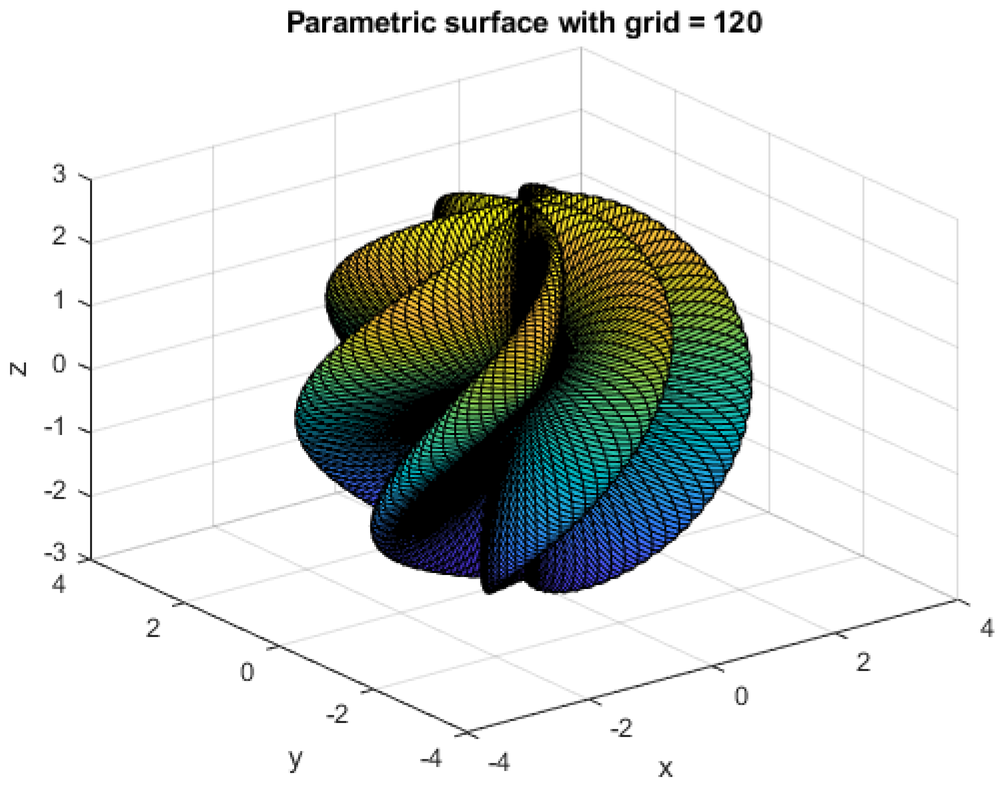
Plot the function using `ezsurf`.

```
ezsurf(x, y, z, [0, 2*pi, 0, pi])
title('Parametric surface')
```



To create a smoother plot, increase the number of mesh points.

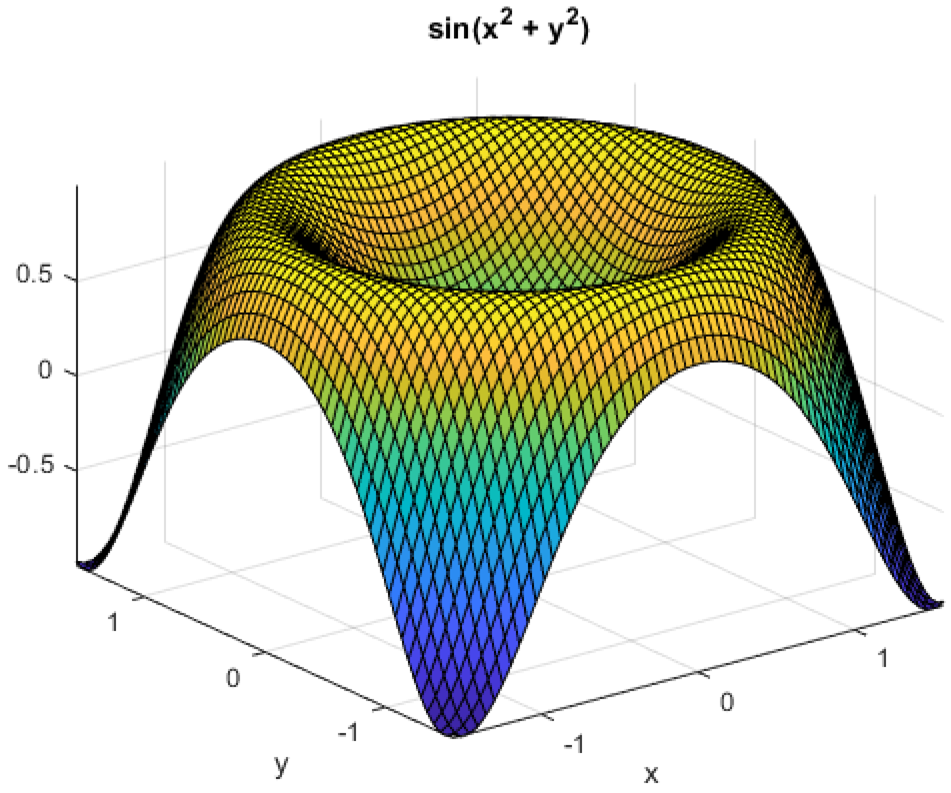
```
ezsurf(x, y, z, [0, 2*pi, 0, pi], 120)
title('Parametric surface with grid = 120')
```



Specify Disk Plotting Range

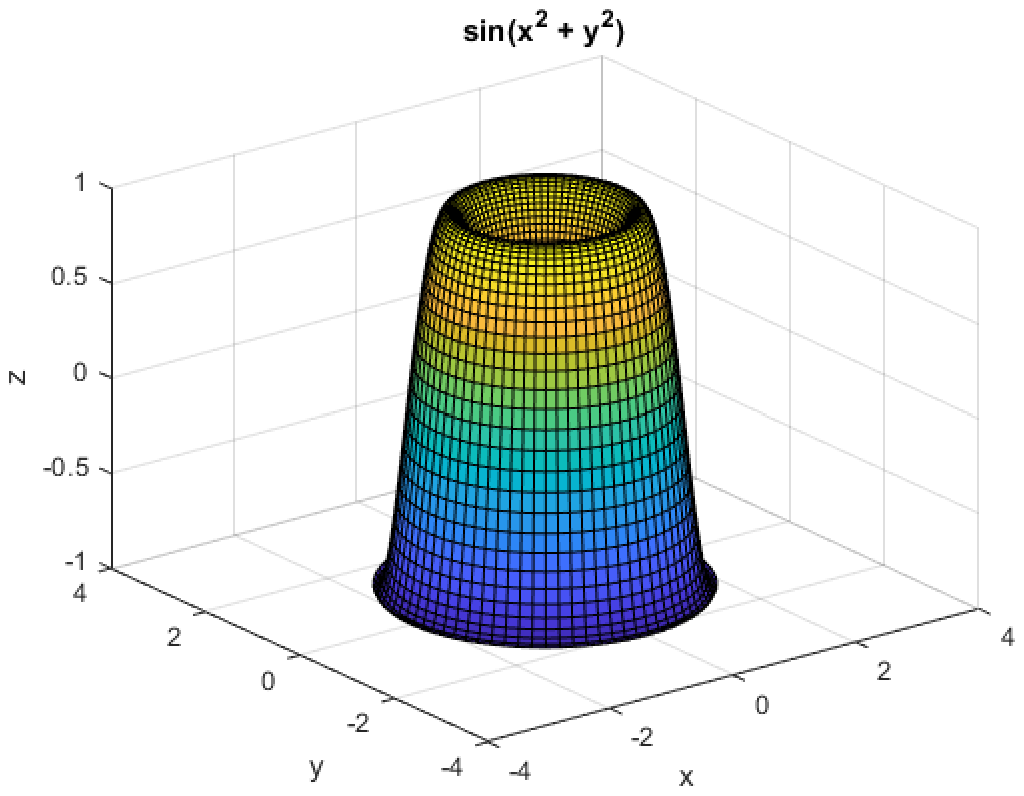
First, plot the expression `sin(x^2 + y^2)` over the square range `-pi/2 < x < pi/2`, `-pi/2 < y < pi/2`.

```
syms x y
ezsurf(sin(x^2 + y^2), [-pi/2, pi/2, -pi/2, pi/2])
```



Now, plot the same expression over the disk range.

```
ezsurf(sin(x^2 + y^2), [-pi/2, pi/2, -pi/2, pi/2],'circ')
```



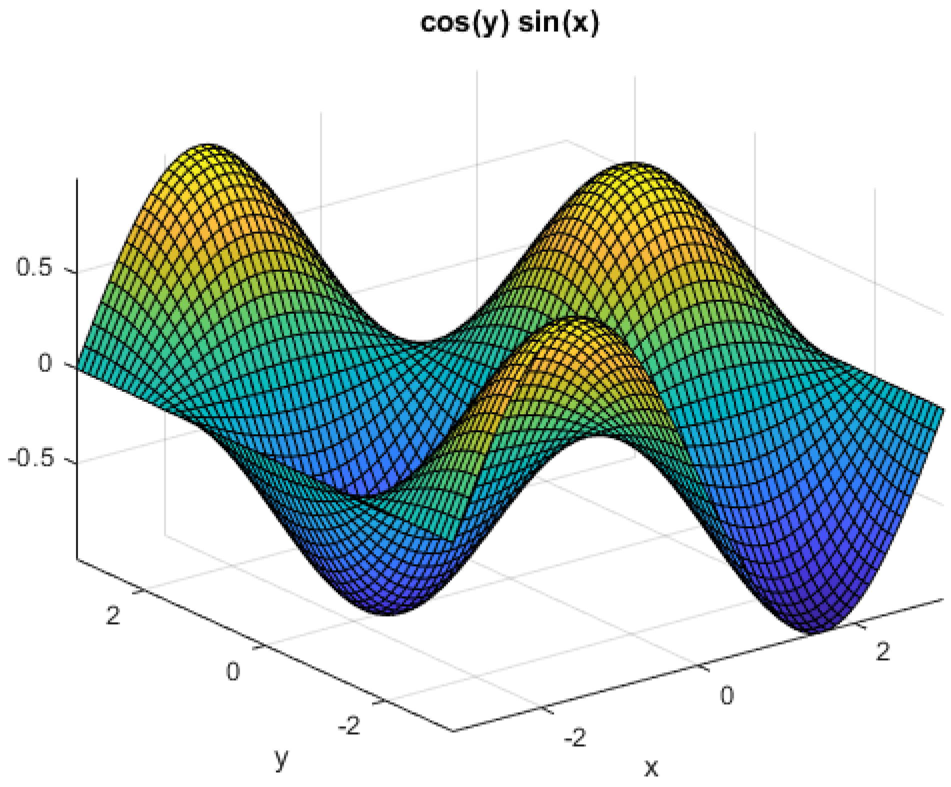
Use Handle to Surface Plot

Plot the symbolic expression `sin(x)cos(x)`, and assign the result to the handle `h`.

```
syms x y
h = ezsurf(sin(x)*cos(y), [-pi, pi])
```

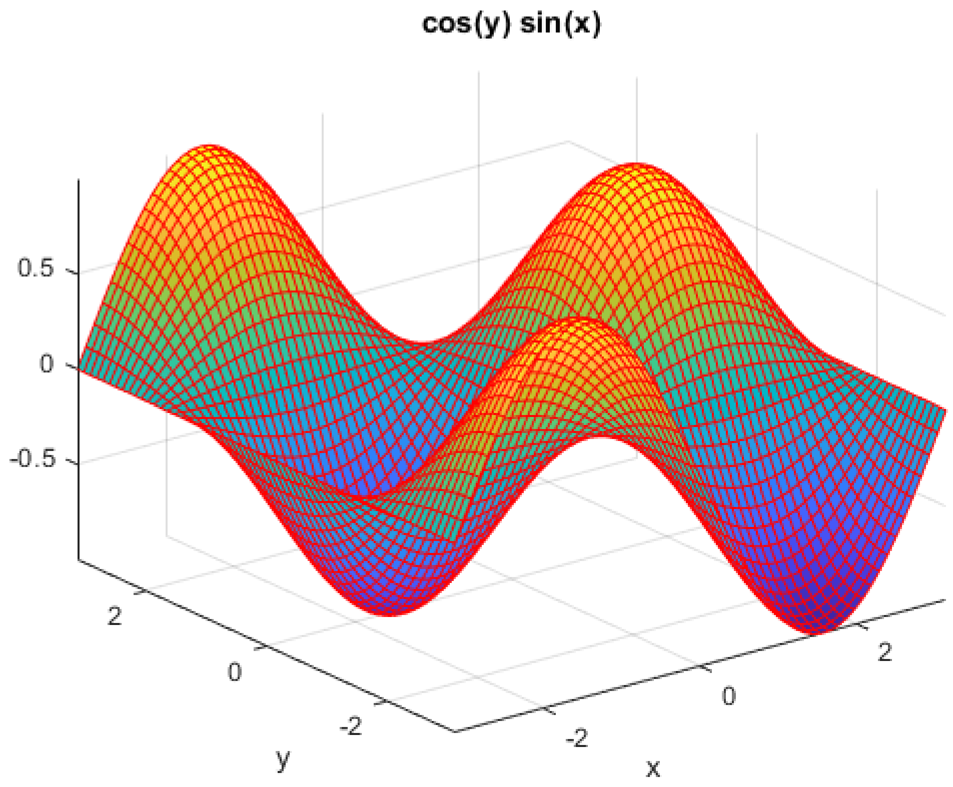
```
h =
  Surface with properties:

       EdgeColor: [0 0 0]
       LineStyle: '-'
```

```
      FaceColor: 'flat'
   FaceLighting: 'flat'
      FaceAlpha: 1
          XData: [60x60 double]
          YData: [60x60 double]
          ZData: [60x60 double]
          CData: [60x60 double]

Show all properties
```



You can use this handle to change properties of the plot. For example, change the color of the area outline.

```
h.EdgeColor = 'red'

h =
  Surface with properties:

       EdgeColor: [1 0 0]
       LineStyle: '-'
       FaceColor: 'flat'
    FaceLighting: 'flat'
       FaceAlpha: 1
           XData: [60x60 double]
           YData: [60x60 double]
           ZData: [60x60 double]
           CData: [60x60 double]

  Show all properties
```



- “Create Plots” on page 2-240

Input Arguments

`f` — Function to plot

symbolic expression with two variables | symbolic function of two variables

Function to plot, specified as a symbolic expression or function of two variables.

Example: `ezsurf(x^2 + y^2)`

x,y,z — Parametric function to plot

three symbolic expressions with two variables | three symbolic functions of two variables

Parametric function to plot, specified as three symbolic expressions or functions of two variables.

Example: `ezsurf(s*cos(t), s*sin(t), t)`

n — Grid value

integer

Grid value, specified as an integer. The default grid value is `60`.

Output Arguments

h — Surface plot handle

scalar

Surface plot handle, returned as a scalar. It is a unique identifier, which you can use to query and modify properties of the surface plot.

Tips

- `ezsurf` chooses the computational grid according to the amount of variation that occurs. If `f` is singular for some points on the grid, then `ezsurf` omits these points. The value at these points is set to `NaN`.

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `surf`

Topics

"Create Plots" on page 2-240

Introduced before R2006a

ezsurfc

Combined surface and contour plotter

Note `ezsurfc` is not recommended. Use `fsurf` instead.

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

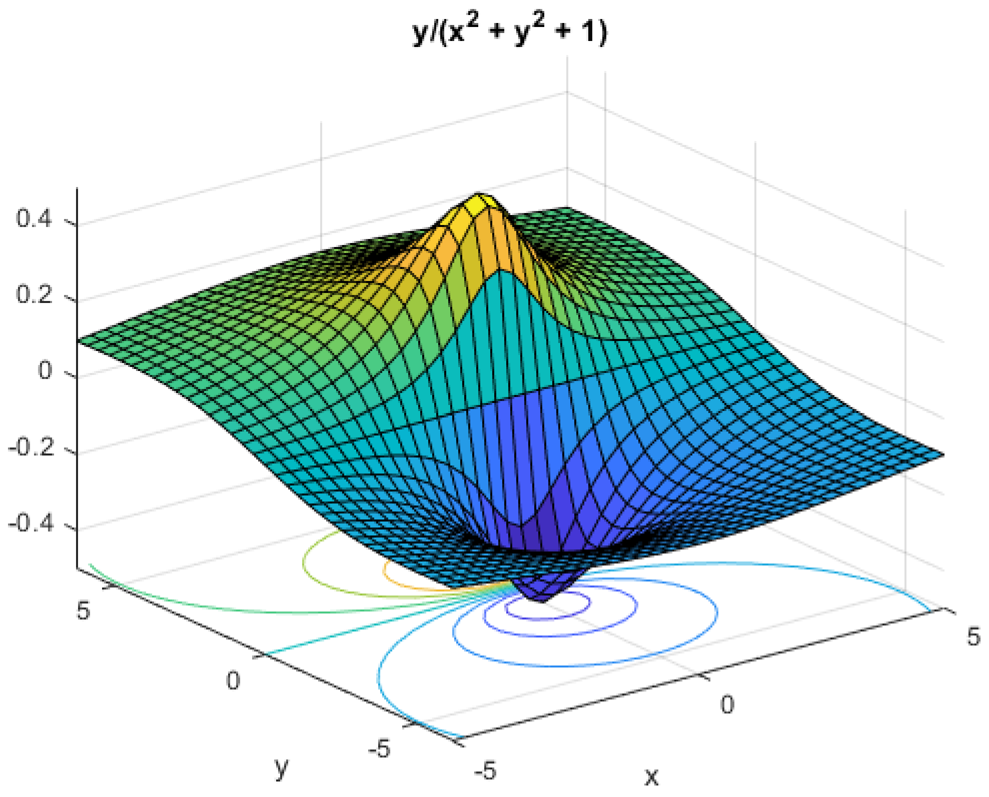
3-D Surface Plot with Contours

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. Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65 and elevation = 26).

```
syms x y
ezsurfc(y/(1 + x^2 + y^2),[-5,5,-2*pi,2*pi],35)
```



See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `surfc`

Introduced before R2006a

factor

Factorization

Syntax

```
F = factor(x)
F = factor(x,vars)
F = factor( ___ ,Name,Value)
```

Description

`F = factor(x)` returns all irreducible factors of `x` in vector `F`. If `x` is an integer, `factor` returns the prime factorization of `x`. If `x` is a symbolic expression, `factor` returns the subexpressions that are factors of `x`.

`F = factor(x,vars)` returns an array of factors `F`, where `vars` specifies the variables of interest. All factors not containing a variable in `vars` are separated into the first entry `F(1)`. The other entries are irreducible factors of `x` that contain one or more variables from `vars`.

`F = factor( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. This syntax can use any of the input arguments from the previous syntaxes.

Examples

Factor Integer Numbers

```
F = factor(823429252)
```

```
F =
           2           2          59         283       12329
```

To factor integers greater than `flintmax`, convert the integer to a symbolic object using `sym`. Then place the number in quotation marks to represent it accurately.

```
F = factor(sym('82342925225632328'))
```

```
F =
[ 2, 2, 2, 251, 401, 18311, 5584781]
```

To factor a negative integer, convert it to a symbolic object using `sym`.

```
F = factor(sym(-92465))
```

```
F =
[ -1, 5, 18493]
```

Perform Prime Factorization of Large Numbers

Perform prime factorization for `41758540882408627201`. Since the integer is greater than `flintmax`, convert it to a symbolic object using `sym`, and place the number in quotation marks to represent it accurately.

```
n = sym('41758540882408627201');
factor(n)
```

```
ans =
[ 479001599, 87178291199]
```

Factor Symbolic Fractions

Factor the fraction `112/81` by converting it into a symbolic object using `sym`.

```
F = factor(sym(112/81))
```

```
F =
[ 2, 2, 2, 2, 7, 1/3, 1/3, 1/3, 1/3]
```

Factor Polynomials

Factor the polynomial `x^6-1`.

```
syms x
F = factor(x^6-1)
```

```
F =
[ x - 1, x + 1, x^2 + x + 1, x^2 - x + 1]
```

Factor the polynomial `y^6-x^6`.

```
syms y
F = factor(y^6-x^6)
```

```
F =
[ -1, x - y, x + y, x^2 + x*y + y^2, x^2 - x*y + y^2]
```

Separate Factors Containing Specified Variables

Factor `y^2*x^2` for factors containing `x`.

```
syms x y
F = factor(y^2*x^2,x)
```

```
F =
[ y^2, x, x]
```

`factor` combines all factors without `x` into the first element. The remaining elements of `F` contain irreducible factors that contain `x`.

Factor the polynomial `y` for factors containing symbolic variables `b` and `c`.

```
syms a b c d
y = -a*b^5*c*d*(a^2 - 1)*(a*d - b*c);
F = factor(y,[b c])
```

```
F =
[ -a*d*(a - 1)*(a + 1), b, b, b, b, b, c, a*d - b*c]
```

`factor` combines all factors without `b` or `c` into the first element of `F`. The remaining elements of `F` contain irreducible factors of `y` that contain either `b` or `c`.

Choose Factorization Modes

Use the `FactorMode` argument to choose a particular factorization mode.

Factor an expression without specifying the factorization mode. By default, `factor` uses factorization over rational numbers. In this mode, `factor` keeps rational numbers in their exact symbolic form.

```
syms x
factor(x^3 + 2, x)
```

```
ans =
x^3 + 2
```

Factor the same expression, but this time use numeric factorization over real numbers. This mode factors the expression into linear and quadratic irreducible polynomials with real coefficients and converts all numeric values to floating-point numbers.

```
factor(x^3 + 2, x, 'FactorMode', 'real')
```

```
ans =
[ x + 1.2599210498948731647672106072782,...
  x^2 - 1.2599210498948731647672106072782*x + 1.5874010519681994747517056392723]
```

Factor this expression using factorization over complex numbers. In this mode, `factor` reduces quadratic polynomials to linear expressions with complex coefficients. This mode converts all numeric values to floating-point numbers.

```
factor(x^3 + 2, x, 'FactorMode', 'complex')
```

```
ans =
[ x + 1.2599210498948731647672106072782,...
  x - 0.62996052494743658238360530363911 + 1.0911236359717214035600726141898i,...
  x - 0.62996052494743658238360530363911 - 1.0911236359717214035600726141898i]
```

Factor this expression using the full factorization mode. This mode factors the expression into linear expressions, reducing quadratic polynomials to linear expressions with complex coefficients. This mode keeps rational numbers in their exact symbolic form.

```
factor(x^3 + 2, x, 'FactorMode', 'full')
```

```
ans =
[ x + 2^(1/3),...
  x - 2^(1/3)*((3^(1/2)*1i)/2 + 1/2),...
  x + 2^(1/3)*((3^(1/2)*1i)/2 - 1/2)]
```

Approximate the result with floating-point numbers by using `vpa`. Because the expression does not contain any symbolic parameters besides the variable `x`, the result is the same as in complex factorization mode.

```
vpa(ans)
```

```
ans =
[ x + 1.2599210498948731647672106072782,...
```

```
  x - 0.62996052494743658238360530363911 - 1.0911236359717214035600726141898i,...
  x - 0.62996052494743658238360530363911 + 1.0911236359717214035600726141898i]
```

Approximate Results Containing `RootOf`

In the full factorization mode,`factor` also can return results as a symbolic sums over polynomial roots expressed as `RootOf`.

Factor this expression.

```
syms x
s = factor(x^3 + x - 3, x, 'FactorMode','full')
```

```
s =
[ x - root(z^3 + z - 3, z, 1),...
  x - root(z^3 + z - 3, z, 2),...
  x - root(z^3 + z - 3, z, 3)]
```

Approximate the result with floating-point numbers by using `vpa`.

```
 vpa(s)
```

```
ans =
[ x - 1.2134116627622296341321313773815,...
  x + 0.60670583138111481706606568869074 + 1.450612249188441526515442203395i,...
  x + 0.60670583138111481706606568869074 - 1.450612249188441526515442203395i]
```

Input Arguments

`x` — Input to factor

number | symbolic number | symbolic expression | symbolic function

Input to factor, specified as a number, or a symbolic number, expression, or function.

`vars` — Variables of interest

symbolic variable | vector of symbolic variables

Variables of interest, specified as a symbolic variable or a vector of symbolic variables. Factors that do not contain a variable specified in `vars` are grouped into the first element of `F`. The remaining elements of `F` contain irreducible factors of `x` that contain a variable in `vars`.

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`.

Example: `factor(x^3 - 2,x,'FactorMode','real')`

`FactorMode` — Factorization mode

`'rational'` (default) | `'real'` | `'complex'` | `'full'`

Factorization mode, specified as the comma-separated pair consisting of `'FactorMode'` and one of these character vectors.

`'rational'`	Factorization over rational numbers.
`'real'`	Factorization over real numbers. A real numeric factorization is a factorization into linear and quadratic irreducible polynomials with real coefficients. This factorization mode requires the coefficients of the input to be convertible to real floating-point numbers. All other inputs (for example, those inputs containing symbolic or complex coefficients) are treated as irreducible.
`'complex'`	Factorization over complex numbers. A complex numeric factorization is a factorization into linear factors whose coefficients are floating-point numbers. Such factorization is only available if the coefficients of the input are convertible to floating-point numbers, that is, if the roots can be determined numerically. Symbolic inputs are treated as irreducible.
`'full'`	Full factorization. A full factorization is a symbolic factorization into linear factors. The result shows these factors using radicals or as a `symsum` ranging over a `RootOf`.

Output Arguments

`F` — Factors of input

symbolic vector

Factors of input, returned as a symbolic vector.

Tips

- To factor an integer greater than `flintmax`, wrap the integer with `sym`. Then place the integer in quotation marks to represent it accurately, for example, `sym('465971235659856452')`.
- To factor a negative integer, wrap the integer with `sym`, for example, `sym(-3)`.

See Also

`collect` | `combine` | `divisors` | `expand` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

Introduced before R2006a

factorial

Factorial function

Syntax

```
factorial(n)
factorial(A)
```

Description

`factorial(n)` returns the factorial on page 4-551 of `n`.

`factorial(A)` returns the factorials of each element of `A`.

Input Arguments

`n`

Symbolic variable or expression representing a nonnegative integer.

`A`

Vector or matrix of symbolic variables or expressions representing nonnegative integers.

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]
```

Definitions

Factorial Function

This product defines the factorial function of a positive integer:

$$n! = \prod_{k=1}^{n} k$$

The factorial function 0! = 1.

Tips

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

See Also

`beta` | `gamma` | `nchoosek` | `psi`

Introduced in R2012a

fcontour

Plot contours

Syntax

```
fcontour(f)
fcontour(f,[min max])
fcontour(f,[xmin xmax ymin ymax])

fcontour( ___ ,LineSpec)
fcontour( ___ ,Name,Value)
fcontour(ax, ___ )
fc = fcontour( ___ )
```

Description

`fcontour(f)` plots the contour lines of symbolic expression *f*(*x*,*y*) over the default interval of `x` and *y*, which is `[-5 5]`.

`fcontour(f,[min max])` plots `f` over the interval `min` < `x` < `max` and `min` < `y` < `max`.

`fcontour(f,[xmin xmax ymin ymax])` plots `f` over the interval `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`. `fcontour` uses `symvar` to order the variables and assign intervals.

`fcontour( ___ ,LineSpec)` uses `LineSpec` to set the line style and color. `fcontour` doesn't support markers.

`fcontour( ___ ,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. `Name,Value` pair settings apply to all the lines plotted. To set options for individual plots, use the objects returned by `fcontour`.

`fcontour(ax, ___ )` plots into the axes object `ax` instead of the current axes object `gca`.

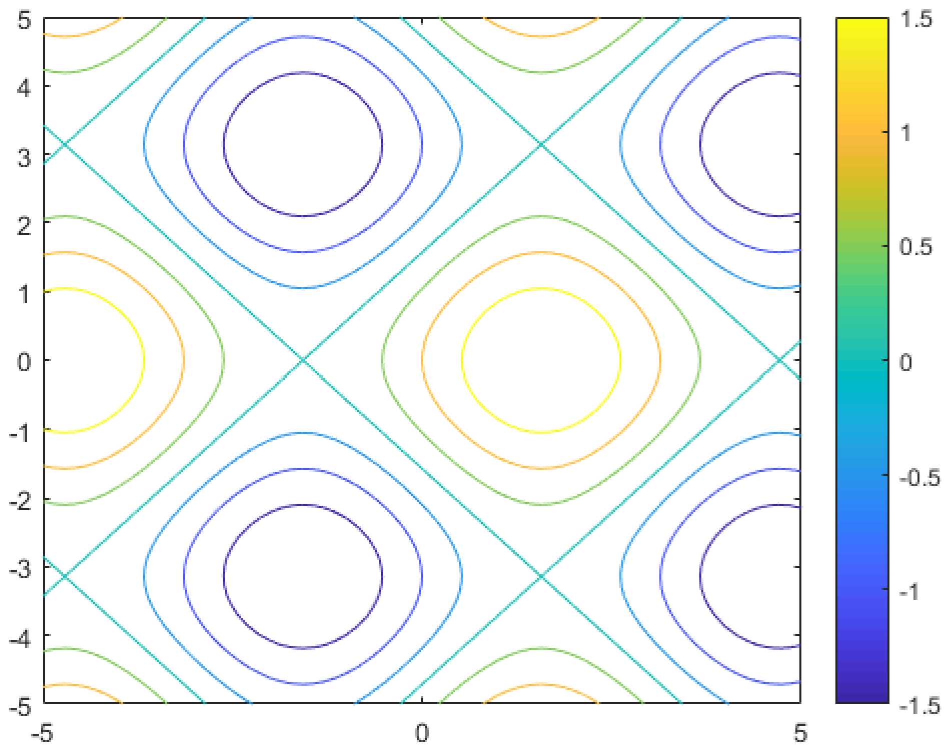
`fc = fcontour( ___ )` returns a function contour object. Use the object to query and modify properties of a specific contour plot. For details, see Function Contour.

Examples

Plot Contours of Symbolic Expression

Plot the contours of $\sin(x) + \cos(y)$ over the default range of $-5 < x < 5$ and $-5 < y < 5$. Show the colorbar. Find a contour's level by matching the contour's color with the colorbar value.

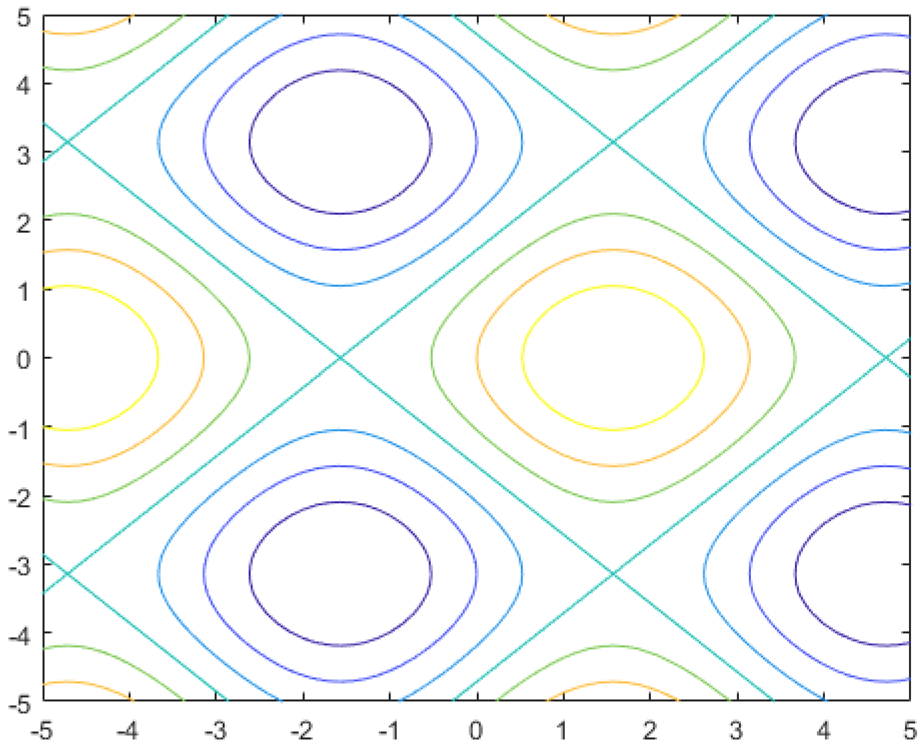
```
syms x y
fcontour(sin(x) + cos(y))
colorbar
```



Plot Contours of Symbolic Function

Plot the contours of $f(x,y) = \sin(x) + \cos(y)$ over the default range of $-5 < x < 5$ and $-5 < y < 5$.

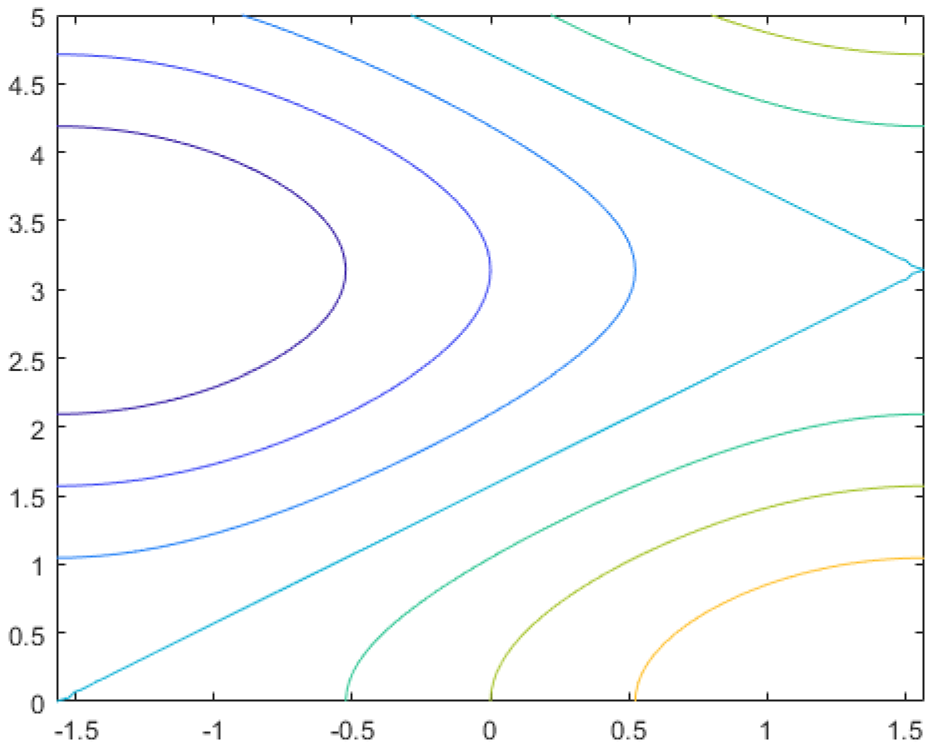
```
syms f(x,y)
f(x,y) = sin(x) + cos(y);
fcontour(f)
```



Specify Plotting Interval

Plot $\sin(x)+\cos(y)$ over $-\pi/2 < x < \pi/2$ and $0 < y < 5$ by specifying the plotting interval as the second argument of `fcontour`.

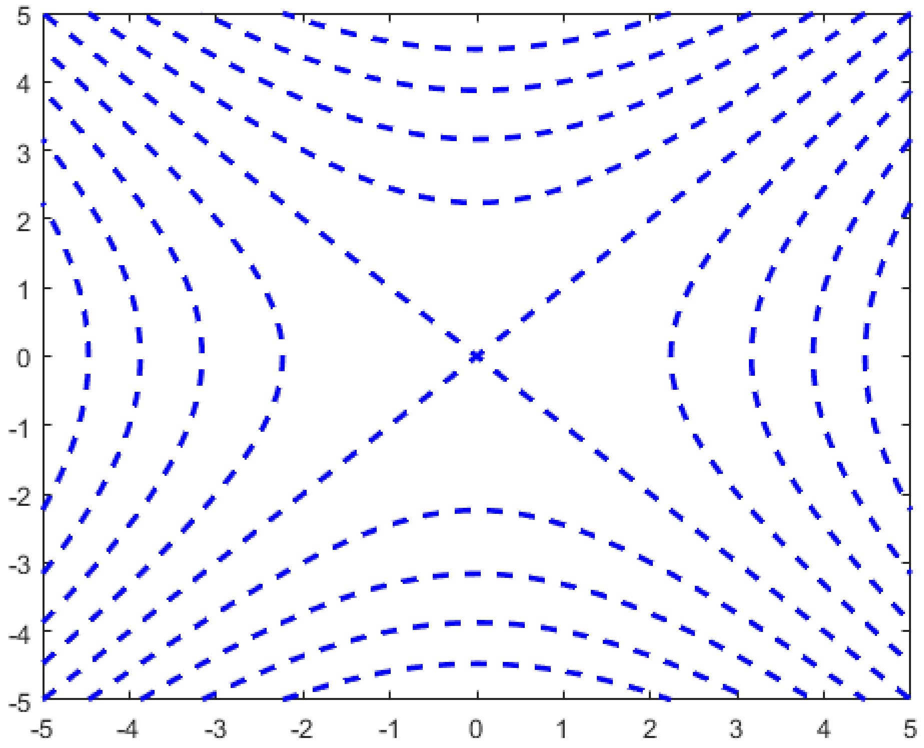
```
syms x y
f = sin(x) + cos(y);
fcontour(f,[-pi/2 pi/2 0 5])
```



Change Line Style, Color and Width

Plot the contours of $x^2 - y^2$ as blue, dashed lines by specifying the `LineSpec` input. Specify a `LineWidth` of `2`. Markers are not supported by `fcontour`.

```
syms x y
fcontour(x^2 - y^2,'--b','LineWidth',2)
```

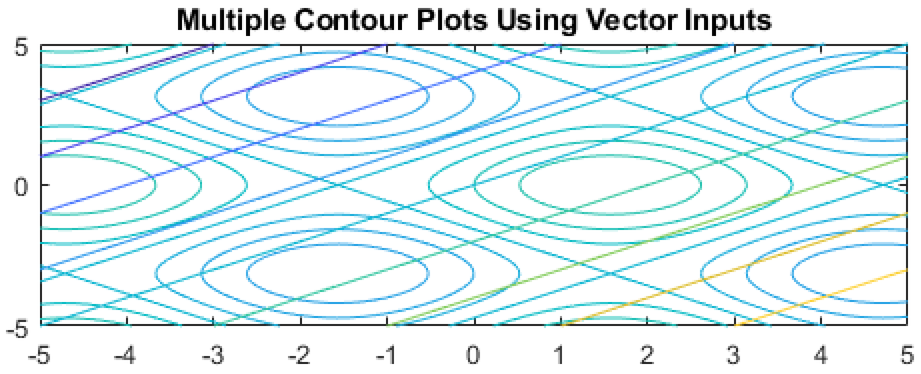


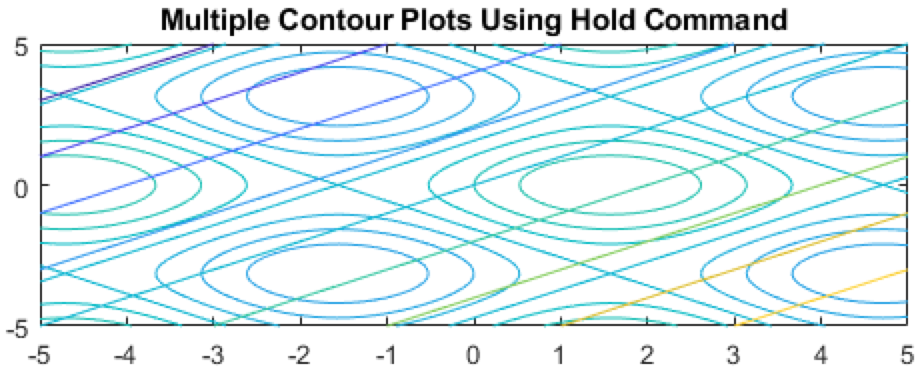
Plot Multiple Contour Plots on Same Figure

Plot multiple contour plots either by passing the inputs as a vector or by using `hold on` to successively plot on the same figure. If you specify `LineStyle` and Name-Value arguments, they apply to all contour plots. You cannot specify individual `LineStyle` and Name-Value pair arguments for each plot.

Divide a figure into two subplots by using `subplot`. On the first subplot, plot $\sin(x) + \cos(y)$ and $x - y$ by using vector input. On the second subplot, plot the same expressions by using `hold on`.

```
syms x y
subplot(2,1,1)
fcontour([sin(x)+cos(y) x-y])
title('Multiple Contour Plots Using Vector Inputs')

subplot(2,1,2)
fcontour(sin(x)+cos(y))
hold on
fcontour(x-y)
title('Multiple Contour Plots Using Hold Command')

hold off
```



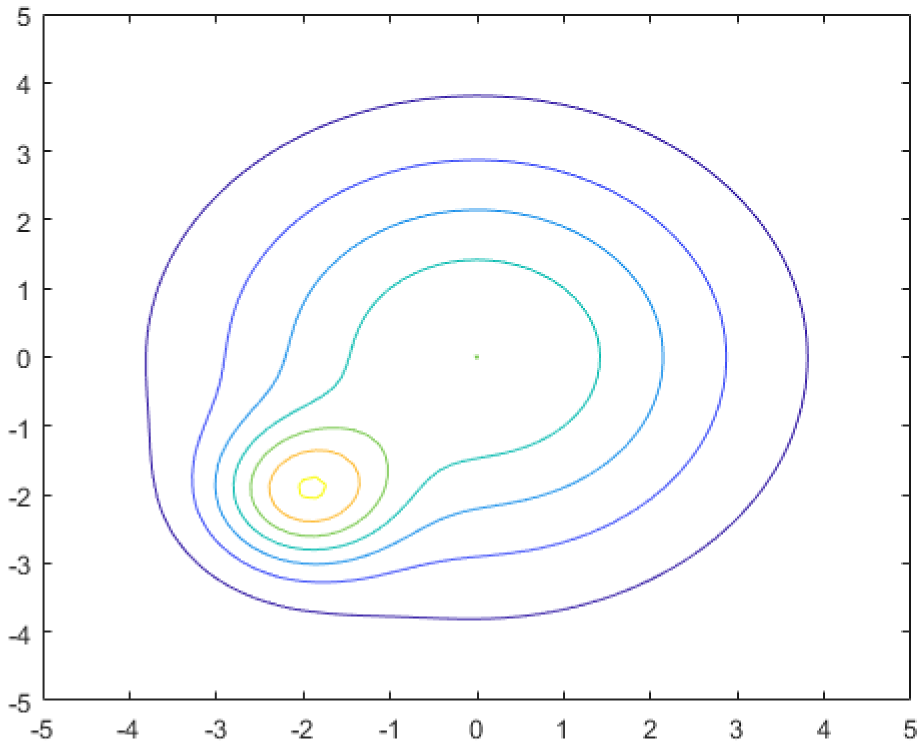


Modify Contour Plot After Creation

Plot the contours of $e^{-(x/3)^2-(y/3)^2}+e^{-(x+2)^2-(y+2)^2}$. Specify an output to make `fcontour` return the plot object.

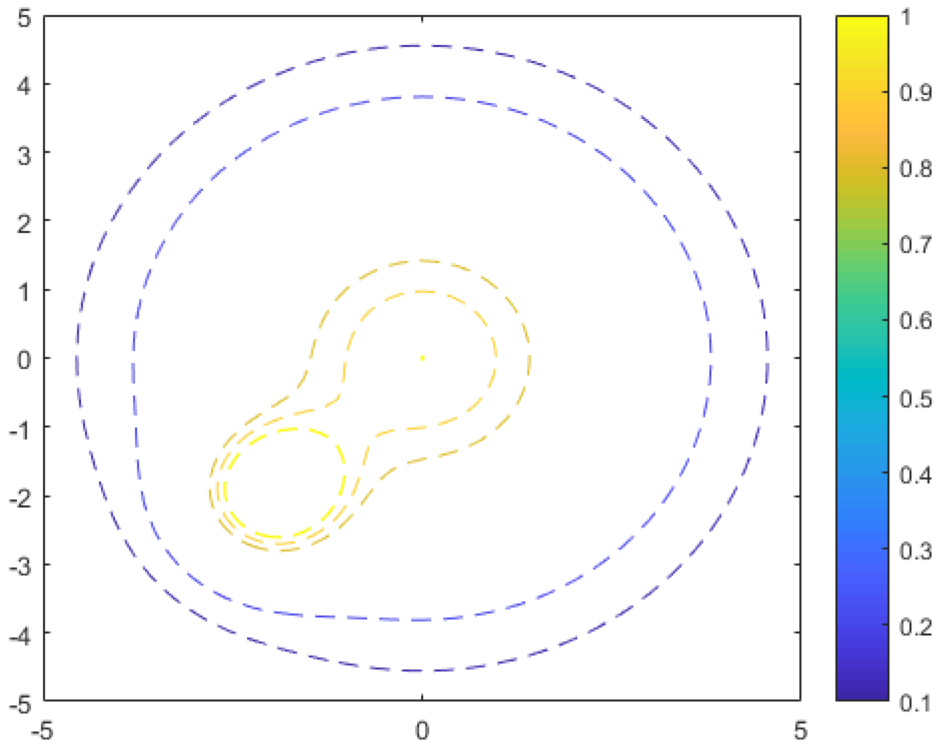
```
syms x y
f = exp(-(x/3)^2-(y/3)^2) + exp(-(x+2)^2-(y+2)^2);
fc = fcontour(f)
```

```
fc = 
  FunctionContour with properties:
     Function: [1x1 sym]
    LineColor: 'flat'
    LineStyle: '-'
    LineWidth: 0.5000
         Fill: 'off'
    LevelList: [0.2000 0.4000 0.6000 0.8000 1 1.2000 1.4000]

  Show all properties
```



Change the `LineWidth` to `1` and the `LineStyle` to a dashed line by using dot notation to set properties of the object `fc`. Visualize contours close to `0` and `1` by setting `LevelList` to `[1 0.9 0.8 0.2 0.1]`.

```
fc.LineStyle = '--';
fc.LineWidth = 1;
fc.LevelList = [1 0.9 0.8 0.2 0.1];
colorbar
```



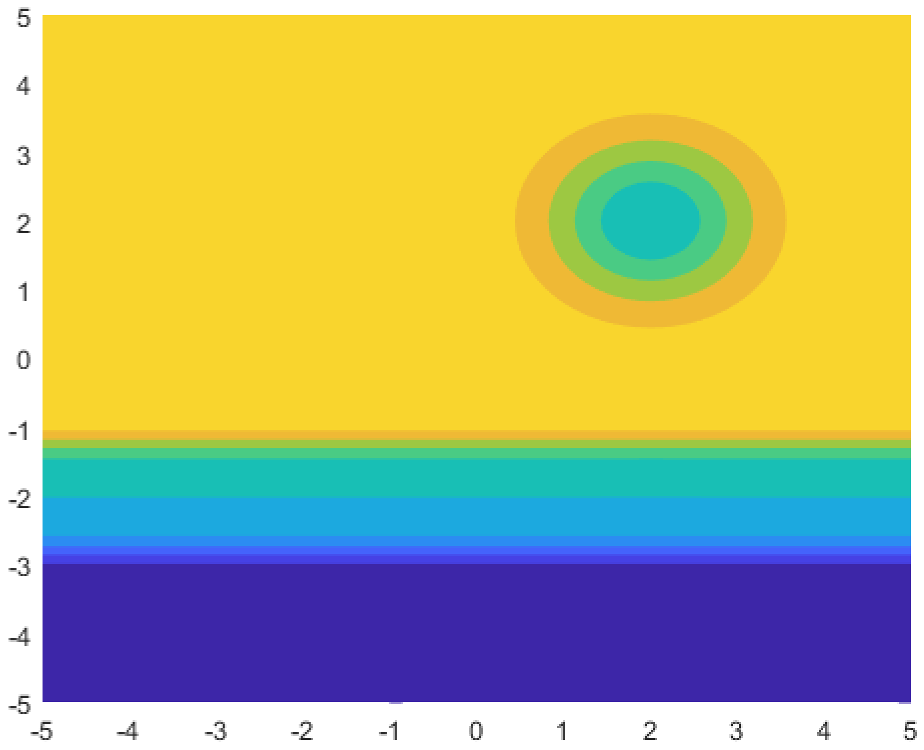
Fill Area Between Contours

Fill the area between contours by setting the `Fill` input of `fcontour` to `'on'`. If you want interpolated shading instead, use the `fsurf` function with its option `'EdgeColor'` set to `'none'` followed by the command `view(0,90)`.

Create a plot that looks like a sunset by filling the contours of

$\mathrm{erf}((y+2)^3) - e^{(-0.65((x-2)^2+(y-2)^2)}$.

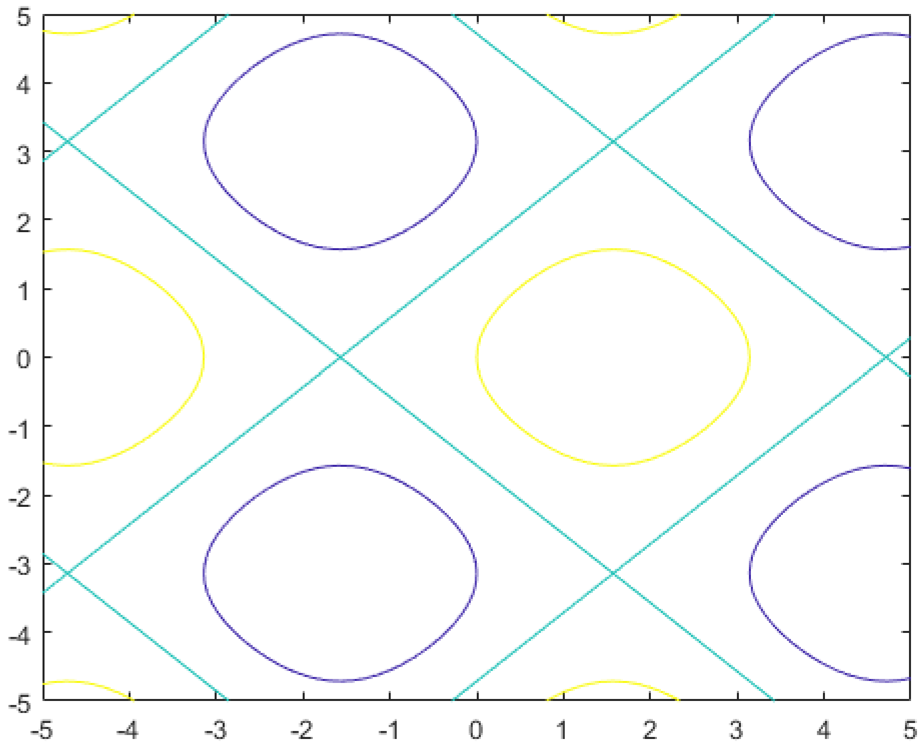
```
syms x y
f = erf((y+2)^3) - exp(-0.65*((x-2)^2+(y-2)^2));
fcontour(f,'Fill','on')
```



Specify Levels for Contour Lines

Set the values at which `fcontour` draws contours by using the `'LevelList'` option.

```
syms x y
f = sin(x) + cos(y);
fcontour(f,'LevelList',[-1 0 1])
```



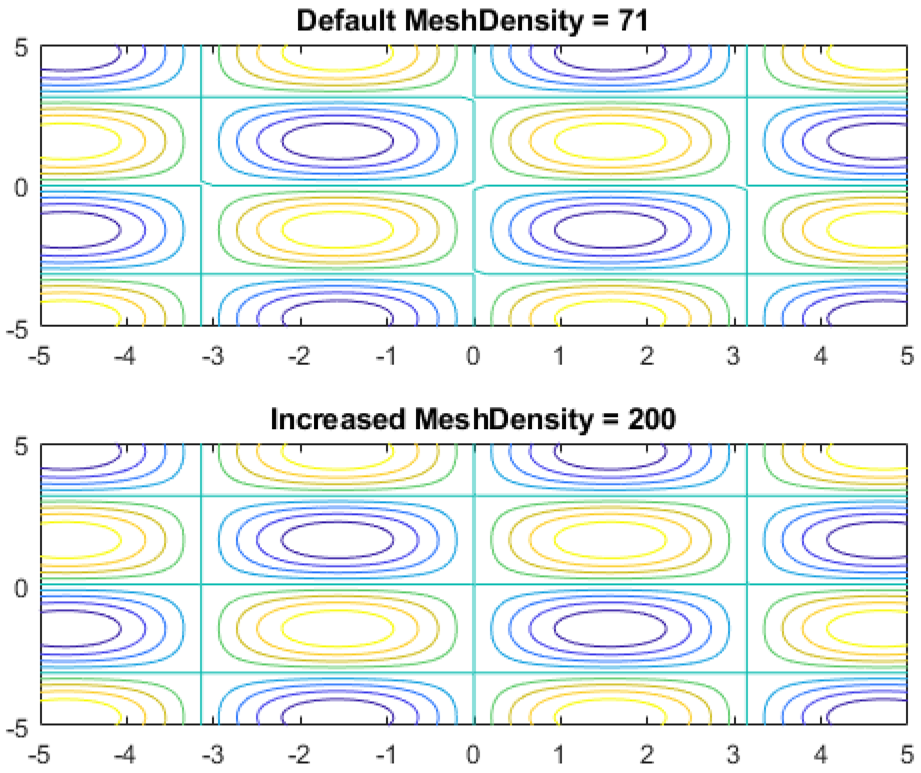
Control Resolution of Contour Lines

Control the resolution of contour lines by using the `'MeshDensity'` option. Increasing `'MeshDensity'` can make smoother, more accurate plots while decreasing it can increase plotting speed.

Divide a figure into two using `subplot`. In the first subplot, plot the contours of $\sin(x)\sin(y)$. The corners of the squares do not meet. To fix this issue, increase

'MeshDensity' to 200 in the second subplot. The corners now meet, showing that by increasing 'MeshDensity' you increase the plot's resolution.

```
syms x y
subplot(2,1,1)
fcontour(sin(x).*sin(y))
title('Default MeshDensity = 71')

subplot(2,1,2)
fcontour(sin(x).*sin(y),'MeshDensity',200)
title('Increased MeshDensity = 200')
```

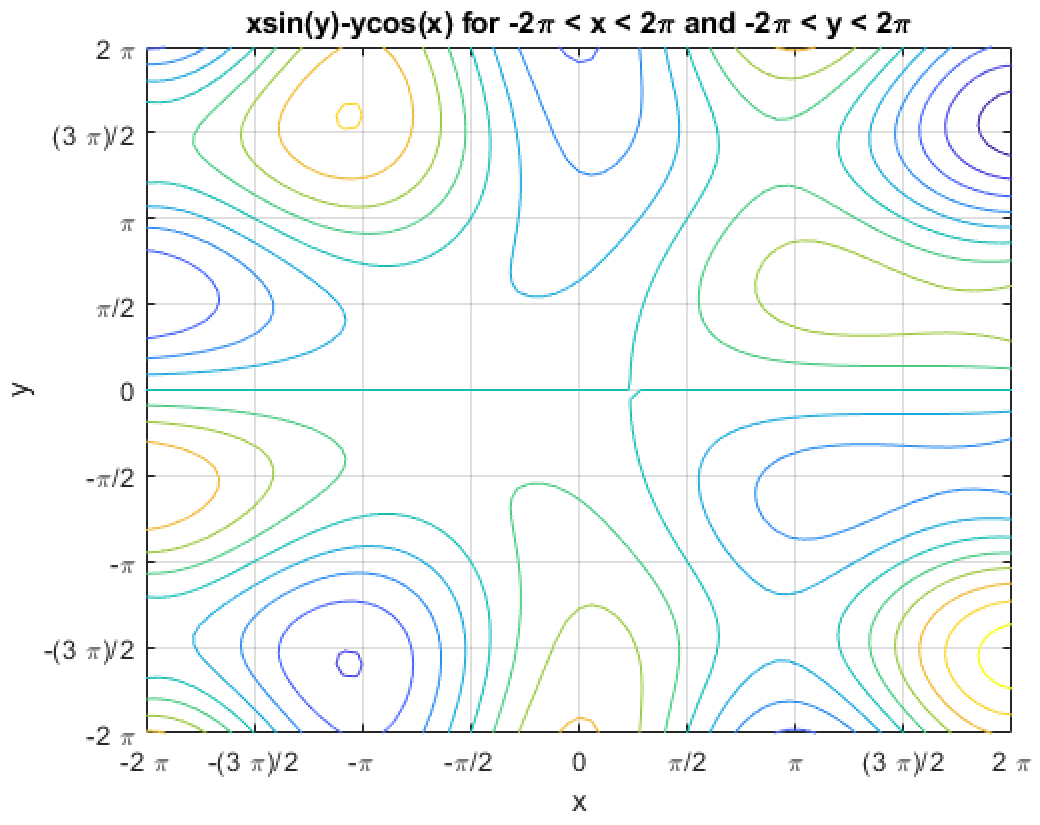


Add Title and Axis Labels and Format Ticks

Plot $x\sin(y) - y\cos(x)$. Add a title and axis labels. Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property. Repeat these steps for the y-axis.

To use LaTeX in plots, see `latex`.

```
syms x y
fcontour(x*sin(y)-y*cos(x), [-2*pi 2*pi])
grid on
title('xsin(y)-ycos(x) for -2\pi < x < 2\pi and -2\pi < y < 2\pi')
xlabel('x')
ylabel('y')
ax = gca;

S = sym(ax.XLim(1):pi/2:ax.XLim(2));
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);

S = sym(ax.YLim(1):pi/2:ax.YLim(2));
ax.YTick = double(S);
ax.YTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);
```

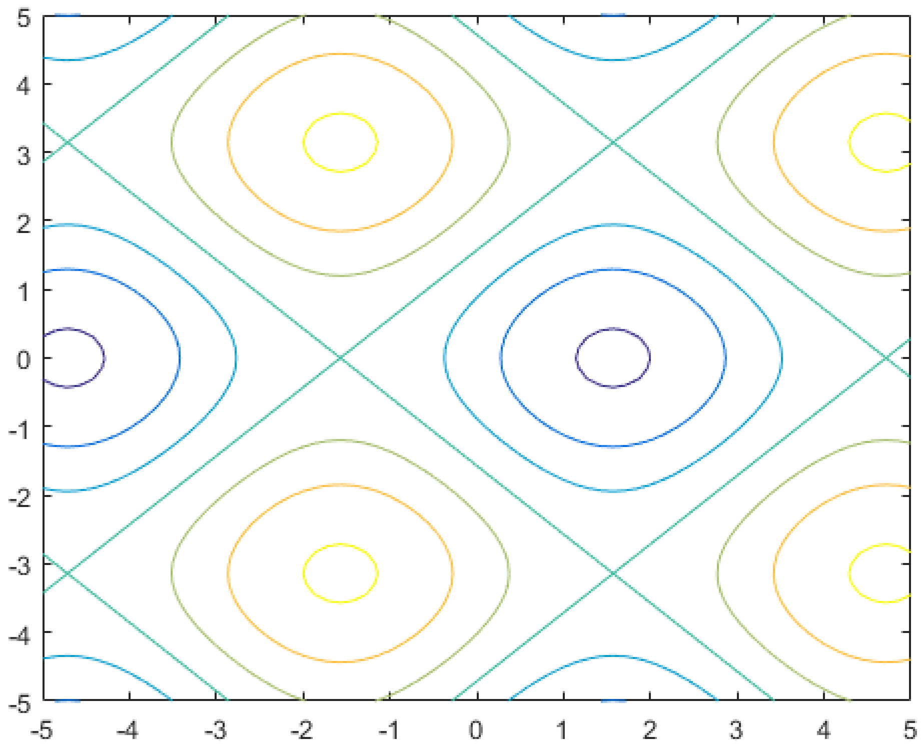


Create Animations

Create animations by changing the displayed expression using the `Function` property of the function handle, and then using `drawnow` to update the plot. To export to GIF, see `imwrite`.

By varying the variable *i* from –π/8 to π/8, animate the parametric curve *i*sin(*x*) + *i*cos(*y*).

```
syms x y
fc = fcontour(-pi/8.*sin(x)-pi/8.*cos(y));
for i=-pi/8:0.01:pi/8
    fc.Function = i.*sin(x)+i.*cos(y);
    drawnow
        pause(0.05)
end
```



Input Arguments

`f` — Expression or function to be plotted

symbolic expression | symbolic function

Expression or function to be plotted, specified as a symbolic expression or function.

`[min max]` — Plotting range for `x` and `y`

[–5 5] (default) | vector of two numbers

Plotting range for x and y, specified as a vector of two numbers. The default range is [-5 5].

`[xmin xmax ymin ymax]` — Plotting range for `x` and `y`
[–5 5 –5 5] (default) | vector of four numbers

Plotting range for x and y, specified as a vector of four numbers. The default range is [-5 5 -5 5].

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then the plot function uses the current axes.

`LineSpec` — Line style and color
character vector | string

Line style and color, specified as a character vector or string containing a line style specifier, a color specifier, or both.

Example: `'--r'` specifies red dashed lines

These two tables list the line style and color options.

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
Color Specifier	**Description**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue

Color Specifier	Description
`w`	white
`k`	black

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`.

Example: `'MeshDensity',30`

The properties listed here are only a subset. For a complete list, see Function Contour.

`MeshDensity` — Number of evaluation points per direction

71 (default) | number

Number of evaluation points per direction, specified as a number. The default is `71`. Because `fcontour` uses adaptive evaluation, the actual number of evaluation points is greater.

Example: `30`

`Fill` — Fill between contour lines

`'off'` (default) | `'on'`

Fill between contour lines, specified as one of these values:

- `'off'` — Do not fill the spaces between contour lines with a color.
- `'on'` — Fill the spaces between contour lines with color.

`LevelList` — Contour levels

vector of z values

Contour levels, specified as a vector of z values. By default, the `fcontour` function chooses values that span the range of values in the `ZData` property.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelListMode` — Selection mode for `LevelList`
`'auto'` (default) | `'manual'`

Selection mode for the `LevelList`, specified as one of these values:

- `'auto'` — Determine the values based on the `ZData` values.
- `'manual'` — Use manually specified values. To specify the values, set the `LevelList` property. When the mode is `'manual'`, the `LevelList` values do not change if you change the `Function` property or the limits.

`LevelStep` — Spacing between contour lines
`0` (default) | scalar numeric value

Spacing between contour lines, specified as a scalar numeric value. For example, specify a value of `2` to draw contour lines at increments of 2. By default, `LevelStep` is determined by using the `ZData` values.

Setting this property sets the associated mode property to `'manual'`.

Example: `3.4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelStepMode` — Selection mode for `LevelStep`
`'auto'` (default) | `'manual'`

Selection mode for the `LevelStep`, specified as one of these values:

- `'auto'` — Determine the value based on the `ZData` values.
- `'manual'` — Use a manually specified value. To specify the value, set the `LevelStep` property. When the mode is `'manual'`, the value of `LevelStepMode` does not change when the `Function` property or the limits change.

`LineColor` — Color of contour lines
`'flat'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Color of contour lines, specified as `'flat'`, an RGB triplet, or one of the color options listed in the table.

- To use a different color for each contour line, specify `'flat'`. The color is determined by the contour value of the line, the colormap, and the scaling of data values into the colormap. For more information on color scaling, see `caxis`.
- To use the same color for all the contour lines, specify an RGB triplet or one of the color options from the table.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	────
`'--'`	Dashed line	- – – – –

Line Style	Description	Resulting Line
`':'`	Dotted line	....
`'-.'`	Dash-dotted line	_._._._.
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

Output Arguments

`fc` — One or more function contour objects
scalar | vector

One or more function contour objects, returned as a scalar or a vector. These objects are unique identifiers, which you can use to query and modify the properties of a specific contour plot. For details, see Function Contour.

See Also

Functions
`fimplicit` | `fimplicit3` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Properties
Function Contour

Topics
“Create Plots” on page 2-240

Introduced in R2016a

feval

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`. Here, the returned value `result` is 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 character vector.

Input Arguments

`F`

MuPAD function name or symbolic object.

`x1,...,xn`

Arguments of `F`.

Output Arguments

`result`

Symbolic object or character vector 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 =
    'Arithmetical expression expected.'

status =
     2
```

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.
- `feval` does not open a MuPAD notebook, and therefore, you cannot use this function to access MuPAD graphics capabilities.

Alternatives

`evalin` lets you evaluate MuPAD expressions without explicitly specifying their arguments.

See Also

`evalin` | `read` | `symengine`

Topics

"Call Built-In MuPAD Functions from MATLAB" on page 3-58
"Evaluations in Symbolic Computations"
"Level of Evaluation"

Introduced in R2008b

fibonacci

Fibonacci numbers

Syntax

`fibonacci(n)`

Description

`fibonacci(n)` returns the n^th “Fibonacci Number” on page 4-581.

Examples

Find Fibonacci Numbers

Find the sixth Fibonacci number by using `fibonacci`.

```
fibonacci(6)
```

```
ans =
     8
```

Find the first 10 Fibonacci numbers.

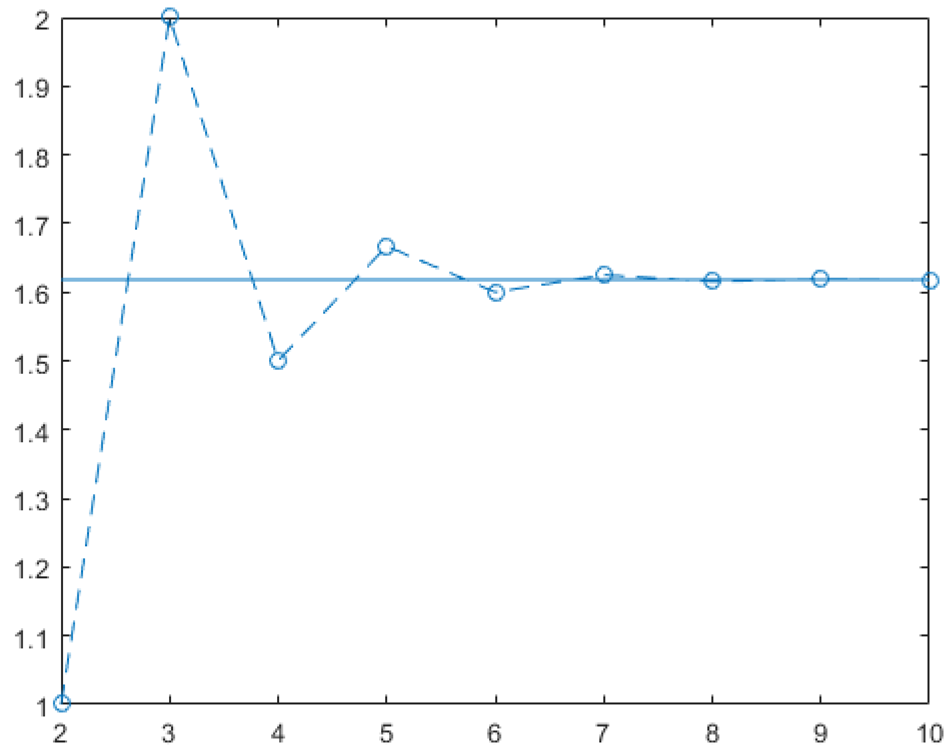
```
n = 1:10;
fibonacci(n)
```

```
ans =
     1     1     2     3     5     8    13    21    34    55
```

Fibonacci Sequence Approximates Golden Ratio

The ratio of successive Fibonacci numbers converges to the golden ratio $1.61803...$. Show this convergence by plotting this ratio against the golden ratio for the first 10 Fibonacci numbers.

```
n = 2:10;
ratio = fibonacci(n)./fibonacci(n-1);

plot(n,ratio,'--o')
hold on

line(xlim,[1.618 1.618])
hold off
```



Symbolically Represent Fibonacci Numbers

Use Fibonacci numbers in symbolic calculations by representing them with symbolic input. `fibonacci` returns the input.

Represent the nth Fibonacci number.

```
syms n
fibonacci(n)

ans =
fibonacci(n)
```

Find Large Fibonacci Numbers

Find large Fibonacci numbers by specifying the input symbolically using `sym`. Symbolic input returns exact symbolic output instead of double output. Convert symbolic numbers to double by using the `double` function.

Find the 300th Fibonacci number.

```
num = sym(300);
fib300 = fibonacci(num)

fib300 =
222232244629420445529739893461909967206666939096499764990979600
```

Convert `fib300` to double. The result is a floating-point approximation.

```
double(fib300)

ans =
   2.2223e+62
```

For more information on symbolic and double arithmetic, see “Choose Symbolic or Numeric Arithmetic” on page 2-114.

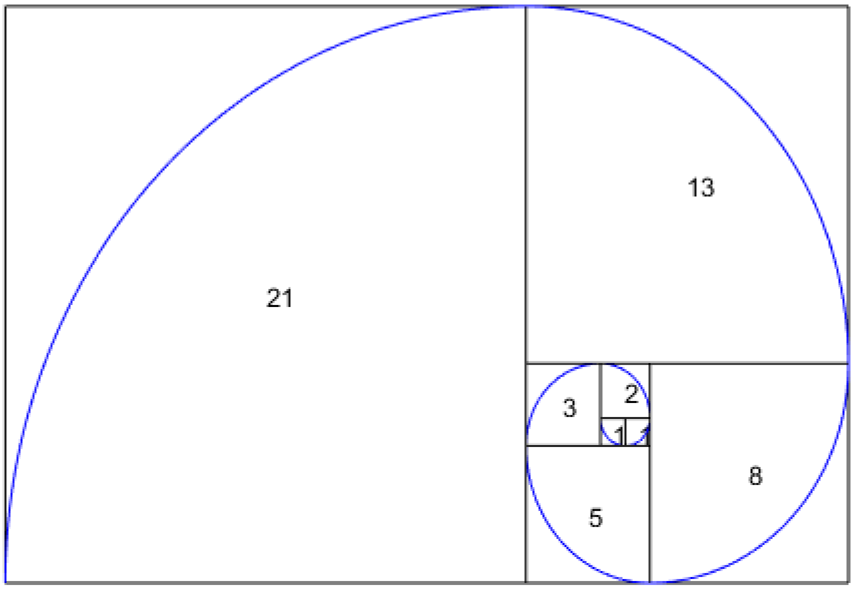
Golden Spiral Using Fibonacci Numbers

The Fibonacci numbers are commonly visualized by plotting the Fibonacci spiral. The Fibonacci spiral approximates the golden spiral.

Approximate the golden spiral for the first 8 Fibonacci numbers. Define the four cases for the right, top, left, and bottom squares in the plot by using a `switch` statement. Form the spiral by defining the equations of arcs through the squares in `eqnArc`. Draw the squares and arcs by using `rectangle` and `fimplicit` respectively.

```
x = 0;
y = 1;
syms v u

axis off
hold on

for n = 1:8
```

```
    a = fibonacci(n);

    % Define squares and arcs
    switch mod(n,4)
        case 0
            y = y - fibonacci(n-2);
            x = x - a;
            eqnArc = (u-(x+a))^2 + (v-y)^2 == a^2;
        case 1
            y = y - a;
            eqnArc = (u-(x+a))^2 + (v-(y+a))^2 == a^2;
        case 2
            x = x + fibonacci(n-1);
            eqnArc = (u-x)^2 + (v-(y+a))^2 == a^2;
        case 3
            x = x - fibonacci(n-2);
            y = y + fibonacci(n-1);
            eqnArc = (u-x)^2 + (v-y)^2 == a^2;
    end

    % Draw square
    pos = [x y a a];
    rectangle('Position', pos)

    % Add Fibonacci number
    xText = (x+x+a)/2;
    yText = (y+y+a)/2;
    text(xText, yText, num2str(a))

    % Draw arc
    interval = [x x+a y y+a];
    fimplicit(eqnArc, interval, 'b')

end
```



Input Arguments

n — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Fibonacci Number

The Fibonacci numbers are the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21….

Given that the first two numbers are 0 and 1, the n^{th} Fibonacci number is

$F_{\mathrm{n}} = F_{\mathrm{n-1}} + F_{\mathrm{n-2}}$.

Applying this formula repeatedly generates the Fibonacci numbers.

Introduced in R2017a

fimplicit

Plot implicit symbolic equation or function

Syntax

```
fimplicit(f)
fimplicit(f,[min max])
fimplicit(f,[xmin xmax ymin ymax])

fimplicit(___,LineSpec)
fimplicit(___,Name,Value)
fimplicit(ax,___)
fi = fimplicit(___)
```

Description

`fimplicit(f)` plots the implicit symbolic equation or function `f` over the default interval `[-5 5]` for `x` and `y`.

`fimplicit(f,[min max])` plots `f` over the interval `min` < `x` < `max` and `min` < `y` < `max`.

`fimplicit(f,[xmin xmax ymin ymax])` plots `f` over the interval `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`. `fimplicit` uses `symvar` to order the variables and assign intervals.

`fimplicit(___,LineSpec)` uses `LineSpec` to set the line style, marker symbol, and line color.

`fimplicit(___,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. `Name,Value` pair settings apply to all the lines plotted. To set options for individual lines, use the objects returned by `fimplicit`.

`fimplicit(ax,___)` plots into the axes specified by `ax` instead of the current axes `gca`.

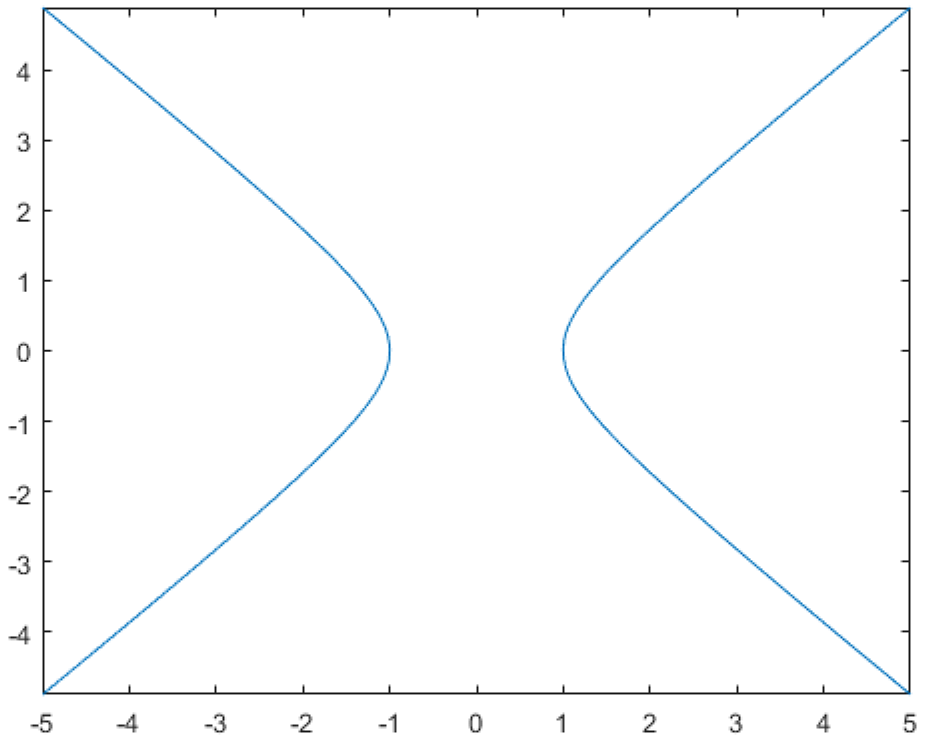
`fi = fimplicit(___)` returns an implicit function line object. Use the object to query and modify properties of a specific line. For details, see Implicit Function Line.

Examples

Plot Implicit Symbolic Equation

Plot the hyperbola $x^2 - y^2 = 1$ by using `fimplicit`. The `fimplicit` function uses the default interval of $[-5, 5]$ for x and y.

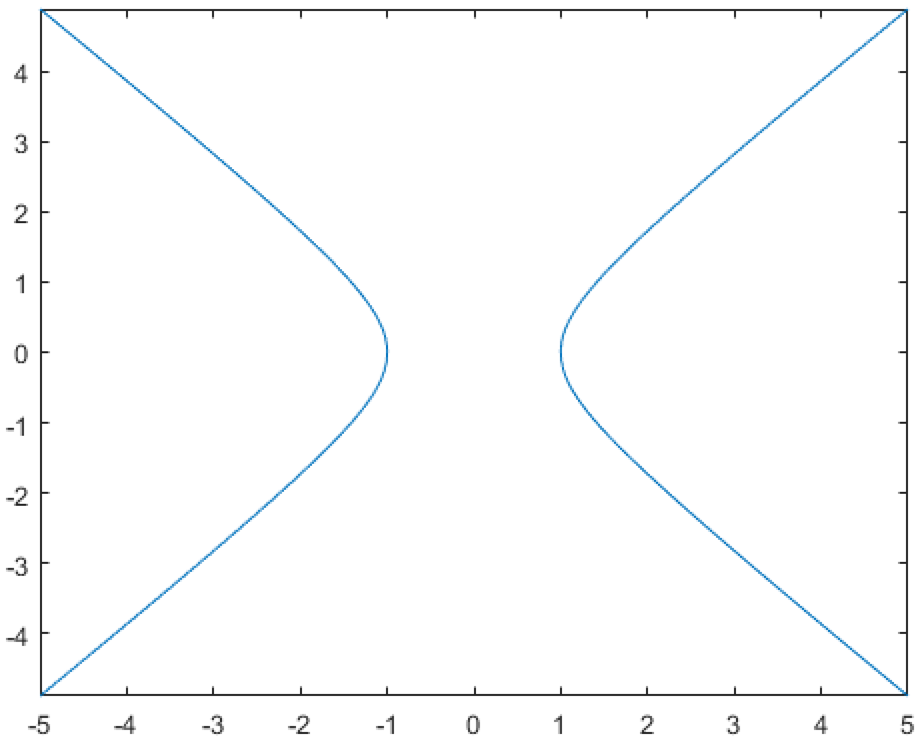
```
syms x y
fimplicit(x^2 - y^2 == 1)
```



Plot Implicit Symbolic Function

Plot the hyperbola described by the function $f(x, y) = x^2 - y^2 - 1$ by first declaring the symbolic function `f(x,y)` using `syms`. The `fimplicit` function uses the default interval of $[-5, 5]$ for x and y.

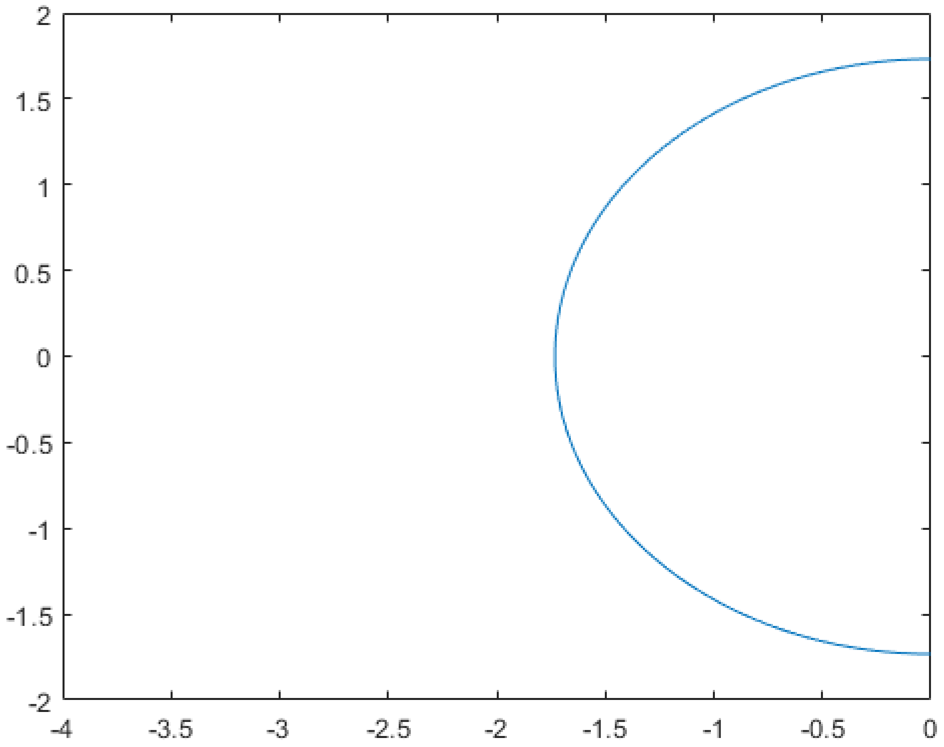
```
syms f(x,y)
f(x,y) = x^2 - y^2 - 1;
fimplicit(f)
```



Specify Plotting Interval

Plot half of the circle $x^2 + y^2 = 3$ by using the intervals $-4 < x < 0$ and $-2 < y < 2$. Specify the plotting interval as the second argument of `fimplicit`.

```
syms x y
circle = x^2 + y^2 == 3;
fimplicit(circle, [-4 0 -2 2])
```

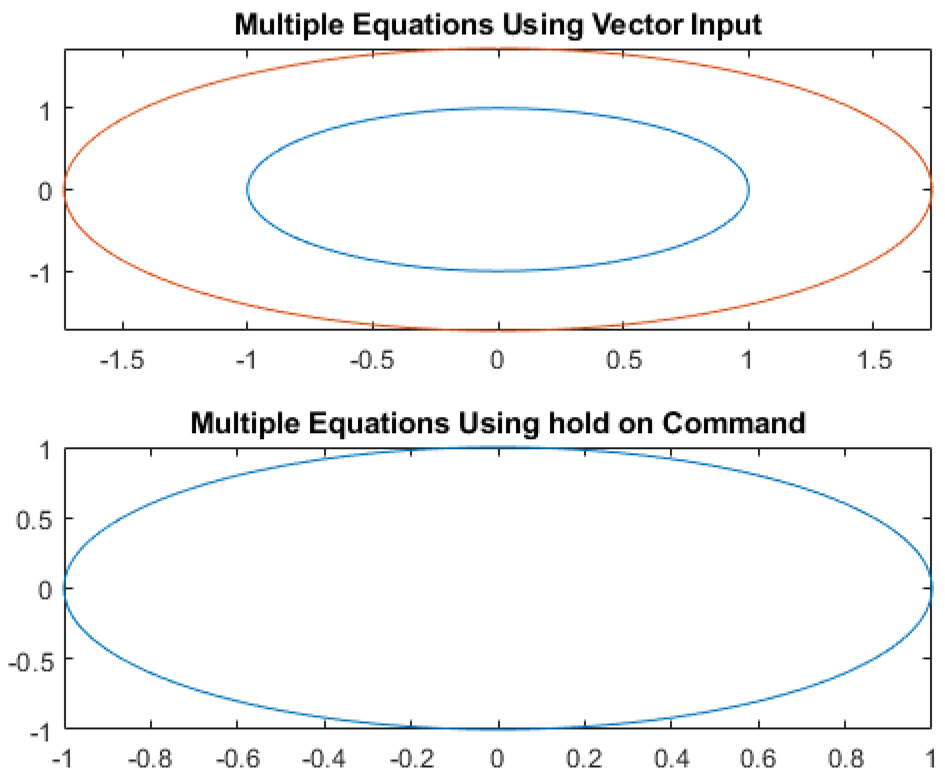


Plot Multiple Implicit Equations

You can plot multiple equations either by passing the inputs as a vector or by using `hold on` to successively plot on the same figure. If you specify `LineSpec` and Name-Value arguments, they apply to all lines. To set options for individual plots, use the function handles returned by `fimplicit`.

Divide a figure into two subplots by using `subplot`. On the first subplot, plot $x^2 + y^2 == 1$ and $x^2 + y^2 == 3$ using vector input. On the second subplot, plot the same inputs by using `hold on`.

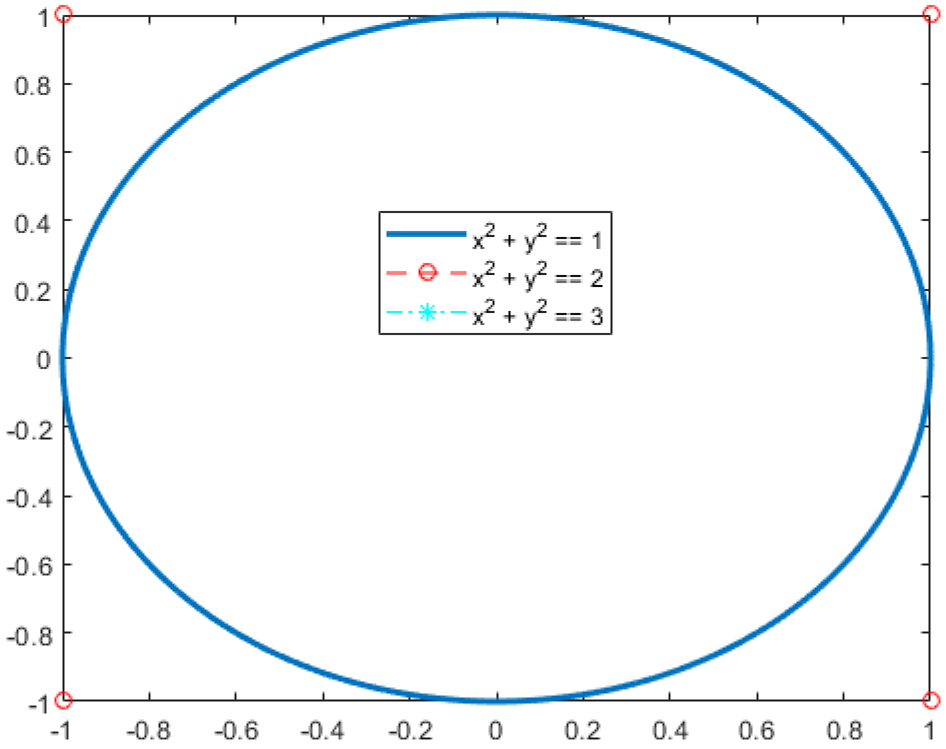
```
syms x y
circle1 = x^2 + y^2 == 1;
circle2 = x^2 + y^2 == 3;

subplot(2,1,1)
fimplicit([circle1 circle2])
title('Multiple Equations Using Vector Input')

subplot(2,1,2)
fimplicit(circle1)
hold on
fimplicit(circle2)
title('Multiple Equations Using hold on Command')

hold off
```



Change Line Properties and Display Markers

Plot three concentric circles of increasing diameter. For the first line, use a linewidth of 2. For the second, specify a dashed red line style with circle markers. For the third, specify a cyan, dash-dot line style with asterisk markers. Display the legend.

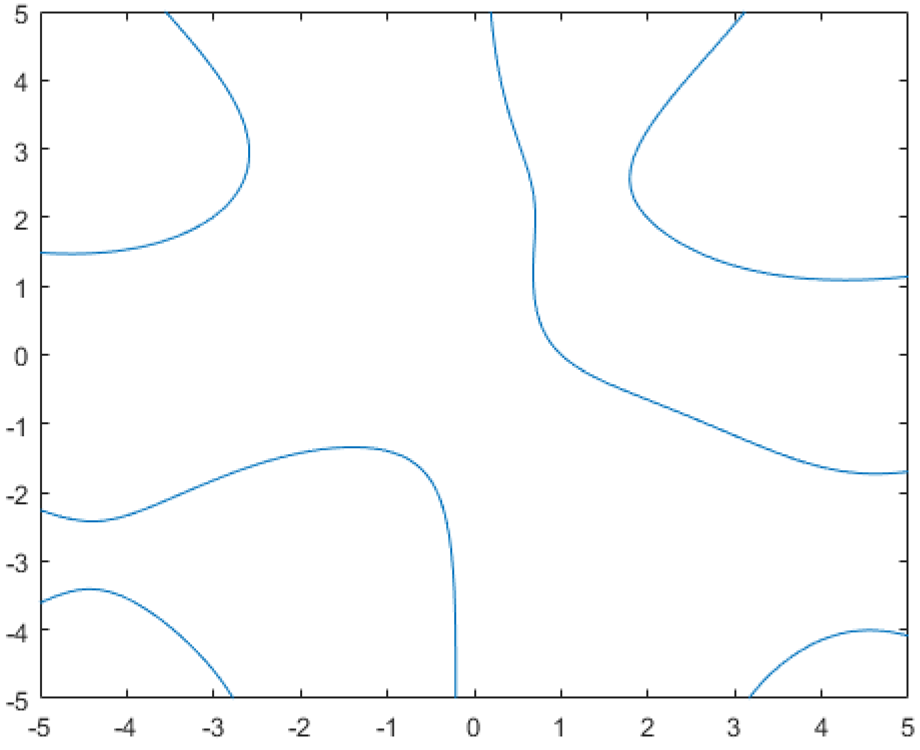
```
syms x y
circle = x^2 + y^2;
fimplicit(circle == 1, 'Linewidth', 2)
hold on
fimplicit(circle == 2, '--or')
fimplicit(circle == 3, '-.*c')
legend('show','Location','best')
hold off
```



Modify Implicit Plot After Creation

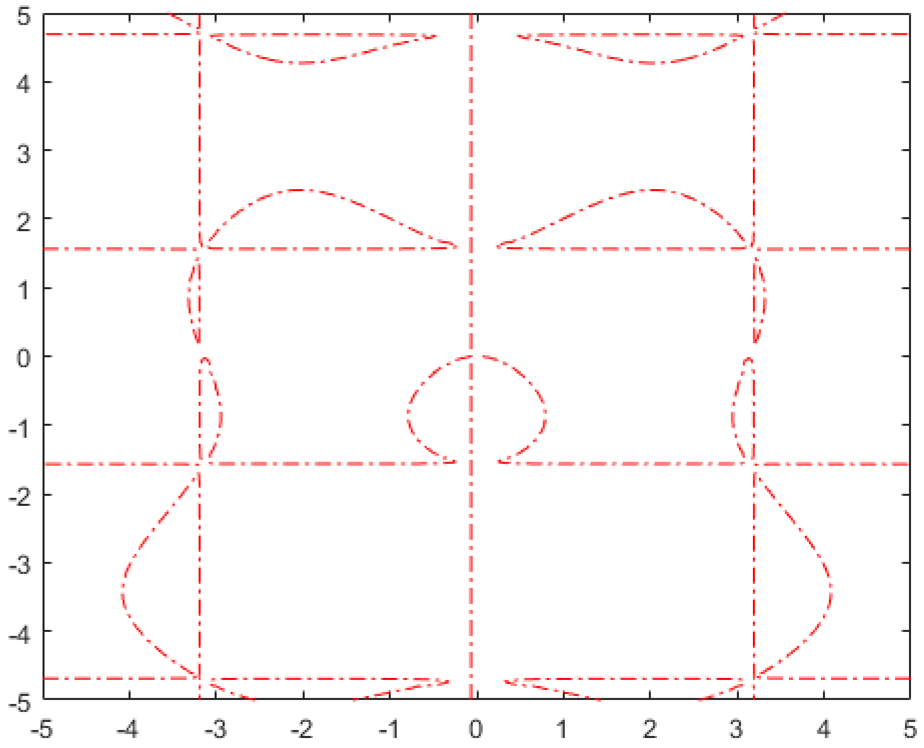
Plot $y\sin(x) + x\cos(y) = 1$. Specify an output to make `fimplicit` return the plot object.

```
syms x y
eqn = y*sin(x) + x*cos(y) == 1;
fi = fimplicit(eqn)

fi = 
  ImplicitFunctionLine with properties:

     Function: [1x1 sym]
        Color: [0 0.4470 0.7410]
    LineStyle: '-'
    LineWidth: 0.5000

  Show all properties
```



Change the plotted equation to $x\cos(y) + y\sin(x) = 0$ by using dot notation to set properties. Similarly, change the line color to red and line style to a dash-dot line. The horizontal and vertical lines in the output are artifacts that should be ignored.

```
fi.Function = x/cos(y) + y/sin(x) == 0;
fi.Color = 'r';
fi.LineStyle = '-.';
```

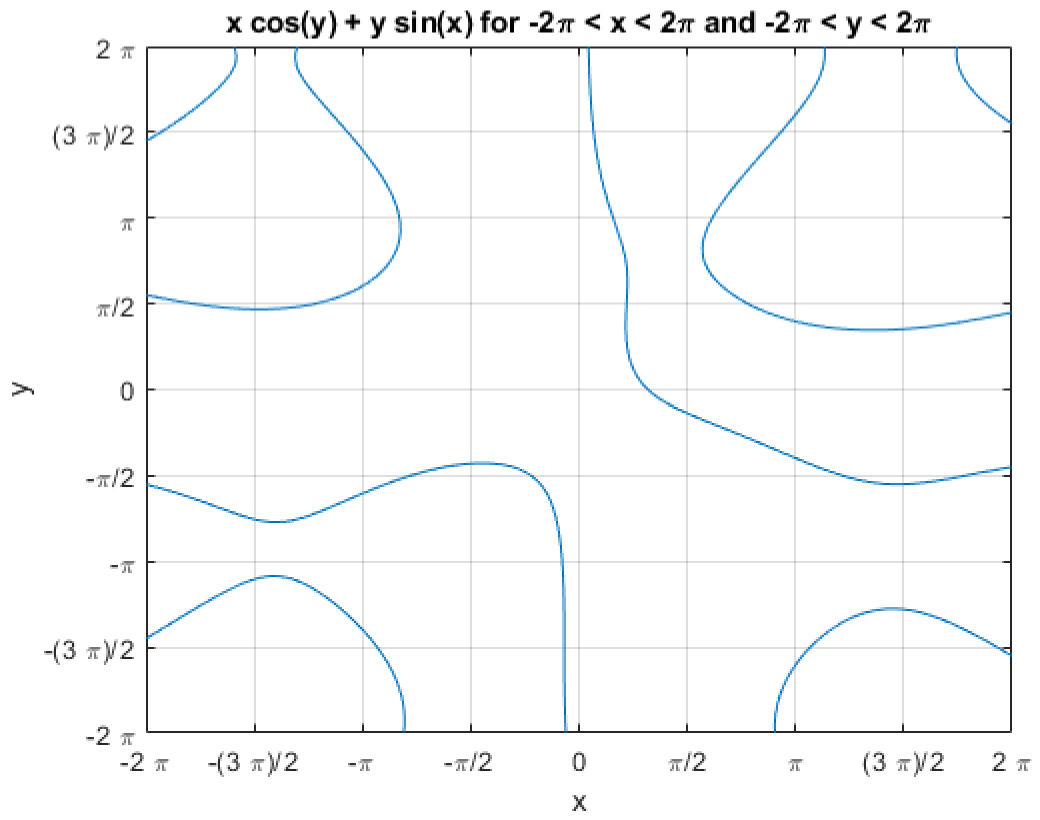


Add Title and Axis Labels and Format Ticks

Plot $x\cos(y) + y\sin(x) = 1$ over the interval $-2\pi < x < 2\pi$ and $-2\pi < y < 2\pi$. Add a title and axis labels. Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property. Repeat these steps for the y-axis.

To use LaTeX in plots, see `latex`.

```
syms x y
eqn = x*cos(y) + y*sin(x) == 1;
fimplicit(eqn, [-2*pi 2*pi])
grid on
title('x cos(y) + y sin(x) for -2\pi < x < 2\pi and -2\pi < y < 2\pi')
xlabel('x')
ylabel('y')
ax = gca;

S = sym(ax.XLim(1):pi/2:ax.XLim(2));
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);

S = sym(ax.YLim(1):pi/2:ax.YLim(2));
ax.YTick = double(S);
ax.YTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);
```



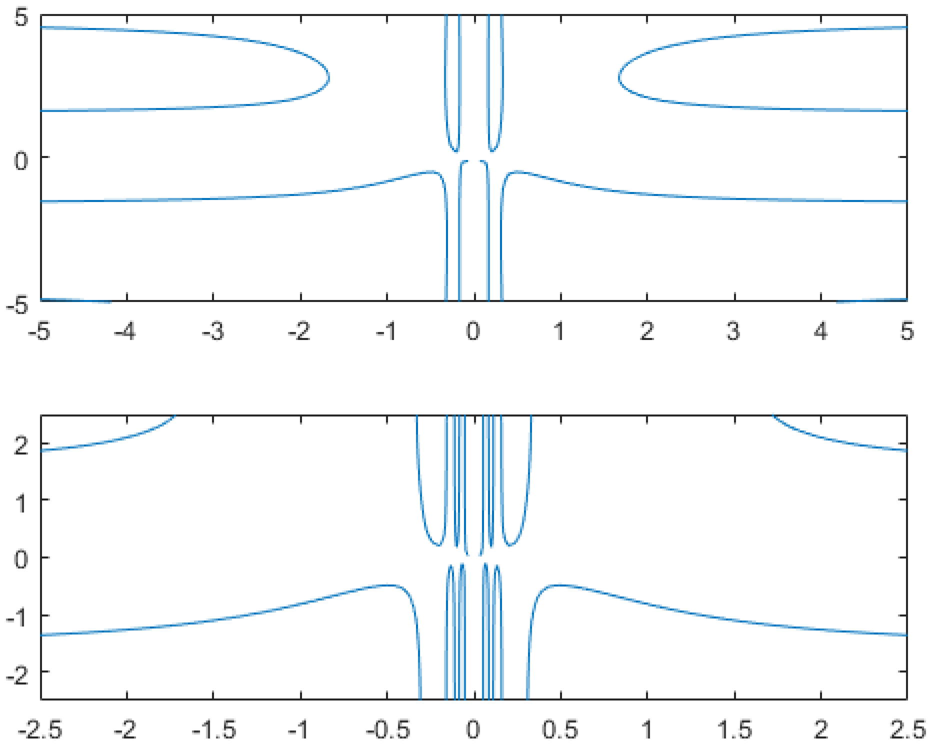
Re-evaluation on Zoom

When you zoom into a plot, `fimplicit` re-evaluates the plot automatically. This re-evaluation on zoom can reveal hidden detail at smaller scales.

Divide a figure into two by using `subplot`. Plot $x\cos(y) + y\sin(1/x) = 0$ in both the first and second subplots. Zoom into the second subplot by using `zoom`. The zoomed subplot shows detail that is not visible in the first subplot.

```
syms x y
eqn = x*cos(y) + y*sin(1/x) == 0;
```

```
subplot(2,1,1)
fimplicit(eqn)

subplot(2,1,2)
fimplicit(eqn)
zoom(2)
```



Input Arguments

`f` — Implicit equation or function to plot

symbolic equation | symbolic expression | symbolic function

Implicit equation or function to plot, specified as a symbolic equation, expression, or function. If the right-hand side is not specified, then it is assumed to be `0`.

`[min max]` — Plotting range for `x` and `y`

[–5 5] (default) | vector of two numbers

Plotting range for `x` and `y`, specified as a vector of two numbers. The default range is `[-5 5]`.

`[xmin xmax ymin ymax]` — Plotting range for `x` and `y`

[–5 5 –5 5] (default) | vector of four numbers

Plotting range for `x` and `y`, specified as a vector of four numbers. The default range is `[-5 5 -5 5]`.

`ax` — Axes object

axes object

Axes object. If you do not specify an axes object, then `fimplicit` uses the current axes `gca`.

`LineSpec` — Line specification

character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
Marker Specifier	**Description**
`o`	Circle
`+`	Plus sign
`*`	Asterisk

Marker Specifier	Description
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram
Color Specifier	**Description**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

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`.

The function line properties listed here are only a subset. For a complete list, see Implicit Function Line.

Example: `'Marker','o','MarkerFaceColor','red'`

`MeshDensity` — **Number of evaluation points per direction**

151 (default) | number

Number of evaluation points per direction, specified as a number. The default is `151`.

`Color` — **Line color**

`[0 0.4470 0.7410]` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet or one of the color options listed in the table.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — **Line style**

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	———

Line Style	Description	Resulting Line
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	....
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)

Value	Description
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color
`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`fi` — One or more implicit function line objects
scalar | vector

One or more implicit function line objects, returned as a scalar or a vector. You can use these objects to query and modify properties of a specific line. For a list of properties, see Implicit Function Line.

See Also

Functions
`fcontour` | `fimplicit3` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Properties
Implicit Function Line

Topics
"Create Plots" on page 2-240

Introduced in R2016b

fimplicit3

Plot 3-D implicit equation or function

Syntax

```
fimplicit3(f)
fimplicit3(f,[min max])
fimplicit3(f,[xmin xmax ymin ymax zmin zmax])

fimplicit3(___,LineSpec)
fimplicit3(___,Name,Value)
fimplicit3(ax,___)
fi = fimplicit3(___)
```

Description

`fimplicit3(f)` plots the 3-D implicit equation or function `f(x,y,z)` over the default interval `[-5 5]` for `x`, `y`, and `z`.

`fimplicit3(f,[min max])` plots `f(x,y,z)` over the interval `[min max]` for `x`, `y`, and `z`.

`fimplicit3(f,[xmin xmax ymin ymax zmin zmax])` plots `f(x,y,z)` over the interval `[xmin xmax]` for `x`, `[ymin ymax]` for `y`, and `[zmin zmax]` for `z`. `fimplicit3` uses `symvar` to order the variables and assign intervals.

`fimplicit3(___,LineSpec)` uses `LineSpec` to set the line style, marker symbol, and face color.

`fimplicit3(___,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

`fimplicit3(ax,___)` plots into the axes with the object `ax` instead of the current axes object `gca`.

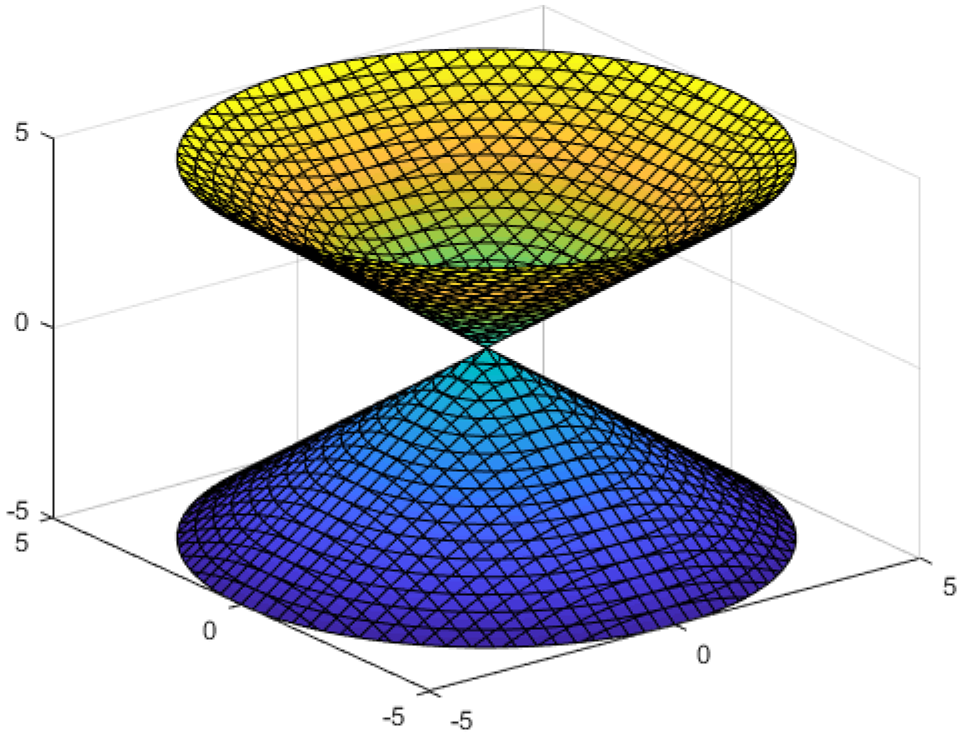
`fi = fimplicit3(___)` returns an implicit function surface object. Use the object to query and modify properties of a specific surface. For details, see Implicit Function Surface.

Examples

Plot 3-D Implicit Symbolic Equation

Plot the hyperboloid $x^2 + y^2 - z^2 = 0$ by using `fimplicit3`. The `fimplicit3` function plots over the default interval of $[-5, 5]$ for x, y, and z.

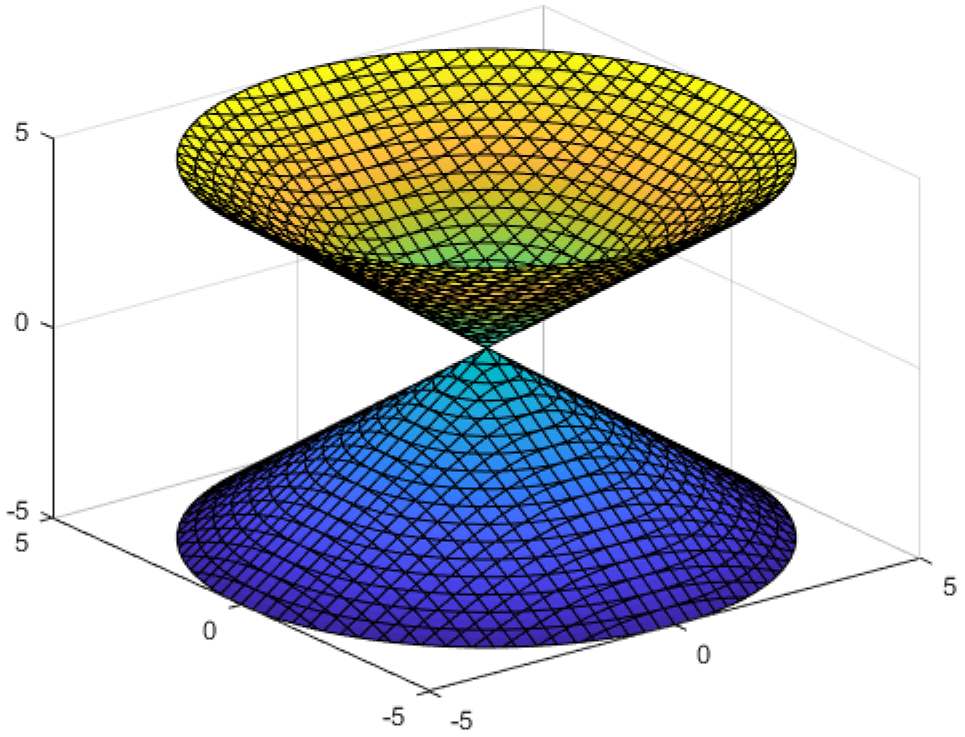
```
syms x y z
fimplicit3(x^2 + y^2 - z^2)
```



Plot 3-D Implicit Symbolic Function

Plot the hyperboloid specified by the function $f(x, y, z) = x^2 + y^2 - z^2$. The `fimplicit3` function plots over the default interval of $[-5, 5]$ for x, y, and z.

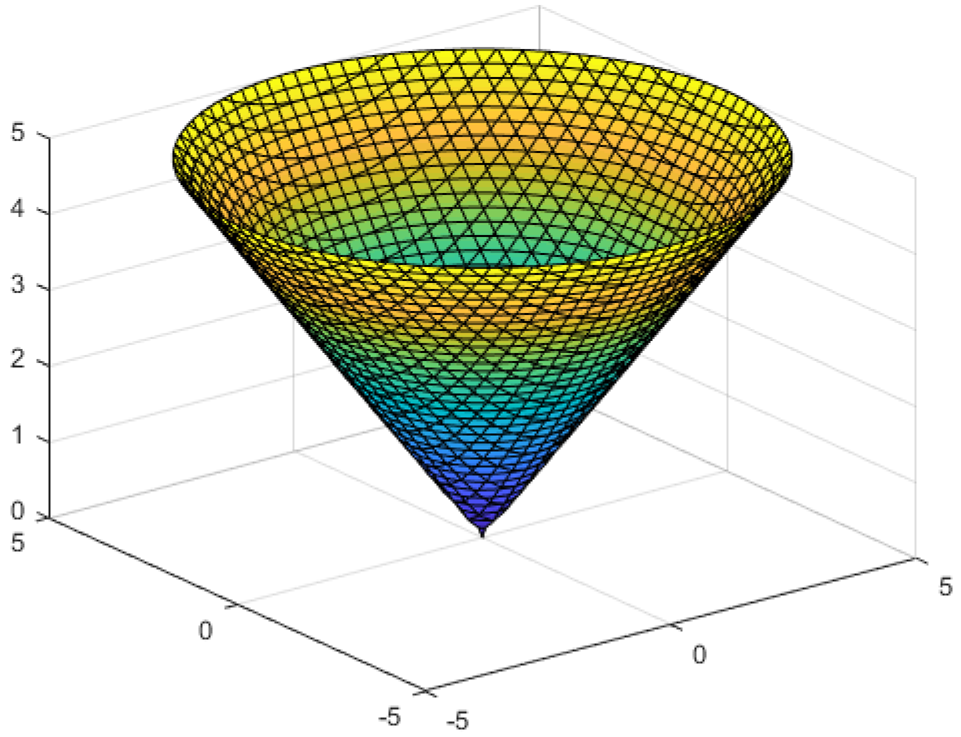
```
syms f(x,y,z)
f(x,y,z) = x^2 + y^2 - z^2;
fimplicit3(f)
```



Specify Plotting Interval

Specify the plotting interval by specifying the second argument to `fimplicit3`. Plot the upper half of the hyperboloid $x^2 + y^2 - z^2 = 0$ by specifying the interval $0 < z < 5$. For x and y, use the default interval $[-5, 5]$.

```
syms x y z
f = x^2 + y^2 - z^2;
interval = [-5 5 -5 5 0 5];
fimplicit3(f, interval)
```



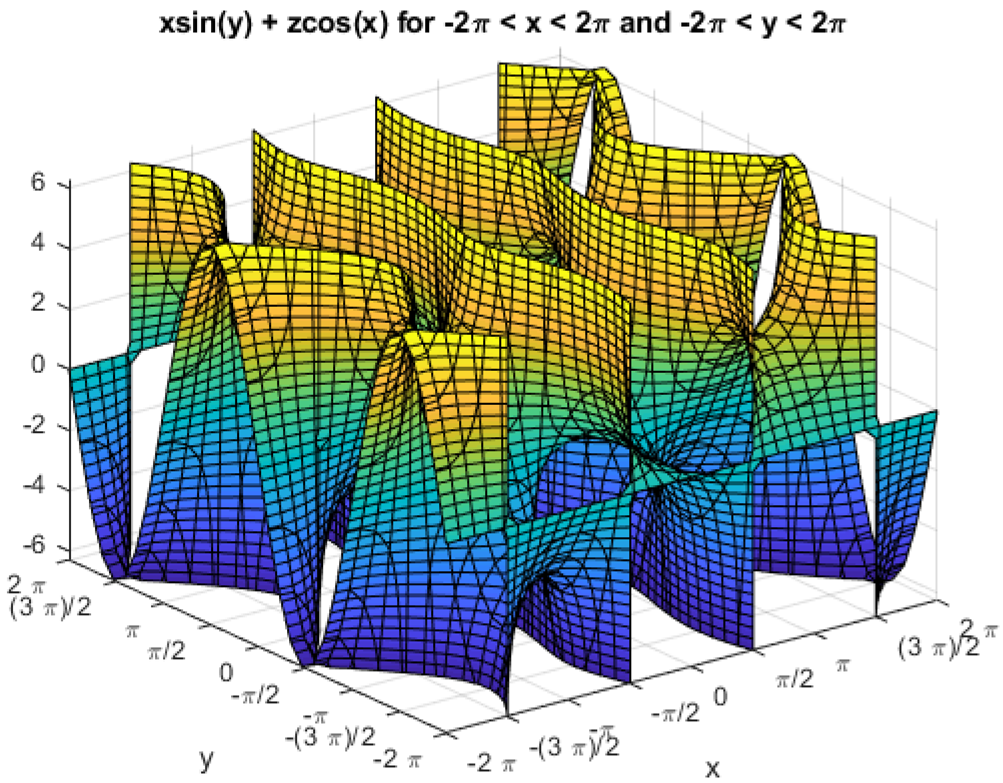
Add Title and Axis Labels and Format Ticks

Plot the implicit equation $x\sin(y)+z\cos(x)=0$ over the interval $(-2\pi,2\pi)$ for all axes.

Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Convert the axis limits to precise multiples of `pi/2` by using `round` and get the symbolic tick values in `S`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property. Repeat these steps for the y-axis.

To use LaTeX in plots, see `latex`.

```
syms x y z
eqn = x*sin(y) + z*cos(x);
fimplicit3(eqn,[-2*pi 2*pi])
title('xsin(y) + zcos(x) for -2\pi < x < 2\pi and -2\pi < y < 2\pi')
xlabel('x')
ylabel('y')
ax = gca;

S = sym(ax.XLim(1):pi/2:ax.XLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel,S,'UniformOutput',false);

S = sym(ax.YLim(1):pi/2:ax.YLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
ax.YTick = double(S);
ax.YTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);
```



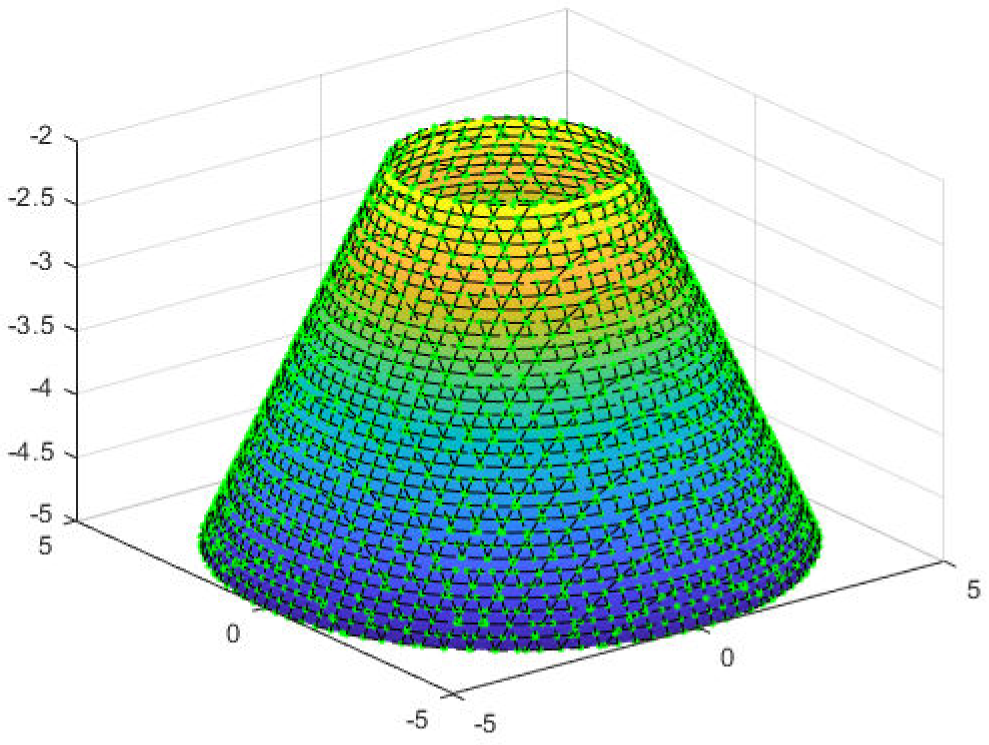
Line Style and Width for Implicit Surface Plot

Plot the implicit surface $x^2 + y^2 - z^2 = 0$ with different line styles for different values of z. For $-5 < z < -2$, use a dashed line with green dot markers. For $-2 < z < 2$, use a `LineWidth` of `1` and a green face color. For $2 < z < 5$, turn off the lines by setting `EdgeColor` to `none`.

```
syms x y z
f = x^2 + y^2 - z^2;
```

```
fimplicit3(f,[-5 5 -5 5 -5 -2],'--.','MarkerEdgeColor','g')
hold on
fimplicit3(f,[-5 5 -5 5 -2 2],'LineWidth',1,'FaceColor','g')
fimplicit3(f,[-5 5 -5 5 2 5],'EdgeColor','none')
```

```
Warning: Error updating ImplicitFunctionSurface.

 The grid must be created from grid vectors which contain unique points.
```

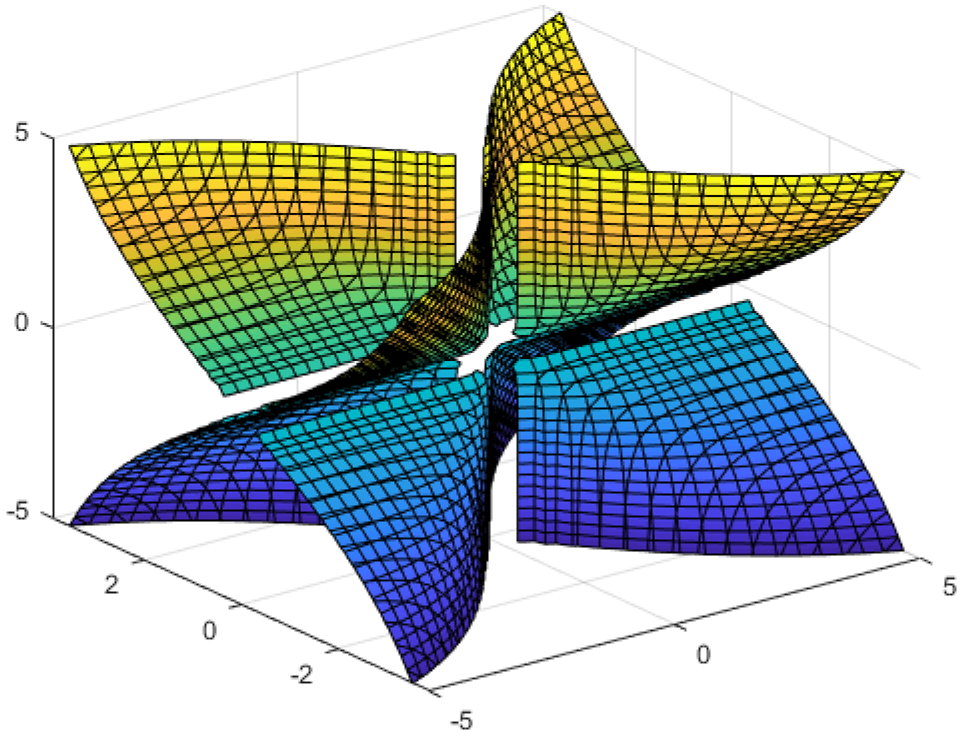


Modify Implicit Surface After Creation

Plot the implicit surface $1/x^2 - 1/y^2 + 1/z^2 = 0$. Specify an output to make `fimplicit3` return the plot object.

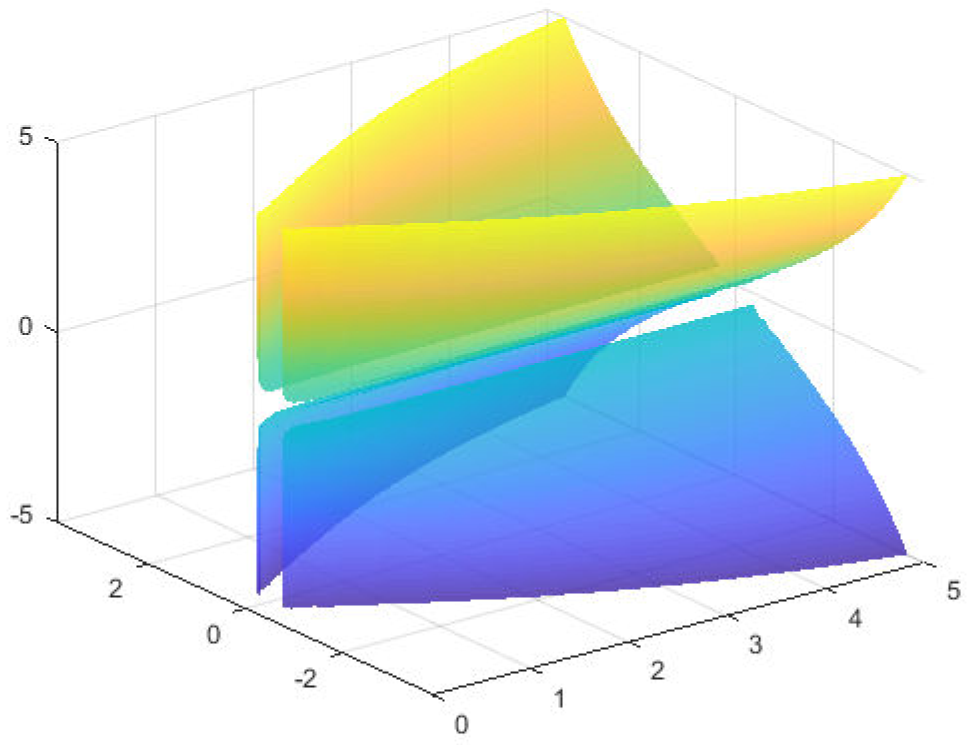
```
syms x y z
f = 1/x^2 - 1/y^2 + 1/z^2;
fi = fimplicit3(f)
```

```
fi = 
  ImplicitFunctionSurface with properties:

     Function: [1x1 sym]
    EdgeColor: [0 0 0]
    LineStyle: '-'
    FaceColor: 'interp'

  Show all properties
```



Show only the positive x-axis by setting the `XRange` property of `fi` to `[0 5]`. Remove the lines by setting the `EdgeColor` property to `'none'`. Visualize the hidden surfaces by making the plot transparent by setting the `FaceAlpha` property to `0.8`.

```
fi.XRange = [0 5];
fi.EdgeColor = 'none';
fi.FaceAlpha = 0.8;
```

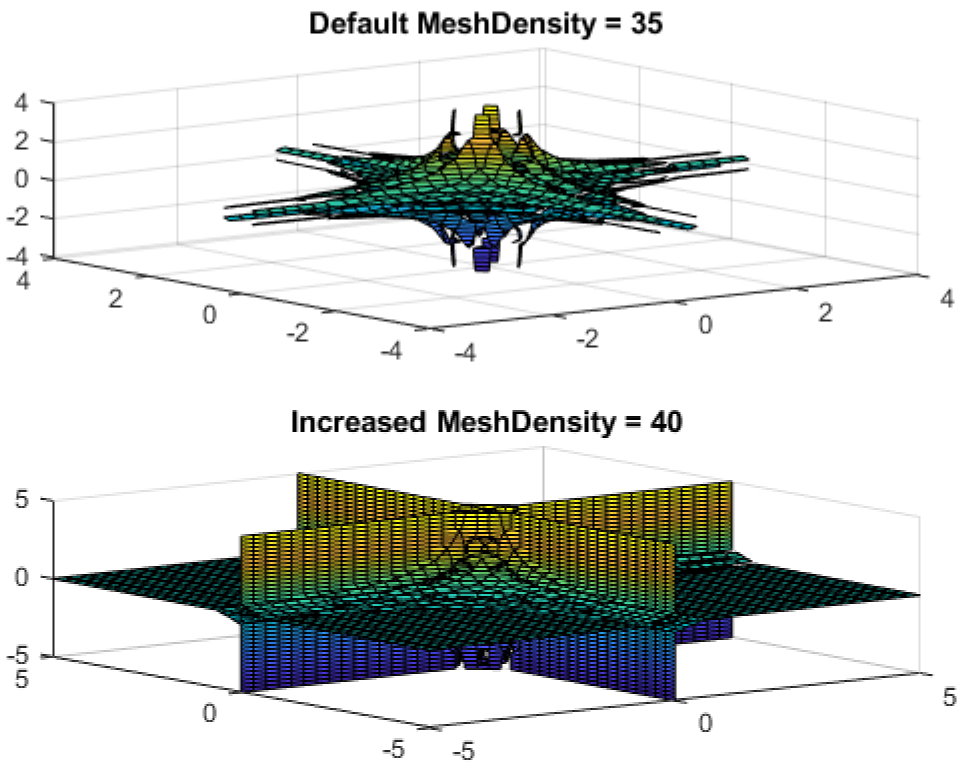


Control Resolution of Implicit Surface Plot

Control the resolution of an implicit surface plot by using the `'MeshDensity'` option. Increasing `'MeshDensity'` can make smoother, more accurate plots while decreasing `'MeshDensity'` can increase plotting speed.

Divide a figure into two by using `subplot`. In the first subplot, plot the implicit surface $\sin(1/(xyz))$. The surface has large gaps. Fix this issue by increasing the `'MeshDensity'` to `40` in the second subplot. `fimplicit3` fills the gaps showing that by increasing `'MeshDensity'` you increased the resolution of the plot.

```
syms x y z
f = sin(1/(x*y*z));

subplot(2,1,1)
fimplicit3(f)
title('Default MeshDensity = 35')

subplot(2,1,2)
fimplicit3(f,'MeshDensity',40)
title('Increased MeshDensity = 40')
```



Input Arguments

`f` — 3-D implicit equation or function to plot

symbolic equation | symbolic expression | symbolic function

3-D implicit equation or function to plot, specified as a symbolic equation, expression, or function. If an expression or function is specified, then `fimplicit3` assumes the right-hand size to be `0`.

`[min max]` — Plotting interval for x-, y- and z- axes

[–5 5] (default) | vector of two numbers

Plotting interval for x-, y- and z- axes, specified as a vector of two numbers. The default is `[-5 5]`.

`[xmin xmax ymin ymax zmin zmax]` — Plotting interval for x-, y- and z- axes

[–5 5 –5 5 –5 5] (default) | vector of six numbers

Plotting interval for x-, y- and z- axes, specified as a vector of six numbers. The default is `[-5 5 -5 5 -5 5]`.

`ax` — Axes object

axes object

Axes object. If you do not specify an axes object, then `fimplicit3` uses the current axes.

`LineSpec` — Line style, marker symbol, and face color

character vector

Line style, marker symbol, and face color, specified as a character vector. The elements of the character vector can appear in any order, and you can omit one or more options from the character vector specifier.

Example: `'--or'` is a red surface with circle markers and dashed lines

Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Specifier	Marker
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram
Specifier	**Color**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

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`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties listed here are only a subset. For a complete list, see Implicit Function Surface.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`.

Example: `100`

`EdgeColor` — Line color
`[0 0 0]` (default) | `'interp'` | `'none'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, or one of the color options listed in the table. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	………………
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`fi` — One or more objects
scalar | vector

One or more objects, returned as a scalar or a vector. The object is an implicit function surface object. You can use these objects to query and modify properties of a specific line. For details, see Implicit Function Surface.

See Also

Functions

`fcontour` | `fimplicit` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Properties

Implicit Function Surface

Topics

“Create Plots” on page 2-240

Introduced in R2016b

findDecoupledBlocks

Search for decoupled blocks in systems of equations

Syntax

```
[eqsBlocks,varsBlocks] = findDecoupledBlocks(eqs,vars)
```

Description

`[eqsBlocks,varsBlocks] = findDecoupledBlocks(eqs,vars)` identifies subsets (blocks) of equations that can be used to define subsets of variables. The number of variables `vars` must coincide with the number of equations `eqs`.

The *i*th block is the set of equations determining the variables in `vars(varsBlocks{i})`. The variables in `vars([varsBlocks{1}, …,varsBlocks{i-1}])` are determined recursively by the previous blocks of equations. After you solve the first block of equations for the first block of variables, the second block of equations, given by `eqs(eqsBlocks{2})`, defines a decoupled subset of equations containing only the subset of variables given by the second block of variables, `vars(varsBlock{2})`, plus the variables from the first block (these variables are known at this time). Thus, if a nontrivial block decomposition is possible, you can split the solution process for a large system of equations involving many variables into several steps, where each step involves a smaller subsystem.

The number of blocks `length(eqsBlocks)` coincides with `length(varsBlocks)`. If `length(eqsBlocks) = length(varsBlocks) = 1`, then a nontrivial block decomposition of the equations is not possible.

Examples

Block Lower Triangular Decomposition of DAE System

Compute a block lower triangular decomposition (BLT decomposition) of a symbolic system of differential algebraic equations (DAEs).

Create the following system of four differential algebraic equations. Here, the symbolic function calls `x1(t)`, `x2(t)`, `x3(t)`, and `x4(t)` represent the state variables of the system. The system also contains symbolic parameters `c1`, `c2`, `c3`, `c4`, and functions `f(t,x,y)` and `g(t,x,y)`.

```
syms x1(t) x2(t) x3(t) x4(t)
syms c1 c2 c3 c4
syms f(t,x,y) g(t,x,y)

eqs = [c1*diff(x1(t),t)+c2*diff(x3(t),t)==c3*f(t,x1(t),x3(t));...
       c2*diff(x1(t),t)+c1*diff(x3(t),t)==c4*g(t,x3(t),x4(t));...
       x1(t)==g(t,x1(t),x3(t));...
       x2(t)==f(t,x3(t),x4(t))];

vars = [x1(t), x2(t), x3(t), x4(t)];
```

Use `findDecoupledBlocks` to find the block structure of the system.

```
[eqsBlocks, varsBlocks] = findDecoupledBlocks(eqs, vars)

eqsBlocks =
  1×3 cell array
    {1×2 double}    {[2]}    {[4]}
varsBlocks =
  1×3 cell array
    {1×2 double}    {[4]}    {[2]}
```

The first block contains two equations in two variables.

```
eqs(eqsBlocks{1})

ans =
 c1*diff(x1(t), t) + c2*diff(x3(t), t) == c3*f(t, x1(t), x3(t))
                                   x1(t) == g(t, x1(t), x3(t))

vars(varsBlocks{1})
```

```
ans =
[ x1(t), x3(t)]
```

After you solve this block for the variables `x1(t)`, `x3(t)`, you can solve the next block of equations. This block consists of one equation.

```
eqs(eqsBlocks{2})
```

```
ans =
c2*diff(x1(t), t) + c1*diff(x3(t), t) == c4*g(t, x3(t), x4(t))
```

The block involves one variable.

```
vars(varsBlocks{2})
```

```
ans =
x4(t)
```

After you solve the equation from block 2 for the variable `x4(t)`, the remaining block of equations, `eqs(eqsBlocks{3})`, defines the remaining variable, `vars(varsBlocks{3})`.

```
eqs(eqsBlocks{3})
vars(varsBlocks{3})
```

```
ans =
x2(t) == f(t, x3(t), x4(t))

ans =
x2(t)
```

Find the permutations that convert the system to a block lower triangular form.

```
eqsPerm = [eqsBlocks{:}]
varsPerm = [varsBlocks{:}]
```

```
eqsPerm =
     1     3     2     4

varsPerm =
     1     3     4     2
```

Convert the system to a block lower triangular system of equations.

```
eqs = eqs(eqsPerm)
vars = vars(varsPerm)
```

```
eqs =
 c1*diff(x1(t), t) + c2*diff(x3(t), t) == c3*f(t, x1(t), x3(t))
                                    x1(t) == g(t, x1(t), x3(t))
 c2*diff(x1(t), t) + c1*diff(x3(t), t) == c4*g(t, x3(t), x4(t))
                                    x2(t) == f(t, x3(t), x4(t))

vars =
[ x1(t), x3(t), x4(t), x2(t)]
```

Find the incidence matrix of the resulting system. The incidence matrix shows that the system of permuted equations has three diagonal blocks of size `2`-by-`2`, `1`-by-`1`, and `1`-by-`1`.

```
incidenceMatrix(eqs, vars)
```

```
ans =
     1     1     0     0
     1     1     0     0
     1     1     1     0
     0     1     1     1
```

BLT Decomposition and Solution of Linear System

Find blocks of equations in a linear algebraic system, and then solve the system by sequentially solving each block of equations starting from the first one.

Create the following system of linear algebraic equations.

```
syms x1 x2 x3 x4 x5 x6 c1 c2 c3

eqs = [c1*x1 + x3 + x5 == c1 + c2 + 1;...
       x1 + x3 + x4 + 2*x6 == 4 + c2;...
       x1 + 2*x3 + c3*x5 == 1 + 2*c2 + c3;...
       x2 + x3 + x4 + x5 == 2 + c2;...
       x1 - c2*x3 + x5 == 2 - c2^2;...
       x1 - x3 + x4 - x6 == 1 - c2];

vars = [x1, x2, x3, x4, x5, x6];
```

Use `findDecoupledBlocks` to convert the system to a lower triangular form. For this system, `findDecoupledBlocks` identifies three blocks of equations and corresponding variables.

```
[eqsBlocks, varsBlocks] = findDecoupledBlocks(eqs, vars)
```

```
eqsBlocks =
  1×3 cell array
    {1×3 double}    {1×2 double}    {[4]}
varsBlocks =
  1×3 cell array
    {1×3 double}    {1×2 double}    {[2]}
```

Identify the variables in the first block. This block consists of three equations in three variables.

```
vars(varsBlocks{1})
```

```
ans =
[ x1, x3, x5]
```

Solve the first block of equations for the first block of variables assigning the solutions to the corresponding variables.

```
[x1,x3,x5] = solve(eqs(eqsBlocks{1}), vars(varsBlocks{1}))
```

```
x1 =
1

x3 =
c2

x5 =
1
```

Identify the variables in the second block. This block consists of two equations in two variables.

```
vars(varsBlocks{2})
```

```
ans =
[ x4, x6]
```

Solve this block of equations assigning the solutions to the corresponding variables.

```
[x4, x6] = solve(eqs(eqsBlocks{2}), vars(varsBlocks{2}))
```

```
x4 =
x3/3 - x1 - c2/3 + 2

x6 =
(2*c2)/3 - (2*x3)/3 + 1
```

Use `subs` to evaluate the result for the already known values of variables `x1`, `x3`, and `x5`.

```
x4 = subs(x4)
x6 = subs(x6)
```

```
x4 =
1

x6 =
1
```

Identify the variables in the third block. This block consists of one equation in one variable.

```
vars(varsBlocks{3})
```

```
ans =
x2
```

Solve this equation assigning the solution to `x2`.

```
x2 = solve(eqs(eqsBlocks{3}), vars(varsBlocks{3}))
```

```
x2 =
c2 - x3 - x4 - x5 + 2
```

Use `subs` to evaluate the result for the already known values of all other variables of this system.

```
x2 = subs(x2)
```

```
x2 =
0
```

Alternatively, you can rewrite this example using the `for`-loop. This approach lets you use the example for larger systems of equations.

```
syms x1 x2 x3 x4 x5 x6 c1 c2 c3

eqs = [c1*x1 + x3 + x5 == c1 + c2 + 1;...
       x1 + x3 + x4 + 2*x6 == 4 + c2;...
       x1 + 2*x3 + c3*x5 == 1 + 2*c2 + c3;...
       x2 + x3 + x4 + x5 == 2 + c2;...
       x1 - c2*x3 + x5 == 2 - c2^2
       x1 - x3 + x4 - x6 == 1 - c2];
```

```
vars = [x1, x2, x3, x4, x5, x6];

[eqsBlocks, varsBlocks] = findDecoupledBlocks(eqs, vars);

vars_sol = vars;

for i = 1:numel(eqsBlocks)
  sol = solve(eqs(eqsBlocks{i}), vars(varsBlocks{i}));
  vars_sol_per_block = subs(vars(varsBlocks{i}), sol);
  for k=1:i-1
    vars_sol_per_block = subs(vars_sol_per_block, vars(varsBlocks{k}),...
                         vars_sol(varsBlocks{k}));
  end
  vars_sol(varsBlocks{i}) = vars_sol_per_block
end
```

```
vars_sol =
[ 1, x2, c2, x4, 1, x6]

vars_sol =
[ 1, x2, c2, 1, 1, 1]

vars_sol =
[ 1, 0, c2, 1, 1, 1]
```

Input Arguments

`eqs` — System of equations

vector of symbolic equations | vector of symbolic expressions

System of equations, specified as a vector of symbolic equations or expressions.

`vars` — Variables

vector of symbolic variables | vector of symbolic functions | vector of symbolic function calls

Variables, specified as a vector of symbolic variables, functions, or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

Output Arguments

`eqsBlocks` **— Indices defining blocks of equations**
cell array

Indices defining blocks of equations, returned as a cell array. Each block of indices is a row vector of double-precision integer numbers. The *i*th block of equations consists of the equations `eqs(eqsBlocks{i})` and involves only the variables in `vars(varsBlocks{1:i})`.

`varsBlocks` **— Indices defining blocks of variables**
cell array

Indices defining blocks of variables, returned as a cell array. Each block of indices is a row vector of double-precision integer numbers. The *i*th block of equations consists of the equations `eqs(eqsBlocks{i})` and involves only the variables in `vars(varsBlocks{1:i})`.

Tips

- The implemented algorithm requires that for each variable in `vars` there must be at least one matching equation in `eqs` involving this variable. The same equation cannot also be matched to another variable. If the system does not satisfy this condition, then `findDecoupledBlocks` throws an error. In particular, `findDecoupledBlocks` requires that `length(eqs) = length(vars)`.
- Applying the permutations `e = [eqsBlocks{:}]` to the vector `eqs` and `v = [varsBlocks{:}]` to the vector `vars` produces an incidence matrix `incidenceMatrix(eqs(e), vars(v))` that has a block lower triangular sparsity pattern.

See Also

`daeFunction` | `decic` | `diag` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies` | `tril` | `triu`

Introduced in R2014b

findUnits

Find units in input

Syntax

```
U = findUnits(expr)
```

Description

`U = findUnits(expr)` returns a row vector of units in the symbolic expression `expr`.

Examples

Find Units in Expression

Find the units in an expression by using `findUnits`.

```
u = symunit;
syms x
units = findUnits(x*u.m + 2*u.N)
```

```
units =
[ [N], [m]]
```

Find Units in Array of Equations or Expressions

Find the units in an array of equations or expressions by using `findUnits`. The `findUnits` function concatenates all units found in the input to return a row vector of units. `findUnits` returns only base units.

```
u = symunit;
array = [2*u.m + 3*u.K, 1*u.N == 1*u.kg/(u.m*u.s^2)];
units = findUnits(array)
units =
[ [K], [N], [kg], [m], [s]]
```

Input Arguments

`expr` — Input
symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

See Also

`checkUnits` | `isUnit` | `newUnit` | `separateUnits` | `str2symunit` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

finverse

Functional inverse

Syntax

```
g = finverse(f)
g = finverse(f,var)
```

Description

`g = finverse(f)` returns the inverse of function `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.

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)
```

Tips

- `finverse` does not issue a warning when the inverse is not unique.

See Also

`compose` | `syms`

Introduced before R2006a

fix

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

`ceil` | `floor` | `frac` | `round`

Introduced before R2006a

floor

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

`ceil` | `fix` | `frac` | `round`

Introduced before R2006a

fmesh

Plot 3-D mesh

Syntax

```
fmesh(f)
fmesh(f,[min max])
fmesh(f,[xmin xmax ymin ymax])

fmesh(funx,funy,funz)
fmesh(funx,funy,funz,[uvmin uvmax])
fmesh(funx,funy,funz,[umin umax vmin vmax])

fmesh(___,LineSpec)
fmesh(___,Name,Value)
fmesh(ax,___)
obj = fmesh(___)
```

Description

`fmesh(f)` creates a mesh plot of the symbolic expression `f(x,y)` over the default interval `[-5 5]` for `x` and `y`.

`fmesh(f,[min max])` plots `f(x,y)` over the interval `[min max]` for `x` and `y`.

`fmesh(f,[xmin xmax ymin ymax])` plots `f(x,y)` over the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

`fmesh(funx,funy,funz)` plots the parametric mesh `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[-5 5]` for `u` and `v`.

`fmesh(funx,funy,funz,[uvmin uvmax])` plots the parametric mesh `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[uvmin uvmax]` for `u` and `v`.

`fmesh(funx,funy,funz,[umin umax vmin vmax])` plots the parametric mesh `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[umin umax]` for `u` and `[vmin vmax]` for `v`.

`fmesh(___,LineSpec)` uses the `LineSpec` to set the line style, marker symbol, and plot color.

`fmesh(___,Name,Value)` specifies surface properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

`fmesh(ax,___)` plots into the axes with the object `ax` instead of the current axes object `gca`.

`obj = fmesh(___)` returns a function surface object or a parameterized function surface object. Use the object to query and modify properties of a specific mesh.

Examples

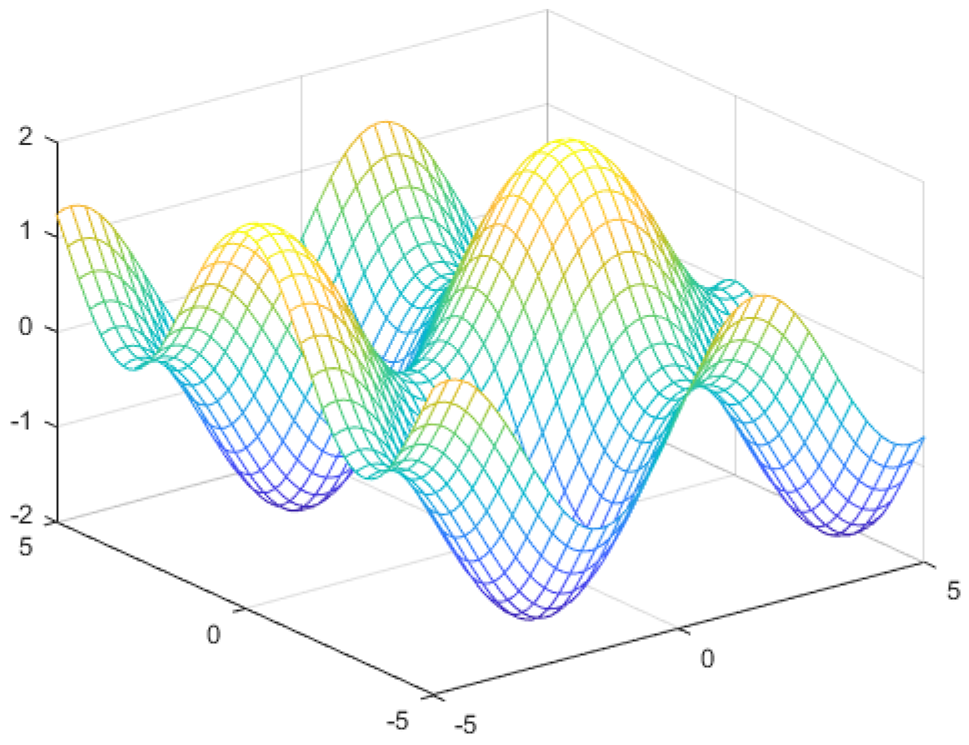
Additional Examples: See **`fsurf`** Page

Note For additional examples, follow the `fsurf` page because `fmesh` and `fsurf` share the same syntax. All examples on the `fsurf` page apply to `fmesh`.

3-D Mesh Plot of Symbolic Expression

Plot a mesh of the input $\sin(x) + \cos(y)$ over the default range $-5 < x < 5$ and $-5 < y < 5$.

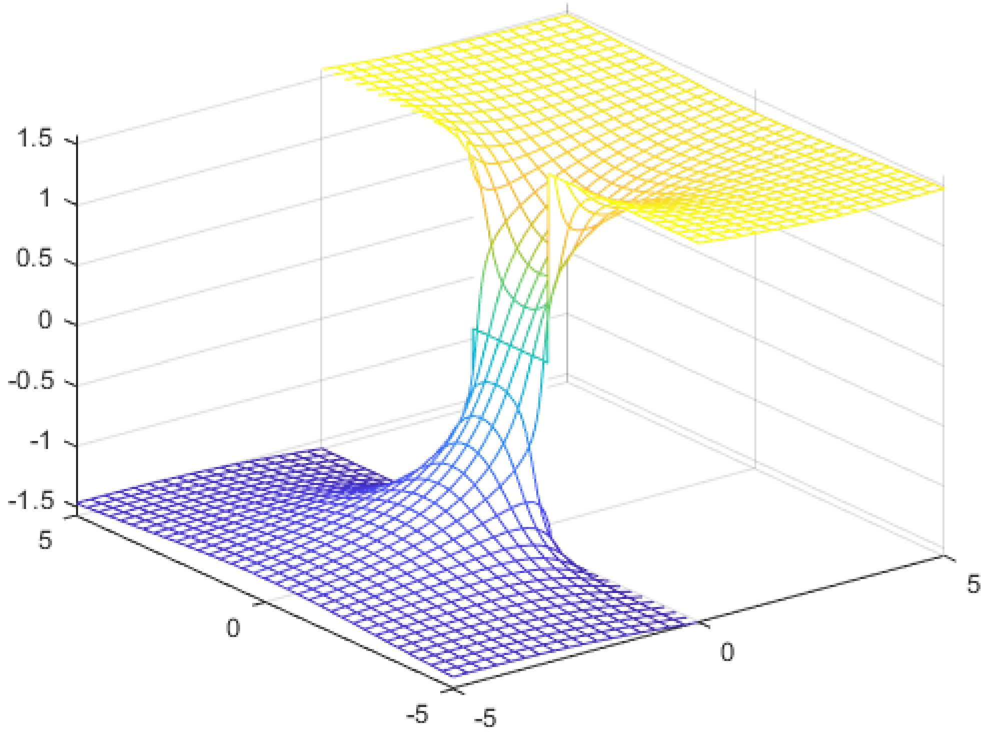
```
syms x y
fmesh(sin(x)+cos(y))
```



3-D Mesh Plot of Symbolic Function

Plot a 3-D mesh of the real part of $\tan^{-1}(x + iy)$ over the default range $-5 < x < 5$ and $-5 < y < 5$.

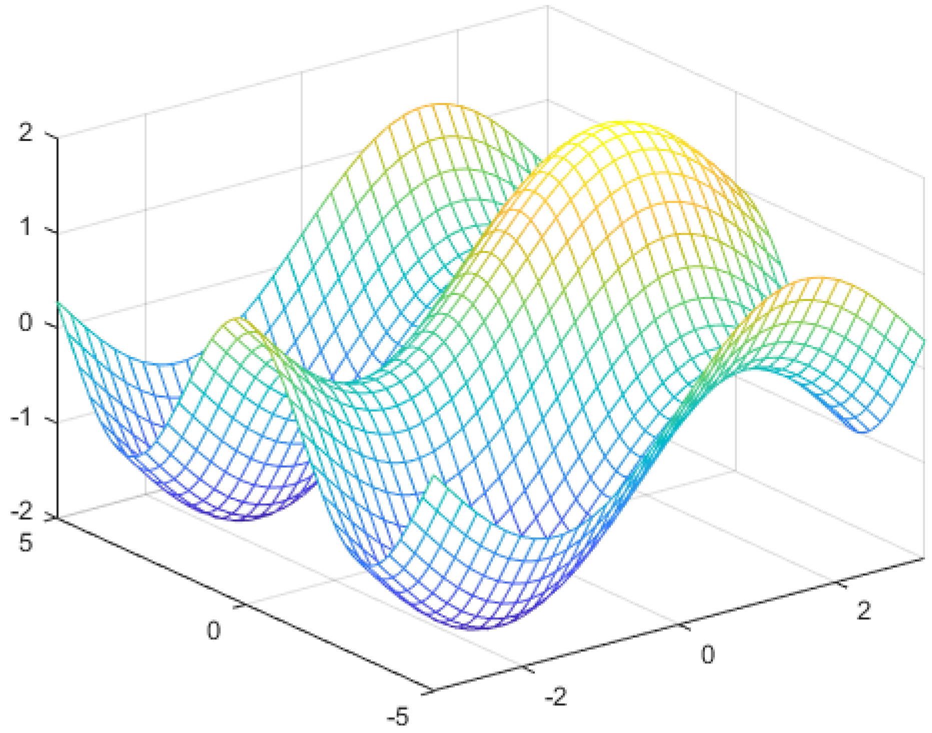
```
syms f(x,y)
f(x,y) = real(atan(x + i*y));
fmesh(f)
```



Specify Plotting Interval of Mesh Plot

Plot $\sin(x) + \cos(y)$ over $-\pi < x < \pi$ and $-5 < y < 5$ by specifying the plotting interval as the second argument of `fmesh`.

```
syms x y
f = sin(x) + cos(y);
fmesh(f, [-pi pi -5 5])
```



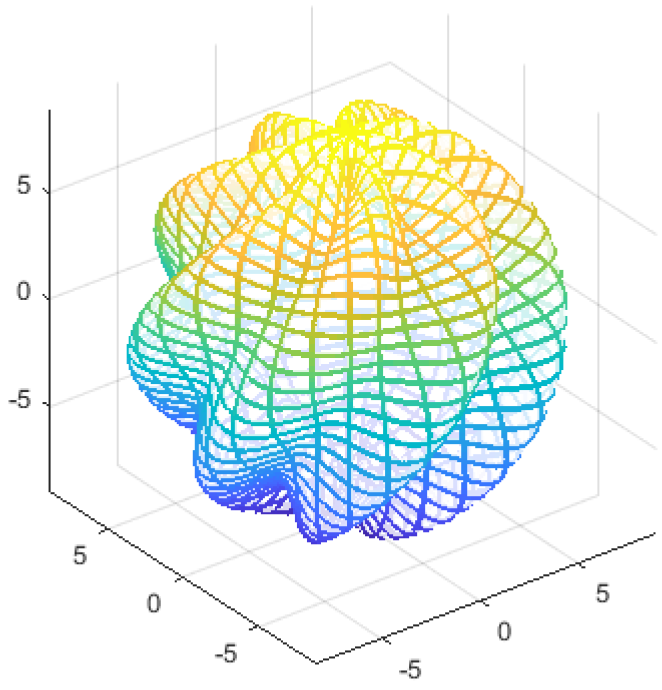
Parameterized Mesh Plot

Plot the parameterized mesh

$$x = r\cos(s)\sin(t)$$
$$y = r\sin(s)\sin(t)$$
$$z = r\cos(t)$$
$$\text{where} \quad r = 8 + \sin(7s + 5t)$$

for $0 < s < 2\pi$ and $0 < t < \pi$. Make the aspect ratio of the axes equal using `axis equal`. See the entire mesh by making the mesh partially transparent using `alpha`.

```
syms s t
r = 8 + sin(7*s + 5*t);
x = r*cos(s)*sin(t);
y = r*sin(s)*sin(t);
z = r*cos(t);
fmesh(x, y, z, [0 2*pi 0 pi], 'Linewidth', 2)
axis equal
```



```
alpha(0.8)
```

Additional Examples: See **`fsurf`** Page

Note For additional examples, follow the `fsurf` page because `fmesh` and `fsurf` share the same syntax. All examples on the `fsurf` page apply to `fmesh`.

Input Arguments

`f` — 3-D expression or function to be plotted
symbolic expression | symbolic function

Expression or function to be plotted, specified as a symbolic expression or function.

`[min max]` — Plotting interval for x- and y-axes
[–5 5] (default) | vector of two numbers

Plotting interval for x- and y-axes, specified as a vector of two numbers. The default is `[-5 5]`.

`[xmin xmax ymin ymax]` — Plotting interval for x- and y-axes
[–5 5 –5 5] (default) | vector of four numbers

Plotting interval for x- and y-axes, specified as a vector of four numbers. The default is `[-5 5 -5 5]`.

`funx,funy,funz` — Parametric functions of `u` and `v`
symbolic expressions | symbolic functions

Parametric functions of `u` and `v`, specified as a symbolic expression or function.

`[uvmin uvmax]` — Plotting interval for `u` and `v`
[–5 5] (default) | vector of two numbers

Plotting interval for `u` and `v` axes, specified as a vector of two numbers. The default is `[-5 5]`.

`[umin umax vmin vmax]` — Plotting interval for `u` and `v`
[–5 5 –5 5] (default) | vector of four numbers

Plotting interval for `u` and `v`, specified as a vector of four numbers. The default is `[-5 5 -5 5]`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fmesh` uses the current axes.

`LineSpec` — Line style, marker symbol, and line color
character vector

Line style, marker symbol, and line color, specified as a character vector. The elements of the character vector can appear in any order, and you can omit one or more options from the character vector specifier.

Example: `'--or'` is a red mesh with circle markers

Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
Specifier	**Marker**
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram

Specifier	Marker
h	Hexagram
Specifier	**Color**
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

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`.

Example: `'Marker','o','MarkerFaceColor','red'`

`MeshDensity` **— Number of evaluation points per direction**
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `fmesh` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` **— Display contour plot under plot**
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

`EdgeColor` **— Line color**
`'interp'` (default) | RGB triplet | `'none'` | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, or one of the color options listed in the table. The default value of `'interp'` colors the edges based on the `ZData` property values.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	———————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	………………

Line Style	Description	Resulting Line
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`obj` — One or more objects
scalar | vector

One or more objects, returned as a scalar or a vector. The object is either a function surface object or parameterized mesh object, depending on the type of plot. You can use these objects to query and modify properties of a specific line. For details, see Function Surface and Parameterized Function Surface.

See Also

Functions

`fcontour` | `fimplicit` | `fimplicit3` | `fplot` | `fplot3` | `fsurf`

Properties

Function Surface | Parameterized Function Surface

Topics

"Create Plots" on page 2-240

Introduced in R2016a

fold

Combine (fold) vector using function

Syntax

```
fold(fun,v)
fold(fun,v,defaultVal)
```

Description

`fold(fun,v)` folds `v` by using `fun`. That is, `fold` calls `fun` on the first two elements of `v`, and then repeatedly calls `fun` on the result and the next element till the last element is combined. Programmatically, the fold operation is `fold(fun,v) = fun(fold(fun,v(1:end-1)),v(end))`.

`fold(fun,v,defaultVal)` returns the value `defaultVal` if `v` is empty.

Examples

Fold Vector Using Function

Fold a vector of symbolic variables using the `power` function. The output shows how `fold` combines elements of the vector from left to right by using the specified function.

```
syms a b c d e
fold(@power, [a b c d e])
```

```
ans =
(((a^b)^c)^d)^e
```

Assume Variable Belongs to Set of Values

Assume the variable `x` belongs to the set of values 1, 2, ..., 10 by applying `or` to the conditions `x == 1`, ..., `x == 10` using `fold`. Check that the assumption is set by using `assumptions`.

```
syms x
cond = fold(@or, x == 1:10);
assume(cond)
assumptions
```

```
ans =
x == 1 | x == 2 | x == 3 | x == 4 | x == 5 |...
 x == 6 | x == 7 | x == 8 | x == 9 | x == 10
```

Specify Default Value of Fold Operation

Specify the default value of `fold` when the input is empty by specifying the third argument. If the third argument is not specified and the input is empty, then `fold` throws an error.

When creating a function to sum a vector, specify a default value of `0`, such that the function returns `0` when the vector is empty.

```
sumVector = @(x) fold(@plus, x, 0);
sumVector([])
```

```
ans =
     0
```

Input Arguments

`fun` — Function used to fold vector
function handle

Function used to fold vector, specified as a function handle.

Example: @or

`v` — Vector to fold
vector | symbolic vector | cell vector

Vector to fold, specified as a vector, symbolic vector, or cell vector. If an element of `v` is a symbolic function, then the formula of the symbolic function is used by calling `formula`.

`defaultVal` — Default value of fold operation
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Default value of fold operation, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

See Also

`prod` | `sum`

Introduced in R2016b

formula

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`

Introduced in R2012a

fortran

Fortran representation of symbolic expression

Syntax

```
fortran(f)
fortran(f,Name,Value)
```

Description

`fortran(f)` returns Fortran code for the symbolic expression `f`.

`fortran(f,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Generate Fortran Code from Symbolic Expression

Generate Fortran code from the symbolic expression `log(1+x)`.

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

```
ans =
    '      t0 = log(x+1.0D0)'
```

Generate Fortran code for the 3-by-3 Hilbert matrix.

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

```
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'
```

Write Fortran Code to File with Comments

Write generated Fortran code to a file by specifying the `File` option. When writing to a file, `fortran` optimizes the code using intermediate variables named `t0`, `t1`, ... Include comments in the file by using the `Comments` option.

```
syms x
f = diff(tan(x));
fortran(f,'File','fortrantest')
```

```
      t2 = tan(x)
      t0 = t2**2+1.0D0
```

Include the comment `Version: 1.1`. Comment lines must be shorter than 71 characters to conform with Fortran 77.

```
fortran(f,'File','fortrantest','Comments','Version: 1.1')
```

```
*Version: 1.1
      t2 = tan(x)
      t0 = t2**2+1.0D0
```

Input Arguments

`f` — Symbolic input

symbolic expression

Symbolic input, specified as a symbolic expression.

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`.

Example: `fortran(x^2,'File','fortrancode','Comments','V1.2')`

`File` — File to write to

character vector | string

File to write to, specified as a character vector or string. When writing to a file, `fortran` optimizes the code using intermediate variables named `t0`, `t1`, ...

`Comments` — Comments to include in file header

character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector. Comment lines must be shorter than 71 characters to conform with Fortran 77.

Tips

- MATLAB is left-associative while Fortran is right-associative. If ambiguity exists in an expression, the `fortran` function must follow MATLAB to create an equivalent representation. For example, `fortran` represents `a^b^c` in MATLAB as `(a**b)**c` in Fortran.

See Also

`ccode` | `latex` | `matlabFunction` | `pretty`

Introduced before R2006a

fourier

Fourier transform

Syntax

```
fourier(f)
fourier(f,transVar)
fourier(f,var,transVar)
```

Description

`fourier(f)` returns the “Fourier Transform” on page 4-662 of `f`. By default, the function `symvar` determines the independent variable, and `w` is the transformation variable.

`fourier(f,transVar)` uses the transformation variable `transVar` instead of `w`.

`fourier(f,var,transVar)` uses the independent variable `var` and the transformation variable `transVar` instead of `symvar` and `w`, respectively.

Examples

Fourier Transform of Symbolic Expression

Compute the Fourier transform of `exp(-t^2)`. By default, the transform is in terms of `w`.

```
syms t
f = exp(-t^2);
ft_f = fourier(f)
```

```
ft_f =
pi^(1/2)*exp(-w^2/4)
```

Specify Independent Variable and Transformation Variable

Compute the Fourier transform of `exp(-t^2-x^2)`. By default, `symvar` determines the independent variable, and `w` is the transformation variable. Here, `symvar` chooses `x`.

```
syms t x
f = exp(-t^2-x^2);
fourier(f)
```

```
ans =
pi^(1/2)*exp(- t^2 - w^2/4)
```

Specify the transformation variable as `y`. If you specify only one variable, that variable is the transformation variable. `symvar` still determines the independent variable.

```
syms y
fourier(f,y)
```

```
ans =
pi^(1/2)*exp(- t^2 - y^2/4)
```

Specify both the independent and transformation variables as `t` and `y` in the second and third arguments, respectively.

```
fourier(f,t,y)
```

```
ans =
pi^(1/2)*exp(- x^2 - y^2/4)
```

Fourier Transforms Involving Dirac and Heaviside Functions

Compute the following Fourier transforms. The results are in terms of the Dirac and Heaviside functions.

```
syms t w
fourier(t^3, t, w)
```

```
ans =
-pi*dirac(3, w)*2i
```

```
syms t0
fourier(heaviside(t - t0), t, w)
```

```
ans =
exp(-t0*w*1i)*(pi*dirac(w) - 1i/w)
```

Specify Fourier Transform Parameters

Specify parameters of the Fourier transform.

Compute the Fourier transform of `f` using the default values of the Fourier parameters `c = 1`, `s = -1`. For details, see “Fourier Transform” on page 4-662.

```
syms t w
f = t*exp(-t^2);
fourier(f,t,w)

ans =
-(w*pi^(1/2)*exp(-w^2/4)*1i)/2
```

Change the Fourier parameters to `c = 1`, `s = 1` by using `sympref`, and compute the transform again. The result changes.

```
sympref('FourierParameters',[1 1]);
fourier(f,t,w)

ans =
(w*pi^(1/2)*exp(-w^2/4)*1i)/2
```

Change the Fourier parameters to `c = 1/(2*pi)`, `s = 1`. The result changes.

```
sympref('FourierParameters', [1/(2*sym(pi)), 1]);
fourier(f,t,w)

ans =
(w*exp(-w^2/4)*1i)/(4*pi^(1/2))
```

Preferences set by `sympref` persist through your current and future MATLAB sessions. Restore the default values of `c` and `s` by setting `FourierParameters` to `'default'`.

```
sympref('FourierParameters','default');
```

Fourier Transform of Array Inputs

Find the Fourier transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `fourier` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x) 1; sin(y) i*z];
vars = [w x; y z];
transVars = [a b; c d];
fourier(M,vars,transVars)

ans =
[                 2*pi*exp(x)*dirac(a),     2*pi*dirac(b)]
[ -pi*(dirac(c - 1) - dirac(c + 1))*1i, -2*pi*dirac(1, d)]
```

If `fourier` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
fourier(x,vars,transVars)

ans =
[ 2*pi*x*dirac(a), pi*dirac(1, b)*2i]
[ 2*pi*x*dirac(c),   2*pi*x*dirac(d)]
```

If Fourier Transform Cannot Be Found

If `fourier` cannot transform the input then it returns an unevaluated call.

```
syms f(t) w
F = fourier(f,t,w)

F =
fourier(f(t), t, w)
```

Return the original expression by using `ifourier`.

```
ifourier(F,w,t)
```

```
ans =
f(t)
```

Input Arguments

`f` — Input

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

`x` (default) | symbolic variable

Independent variable, specified as a symbolic variable. This variable is often called the "time variable" or the "space variable." If you do not specify the variable, then `fourier` uses the function `symvar` to determine the independent variable.

`transVar` — Transformation variable

`w` (default) | `v` | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. This variable is often called the "frequency variable." By default, `fourier` uses `w`. If `w` is the independent variable of `f`, then `fourier` uses `v`.

Definitions

Fourier Transform

The Fourier transform of the expression $f = f(x)$ with respect to the variable x at the point w is

$$F(w) = c\int_{-\infty}^{\infty} f(x)\, e^{iswx}\, dx.$$

c and s are parameters of the Fourier transform. The `fourier` function uses $c = 1$, $s = –1$.

Tips

- If any argument is an array, then `fourier` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.
- To compute the inverse Fourier transform, use `ifourier`.

References

[1] Oberhettinger F., "Tables of Fourier Transforms and Fourier Transforms of Distributions." Springer, 1990.

See Also

`ifourier` | `ilaplace` | `iztrans` | `laplace` | `sympref` | `ztrans`

Topics

"Fourier and Inverse Fourier Transforms" on page 2-220

Introduced before R2006a

fplot

Plot symbolic expression or function

Syntax

```
fplot(f)
fplot(f,[xmin xmax])

fplot(xt,yt)
fplot(xt,yt,[tmin tmax])

fplot( ___ ,LineSpec)
fplot( ___ ,Name,Value)
fplot(ax, ___ )
fp = fplot( ___ )
```

Description

`fplot(f)` plots symbolic input `f` over the default interval `[-5 5]`.

`fplot(f,[xmin xmax])` plots `f` over the interval `[xmin xmax]`.

`fplot(xt,yt)` plots $xt = x(t)$ and $yt = y(t)$ over the default range of `t`, which is `[−5 5]`.

`fplot(xt,yt,[tmin tmax])` plots $xt = x(t)$ and $yt = y(t)$ over the specified range `[tmin tmax]`.

`fplot( ___ ,LineSpec)` uses `LineSpec` to set the line style, marker symbol, and line color.

`fplot( ___ ,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. `Name,Value` pair settings apply to all the lines plotted. To set options for individual lines, use the objects returned by `fplot`.

`fplot(ax, ___ )` plots into the axes specified by `ax` instead of the current axes `gca`.

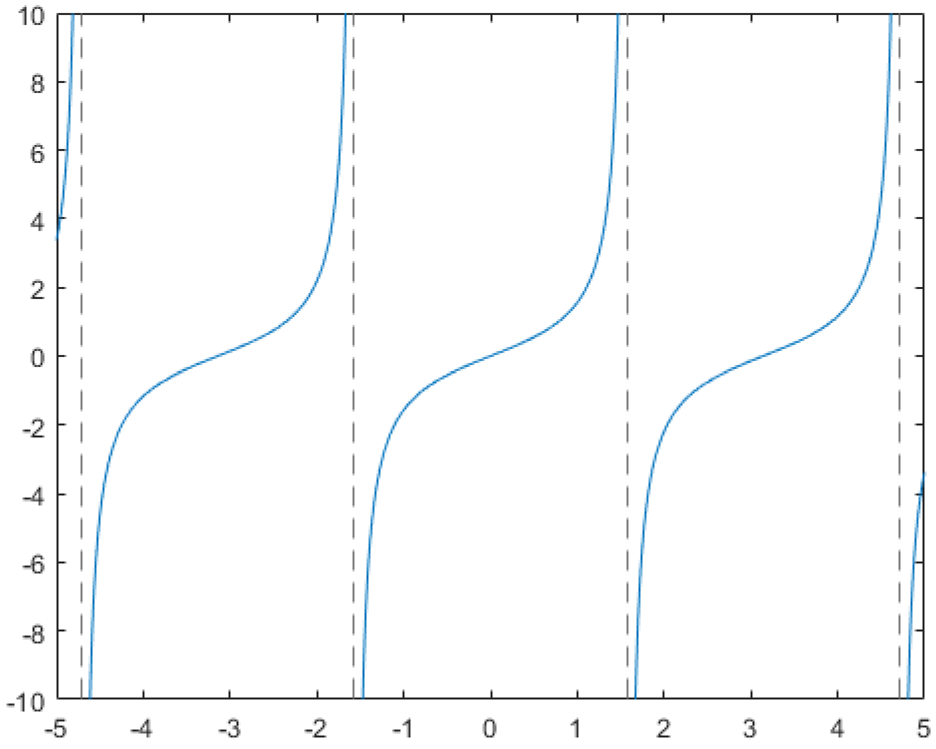
`fp = fplot(___)` returns a function line object or parameterized line object, depending on the type of plot. Use the object to query and modify properties of a specific line. For details, see Function Line and Parameterized Function Line.

Examples

Plot Symbolic Expression

Plot `tan(x)` over the default range of `[-5 5]`. `fplot` shows poles by default. For details, see the `ShowPoles` argument in “Name-Value Pair Arguments” on page 4-682.

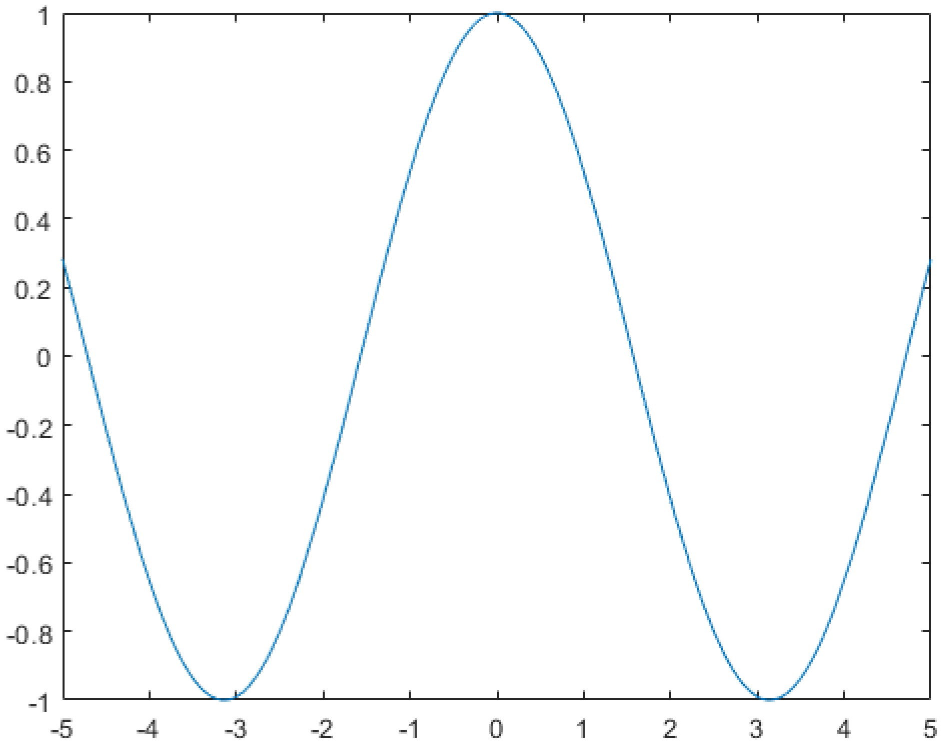
```
syms x
fplot(tan(x))
```



Plot Symbolic Function

Plot the symbolic function $f(x) = cos(x)$ over the default range `[-5 5]`.

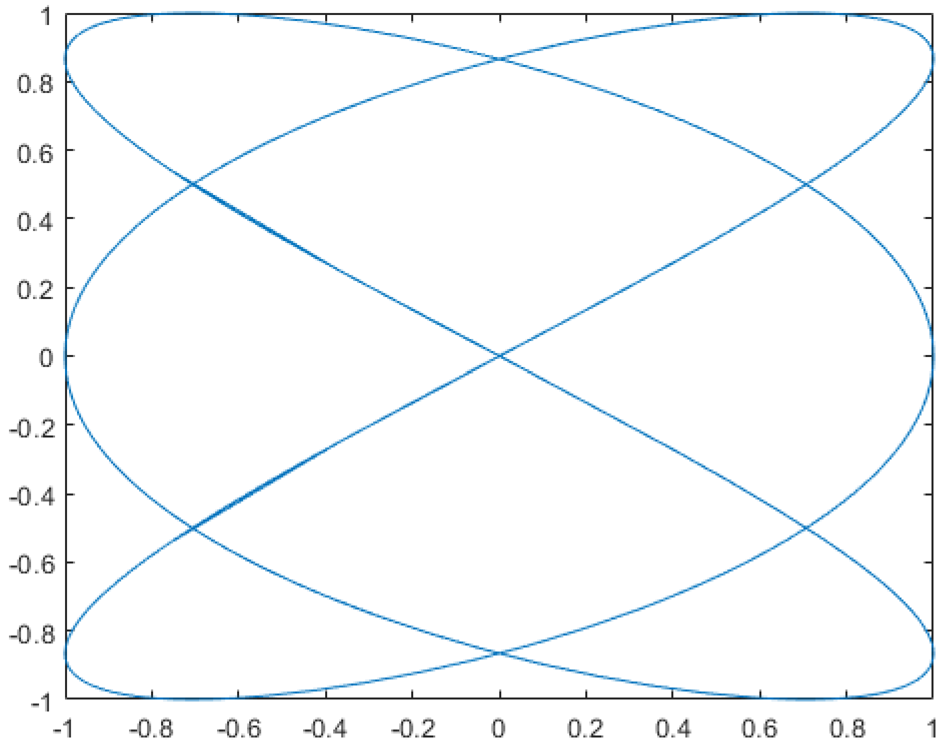
```
syms f(x)
f(x) = cos(x);
fplot(f)
```



Plot Parametric Curve

Plot the parameteric curve $x = \cos(3t)$ and $y = \sin(2t)$.

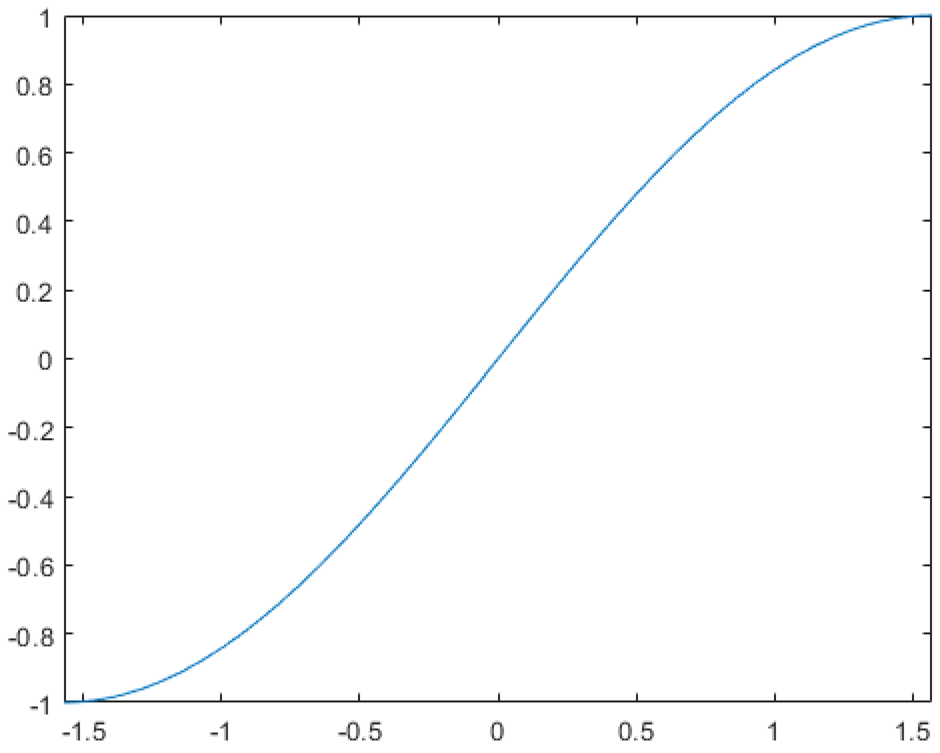
```
syms t
x = cos(3*t);
y = sin(2*t);
fplot(x,y)
```



Specify Plotting Interval

Plot $\sin(x)$ over $[-\pi/2, \pi/2]$ by specifying the plotting interval as the second input to `fplot`.

```
syms x
fplot(sin(x),[-pi/2 pi/2])
```

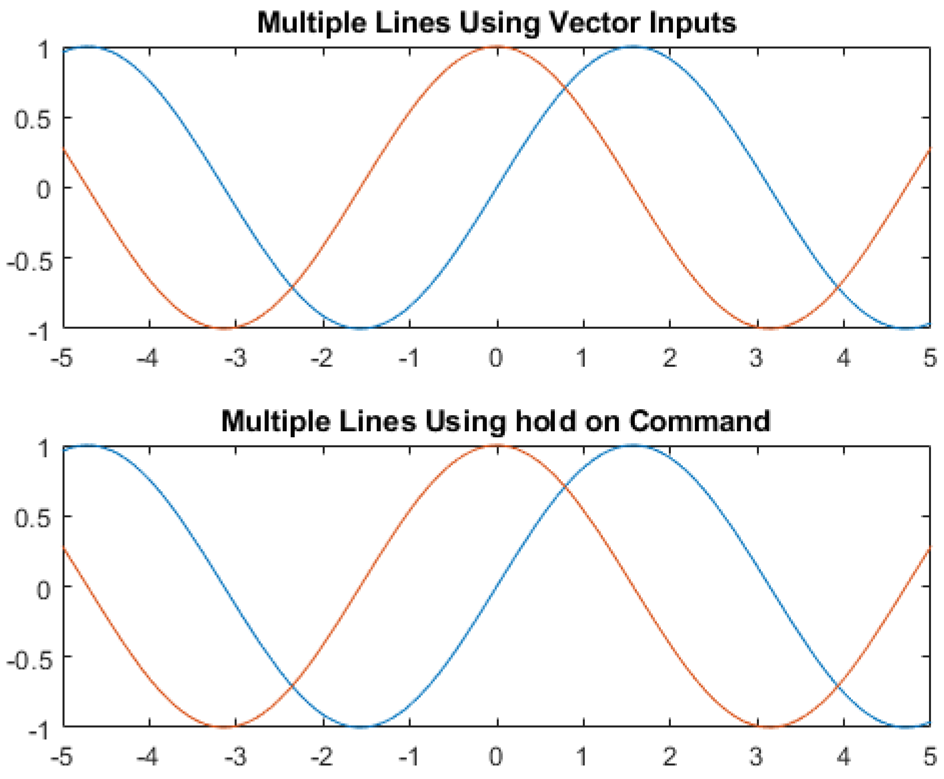


Plot Multiple Lines on Same Figure

You can plot multiple lines either by passing the inputs as a vector or by using `hold on` to successively plot on the same figure. If you specify `LineSpec` and Name-Value arguments, they apply to all lines. To set options for individual plots, use the function handles returned by `fplot`.

Divide a figure into two subplots using `subplot`. On the first subplot, plot $\sin(x)$ and $\cos(x)$ using vector input. On the second subplot, plot $\sin(x)$ and $\cos(x)$ using `hold on`.

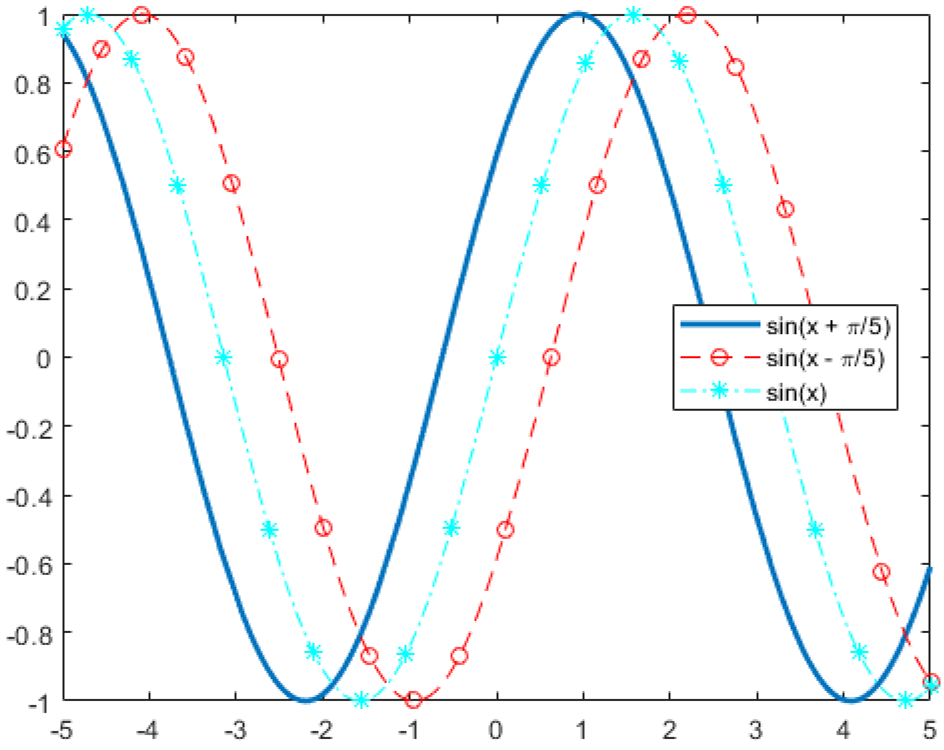
```
syms x
subplot(2,1,1)
fplot([sin(x) cos(x)])
title('Multiple Lines Using Vector Inputs')

subplot(2,1,2)
fplot(sin(x))
hold on
fplot(cos(x))
title('Multiple Lines Using hold on Command')

hold off
```



Change Line Properties and Display Markers

Plot three sine curves with a phase shift between each line. For the first line, use a linewidth of 2. For the second, specify a dashed red line style with circle markers. For the third, specify a cyan, dash-dot line style with asterisk markers. Display the legend.

```
syms x
fplot(sin(x+pi/5),'Linewidth',2)
hold on
fplot(sin(x-pi/5),'--or')
fplot(sin(x),'-.*c')
legend('show','Location','best')
hold off
```

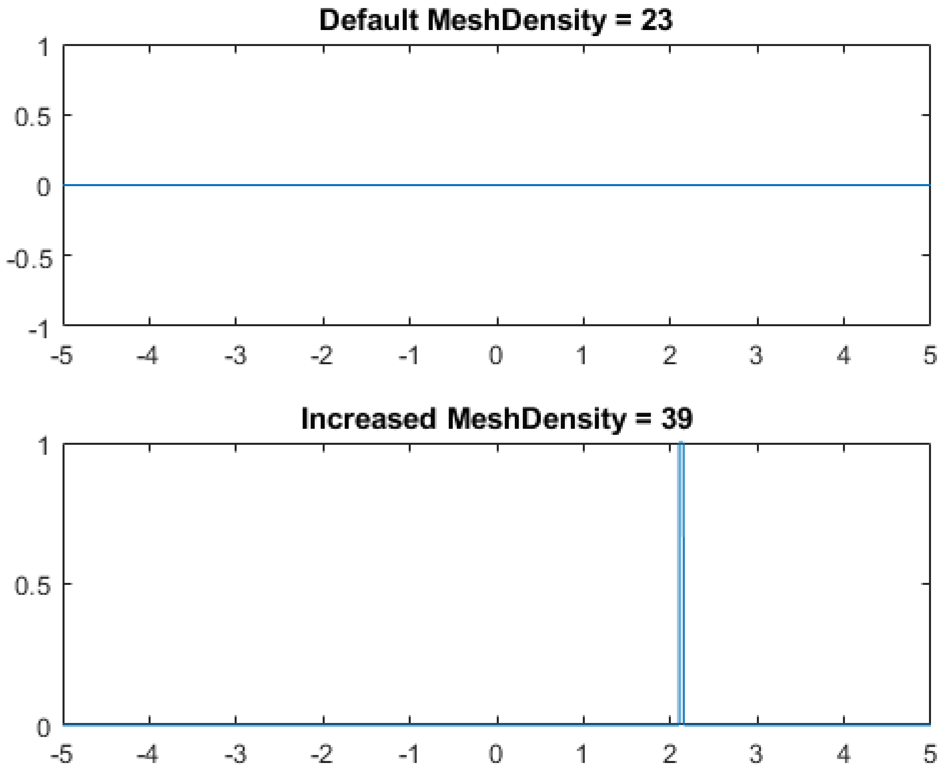


Control Resolution of Plot

Control the resolution of a plot by using the `MeshDensity` option. Increasing `MeshDensity` can make smoother, more accurate plots, while decreasing it can increase plotting speed.

Divide a figure into two by using `subplot`. In the first subplot, plot a step function from `x = 2.1` to `x = 2.15`. The plot's resolution is too low to detect the step function. Fix this issue by increasing `MeshDensity` to `39` in the second subplot. The plot now detects the step function and shows that by increasing `MeshDensity` you increased the plot's resolution.

```
syms x
stepFn = rectangularPulse(2.1, 2.15, x);

subplot(2,1,1)
fplot(stepFn);
title('Default MeshDensity = 23')

subplot(2,1,2)
fplot(stepFn,'MeshDensity',39);
title('Increased MeshDensity = 39')
```

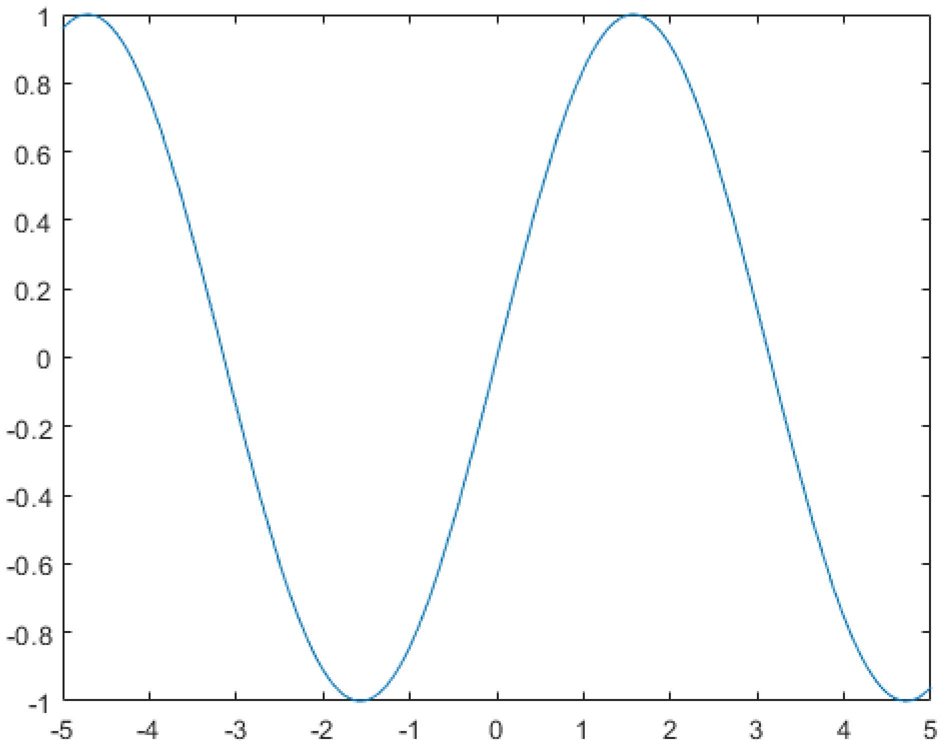


Modify Plot After Creation

Plot `sin(x)`. Specify an output to make `fplot` return the plot object.

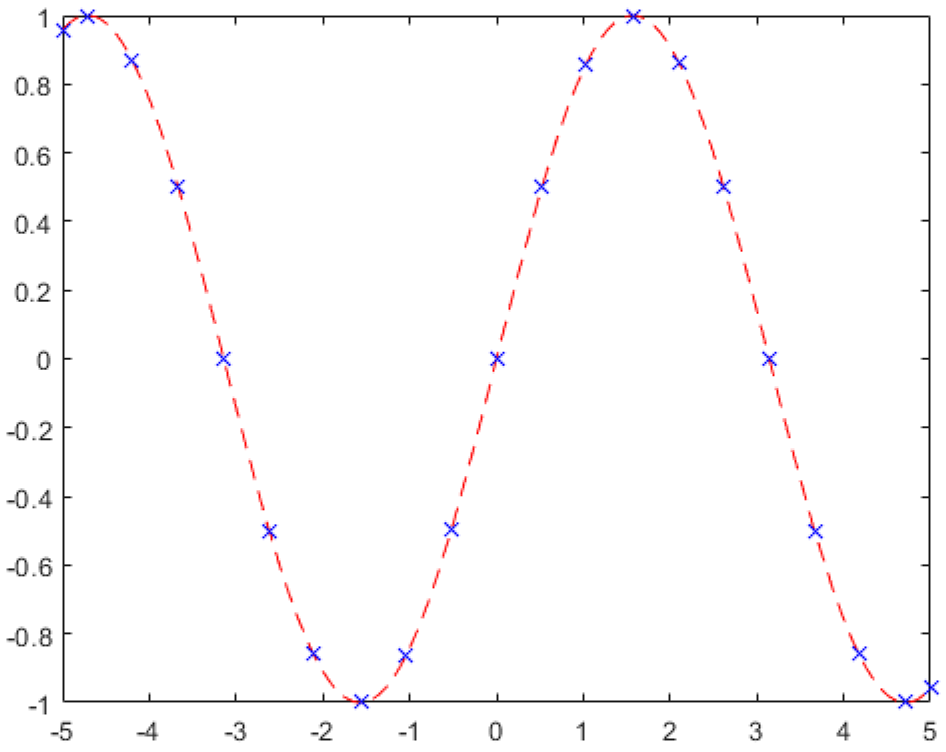
```
syms x
h = fplot(sin(x))
```

```
h = 
  FunctionLine with properties:

      Function: [1x1 sym]
         Color: [0 0.4470 0.7410]
     LineStyle: '-'
     LineWidth: 0.5000

  Show all properties
```



Change the default blue line to a dashed red line by using dot notation to set properties. Similarly, add `'x'` markers and set the marker color to blue.

```
h.LineStyle = '--';
h.Color = 'r';
h.Marker = 'x';
h.MarkerEdgeColor = 'b';
```



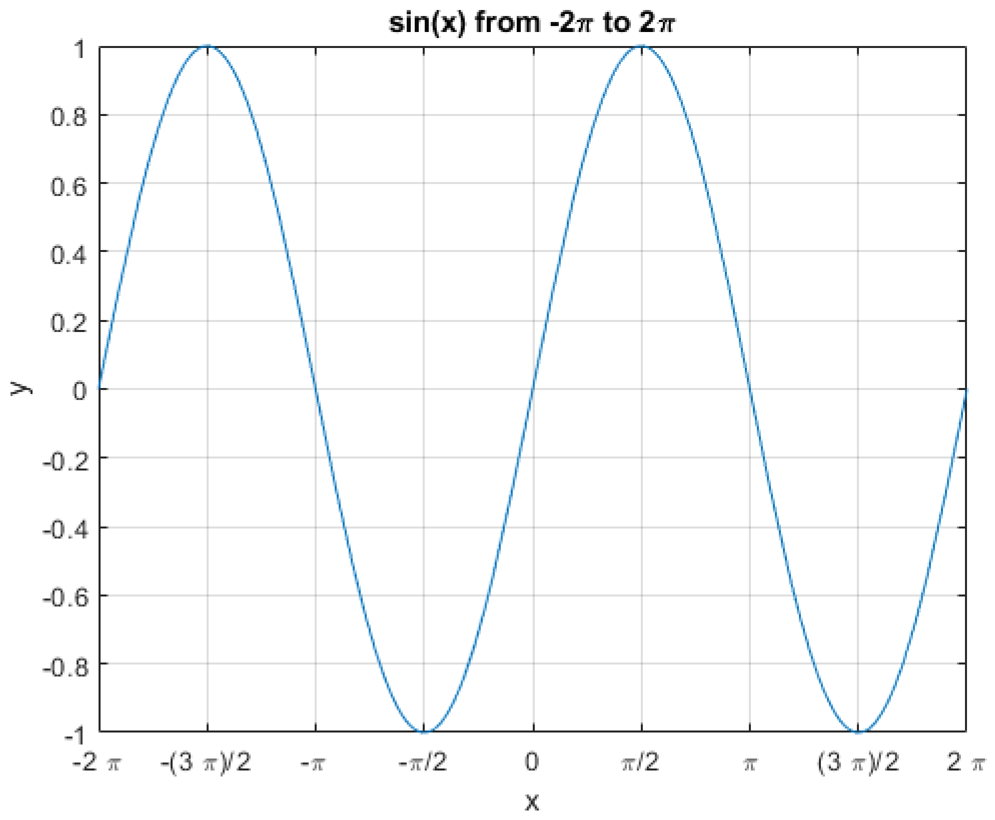
Add Title and Axis Labels and Format Ticks

For x from -2π to 2π, plot $sin(x)$. Add a title and axis labels. Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property.

To use LaTeX in plots, see `latex`.

```
syms x
fplot(sin(x),[-2*pi 2*pi])
```

```
grid on
title('sin(x) from -2\pi to 2\pi')
xlabel('x')
ylabel('y')

ax = gca;
S = sym(ax.XLim(1):pi/2:ax.XLim(2));
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel,S,'UniformOutput',false);
```

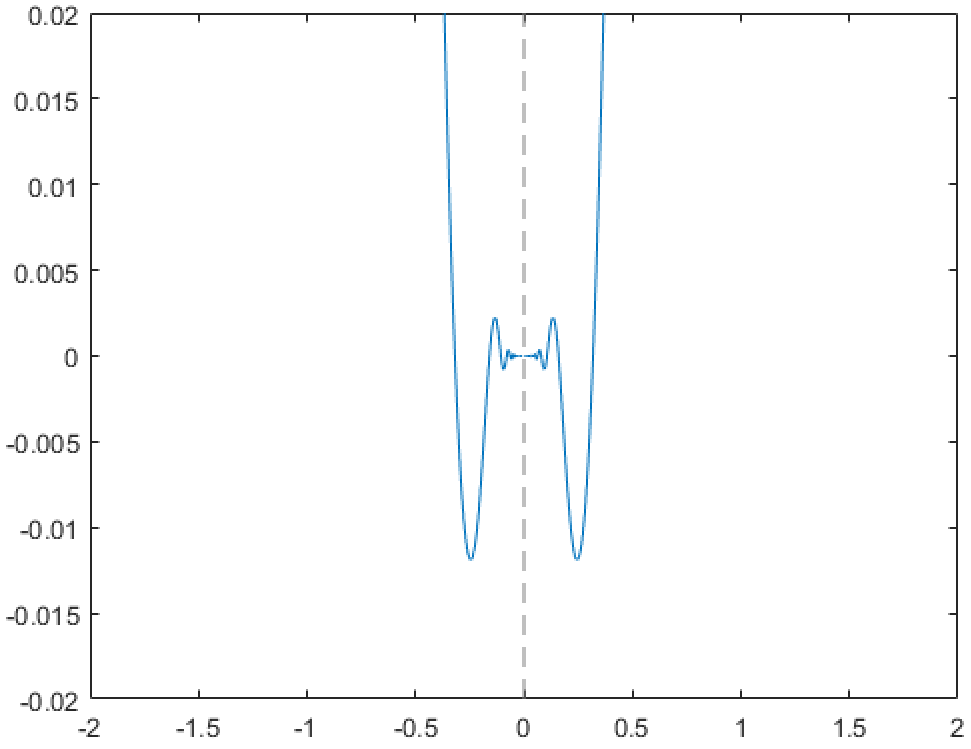


Re-evaluation on Zoom

When you zoom into a plot, `fplot` re-evaluates the plot automatically. This re-evaluation on zoom reveals hidden detail at smaller scales.

Plot `x^3*sin(1/x)` for `-2 < x < 2` and `-0.02 < y < 0.02`. Zoom in on the plot using `zoom` and redraw the plot using `drawnow`. Because of re-evaluation on zoom, `fplot` reveals smaller-scale detail. Repeat the zoom 6 times to view smaller-scale details. To play the animation, click the image.

```
syms x
fplot(x^3*sin(1/x));
axis([-2 2 -0.02 0.02]);
for i=1:6
    zoom(1.7)
    pause(0.5)
end
```



Create Animations

Create animations by changing the displayed expression using the `Function`, `XFunction`, and `YFunction` properties and then by using `drawnow` to update the plot. To export to GIF, see `imwrite`.

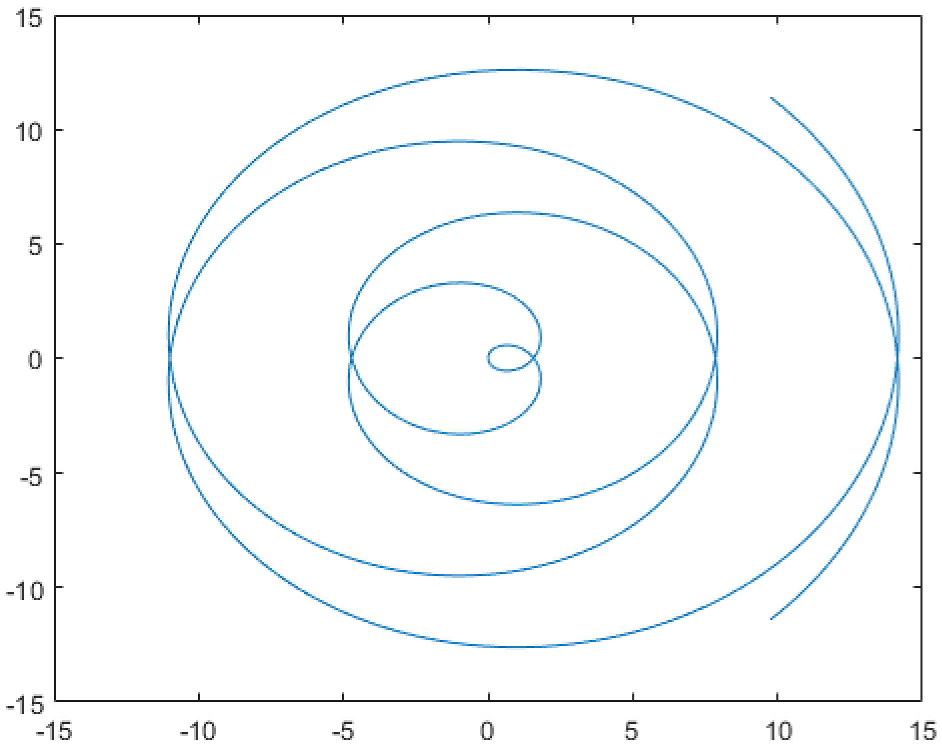
By varying the variable i from 0.1 to 3, animate the parametric curve

$$x = it\sin(it)$$

$$y = it\cos(it).$$

To play the animation, click the image.

```
syms t
fp = fplot(t, t);
axis([-15 15 -15 15])
for i=0.1:0.05:3
    fp.XFunction = i.*t.*sin(i*t);
    fp.YFunction = i.*t.*cos(i*t);
    drawnow
end
```



Input Arguments

`f` — Expression or function to plot
symbolic expression | symbolic function

Expression or function to plot, specified as a symbolic expression or function.

`[xmin xmax]` — Plotting interval for x-coordinates
[–5 5] (default) | vector of two numbers

Plotting interval for x-coordinates, specified as a vector of two numbers. The default range is `[-5 5]`. However, if `fplot` detects a finite number of discontinuities in `f`, then `fplot` expands the range to show them.

`xt` — Parametric input for x-coordinates
symbolic expression | symbolic function

Parametric input for x-coordinates, specified as a symbolic expression or function. `fplot` uses `symvar` to find the parameter.

`yt` — Parametric input for y-axis
symbolic expression | symbolic function

Parametric input for y-axis, specified as a symbolic expression or function. `fplot` uses `symvar` to find the parameter.

`[tmin tmax]` — Range of values of parameter `t`
[–5 5] (default) | vector of two numbers

Range of values of parameter `t`, specified as a vector of two numbers. The default range is `[-5 5]`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fplot` uses the current axes `gca`.

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
Marker Specifier	**Description**
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram
Color Specifier	**Description**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green

Color Specifier	Description
`b`	blue
`w`	white
`k`	black

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`.

The function line properties listed here are only a subset. For a complete list, see Function Line.

Example: `'Marker','o','MarkerFaceColor','red'`

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `fplot` uses adaptive evaluation, the actual number of evaluation points is greater.

`ShowPoles` — Display asymptotes at poles
`'on'` (default) | `'off'`

Display asymptotes at poles, specified as `'on'` (default) or `'off'`. The asymptotes display as gray, dashed vertical lines. `fplot` displays asymptotes only with the `fplot(f)` syntax or variants, and not with the `fplot(xt,yt)` syntax.

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet or one of the color options listed in the table.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	·················
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`

Option	Description	Equivalent RGB Triplet
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`fp` — One or more function or parameterized line objects
scalar | vector

One or more function or parameterized function line objects, returned as a scalar or a vector.

- If you use the `fplot(f)` syntax or a variation of this syntax, then `fplot` returns function line objects.
- If you use the `fplot(xt,yt)` syntax or a variation of this syntax, then `fplot` returns parameterized line objects.

You can use these objects to query and modify properties of a specific line. For a list of properties, see Function Line and Parameterized Function Line.

Tips

- If `fplot` detects a finite number of discontinuities in `f`, then `fplot` expands the range to show them.

See Also

Functions

`fcontour` | `fimplicit` | `fimplicit3` | `fmesh` | `fplot3` | `fsurf`

Properties

Function Line | Parameterized Function Line

Topics

"Create Plots" on page 2-240

Introduced in R2016a

fplot3

Plot 3-D parametric curve

Syntax

```
fplot3(xt,yt,zt)
fplot3(xt,yt,zt,[tmin tmax])

fplot3(___,LineSpec)
fplot3(___,Name,Value)
fplot3(ax,___)
fp = fplot3(___)
```

Description

`fplot3(xt,yt,zt)` plots the parametric curve $xt = x(t)$, $yt = y(t)$, and $zt = z(t)$ over the default interval $-5 < t < 5$.

`fplot3(xt,yt,zt,[tmin tmax])` plots $xt = x(t)$, $yt = y(t)$, and $zt = z(t)$ over the interval $tmin < t < tmax$.

`fplot3(___,LineSpec)` uses `LineSpec` to set the line style, marker symbol, and line color.

`fplot3(___,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. `Name,Value` pair settings apply to all the lines plotted. To set options for individual lines, use the objects returned by `fplot3`.

`fplot3(ax,___)` plots into the axes object `ax` instead of the current axes `gca`.

`fp = fplot3(___)` returns a parameterized function line object. Use the object to query and modify properties of a specific parameterized line. For details, see Parameterized Function Line.

Examples

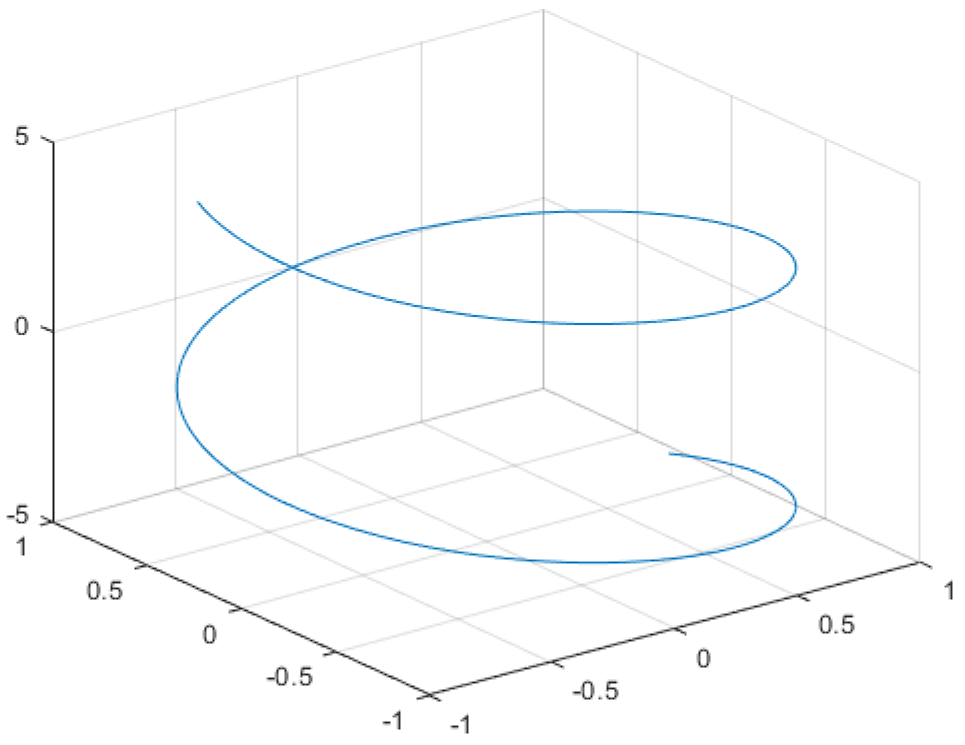
Plot 3-D Parametric Line

Plot the 3-D parametric line

$$x = \sin(t)$$
$$y = \cos(t)$$
$$z = t$$

over the default parameter range `[-5 5]`.

```
syms t
xt = sin(t);
yt = cos(t);
zt = t;
fplot3(xt,yt,zt)
```



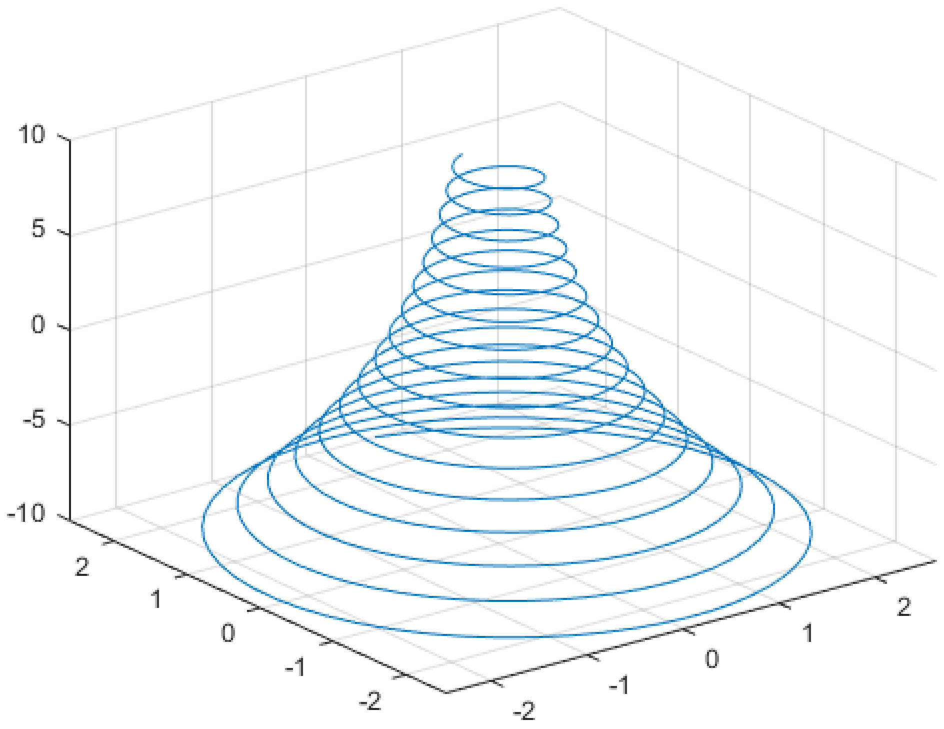
Specify Parameter Range

Plot the parametric line

$$x = e^{-t/10}\sin(5t)$$
$$y = e^{-t/10}\cos(5t)$$
$$z = t$$

over the parameter range `[-10 10]` by specifying the fourth argument of `fplot3`.

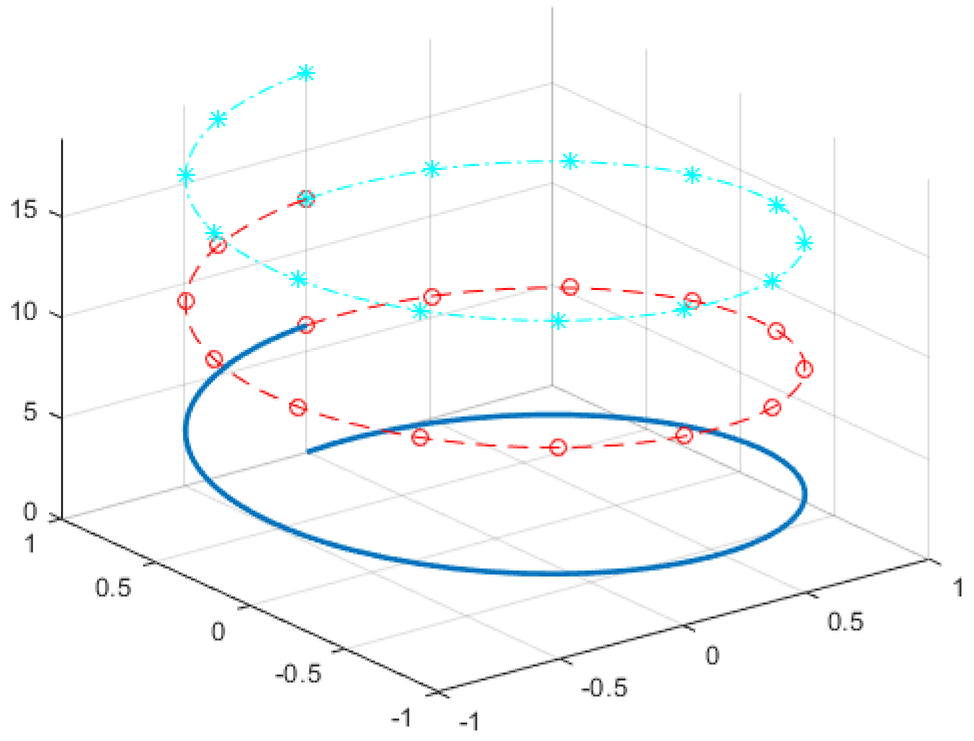
```
syms t
xt = exp(-t/10).*sin(5*t);
yt = exp(-t/10).*cos(5*t);
zt = t;
fplot3(xt,yt,zt,[-10 10])
```



Change Line Properties and Display Markers

Plot the same 3-D parametric curve three times over different intervals of the parameter. For the first curve, use a linewidth of 2. For the second, specify a dashed red line style with circle markers. For the third, specify a cyan, dash-dot line style with asterisk markers.

```
syms t
fplot3(sin(t), cos(t), t, [0 2*pi], 'LineWidth', 2)
hold on
fplot3(sin(t), cos(t), t, [2*pi 4*pi], '--or')
fplot3(sin(t), cos(t), t, [4*pi 6*pi], '-.*c')
```

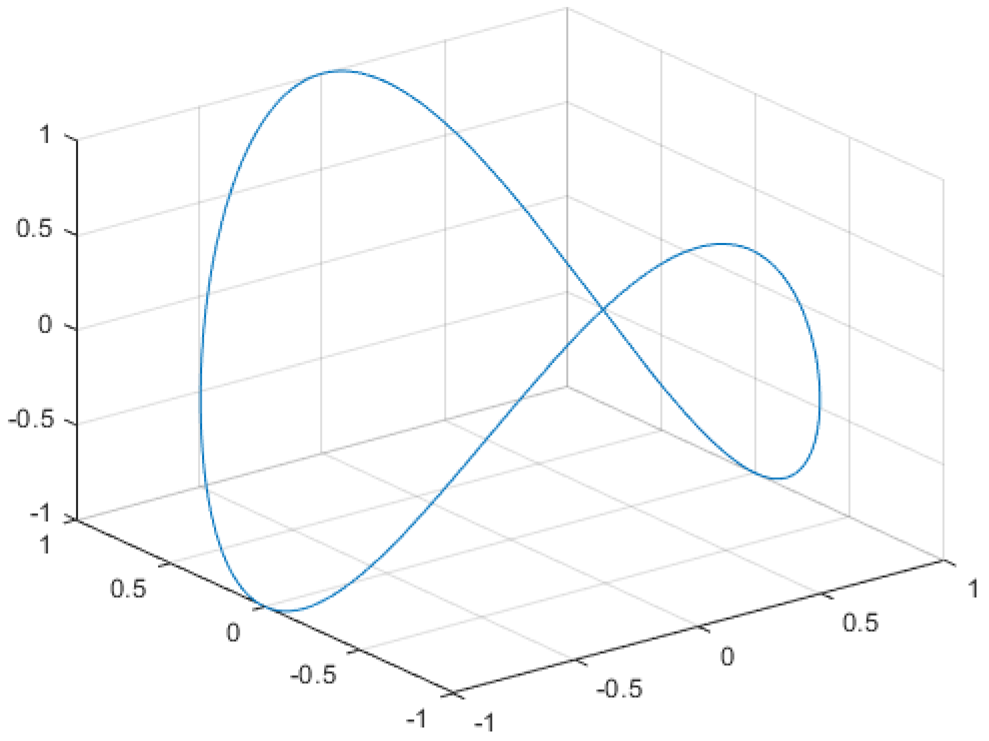


Plot 3-D Parametric Line Using Symbolic Functions

Plot the 3-D parametric line

$$x(t) = \sin(t)$$
$$y(t) = \cos(t)$$
$$z(t) = \cos(2t).$$

```
syms x(t) y(t) z(t)
x(t) = sin(t);
y(t) = cos(t);
z(t) = cos(2*t);
fplot3(x,y,z)
```

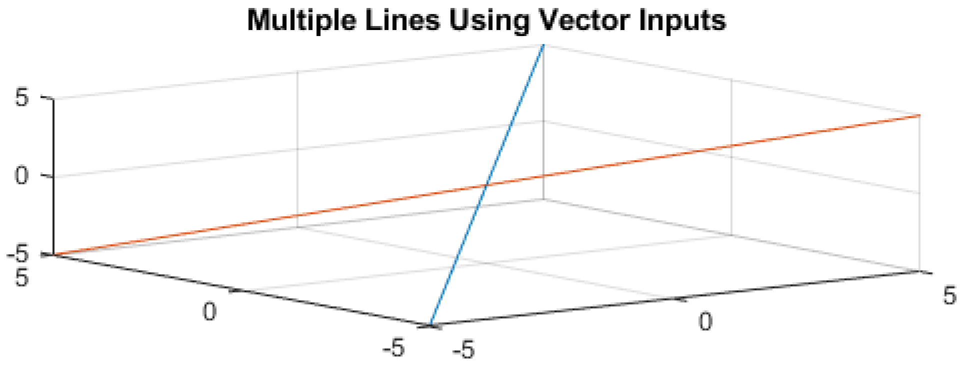


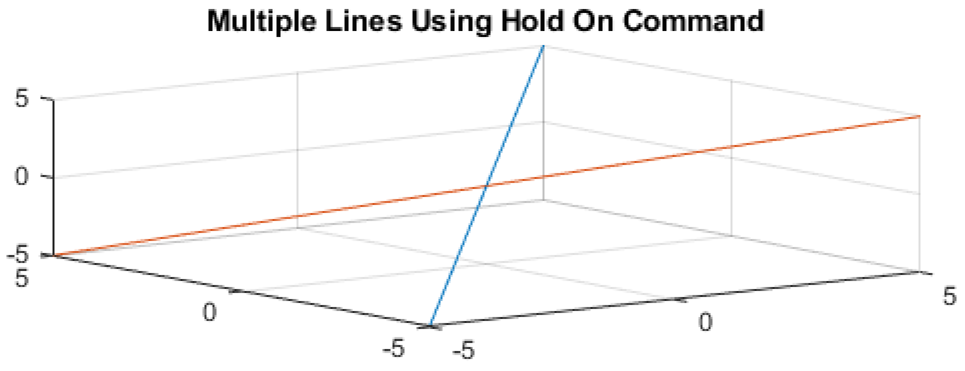
Plot Multiple Lines on Same Figure

Plot multiple lines either by passing the inputs as a vector or by using `hold on` to successively plot on the same figure. If you specify `LineSpec` and Name-Value arguments, they apply to all lines. To set options for individual lines, use the function handles returned by `fplot3`.

Divide a figure into two subplots using `subplot`. On the first subplot, plot two parameterized lines using vector input. On the second subplot, plot the same lines using `hold on`.

```
syms t
subplot(2,1,1)
fplot3([t -t], t, [t -t])
title('Multiple Lines Using Vector Inputs')

subplot(2,1,2)
fplot3(t, t, t)
hold on
fplot3(-t, t, -t)
title('Multiple Lines Using Hold On Command')

hold off
```





Modify 3-D Parametric Line After Creation

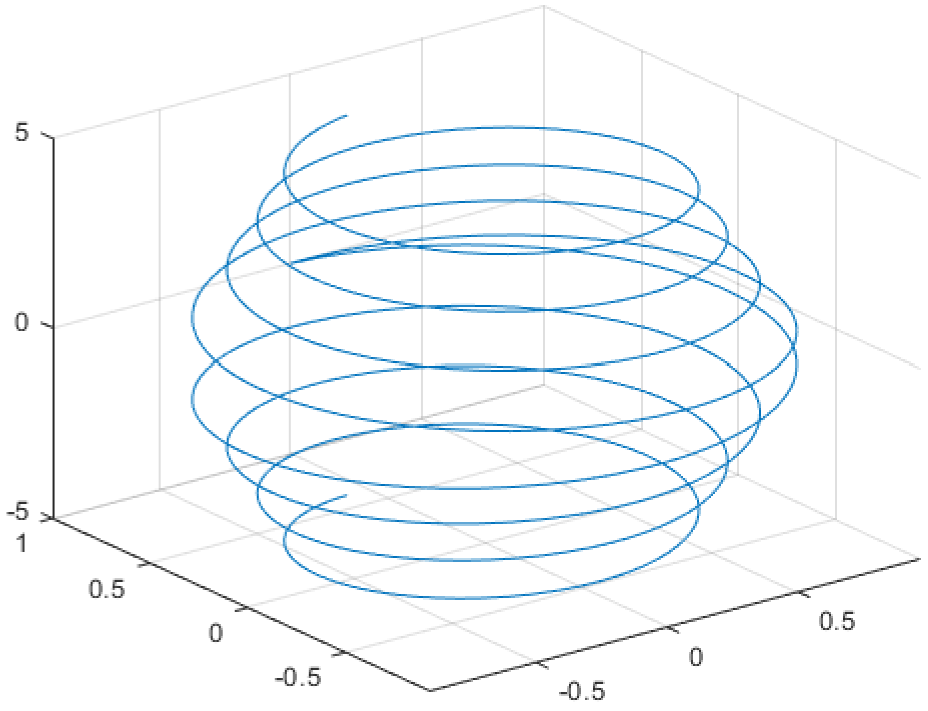
Plot the parametric line

$$x = e^{-|t|/10}\sin(5|t|)$$
$$y = e^{-|t|/10}\cos(5|t|)$$
$$z = t.$$

Provide an output to make `fplot` return the plot object.

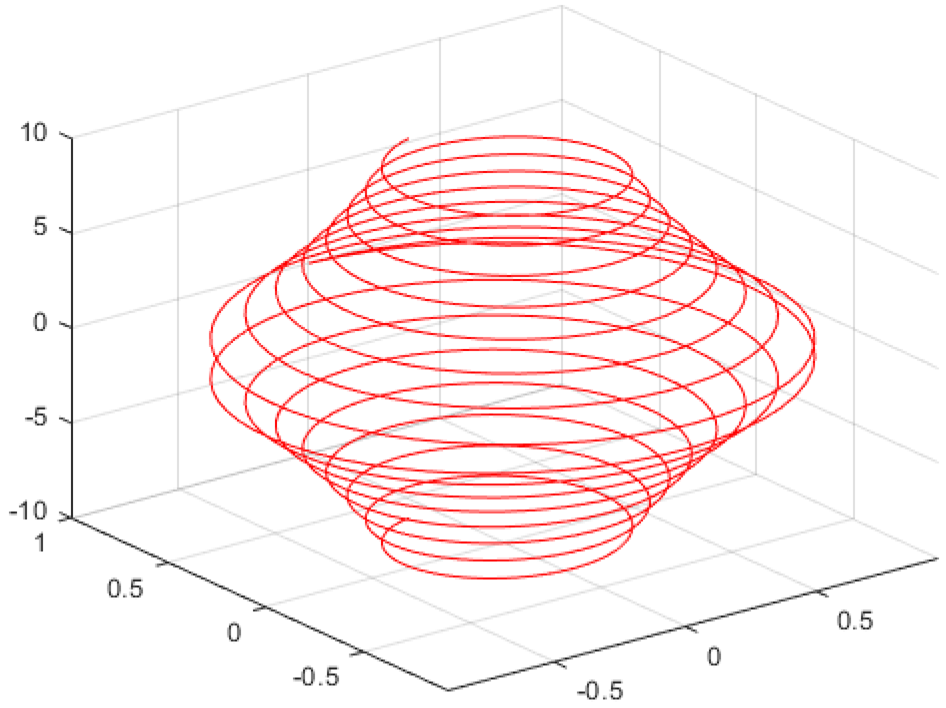
```
syms t
xt = exp(-abs(t)/10).*sin(5*abs(t));
yt = exp(-abs(t)/10).*cos(5*abs(t));
zt = t;
fp = fplot3(xt,yt,zt)
```

```
fp = 
  ParameterizedFunctionLine with properties:

    XFunction: [1x1 sym]
    YFunction: [1x1 sym]
    ZFunction: [1x1 sym]
        Color: [0 0.4470 0.7410]
    LineStyle: '-'
    LineWidth: 0.5000

  Show all properties
```



Change the range of parameter values to `[-10 10]` and the line color to red by using the `TRange` and `Color` properties of `fp` respectively.

```
fp.TRange = [-10 10];
fp.Color = 'r';
```



Add Title and Axis Labels and Format Ticks

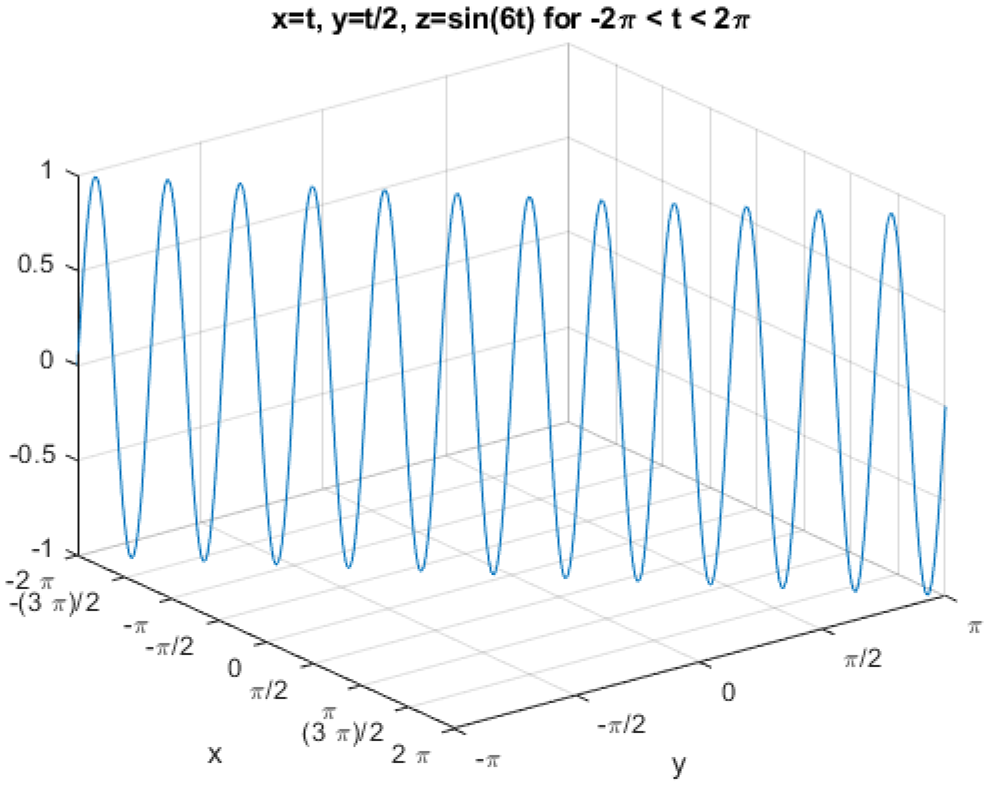
For t values in the range -2π to 2π, plot the parametric line

$$x = t$$
$$y = t/2$$
$$z = \sin(6t).$$

Add a title and axis labels. Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property. Repeat these steps for the y-axis.

To use LaTeX in plots, see `latex`.

```
syms t
xt = t;
yt = t/2;
zt = sin(6*t);
fplot3(xt,yt,zt,[-2*pi 2*pi],'MeshDensity',30)
view(52.5,30)
xlabel('x')
ylabel('y')
title('x=t, y=t/2, z=sin(6t) for -2\pi < t < 2\pi')
ax = gca;

S = sym(ax.XLim(1):pi/2:ax.XLim(2));
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);

S = sym(ax.YLim(1):pi/2:ax.YLim(2));
ax.YTick = double(S);
ax.YTickLabel = arrayfun(@texlabel, S, 'UniformOutput', false);
```



Create Animations

Create animations by changing the displayed expression using the `XFunction`, `YFunction`, and `ZFunction` properties and then by using `drawnow` to update the plot. To export to GIF, see `imwrite`.

By varying the variable i from 0 to 4π, animate the parametric curve

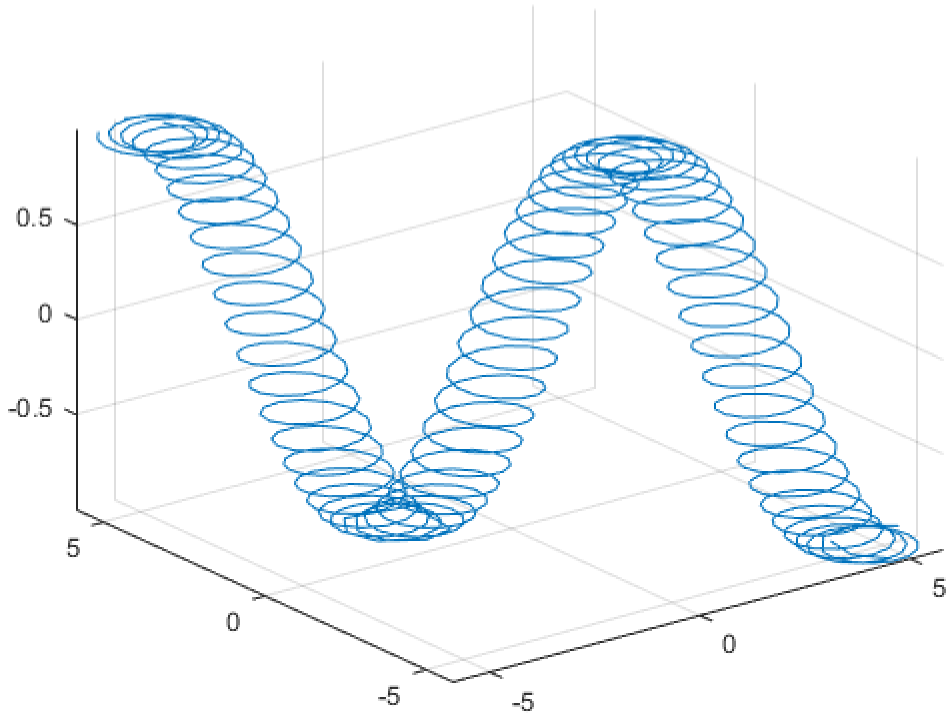
$x = t + \sin(40t)$

$y = -t + \cos(40t)$

$z = \sin(t + i).$

To play the animation, click the image.

```
syms t
fp = fplot3(t+sin(40*t),-t+cos(40*t), sin(t));
for i=0:pi/10:4*pi
    fp.ZFunction = sin(t+i);
drawnow
end
```



Input Arguments

xt — Parametric input for x-axis
symbolic expression | symbolic function

Parametric input for x-axis, specified as a symbolic expression or function. `fplot3` uses `symvar` to find the parameter.

`yt` — Parametric input for y-axis

symbolic expression | symbolic function

Parametric input for y-axis, specified as a symbolic expression or function. `fplot3` uses `symvar` to find the parameter.

`zt` — Parametric input for z-axis

symbolic expression | symbolic function

Parametric input for z-axis, specified as a symbolic expression or function. `fplot3` uses `symvar` to find the parameter.

`[tmin tmax]` — Range of values of parameter

[–5 5] (default) | vector of two numbers

Range of values of parameter, specified as a vector of two numbers. The default range is `[-5 5]`.

`ax` — Axes object

axes object

Axes object. If you do not specify an axes object, then `fplot3` uses the current axes.

`LineSpec` — Line specification

character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram
Color Specifier	**Description**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

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`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties listed here are only a subset. For a complete list, see Parameterized Function Line.

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `fplot3` uses adaptive evaluation, the actual number of evaluation points is greater.

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet or one of the color options listed in the table.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	………………
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`fp` — One or more parameterized function line objects
scalar | vector

One or more parameterized line objects, returned as a scalar or a vector. You can use these objects to query and modify properties of a specific parameterized line. For details, see Parameterized Function Line.

See Also

Functions

`fcontour` | `fimplicit` | `fimplicit3` | `fmesh` | `fplot` | `fsurf`

Properties

Parameterized Function Line

Topics

"Create Plots" on page 2-240

Introduced in R2016a

frac

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

`ceil` | `fix` | `floor` | `round`

Introduced before R2006a

fresnelc

Fresnel cosine integral function

Syntax

```
fresnelc(z)
```

Description

`fresnelc(z)` returns the Fresnel cosine integral on page 4-714 of `z`.

Examples

Fresnel Cosine Integral Function for Numeric and Symbolic Input Arguments

Find the Fresnel cosine integral function for these numbers. Since these are not symbolic objects, you receive floating-point results.

```
fresnelc([-2 0.001 1.22+0.31i])
```

```
ans =
-0.4883 + 0.0000i   0.0010 + 0.0000i   0.8617 - 0.2524i
```

Find the Fresnel cosine integral function symbolically by converting the numbers to symbolic objects:

```
y = fresnelc(sym([-2 0.001 1.22+0.31i]))
```

```
y =
[ -fresnelc(2), fresnelc(1/1000), fresnelc(61/50 + 31i/100)]
```

Use `vpa` to approximate results:

```
vpa(y)
```

```
ans =
[ -0.48825340607534075450022350335726, 0.00099999999999997532598899727942 2003,...
 0.86166573430841730950055370401908 - 0.25236540291386150167658349493972i]
```

Fresnel Cosine Integral Function for Special Values

Find the Fresnel cosine integral function for special values:

```
fresnelc([0 Inf -Inf i*Inf -i*Inf])

ans =
0.0000 + 0.0000i   0.5000 + 0.0000i  -0.5000 + 0.0000i...
   0.0000 + 0.5000i   0.0000 - 0.5000i
```

Fresnel Cosine Integral for Symbolic Functions

Find the Fresnel cosine integral for the function `exp(x) + 2*x`:

```
syms f(x)
f = exp(x)+2*x;
fresnelc(f)

ans =
fresnelc(2*x + exp(x))
```

Fresnel Cosine Integral for Symbolic Vectors and Arrays

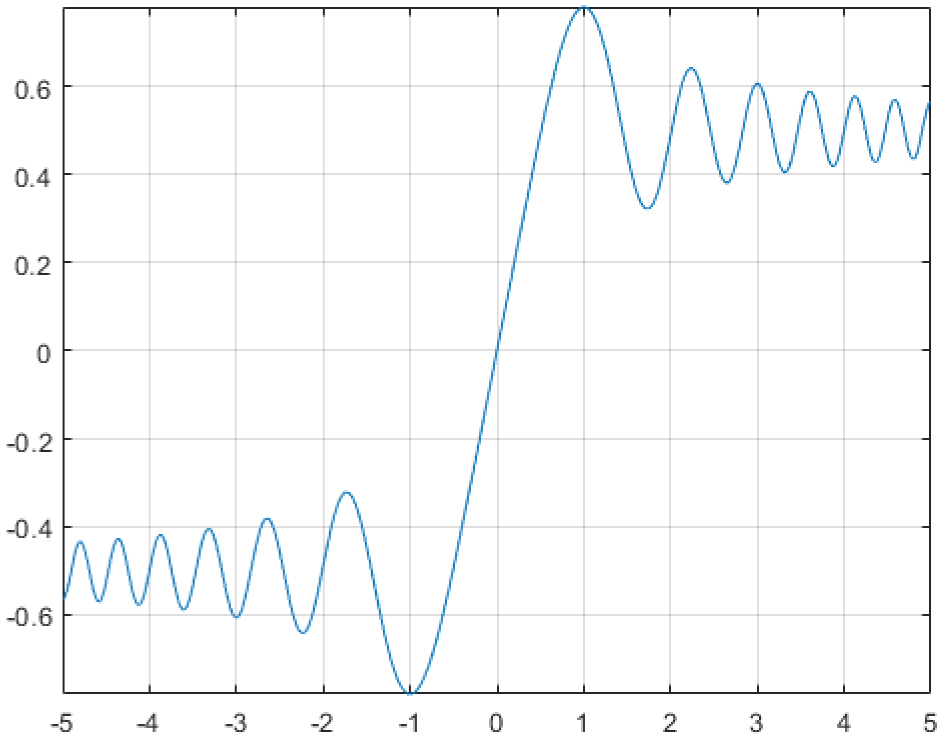
Find the Fresnel cosine integral for elements of vector `V` and matrix `M`:

```
syms x
V = [sin(x) 2i -7];
M = [0 2; i exp(x)];
fresnelc(V)
fresnelc(M)

ans =
[ fresnelc(sin(x)), fresnelc(2i), -fresnelc(7)]
ans =
[            0,      fresnelc(2)]
[ fresnelc(1i), fresnelc(exp(x))]
```

Plot Fresnel Cosine Integral Function

Plot the Fresnel cosine integral function from `x = -5` to `x = 5`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(fresnelc(x),[-5,5])
grid on
```



Differentiate and Find Limits of Fresnel Cosine Integral

The functions `diff` and `limit` handle expressions containing `fresnelc`.

Find the third derivative of the Fresnel cosine integral function:

```
syms x
diff(fresnelc(x),x,3)

ans =
- pi*sin((pi*x^2)/2) - x^2*pi^2*cos((pi*x^2)/2)
```

Find the limit of the Fresnel cosine integral function as *x* tends to infinity:

```
syms x
limit(fresnelc(x),Inf)

ans =
1/2
```

Taylor Series Expansion of Fresnel Cosine Integral

Use `taylor` to expand the Fresnel cosine integral in terms of the Taylor series:

```
syms x
taylor(fresnelc(x))

ans =
x - (x^5*pi^2)/40
```

Simplify Expressions Containing fresnelc

Use `simplify` to simplify expressions:

```
syms x
simplify(3*fresnelc(x)+2*fresnelc(-x))

ans =
fresnelc(x)
```

Input Arguments

z — Upper limit on Fresnel cosine integral
numeric value | vector | matrix | multidimensional array | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic function

Upper limit on the Fresnel cosine integral, specified as a numeric value, vector, matrix, or as a multidimensional array, or a symbolic variable, expression, vector, matrix, or function.

Definitions

Fresnel Cosine Integral

The Fresnel cosine integral of z is

$$\mathrm{fresnelc}(z) = \int_0^z \cos\left(\frac{\pi t^2}{2}\right) dt.$$

Algorithms

`fresnelc` is analytic throughout the complex plane. It satisfies fresnelc(-z) = -fresnelc(z), conj(fresnelc(z)) = fresnelc(conj(z)), and fresnelc(i*z) = i*fresnelc(z) for all complex values of z.

`fresnelc` returns special values for $z = 0$, $z = \pm\infty$, and $z = \pm i\infty$ which are 0, ±5, and ±0.5i. `fresnelc(z)` returns symbolic function calls for all other symbolic values of `z`.

See Also

`erf` | `fresnels`

Introduced in R2014a

fresnels

Fresnel sine integral function

Syntax

```
fresnels(z)
```

Description

`fresnels(z)` returns the Fresnel sine integral on page 4-719 of `z`.

Examples

Fresnel Sine Integral Function for Numeric and Symbolic Arguments

Find the Fresnel sine integral function for these numbers. Since these are not symbolic objects, you receive floating-point results.

```
fresnels([-2 0.001 1.22+0.31i])
```

```
ans =
-0.3434 + 0.0000i   0.0000 + 0.0000i   0.7697 + 0.2945i
```

Find the Fresnel sine integral function symbolically by converting the numbers to symbolic objects:

```
y = fresnels(sym([-2 0.001 1.22+0.31i]))
```

```
y =
[ -fresnels(2), fresnels(1/1000), fresnels(61/50 + 31i/100)]
```

Use `vpa` to approximate the results:

```
vpa(y)
```

```
ans =
[ -0.34341567836369824219530081595807, 0.00000000052359877559820659249174920261227,...
 0.76969209233306959998384249252902 + 0.29449530344285433030167256417637i]
```

Fresnel Sine Integral for Special Values

Find the Fresnel sine integral function for special values:

```
fresnels([0 Inf -Inf i*Inf -i*Inf])
```

```
ans =
0.0000 + 0.0000i   0.5000 + 0.0000i  -0.5000 + 0.0000i   0.0000 - 0.5000i...
   0.0000 + 0.5000i
```

Fresnel Sine Integral for Symbolic Functions

Find the Fresnel sine integral for the function `exp(x) + 2*x`:

```
syms x
f = symfun(exp(x)+2*x,x);
fresnels(f)
```

```
ans(x) =
fresnels(2*x + exp(x))
```

Fresnel Sine Integral for Symbolic Vectors and Arrays

Find the Fresnel sine integral for elements of vector `V` and matrix `M`:

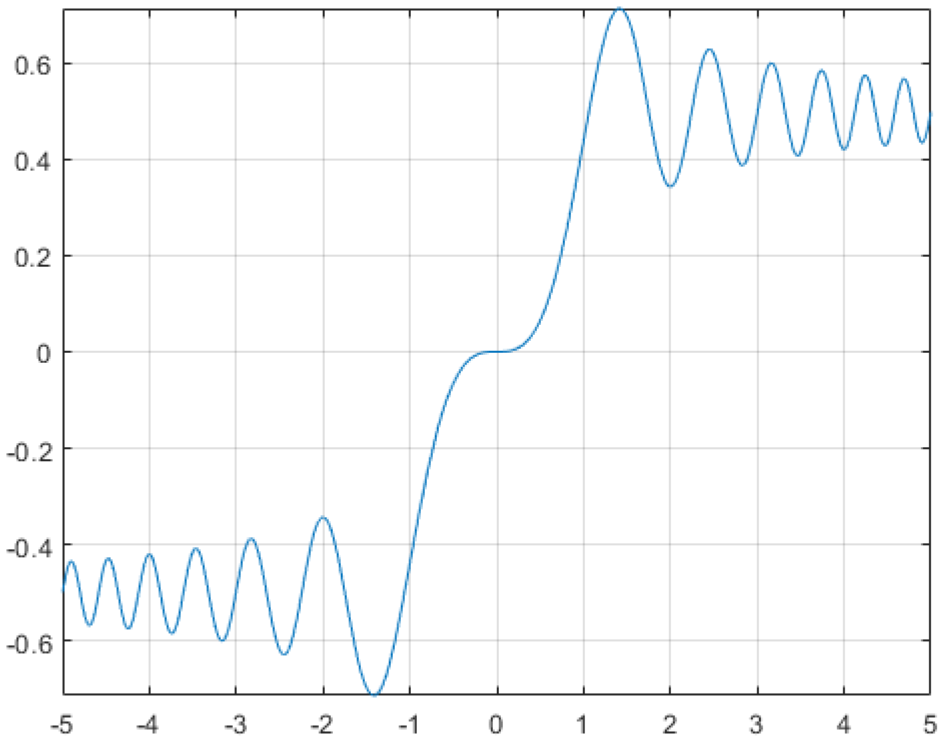
```
syms x
V = [sin(x) 2i -7];
M = [0 2; i exp(x)];
fresnels(V)
fresnels(M)
```

```
ans =
[ fresnels(sin(x)), fresnels(2i), -fresnels(7)]
ans =
[           0,      fresnels(2)]
[ fresnels(1i), fresnels(exp(x))]
```

Plot Fresnel Sine Integral Function

Plot the Fresnel sine integral function from `x = -5` to `x = 5`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(fresnels(x),[-5,5])
grid on
```



Differentiate and Find Limits of Fresnel Sine Integral

The functions `diff` and `limit` handle expressions containing `fresnels`.

Find the fourth derivative of the Fresnel sine integral function:

```
syms x
diff(fresnels(x),x,4)

ans =
- 3*x*pi^2*sin((pi*x^2)/2) - x^3*pi^3*cos((pi*x^2)/2)
```

Find the limit of the Fresnel sine integral function as *x* tends to infinity:

```
syms x
limit(fresnels(x),Inf)

ans =
1/2
```

Taylor Series Expansion of Fresnel Sine Integral

Use `taylor` to expand the Fresnel sine integral in terms of the Taylor series:

```
syms x
taylor(fresnels(x))

ans =
(pi*x^3)/6
```

Simplify Expressions Containing fresnels

Use `simplify` to simplify expressions:

```
syms x
simplify(3*fresnels(x)+2*fresnels(-x))

ans =
fresnels(x)
```

Input Arguments

z — Upper limit on the Fresnel sine integral
numeric value | vector | matrix | multidimensional array | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic function

Upper limit on the Fresnel sine integral, specified as a numeric value, vector, matrix, or a multidimensional array or as a symbolic variable, expression, vector, matrix, or function.

Definitions

Fresnel Sine Integral

The Fresnel sine integral of z is

$$\mathrm{fresnels}(z) = \int_0^z \sin\left(\frac{\pi t^2}{2}\right) dt$$

.

Algorithms

The `fresnels(z)` function is analytic throughout the complex plane. It satisfies fresnels(-z) = -fresnels(z), conj(fresnels(z)) = fresnels(conj(z)), and fresnels(i*z) = -i*fresnels(z) for all complex values of `z`.

`fresnels(z)` returns special values for z = 0, z = ±∞, and z = ±i∞ which are 0, ±5, and ∓0.5i. `fresnels(z)` returns symbolic function calls for all other symbolic values of `z`.

See Also

`erf` | `fresnelc`

Introduced in R2014a

fsurf

Plot 3-D surface

Syntax

```
fsurf(f)
fsurf(f,[min max])
fsurf(f,[xmin xmax ymin ymax])

fsurf(funx,funy,funz)
fsurf(funx,funy,funz,[uvmin uvmax])
fsurf(funx,funy,funz,[umin umax vmin vmax])

fsurf(___,LineSpec)
fsurf(___,Name,Value)
fsurf(ax,___)
fs = fsurf(___)
```

Description

`fsurf(f)` creates a surface plot of the symbolic expression `f(x,y)` over the default interval `[-5 5]` for `x` and `y`.

`fsurf(f,[min max])` plots `f(x,y)` over the interval `[min max]` for `x` and `y`.

`fsurf(f,[xmin xmax ymin ymax])` plots `f(x,y)` over the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

`fsurf(funx,funy,funz)` plots the parametric surface `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[-5 5]` for `u` and `v`.

`fsurf(funx,funy,funz,[uvmin uvmax])` plots the parametric surface `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[uvmin uvmax]` for `u` and `v`.

`fsurf(funx,funy,funz,[umin umax vmin vmax])` plots the parametric surface `x = x(u,v)`, `y = y(u,v)`, `z = z(u,v)` over the interval `[umin umax]` for `u` and `[vmin vmax]` for `v`.

`fsurf(___,LineSpec)` uses `LineSpec` to set the line style, marker symbol, and face color.

`fsurf(___,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

`fsurf(ax,___)` plots into the axes with the object `ax` instead of the current axes object `gca`.

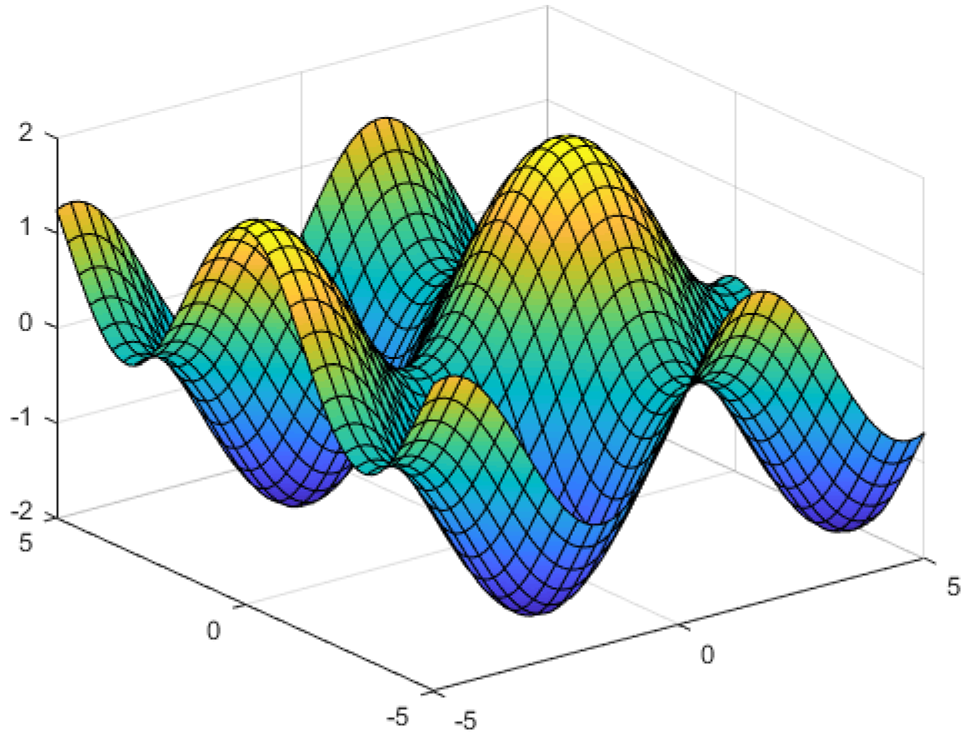
`fs = fsurf(___)` returns a function surface object or parameterized function surface object, depending on the type of surface. Use the object to query and modify properties of a specific surface. For details, see Function Surface and Parameterized Function Surface.

Examples

3-D Surface Plot of Symbolic Expression

Plot the input $\sin(x) + \cos(y)$ over the default range $-5 < x < 5$ and $-5 < y < 5$.

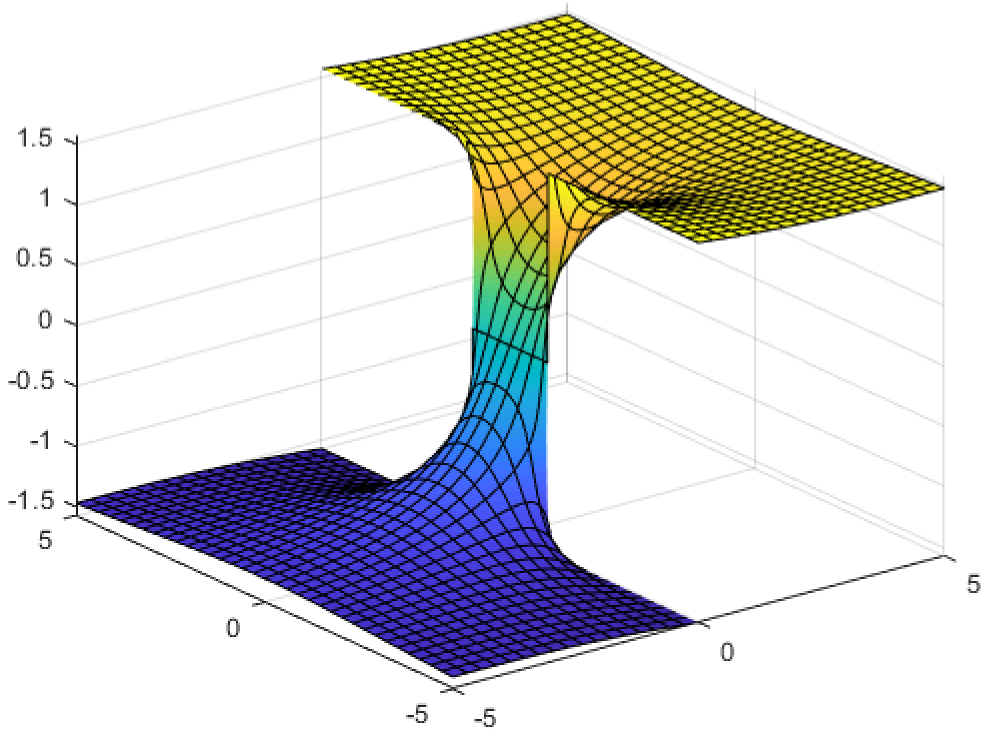
```
syms x y
fsurf(sin(x)+cos(y))
```



3-D Surface Plot of Symbolic Function

Plot the real part of $\tan^{-1}(x+iy)$ over the default range $-5 < x < 5$ and $-5 < y < 5$.

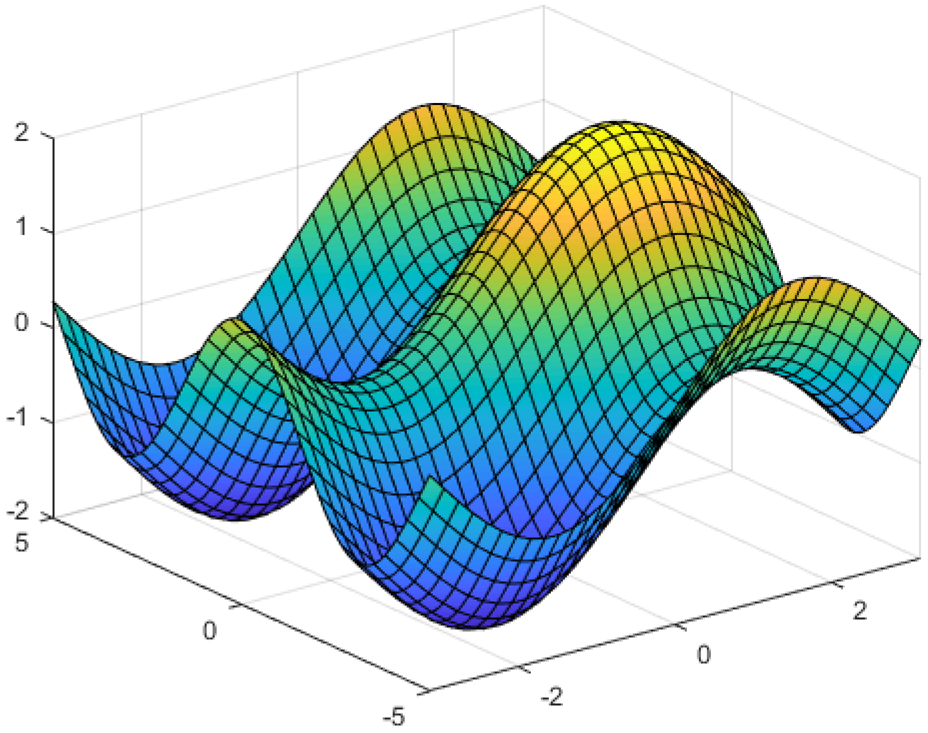
```
syms f(x,y)
f(x,y) = real(atan(x + i*y));
fsurf(f)
```



Specify Plotting Interval of Surface Plot

Plot $\sin(x) + \cos(y)$ over $-\pi < x < \pi$ and $-5 < y < 5$ by specifying the plotting interval as the second argument of `fsurf`.

```
syms x y
f = sin(x) + cos(y);
fsurf(f, [-pi pi -5 5])
```



Parameterized Surface Plot

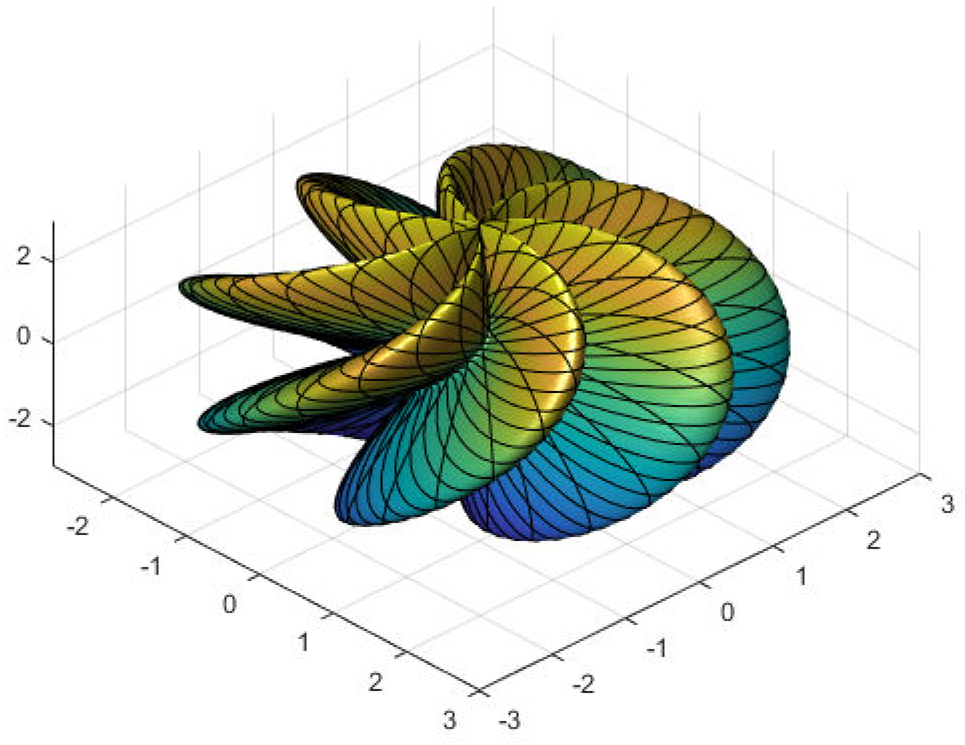
Plot the parameterized surface

$$\begin{aligned} x &= r\cos(s)\sin(t) \\ y &= r\sin(s)\sin(t) \\ z &= r\cos(t) \\ \text{where}\quad r &= 2+\sin(7s+5t) \end{aligned}$$

for $0 < s < 2\pi$ and $0 < t < \pi$.

Improve the plot's appearance by using `camlight`.

```
syms s t
r = 2 + sin(7*s + 5*t);
x = r*cos(s)*sin(t);
y = r*sin(s)*sin(t);
z = r*cos(t);
fsurf(x, y, z, [0 2*pi 0 pi])
camlight
view(46,52)
```



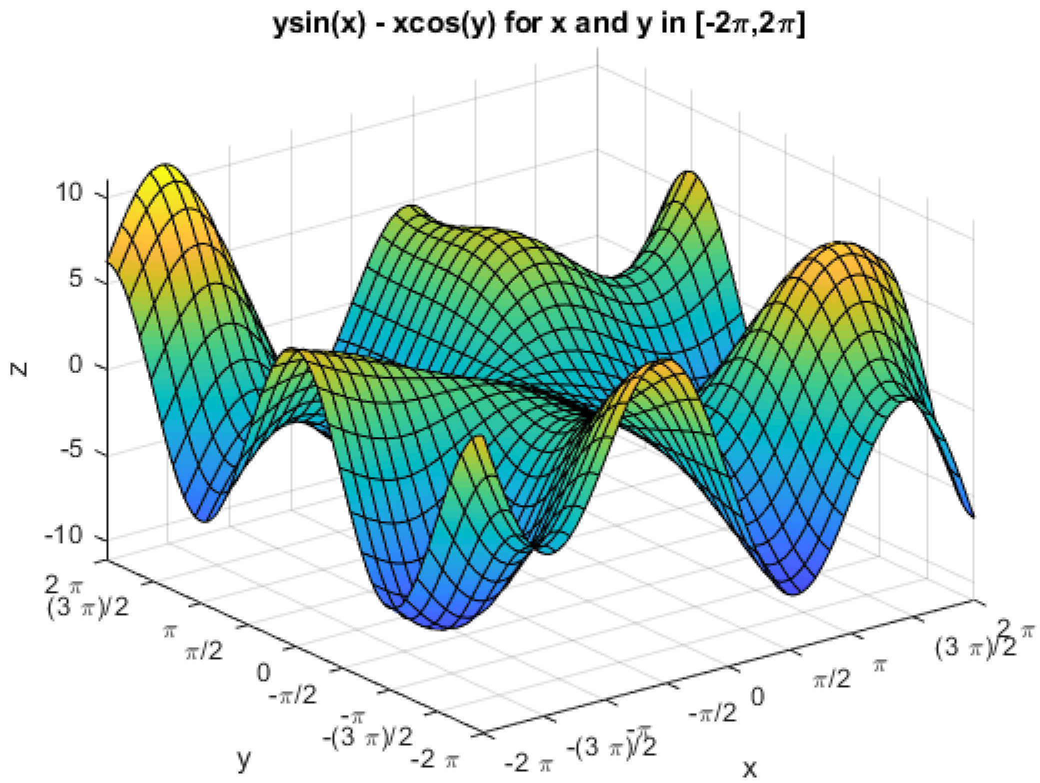
Add Title and Axis Labels and Format Ticks

For x and y from -2π to 2π, plot the 3-D surface $y\sin(x) - x\cos(y)$. Add a title and axis labels.

Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Convert the axis limits to precise multiples of `pi/2` by using `round` and get the symbolic tick values in `S`. Display these ticks by using the `XTick` property. Create x-axis labels by using `arrayfun` to apply `texlabel` to `S`. Display these labels by using the `XTickLabel` property. Repeat these steps for the y-axis.

To use LaTeX in plots, see `latex`.

```
syms x y
fsurf(y.*sin(x)-x.*cos(y), [-2*pi 2*pi])
title('ysin(x) - xcos(y) for x and y in [-2\pi,2\pi]')
xlabel('x')
ylabel('y')
zlabel('z')

ax = gca;
S = sym(ax.XLim(1):pi/2:ax.XLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
ax.XTick = double(S);
ax.XTickLabel = arrayfun(@texlabel,S,'UniformOutput',false);

S = sym(ax.YLim(1):pi/2:ax.YLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
ax.YTick = double(S);
ax.YTickLabel = arrayfun(@texlabel,S,'UniformOutput',false);
```

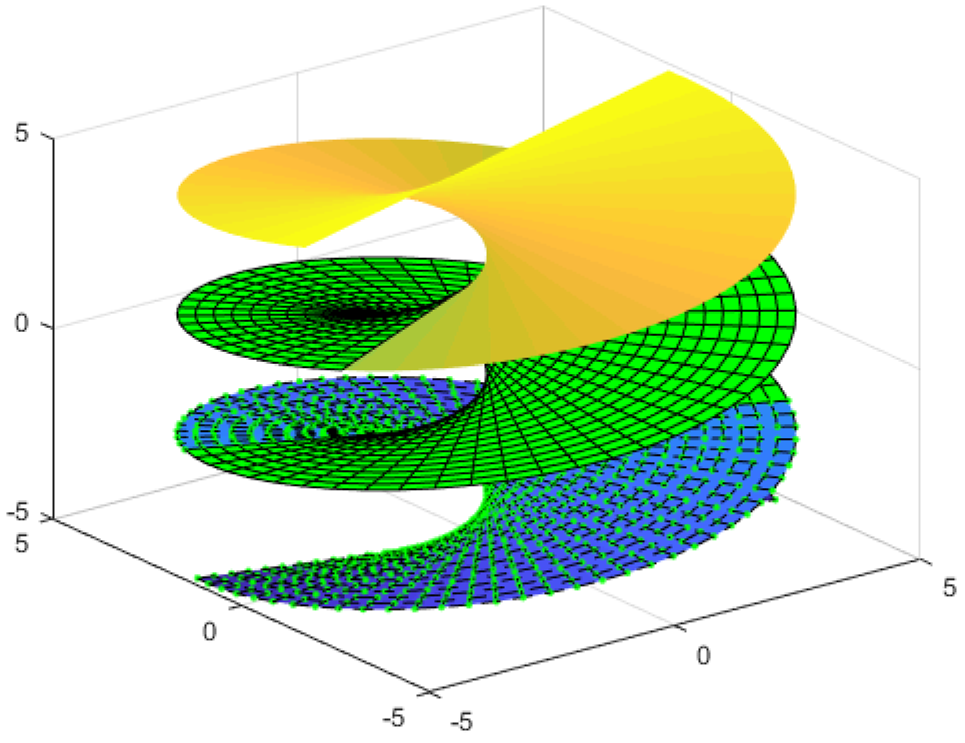


Line Style and Width for Surface Plot

Plot the parametric surface $x = s\sin(t)$, $y = -s\cos(t)$, $z = t$ with different line styles for different values of t. For $-5 < t < -2$, use a dashed line with green dot markers. For $-2 < t < 2$, use a `LineWidth` of `1` and a green face color. For $2 < t < 5$, turn off the lines by setting `EdgeColor` to `none`.

```
syms s t
fsurf(s*sin(t),-s*cos(t),t,[-5 5 -5 -2],'--.','MarkerEdgeColor','g')
hold on
```

```
fsurf(s*sin(t),-s*cos(t),t,[-5 5 -2 2],'LineWidth',1,'FaceColor','g')
fsurf(s*sin(t),-s*cos(t),t,[-5 5 2 5],'EdgeColor','none')
```



Modify Surface After Creation

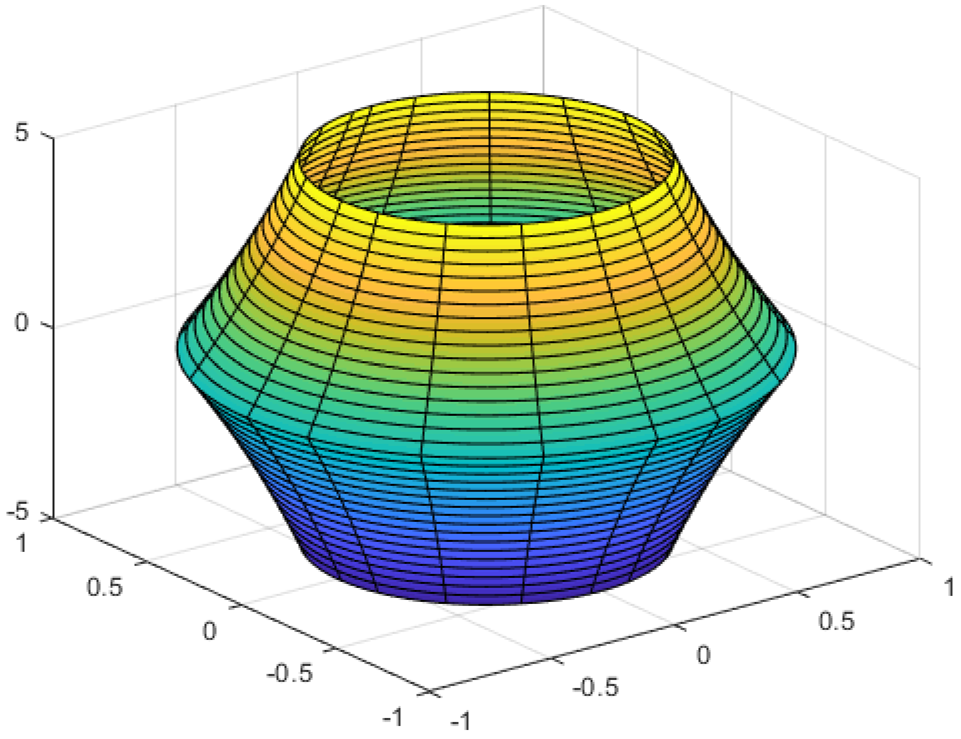
Plot the parametric surface

$$x = e^{-|u|/10}\sin(5|v|)$$
$$y = e^{-|u|/10}\cos(5|v|)$$
$$z = u.$$

Specify an output to make `fcontour` return the plot object.

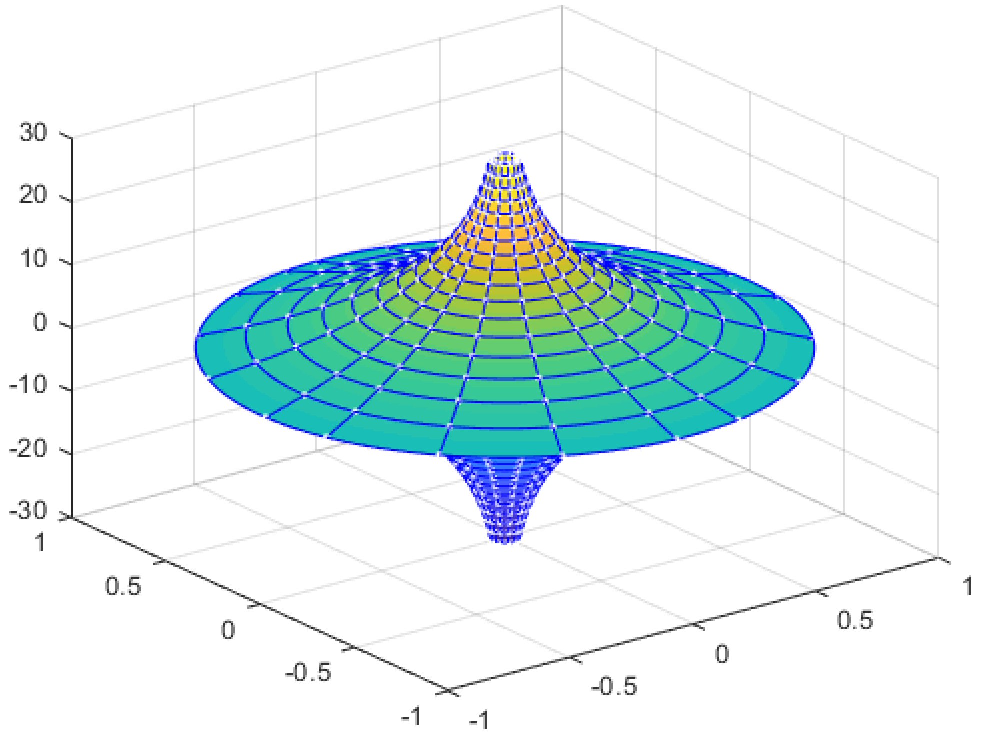
```
syms u v
x = exp(-abs(u)/10).*sin(5*abs(v));
y = exp(-abs(u)/10).*cos(5*abs(v));
z = u;
fs = fsurf(x,y,z)
```

```
fs =
  ParameterizedFunctionSurface with properties:

    XFunction: [1x1 sym]
    YFunction: [1x1 sym]
    ZFunction: [1x1 sym]
    EdgeColor: [0 0 0]
    LineStyle: '-'
    FaceColor: 'interp'

  Show all properties
```



Change the range of `u` to `[-30 30]` by using the `URange` property of `fs`. Set the line color to blue by using the `EdgeColor` property and specify white, dot markers by using the `Marker` and `MarkerEdgeColor` properties.

```
fs.URange = [-30 30];
fs.EdgeColor = 'b';
fs.Marker = '.';
fs.MarkerEdgeColor = 'w';
```

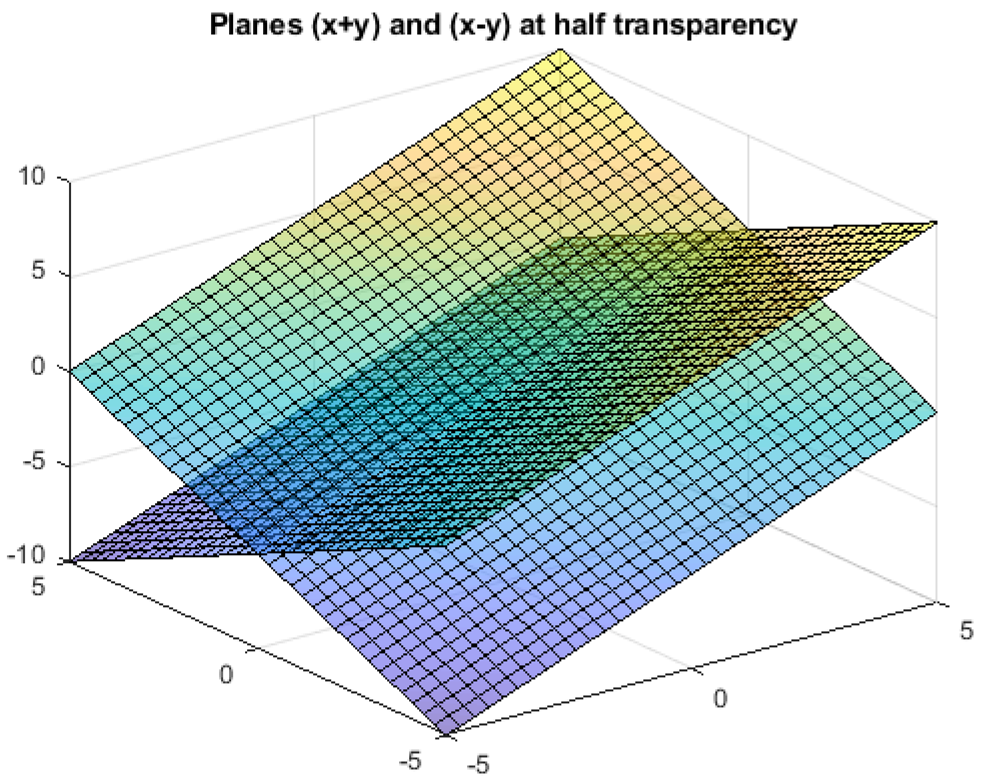


Multiple Surface Plots and Transparent Surfaces

Plot multiple surfaces using vector input to `fsurf`. Alternatively, use `hold on` to plot successively on the same figure. When displaying multiple surfaces on the same figure, transparency is useful. Adjust the transparency of surface plots by using the `FaceAlpha` property. `FaceAlpha` varies from `0` to `1`, where `0` is full transparency and `1` is no transparency.

Plot the planes $x + y$ and $x - y$ using vector input to `fsurf`. Show both planes by making them half transparent using `FaceAlpha`.

```
syms x y
h = fsurf([x+y x-y]);
h(1).FaceAlpha = 0.5;
h(2).FaceAlpha = 0.5;
title('Planes (x+y) and (x-y) at half transparency')
```

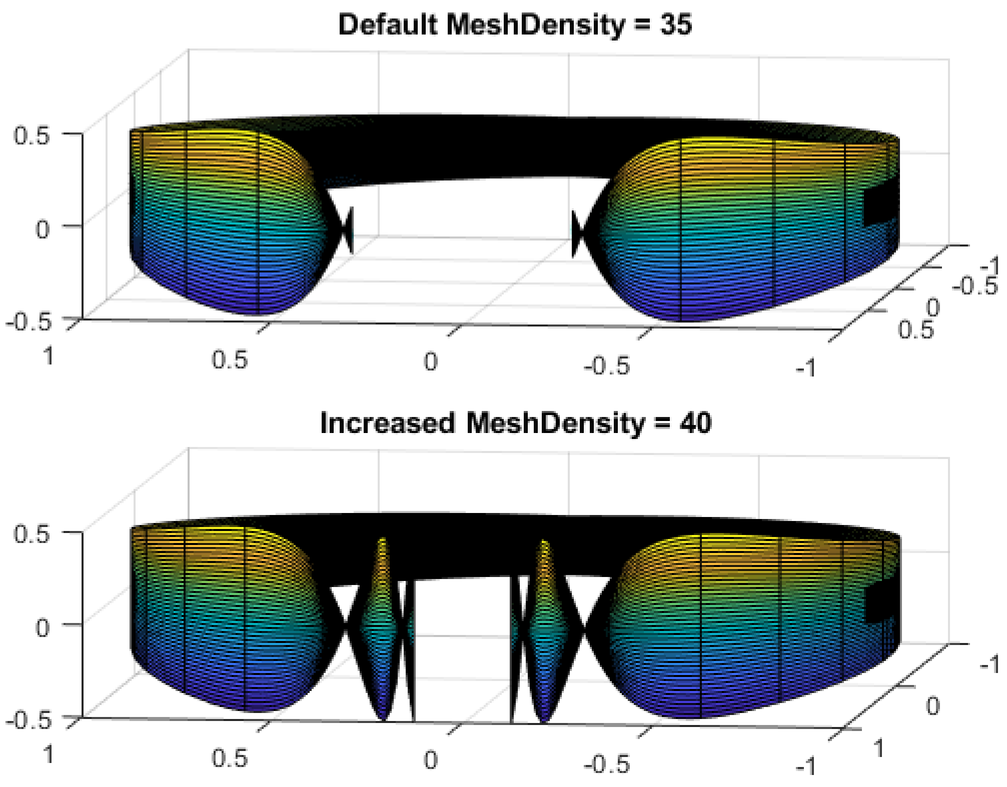


Control Resolution of Surface Plot

Control the resolution of a surface plot using the `'MeshDensity'` option. Increasing `'MeshDensity'` can make smoother, more accurate plots while decreasing it can increase plotting speed.

Divide a figure into two using `subplot`. In the first subplot, plot the parametric surface $x = \sin(s)$, $y = \cos(s)$, and $z = (t/10)\sin(1/s)$. The surface has a large gap. Fix this issue by increasing the `'MeshDensity'` to `40` in the second subplot. `fsurf` fills the gap showing that by increasing `'MeshDensity'` you increased the plot's resolution.

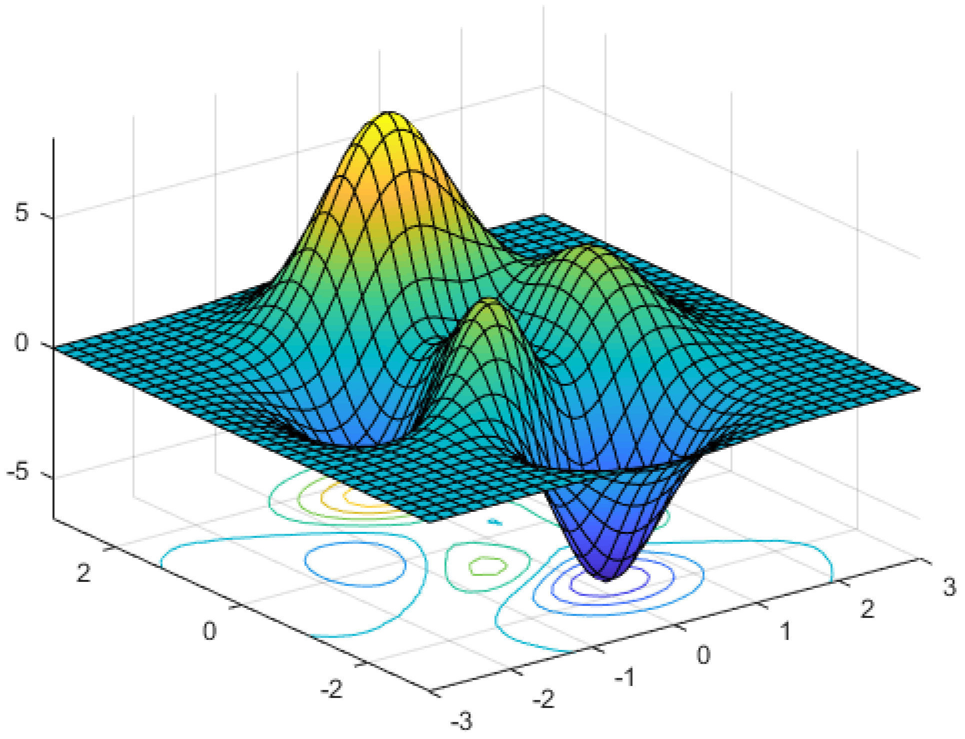
```
syms s t

subplot(2,1,1)
fsurf(sin(s), cos(s), t/10.*sin(1./s))
view(-172,25)
title('Default MeshDensity = 35')

subplot(2,1,2)
fsurf(sin(s), cos(s), t/10.*sin(1./s),'MeshDensity',40)
view(-172,25)
title('Increased MeshDensity = 40')
```



Show Contours Below Surface Plot

Show contours for the surface plot of the expression `f` by setting the `'ShowContours'` option to `'on'`.

```
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);
fsurf(f,[-3 3],'ShowContours','on')
```



Create Animations of Surface Plots

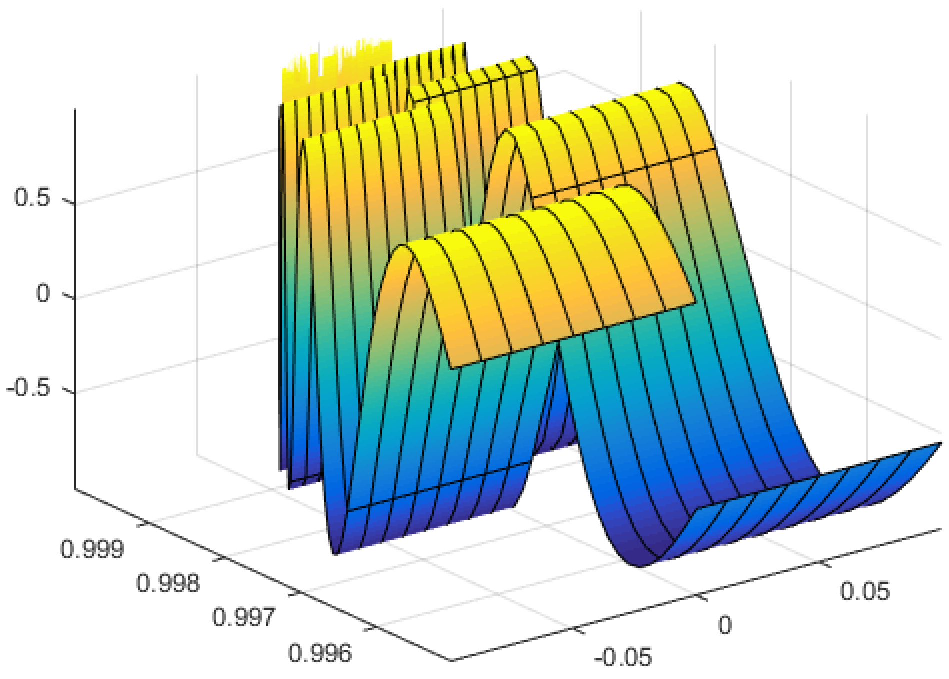
Create animations by changing the displayed expression using the `Function`, `XFunction`, `YFunction`, and `ZFunction` properties and then by using `drawnow` to update the plot. To export to GIF, see `imwrite`.

By varying the variable *i* from 1 to 3, animate the parametric surface

$$x = t\sin(s)$$
$$y = \cos(s)$$
$$z = \sin\left(\frac{i}{s}\right).$$

for -0.1 < u < 0.1 and 0 < v < 1. Increase plotting speed by reducing `MeshDensity` to `9`.

```
syms s t
h = fsurf(t.*sin(s), cos(s), sin(1./s), [-0.1 0.1 0 1]);
h.MeshDensity = 9;
for i=1:0.05:3
    h.ZFunction = sin(i./s);
    drawnow
end
```



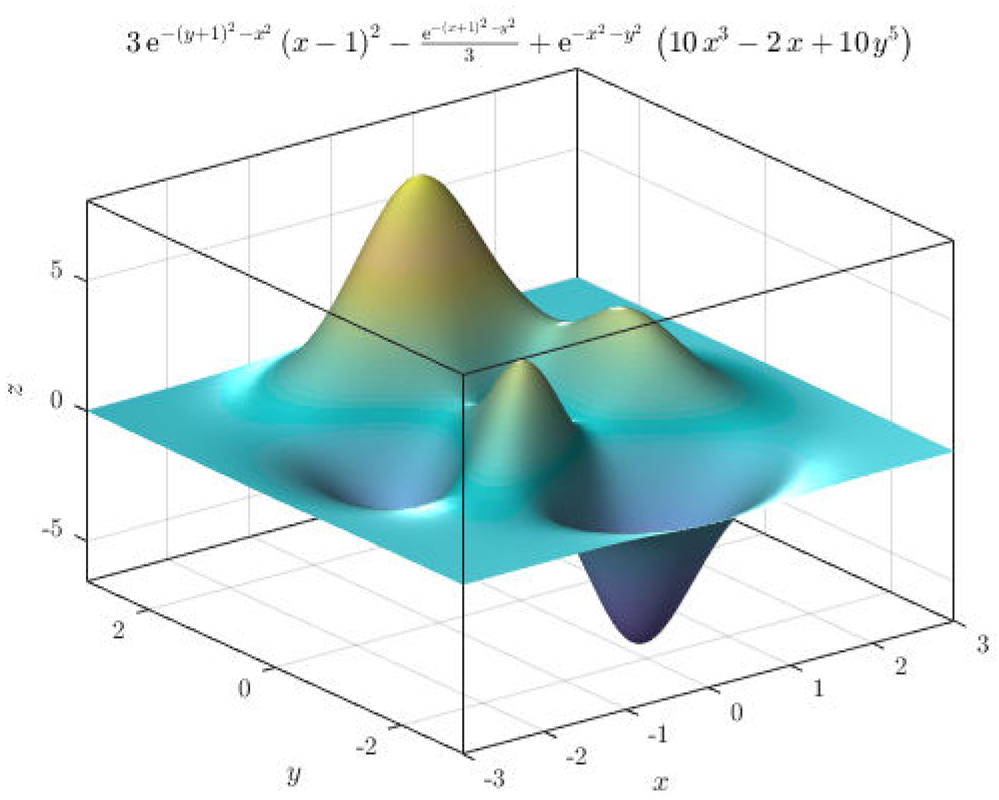
Improve Appearance of Surface Plot

Plot the expression `f`. Improve the appearance of the surface plot by using the properties of the handle returned by `fsurf`, the lighting properties, and the `colormap`.

Create a light by using `camlight`. Increase brightness by using `brighten`. Remove the lines by setting `EdgeColor` to `'none'`. Increase the ambient light using `AmbientStrength`. For details, see “Lighting Overview” (MATLAB). Turn the axes box on. For the title, convert `f` to LaTeX using `latex`. Finally, to improve the appearance of the axes ticks, axes labels, and title, set `'Interpreter'` to `'latex'`.

```
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);
h = fsurf(f,[-3 3]);

camlight(110,70)
brighten(0.6)
h.EdgeColor = 'none';
h.AmbientStrength = 0.4;

a = gca;
a.TickLabelInterpreter = 'latex';
a.Box = 'on';
a.BoxStyle = 'full';

xlabel('$x$','Interpreter','latex')
ylabel('$y$','Interpreter','latex')
zlabel('$z$','Interpreter','latex')
title_latex = ['$' latex(f) '$'];
title(title_latex,'Interpreter','latex')
```



Input Arguments

`f` — 3-D expression or function to be plotted
symbolic expression | symbolic function

Expression or function to be plotted, specified as a symbolic expression or function.

`[min max]` — Plotting interval for x- and y-axes
[–5 5] (default) | vector of two numbers

Plotting interval for x- and y-axes, specified as a vector of two numbers. The default is `[-5 5]`.

`[xmin xmax ymin ymax]` — Plotting interval for x- and y-axes

[–5 5 –5 5] (default) | vector of four numbers

Plotting interval for x- and y-axes, specified as a vector of four numbers. The default is `[-5 5 -5 5]`.

`funx,funy,funz` — Parametric functions of `u` and `v`

symbolic expressions | symbolic functions

Parametric functions of `u` and `v`, specified as a symbolic expression or function.

`[uvmin uvmax]` — Plotting interval for `u` and `v`

[–5 5] (default) | vector of two numbers

Plotting interval for `u` and `v` axes, specified as a vector of two numbers. The default is `[-5 5]`.

`[umin umax vmin vmax]` — Plotting interval for `u` and `v`

[–5 5 –5 5] (default) | vector of four numbers

Plotting interval for `u` and `v`, specified as a vector of four numbers. The default is `[-5 5 -5 5]`.

`ax` — Axes object

axes object

Axes object. If you do not specify an axes object, then `fsurf` uses the current axes.

`LineSpec` — Line style, marker symbol, and face color

character vector

Line style, marker symbol, and color, specified as a character vector. The elements of the character vector can appear in any order, and you can omit one or more options from the character vector specifier.

Example: `'--or'` is a red surface with circle markers and dashed lines

Specifier	Line Style
`–`	Solid line (default)

Specifier	Line Style
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line
Specifier	**Marker**
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram
Specifier	**Color**
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

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`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties listed here are only a subset. For a complete list, see Function Surface.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `fsurf` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` — Display contour plot under plot
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

`EdgeColor` — Line color
`[0 0 0]` (default) | `'interp'` | `'none'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, or one of the color options listed in the table. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`

Option	Description	Equivalent RGB Triplet
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	————————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	………………
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points. If the line has markers, then the line width also affects the marker edges.

Example: `0.75`

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`

Option	Description	Equivalent RGB Triplet
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

`MarkerFaceColor` — Marker fill color
`'none'` (default) | `'auto'` | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, or one of the color options listed in the table. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`MarkerSize` — Marker size

`6` (default) | positive value

Marker size, specified as a positive value in points.

Example: `10`

Output Arguments

`fs` — One or more objects

scalar | vector

One or more objects, returned as a scalar or a vector. The object is either a function surface object or parameterized surface object, depending on the type of plot. You can use these objects to query and modify properties of a specific line. For details, see Function Surface and Parameterized Function Surface.

See Also

Functions

`fcontour` | `fimplicit` | `fimplicit3` | `fmesh` | `fplot` | `fplot3`

Properties

Function Surface | Parameterized Function Surface

Topics

"Create Plots" on page 2-240

Introduced in R2016a

functionalDerivative

Functional derivative

Syntax

```
D = functionalDerivative(f,y)
```

Description

`D = functionalDerivative(f,y)` returns the “Functional Derivative” on page 4-751

of the functional $F = \int f\left(x, y(x), y'(x) ...\right) dx$ with respect to the function $y = y(x)$, where x represents one or more independent variables. If `y` is a vector of symbolic functions, `functionalDerivative` returns a vector of functional derivatives with respect to the functions in `y`, where all functions in `y` must depend on the same independent variables.

Examples

Find Functional Derivative

Find the functional derivative of the function given by $f(y) = y(x)\sin\left(y(x)\right)$ with respect to the function `y`.

```
syms y(x)
f = y*sin(y);
D = functionalDerivative(f,y)
```

```
D(x) =
sin(y(x)) + cos(y(x))*y(x)
```

Find Functional Derivative of Vector of Functionals

Find the functional derivative of the function given by $H(u,v) = u^2 \frac{dv}{dx} + v \frac{d^2u}{dx^2}$ with respect to the functions u and v.

```
syms u(x) v(x)
H = u^2*diff(v,x)+v*diff(u,x,x);
D = functionalDerivative(H,[u v])
```

```
D(x) =
 2*u(x)*diff(v(x), x) + diff(v(x), x, x)
 diff(u(x), x, x) - 2*u(x)*diff(u(x), x)
```

functionalDerivative returns a vector of symbolic functions containing the functional derivatives of H with respect to u and v, respectively.

Find Euler-Lagrange Equation for Spring

First find the Lagrangian for a spring with mass m and spring constant k, and then derive the Euler-Lagrange equation. The Lagrangian is the difference of kinetic energy T and potential energy V which are functions of the displacement x(t).

```
syms m k x(t)
T = sym(1)/2*m*diff(x,t)^2;
V = sym(1)/2*k*x^2;
L = T - V
```

```
L(t) =
(m*diff(x(t), t)^2)/2 - (k*x(t)^2)/2
```

Find the Euler-Lagrange equation by finding the functional derivative of L with respect to x, and equate it to 0.

```
eqn = functionalDerivative(L,x) == 0
```

```
eqn(t) =
- m*diff(x(t), t, t) - k*x(t) == 0
```

diff(x(t), t, t) is the acceleration. The equation eqn represents the expected differential equation that describes spring motion.

Solve `eqn` using `dsolve`. Obtain the expected form of the solution by assuming mass `m` and spring constant `k` are positive.

```
assume(m,'positive')
assume(k,'positive')
xSol = dsolve(eqn,x(0) == 0)

xSol =
C5*sin((k^(1/2)*t)/m^(1/2))
```

Clear assumptions for further calculations.

```
assume([k m],'clear')
```

Find Differential Equation for Brachistochrone Problem

The Brachistochrone problem is to find the quickest path of descent under gravity. The time for a body to move along a curve `y(x)` under gravity is given by

$$f = \sqrt{\frac{1 + y'^2}{2gy}},$$

where g is the acceleration due to gravity.

Find the quickest path by minimizing `f` with respect to the path `y`. The condition for a minimum is

$$\frac{\delta f}{\delta y} = 0.$$

Compute this condition to obtain the differential equation that describes the Brachistochrone problem. Use `simplify` to simplify the solution to its expected form.

```
syms g y(x)
assume(g,'positive')
f = sqrt((1+diff(y)^2)/(2*g*y));
eqn = functionalDerivative(f,y) == 0;
eqn = simplify(eqn)

eqn(x) =
diff(y(x), x)^2 + 2*y(x)*diff(y(x), x, x) == -1
```

This equation is the standard differential equation for the Brachistochrone problem.

Find Minimal Surface in 3-D Space

If the function $u(x,y)$ describes a surface in 3-D space, then the surface area is found by the functional

$$F(u) = \iint f\left(x,y,u,u_x,u_y\right)dxdy = \iint \sqrt{1+u_x^2+u_y^2}\,dxdy,$$

where u_x and u_y are the partial derivatives of u with respect to x and y.

Find the equation that describes the minimal surface for a 3-D surface described by the function `u(x,y)` by finding the functional derivative of `f` with respect to `u`.

```
syms u(x,y)
f = sqrt(1 + diff(u,x)^2 + diff(u,y)^2);
D = functionalDerivative(f,u)
```

```
D(x, y) =
-(diff(u(x, y), y)^2*diff(u(x, y), x, x)...
 + diff(u(x, y), x)^2*diff(u(x, y), y, y)...
 - 2*diff(u(x, y), x)*diff(u(x, y), y)*diff(u(x, y), x, y)...
 + diff(u(x, y), x, x)...
 + diff(u(x, y), y, y))/(diff(u(x, y), x)^2...
 + diff(u(x, y), y)^2 + 1)^(3/2)
```

The solutions to this equation `D` describe minimal surfaces in 3-D space such as soap bubbles.

Input Arguments

`f` — Expression to find functional derivative of

symbolic variable | symbolic function | symbolic expression

Expression to find functional derivative of, specified as a symbolic variable, function, or expression. The argument `f` represents the density of the functional.

`y` — Differentiation function

symbolic function | vector of symbolic functions | matrix of symbolic functions | multidimensional array of symbolic functions

Differentiation function, specified as a symbolic function or a vector, matrix, or multidimensional array of symbolic functions. The argument `y` can be a function of one or more independent variables. If `y` is a vector of symbolic functions, `functionalDerivative` returns a vector of functional derivatives with respect to the functions in `y`, where all functions in `y` must depend on the same independent variables.

Output Arguments

D — Functional derivative

symbolic function | vector of symbolic functions

Functional derivative, returned as a symbolic function or a vector of symbolic functions. If input `y` is a vector, then `D` is a vector.

Definitions

Functional Derivative

Consider functionals

$$F(y) = \int_{\Omega} f\left(x, y(x), y'(x), y''(x), \ldots\right) dx,$$

where Ω is a region in x-space.

For a small change in the value of y, δy, the change in the functional F is

$$\frac{\delta F}{\delta y} = \left.\frac{d}{d\varepsilon}\right|_{\varepsilon=0} F(y + \varepsilon \delta y) = \int_{\Omega} \frac{\delta f(x)}{\delta y} \delta y(x) dx + \text{boundary terms}.$$

The expression $\frac{\delta f(x)}{\delta y}$ is the functional derivative of f with respect to y.

See Also

`diff` | `dsolve` | `int`

Topics

"Functional Derivatives Tutorial" on page 2-45

Introduced in R2015a

funm

General matrix function

Syntax

```
F = funm(A,f)
```

Description

`F = funm(A,f)` computes the function `f`(`A`) for the square matrix `A`. For details, see “Matrix Function” on page 4-758.

Examples

Matrix Cube Root

Find matrix `B`, such that `B`^{3} `= A`, where `A` is a 3-by-3 identity matrix.

To solve `B`^{3} `= A`, compute the cube root of the matrix `A` using the `funm` function. Create the symbolic function `f(x) = x^(1/3)` and use it as the second argument for `funm`. The cube root of an identity matrix is the identity matrix itself.

```
A = sym(eye(3))

syms f(x)
f(x) = x^(1/3);

B = funm(A,f)
```

```
A =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]

B =
[ 1, 0, 0]
[ 0, 1, 0]
[ 0, 0, 1]
```

Replace one of the `0` elements of matrix `A` with `1` and compute the matrix cube root again.

```
A(1,2) = 1
B = funm(A,f)
```

```
A =
[ 1, 1, 0]
[ 0, 1, 0]
[ 0, 0, 1]

B =
[ 1, 1/3, 0]
[ 0,   1, 0]
[ 0,   0, 1]
```

Now, compute the cube root of the upper triangular matrix.

```
A(1:2,2:3) = 1
B = funm(A,f)
```

```
A =
[ 1, 1, 1]
[ 0, 1, 1]
[ 0, 0, 1]

B =
[ 1, 1/3, 2/9]
[ 0,   1, 1/3]
[ 0,   0,   1]
```

Verify that `B`^{3} `= A`.

```
B^3
```

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

Matrix Lambert W Function

Find the matrix Lambert W function.

First, create a 3-by-3 matrix `A` using variable-precision arithmetic with five digit accuracy. In this example, using variable-precision arithmetic instead of exact symbolic numbers lets you speed up computations and decrease memory usage. Using only five digits helps the result to fit on screen.

```
savedefault = digits(5);
A = vpa(magic(3))
```

```
A =
[ 8.0, 1.0, 6.0]
[ 3.0, 5.0, 7.0]
[ 4.0, 9.0, 2.0]
```

Create the symbolic function `f(x) = lambertw(x)`.

```
syms f(x)
f(x) = lambertw(x);
```

To find the Lambert W function (W_0 branch) in a matrix sense, call`funm` using `f(x)` as its second argument.

```
W0 = funm(A,f)
```

```
W0 =
[  1.5335 + 0.053465i, 0.11432 + 0.47579i, 0.36208 - 0.52925i]
[ 0.21343 + 0.073771i,  1.3849 + 0.65649i, 0.41164 - 0.73026i]
[  0.26298 - 0.12724i,  0.51074 - 1.1323i,   1.2362 + 1.2595i]
```

Verify that this result is a solution of the matrix equation $A = W0 \cdot e^{W0}$ within the specified accuracy.

```
W0*expm(W0)
```

```
ans =
[               8.0, 1.0 - 5.6843e-13i, 6.0 + 1.1369e-13i]
[ 3.0 - 2.2737e-13i, 5.0 - 2.8422e-14i, 7.0 - 4.1211e-13i]
[ 4.0 - 2.2737e-13i, 9.0 - 9.9476e-14i, 2.0 + 1.4779e-12i]
```

Now, create the symbolic function `f(x)` representing the branch W_{-1} of the Lambert W function.

```
f(x) = lambertw(-1,x);
```

Find the `W_-1` branch for the matrix `A`.

```
Wm1 = funm(A,f)
```

```
Wm1 =
[   0.40925 - 4.7154i, 0.54204 + 0.5947i, 0.13764 - 0.80906i]
[ 0.38028 + 0.033194i, 0.65189 - 3.8732i, 0.056763 - 1.0898i]
[   0.2994 - 0.24756i, - 0.105 - 1.6513i,  0.89453 - 3.0309i]
```

Verify that this result is the solution of the matrix equation $A = Wm1 \cdot e^{Wm1}$ within the specified accuracy.

```
Wm1*expm(Wm1)
```

```
ans =
[ 8.0 - 8.3844e-13i,  1.0 - 3.979e-13i, 6.0 - 9.0949e-13i]
[ 3.0 - 9.6634e-13i,  5.0 + 1.684e-12i, 7.0 + 4.5475e-13i]
[ 4.0 - 1.3642e-12i, 9.0 + 1.6698e-12i, 2.0 + 1.7053e-13i]
```

Matrix Exponential, Logarithm, and Square Root

You can use `funm` with appropriate second arguments to find matrix exponential, logarithm, and square root. However, the more efficient approach is to use the functions `expm`, `logm`, and `sqrtm` for this task.

Create this square matrix and find its exponential, logarithm, and square root.

```
syms x
A = [1 -1; 0 x]
expA = expm(A)
logA = logm(A)
sqrtA = sqrtm(A)
```

```
A =
[ 1, -1]
[ 0,  x]

expA =
[ exp(1), (exp(1) - exp(x))/(x - 1)]
[      0,                    exp(x)]

logA =
[ 0, -log(x)/(x - 1)]
[ 0,          log(x)]

sqrtA =
[ 1, 1/(x - 1) - x^(1/2)/(x - 1)]
[ 0,                     x^(1/2)]
```

Find the matrix exponential, logarithm, and square root of `A` using `funm`. Use the symbolic expressions `exp(x)`, `log(x)`, and `sqrt(x)` as the second argument of `funm`. The results are identical.

```
expA = funm(A,exp(x))
logA = funm(A,log(x))
sqrtA = funm(A,sqrt(x))
```

```
expA =
[ exp(1), exp(1)/(x - 1) - exp(x)/(x - 1)]
[      0,                          exp(x)]

logA =
[ 0, -log(x)/(x - 1)]
[ 0,          log(x)]

sqrtA =
[ 1, 1/(x - 1) - x^(1/2)/(x - 1)]
[ 0,                     x^(1/2)]
```

Input Arguments

`A` — Input matrix

square matrix

Input matrix, specified as a square symbolic or numeric matrix.

`f` — Function

symbolic function | symbolic expression

Function, specified as a symbolic function or expression.

Output Arguments

F — Resulting matrix
symbolic matrix

Resulting function, returned as a symbolic matrix.

Definitions

Matrix Function

Matrix function is a scalar function that maps one matrix to another.

Suppose, `f(x)`, where `x` is a scalar, has a Taylor series expansion. Then the matrix function `f(A)`, where `A` is a matrix, is defined by the Taylor series of `f(A)`, with addition and multiplication performed in the matrix sense.

If `A` can be represented as `A = P·D·P`$^{-1}$, where `D` is a diagonal matrix, such that

$$D = \begin{pmatrix} d_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_n \end{pmatrix}$$

then the matrix function `f(A)` can be computed as follows:

$$\mathrm{f}(A) = P \cdot \begin{pmatrix} \mathrm{f}(d_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathrm{f}(d_n) \end{pmatrix} \cdot P^{-1}$$

Non-diagonalizable matrices can be represented as `A = P·J·P`$^{-1}$, where `J` is a Jordan form of the matrix `A`. Then, the matrix function `f(A)` can be computed by using the following definition on each Jordan block:

$$f\left(\begin{pmatrix}\lambda & 1 & 0 & \cdots & 0\\ 0 & \ddots & \ddots & \ddots & \vdots\\ \vdots & \ddots & \ddots & \ddots & 0\\ \vdots & \ddots & \ddots & \ddots & 1\\ 0 & \cdots & \cdots & 0 & \lambda\end{pmatrix}\right)=\begin{pmatrix}\frac{f(\lambda)}{0!} & \frac{f'(\lambda)}{1!} & \frac{f''(\lambda)}{2!} & \cdots & \frac{f^{(n-1)}(\lambda)}{(n-1)!}\\ 0 & \ddots & \ddots & \ddots & \vdots\\ \vdots & \ddots & \ddots & \ddots & \frac{f''(\lambda)}{2!}\\ \vdots & \ddots & \ddots & \ddots & \frac{f'(\lambda)}{1!}\\ 0 & \cdots & \cdots & 0 & \frac{f(\lambda)}{0!}\end{pmatrix}$$

Tips

- For compatibility with the MATLAB `funm` function, `funm` accepts the following arguments:
  - Function handles such as `@exp` and `@sin`, as its second input argument.
  - The `options` input argument, such as `funm(A,f,options)`.
  - Additional input arguments of the function `f`, such as `funm(A,f,options,p1,p2,...)`
  - The `exitflag` output argument, such as `[F,exitflag] = funm(A,f)`. Here, `exitflag` is `1` only if the `funm` function call errors, for example, if it encounters a division by zero. Otherwise, `exitflag` is `0`.

  For more details and a list of all acceptable function handles, see the MATLAB `funm` function.
- If the input matrix `A` is numeric (not a symbolic object) and the second argument `f` is a function handle, then the `funm` call invokes the MATLAB `funm` function.

See Also

`eig` | `expm` | `jordan` | `logm` | `sqrtm`

Introduced in R2014b

funtool

Function calculator

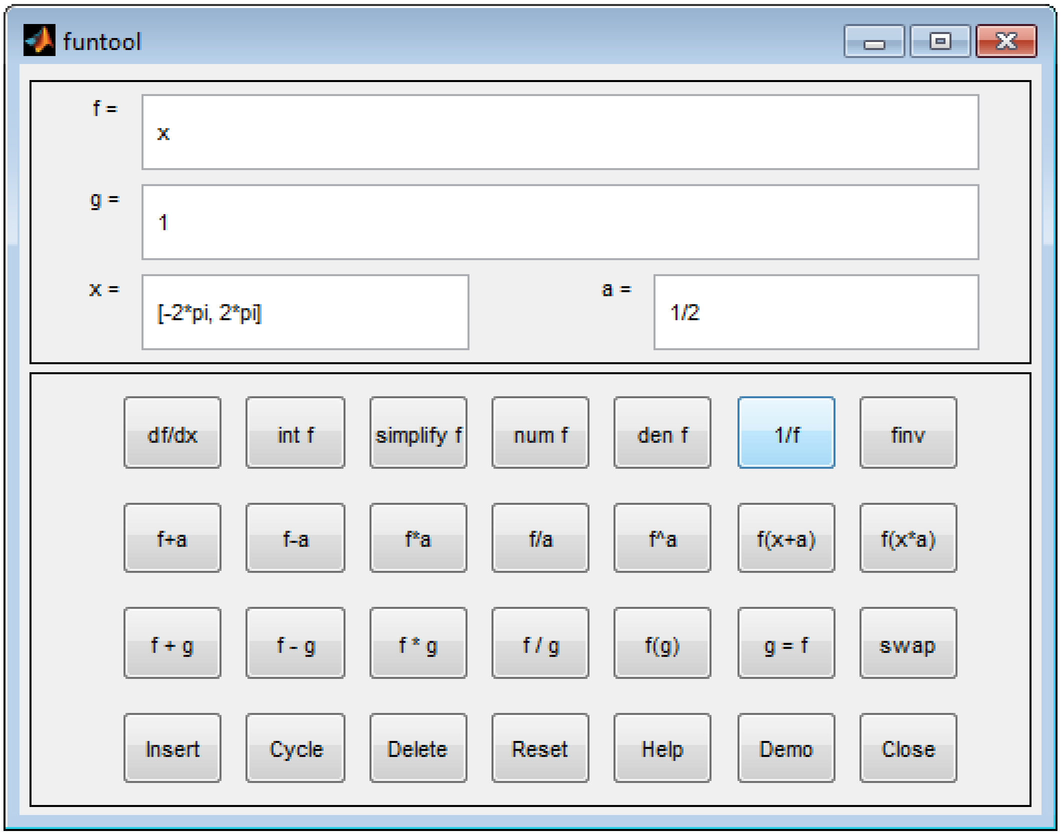
Syntax

```
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

`fplot` | `syms`

Introduced before R2006a

gamma

Gamma function

Syntax

```
gamma(X)
```

Description

`gamma(X)` returns the gamma function on page 4-767 of a symbolic variable or symbolic expression `X`.

Examples

Gamma Function for Numeric and Symbolic Arguments

Depending on its arguments, `gamma` returns floating-point or exact symbolic results.

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

```
A = gamma([-11/3, -7/5, -1/2, 1/3, 1, 4])
```

```
A =
    0.2466    2.6593   -3.5449    2.6789    1.0000    6.0000
```

Compute the gamma function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `gamma` returns unresolved symbolic calls.

```
symA = gamma(sym([-11/3, -7/5, -1/2, 1/3, 1, 4]))
```

```
symA =
[ (27*pi*3^(1/2))/(440*gamma(2/3)), gamma(-7/5),...
-2*pi^(1/2), (2*pi*3^(1/2))/(3*gamma(2/3)), 1, 6]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

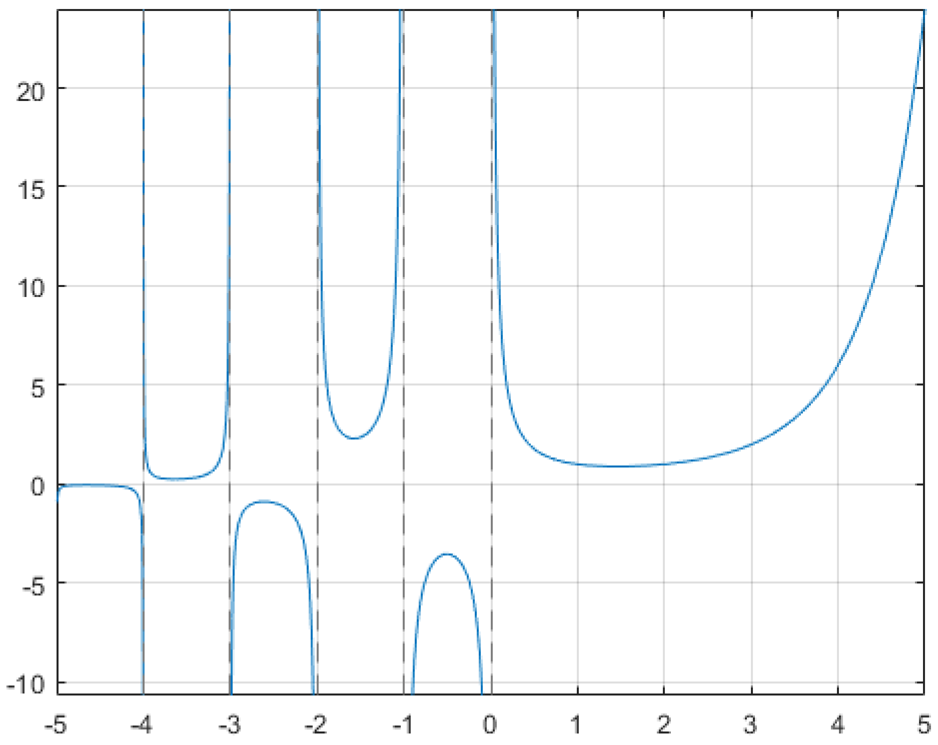
```
vpa(symA)
```

```
ans =
[ 0.24658411512650858900694446388517,...
2.6592718728800305399898810505738,...
-3.5449077018110320545963349666823,...
2.6789385347077476336556929409747,...
1.0, 6.0]
```

Plot Gamma Function

Plot the gamma function and add grid lines. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(gamma(x))
grid on
```



Handle Expressions Containing Gamma Function

Many functions, such as `diff`, `limit`, and `simplify`, can handle expressions containing `gamma`.

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, t, 1)
```

```
u =
3*t^2*gamma(t^3 + 1)*psi(t^3 + 1)
```

```
u1 =
3 - 3*eulergamma
```

Approximate the result using `vpa`:

```
vpa(u1)
```

```
ans =
1.2683530052954014181804637297528
```

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)
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as symbolic number, variable, expression, function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Gamma Function

The following integral defines the gamma function:

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

See Also

`beta` | `factorial` | `gammaln` | `igamma` | `nchoosek` | `pochhammer` | `psi`

Introduced before R2006a

gammaln

Logarithmic gamma function

Syntax

```
gammaln(X)
```

Description

`gammaln(X)` returns the logarithmic gamma function for each element of `X`.

Examples

Logarithmic Gamma Function for Numeric and Symbolic Arguments

Depending on its arguments, `gammaln` returns floating-point or exact symbolic results.

Compute the logarithmic gamma function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
A = gammaln([1/5, 1/2, 2/3, 8/7, 3])
```

```
A =
    1.5241    0.5724    0.3032   -0.0667    0.6931
```

Compute the logarithmic gamma function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `gammaln` returns results in terms of the `gammaln`, `log`, and `gamma` functions.

```
symA = gammaln(sym([1/5, 1/2, 2/3, 8/7, 3]))
```

```
symA =
[ gammaln(1/5), log(pi^(1/2)), gammaln(2/3),...
log(gamma(1/7)/7), log(2)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 1.5240638224307845248810564939263,...
0.57236494292470008707171367567653,...
0.30315027514752356867586281737201,...
-0.066740877459477468649396334098109,...
0.69314718055994530941723212145818]
```

Definition of the Logarithmic Gamma Function on Complex Plane

`gammaln` is defined for all complex arguments, except negative infinity.

Compute the logarithmic gamma function for positive integer arguments. For such arguments, the logarithmic gamma function is defined as the natural logarithm of the gamma function, `gammaln(x) = log(gamma(x))`.

```
pos = gammaln(sym([1/4, 1/3, 1, 5, Inf]))

pos =
[ log((pi*2^(1/2))/gamma(3/4)), log((2*pi*3^(1/2))/(3*gamma(2/3))), 0, log(24), Inf]
```

Compute the logarithmic gamma function for nonpositive integer arguments. For nonpositive integers, `gammaln` returns `Inf`.

```
nonposint = gammaln(sym([0, -1, -2, -5, -10]))

nonposint =
[ Inf, Inf, Inf, Inf, Inf]
```

Compute the logarithmic gamma function for complex and negative rational arguments. For these arguments, `gammaln` returns unresolved symbolic calls.

```
complex = gammaln(sym([i, -1 + 2*i , -2/3, -10/3]))

complex =
[ gammaln(1i), gammaln(- 1 + 2i), gammaln(-2/3), gammaln(-10/3)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(complex)
```

```
ans =
[ - 0.65092319930185633888521683150395 - 1.8724366472624298171188533494366i,...
- 3.3739449232079248379476073664725 - 3.4755939462808110432931921583558i,...
1.3908857550359314511651871524423 - 3.1415926535897932384626433832795i,...
- 0.93719017334928727370096467598178 - 12.566370614359172953850573533118i]
```

Compute the logarithmic gamma function of negative infinity:

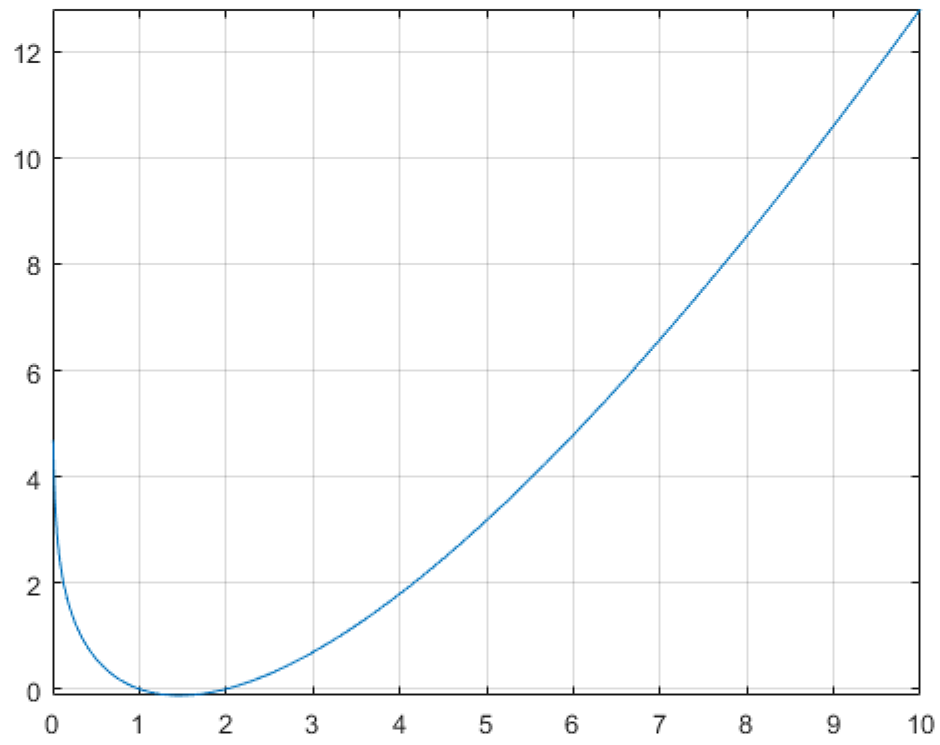
```
gammaln(sym(-Inf))
```

```
ans =
NaN
```

Plot Logarithmic Gamma Function

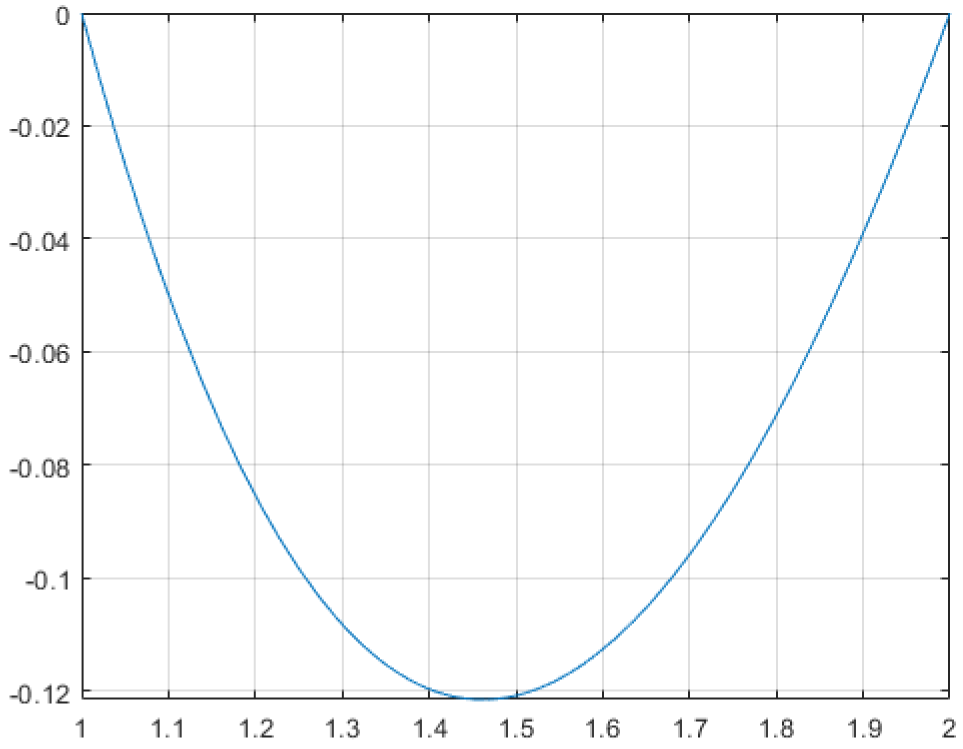
Plot the logarithmic gamma function on the interval from 0 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(gammaln(x), [0 10])
grid on
```



To see the negative values better, plot the same function on the interval from 1 to 2.

```
fplot(gammaln(x), [1 2])
grid on
```



Handle Expressions Containing Logarithmic Gamma Function

Many functions, such as `diff` and `limit`, can handle expressions containing `lngamma`.

Differentiate the logarithmic gamma function:

```
syms x
diff(gammaln(x), x)
```

```
ans =
psi(x)
```

Compute the limits of these expressions containing the logarithmic gamma function:

```
syms x
limit(1/gammaln(x), x, Inf)

ans =
0

limit(gammaln(x - 1) - gammaln(x - 2), x, 0)

ans =
log(2) + pi*1i
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as symbolic number, variable, expression, function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Algorithms

For single or double input to `gammaln(x)`, `x` must be real and positive.

For symbolic input,

- `gammaln(x)` is defined for all complex `x` except the singular points 0, -1, -2, ...
- For positive real `x`, `gammaln(x)` represents the logarithmic gamma function `log(gamma(x))`.
- For negative real `x` or for complex `x`, gammaln(x) = log(gamma(x)) + f(x)2πi where f(x) is some integer valued function. The integer multiples of 2πi are chosen such that `gammaln(x)` is analytic throughout the complex plane with a branch cut along the negative real semi axis.
- For negative real `x`, `gammaln(x)` is equal to the limit of `log(gamma(x))` from ‘above’.

See Also

`beta` | `gamma` | `log` | `nchoosek` | `psi`

Introduced in R2014a

gcd

Greatest common divisor

Syntax

```
G = gcd(A)
G = gcd(A,B)
[G,C,D] = gcd(A,B,X)
```

Description

`G = gcd(A)` finds the greatest common divisor of all elements of `A`.

`G = gcd(A,B)` finds the greatest common divisor of `A` and `B`.

`[G,C,D] = gcd(A,B,X)` finds the greatest common divisor of `A` and `B`, and also returns the Bézout coefficients, `C` and `D`, such that `G = A*C + B*D`, and `X` does not appear in the denominator of `C`and `D`. If you do not specify `X`, then `gcd` uses the default variable determined by `symvar`.

Examples

Greatest Common Divisor of Four Integers

To find the greatest common divisor of three or more values, specify those values as a symbolic vector or matrix.

Find the greatest common divisor of these four integers, specified as elements of a symbolic vector.

```
A = sym([4420, -128, 8984, -488])
gcd(A)
```

```
A =
[ 4420, -128, 8984, -488]
```

```
ans =
4
```

Alternatively, specify these values as elements of a symbolic matrix.

```
A = sym([4420, -128; 8984, -488])
gcd(A)

A =
[ 4420, -128]
[ 8984, -488]

ans =
4
```

Greatest Common Divisor of Rational Numbers

The greatest common divisor of rational numbers $a_1, a_2, ...$ is a number `g`, such that $g/a_1, g/a_2, ...$ are integers, and `gcd(g) = 1`.

Find the greatest common divisor of these rational numbers, specified as elements of a symbolic vector.

```
gcd(sym([1/4, 1/3, 1/2, 2/3, 3/4]))

ans =
1/12
```

Greatest Common Divisor of Complex Numbers

`gcd` computes the greatest common divisor of complex numbers over the Gaussian integers (complex numbers with integer real and imaginary parts). It returns a complex number with a positive real part and a nonnegative imaginary part.

Find the greatest common divisor of these complex numbers.

```
gcd(sym([10 - 5*i, 20 - 10*i, 30 - 15*i]))

ans =
5 + 10i
```

Greatest Common Divisor of Elements of Matrices

For vectors and matrices, `gcd` finds the greatest common divisors element-wise. Nonscalar arguments must be the same size.

Find the greatest common divisors for the elements of these two matrices.

```
A = sym([309, 186; 486, 224]);
B = sym([558, 444; 1024, 1984]);
gcd(A,B)
```

```
ans =
[ 3,  6]
[ 2, 32]
```

Find the greatest common divisors for the elements of matrix `A` and the value `200`. Here, `gcd` expands `200` into the `2`-by-`2` matrix with all elements equal to `200`.

```
gcd(A,200)
```

```
ans =
[ 1, 2]
[ 2, 8]
```

Greatest Common Divisor of Polynomials

Find the greatest common divisor of univariate and multivariate polynomials.

Find the greatest common divisor of these univariate polynomials.

```
syms x
gcd(x^3 - 3*x^2 + 3*x - 1, x^2 - 5*x + 4)
```

```
ans =
x - 1
```

Find the greatest common divisor of these multivariate polynomials. Because there are more than two polynomials, specify them as elements of a symbolic vector.

```
syms x y
gcd([x^2*y + x^3, (x + y)^2, x^2 + x*y^2 + x*y + x + y^3 + y])
```

```
ans =
x + y
```

Bézout Coefficients

Find the greatest common divisor and the Bézout coefficients of these polynomials. For multivariate expressions, use the third input argument to specify the polynomial variable. When computing Bézout coefficients, `gcd` ensures that the polynomial variable does not appear in their denominators.

Find the greatest common divisor and the Bézout coefficients of these polynomials with respect to variable `x`.

```
[G,C,D] = gcd(x^2*y + x^3, (x + y)^2, x)
```

```
G =
x + y

C =
1/y^2

D =
1/y - x/y^2
```

Find the greatest common divisor and the Bézout coefficients of the same polynomials with respect to variable `y`.

```
[G,C,D] = gcd(x^2*y + x^3, (x + y)^2, y)
```

```
G =
x + y

C =
1/x^2

D =
0
```

If you do not specify the polynomial variable, then the toolbox uses `symvar` to determine the variable.

```
[G,C,D] = gcd(x^2*y + x^3, (x + y)^2)
```

```
G =
x + y

C =
1/y^2

D =
1/y - x/y^2
```

Solution to Diophantine Equation

Solve the Diophantine equation, `30x + 56y = 8`, for `x` and `y`.

Find the greatest common divisor and a pair of Bézout coefficients for `30` and `56`.

```
[G,C,D] = gcd(sym(30),56)
```

```
G =
2

C =
-13

D =
7
```

`C = -13` and `D = 7` satisfy the Bézout's identity, `(30*(-13)) + (56*7) = 2`.

Rewrite Bézout's identity so that it looks more like the original equation. Do this by multiplying by `4`. Use `==` to verify that both sides of the equation are equal.

```
isAlways((30*C*4) + (56*D*4) == G*4)
```

```
ans =
  logical
   1
```

Calculate the values of `x` and `y` that solve the problem.

```
x = C*4
y = D*4
```

```
x =
-52

y =
28
```

Input Arguments

`A` — Input value

number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input value, specified as a number, symbolic number, variable, expression, function, or a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

`B` — Input value

number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input value, specified as a number, symbolic number, variable, expression, function, or a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

`X` — Polynomial variable

symbolic variable

Polynomial variable, specified as a symbolic variable.

Output Arguments

`G` — Greatest common divisor

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Greatest common divisor, returned as a symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions.

`C,D` — Bézout coefficients

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Bézout coefficients, returned as symbolic numbers, variables, expressions, functions, or vectors or matrices of symbolic numbers, variables, expressions, or functions.

Tips

- Calling `gcd` for numbers that are not symbolic objects invokes the MATLAB `gcd` function.
- The MATLAB `gcd` function does not accept rational or complex arguments. To find the greatest common divisor of rational or complex numbers, convert these numbers to symbolic objects by using `sym`, and then use `gcd`.
- Nonscalar arguments must be the same size. If one input argument is nonscalar, then `gcd` expands the scalar into a vector or matrix of the same size as the nonscalar argument, with all elements equal to the corresponding scalar.

See Also

`lcm`

Introduced in R2014b

ge

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`.

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
```

Tips

- Calling >= or `ge` for non-symbolic `A` and `B` invokes the MATLAB `ge` function. This function returns a logical array with elements set to logical `1 (true)` where `A` is greater than or equal to `B`; otherwise, it returns logical `0 (false)`.
- 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`.

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` | `lt` | `ne`

Topics

"Set Assumptions" on page 1-28

Introduced in R2012a

gegenbauerC

Gegenbauer polynomials

Syntax

```
gegenbauerC(n,a,x)
```

Description

`gegenbauerC(n,a,x)` represents the `n`th-degree Gegenbauer (ultraspherical) polynomial on page 4-790 with parameter `a` at the point `x`.

Examples

First Four Gegenbauer Polynomials

Find the first four Gegenbauer polynomials for the parameter `a` and variable `x`.

```
syms a x
gegenbauerC([0, 1, 2, 3], a, x)
```

```
ans =
[ 1, 2*a*x, (2*a^2 + 2*a)*x^2 - a,...
((4*a^3)/3 + 4*a^2 + (8*a)/3)*x^3 + (- 2*a^2 - 2*a)*x]
```

Gegenbauer Polynomials for Numeric and Symbolic Arguments

Depending on its arguments, `gegenbauerC` returns floating-point or exact symbolic results.

Find the value of the fifth-degree Gegenbauer polynomial for the parameter `a = 1/3` at these points. Because these numbers are not symbolic objects, `gegenbauerC` returns floating-point results.

```
gegenbauerC(5, 1/3, [1/6, 1/4, 1/3, 1/2, 2/3, 3/4])

ans =
    0.1520    0.1911    0.1914    0.0672   -0.1483   -0.2188
```

Find the value of the fifth-degree Gegenbauer polynomial for the same numbers converted to symbolic objects. For symbolic numbers, `gegenbauerC` returns exact symbolic results.

```
gegenbauerC(5, 1/3, sym([1/6, 1/4, 1/3, 1/2, 2/3, 3/4]))

ans =
[ 26929/177147, 4459/23328, 33908/177147, 49/729, -26264/177147, -7/32]
```

Evaluate Chebyshev Polynomials with Floating-Point Numbers

Floating-point evaluation of Gegenbauer polynomials by direct calls of `gegenbauerC` is numerically stable. However, first computing the polynomial using a symbolic variable, and then substituting variable-precision values into this expression can be numerically unstable.

Find the value of the 500th-degree Gegenbauer polynomial for the parameter `4` at `1/3` and `vpa(1/3)`. Floating-point evaluation is numerically stable.

```
gegenbauerC(500, 4, 1/3)
gegenbauerC(500, 4, vpa(1/3))

ans =
  -1.9161e+05

ans =
-191609.10250897532784888518393655
```

Now, find the symbolic polynomial `C500 = gegenbauerC(500, 4, x)`, and substitute `x = vpa(1/3)` into the result. This approach is numerically unstable.

```
syms x
C500 = gegenbauerC(500, 4, x);
subs(C500, x, vpa(1/3))

ans =
-8.0178726380235741521208852037291e35
```

Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

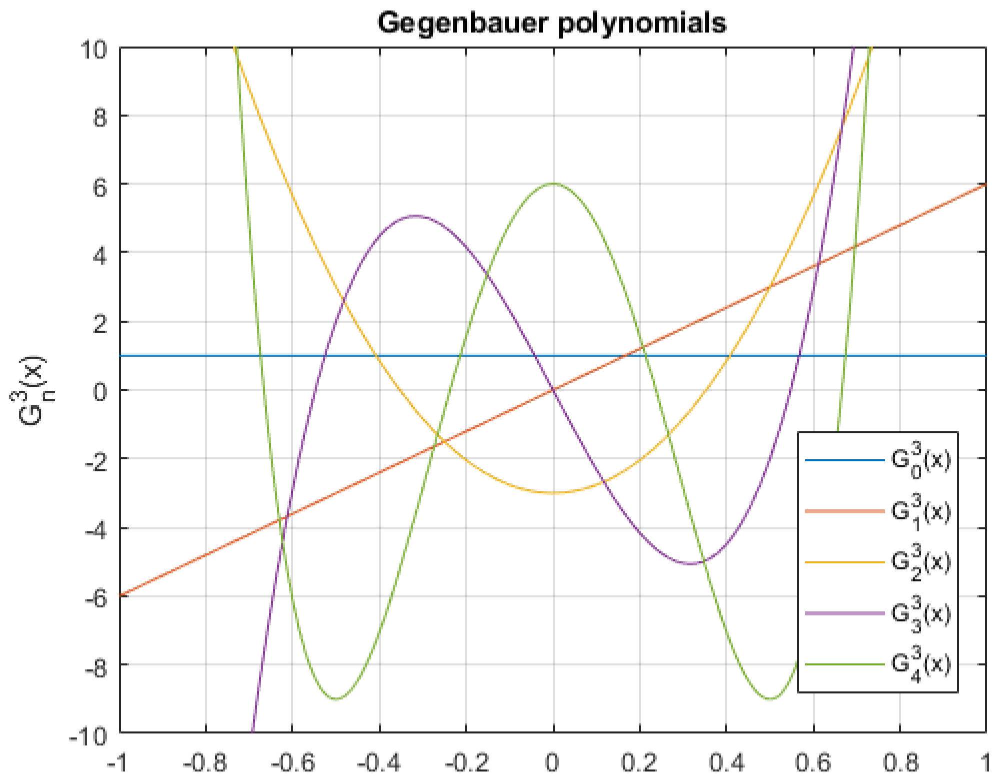
```
subs(vpa(C500), x, sym(1/3))
```

```
ans =
-8.1125412405858470246887213923167e36
```

Plot Gegenbauer Polynomials

Plot the first five Gegenbauer polynomials for the parameter `a = 3`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(gegenbauerC(0:4, 3, x))
axis([-1 1 -10 10])
grid on

ylabel('G_n^3(x)')
title('Gegenbauer polynomials')
legend('G_0^3(x)', 'G_1^3(x)', 'G_2^3(x)', 'G_3^3(x)', 'G_4^3(x)',...
                                                'Location', 'Best')
```



Input Arguments

`n` — Degree of polynomial

nonnegative integer | symbolic variable | symbolic expression | symbolic function | vector | matrix

Degree of the polynomial, specified as a nonnegative integer, symbolic variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

a — Parameter
number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Parameter, specified as a nonnegative integer, symbolic variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

x — Evaluation point
number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Evaluation point, specified as a number, symbolic number, variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Gegenbauer Polynomials

Gegenbauer polynomials are defined by this recursion formula.

$$G(0,a,x)=1,\quad G(1,a,x)=2ax,\quad G(n,a,x)=\frac{2x(n+a-1)}{n}G(n-1,a,x)-\frac{n+2a-2}{n}G(n-2,a,x)$$

For all real a > -1/2, Gegenbauer polynomials are orthogonal on the interval $-1 \le x \le 1$ with respect to the weight function

$$w(x)=\left(1-x^2\right)^{a-\frac{1}{2}}$$

Chebyshev polynomials of the first and second kinds are a special case of the Gegenbauer polynomials.

$$T(n,x)=\frac{n}{2}G(n,0,x)$$

$$U(n,x)=G(n,1,x)$$

Legendre polynomials are also a special case of the Gegenbauer polynomials.

$$P(n,x) = G\left(n,\frac{1}{2},x\right)$$

Tips

- `gegenbauerC` returns floating-point results for numeric arguments that are not symbolic objects.
- `gegenbauerC` acts element-wise on nonscalar inputs.
- All nonscalar arguments must have the same size. If one or two input arguments are nonscalar, then `gegenbauerC` expands the scalars into vectors or matrices of the same size as the nonscalar arguments, with all elements equal to the corresponding scalar.

References

[1] Hochstrasser, U. W. “Orthogonal Polynomials.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`chebyshevT` | `chebyshevU` | `hermiteH` | `jacobiP` | `laguerreL` | `legendreP`

Introduced in R2014b

getVar

Get variable from MuPAD notebook

Syntax

```
MATLABvar = getVar(nb,'MuPADvar')
```

Description

`MATLABvar = getVar(nb,'MuPADvar')` assigns the variable `MuPADvar` in the MuPAD notebook `nb` to a symbolic variable `MATLABvar` in the MATLAB workspace.

Examples

Copy Variable from MuPAD to MATLAB

Copy a variable with a value assigned to it from a MuPAD notebook to the MATLAB workspace.

Create a new MuPAD notebook and specify a handle `mpnb` to that notebook:

```
mpnb = mupad;
```

In the MuPAD notebook, enter the following command. 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.

```
f := x^2
```

Return to the MATLAB Command Window and use the `getVar` function:

```
f = getVar(mpnb,'f')
```

```
f =
x^2
```

After you call `getVar`, the variable `f` appears in the MATLAB workspace. The value of the variable `f` in the MATLAB workspace is `x^2`.

Now, use `getVar` to copy variables `a` and `b` from the same notebook. Although you do not specify these variables explicitly, and they do not have any values assigned to them, they exist in MuPAD.

```
a = getVar(mpnb,'a')
b = getVar(mpnb,'b')
```

```
a =
a

b =
b
```

- “Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52

Input Arguments

`nb` — Pointer to MuPAD notebook

handle to notebook

Pointer to a MuPAD notebook, specified as a MuPAD notebook handle. You create the notebook handle when opening a notebook with the `mupad` or `openmn` function.

`MuPADvar` — Variable in MuPAD notebook

variable

Variable in a MuPAD notebook, specified as a variable. A variable exists in MuPAD even if it has no value assigned to it.

Output Arguments

`MATLABvar` — Variable in MATLAB workspace

symbolic variable

Variable in the MATLAB workspace, returned as a symbolic variable.

See Also

`mupad` | `openmu` | `setVar`

Topics

“Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52

Introduced in R2008b

gradient

Gradient vector of scalar function

Syntax

```
gradient(f,v)
```

Description

`gradient(f,v)` finds the gradient vector of the scalar function `f` with respect to vector `v` in Cartesian coordinates.

If you do not specify `v`, then `gradient(f)` finds 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`.

Examples

Find Gradient of Function

The gradient of a function `f` with respect to the vector `v` is the vector of the first partial derivatives of `f` with respect to each element of `v`.

Find the gradient vector of `f(x, y, z)` with respect to vector `[x, y, z]`. The gradient is a vector with these components.

```
syms x y z
f = 2*y*z*sin(x) + 3*x*sin(z)*cos(y);
gradient(f, [x, y, z])
```

```
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)
```

Plot Gradient of Function

Find the gradient of a function `f(x, y)`, and plot it as a quiver (velocity) plot.

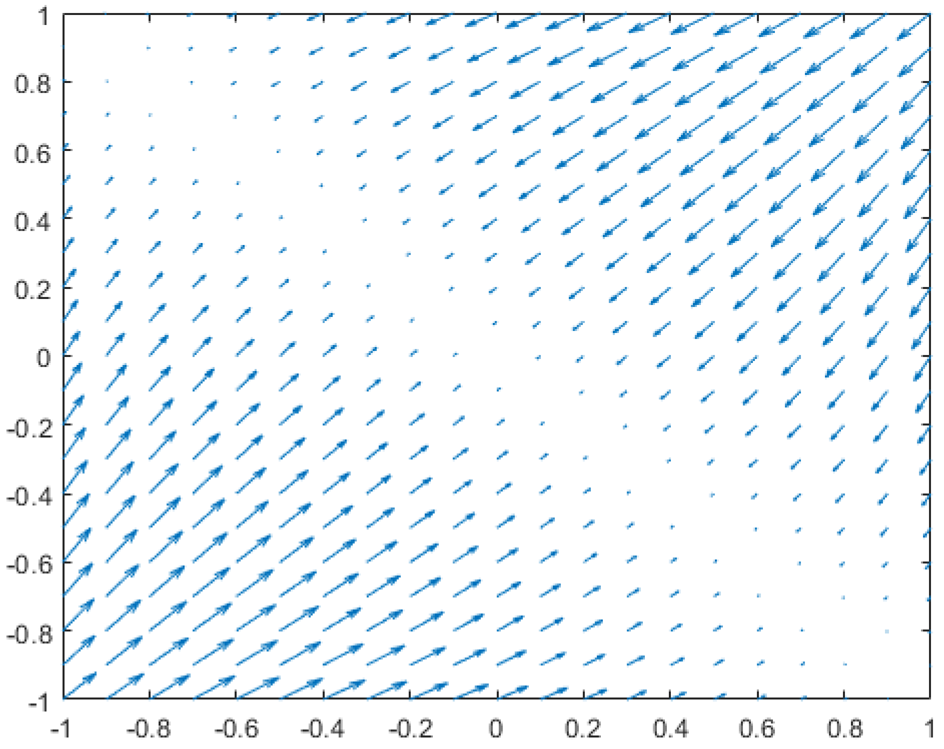
Find the gradient vector of `f(x, y)` with respect to vector `[x, y]`. The gradient is vector `g` with these components.

```
syms x y
f = -(sin(x) + sin(y))^2;
g = gradient(f, [x, y])
```

```
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)
```



Input Arguments

`f` — Scalar function

symbolic expression | symbolic function

Scalar function, specified as symbolic expression or symbolic function.

`v` — Vector with respect to which you find gradient vector

symbolic vector

Vector with respect to which you find gradient vector, specified as a symbolic vector. By default, `v` is a vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

If `v` is a scalar, `gradient(f,v) = diff(f,v)`. If `v` is an empty symbolic object, such as `sym([])`, then `gradient` returns an empty symbolic object.

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`.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

See Also

`curl` | `diff` | `divergence` | `hessian` | `jacobian` | `laplacian` | `potential` | `quiver` | `vectorPotential`

Introduced in R2011b

gt

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`.

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
```

Tips

- Calling > or `gt` for non-symbolic `A` and `B` invokes the MATLAB `gt` function. This function returns a logical array with elements set to logical `1 (true)` where `A` is greater than `B`; otherwise, it returns logical `0 (false)`.
- 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`.

See Also

`eq` | `ge` | `isAlways` | `le` | `lt` | `ne`

Topics

"Set Assumptions" on page 1-28

Introduced in R2012a

harmonic

Harmonic function (harmonic number)

Syntax

```
harmonic(x)
```

Description

`harmonic(x)` returns the harmonic function on page 4-807 of `x`. For integer values of `x`, `harmonic(x)` generates harmonic numbers.

Examples

Generate Harmonic Numbers

Generate the first 10 harmonic numbers.

```
harmonic(sym(1:10))
```

```
ans =
[ 1, 3/2, 11/6, 25/12, 137/60, 49/20, 363/140, 761/280, 7129/2520, 7381/2520]
```

Harmonic Function for Numeric and Symbolic Arguments

Find the harmonic function for these numbers. Since these are not symbolic objects, you get floating-point results.

```
harmonic([2 i 13/3])
```

```
ans =
   1.5000 + 0.0000i   0.6719 + 1.0767i   2.1545 + 0.0000i
```

Find the harmonic function symbolically by converting the numbers to symbolic objects.

```
y = harmonic(sym([2 i 13/3]))

y =
[ 3/2, harmonic(1i), 8571/1820 - (pi*3^(1/2))/6 - (3*log(3))/2]
```

If the denominator of `x` is 2, 3, 4, or 6, and $|x| < 500$, then the result is expressed in terms of `pi` and `log`.

Use `vpa` to approximate the results obtained.

```
vpa(y)

ans =
[ 1.5, 0.67186598552400983787839057280431...
 + 1.0766740474685811741340507947 5i,...
 2.1545225442213858782694336751358]
```

For $|x| > 1000$, `harmonic` returns the function call as it is. Use `vpa` to force `harmonic` to evaluate the function call.

```
harmonic(sym(1001))
vpa(harmonic(sym(1001)))

ans =
harmonic(1001)
ans =
7.4864698615493459116575172053329
```

Harmonic Function for Special Values

Find the harmonic function for special values.

```
harmonic([0 1 -1 Inf -Inf])

ans =
     0     1   Inf   Inf   NaN
```

Harmonic Function for Symbolic Functions

Find the harmonic function for the symbolic function `f`.

```
syms f(x)
f(x) = exp(x) + tan(x);
y = harmonic(f)
```

```
y(x) =
harmonic(exp(x) + tan(x))
```

Harmonic Function for Symbolic Vectors and Matrices

Find the harmonic function for elements of vector `V` and matrix `M`.

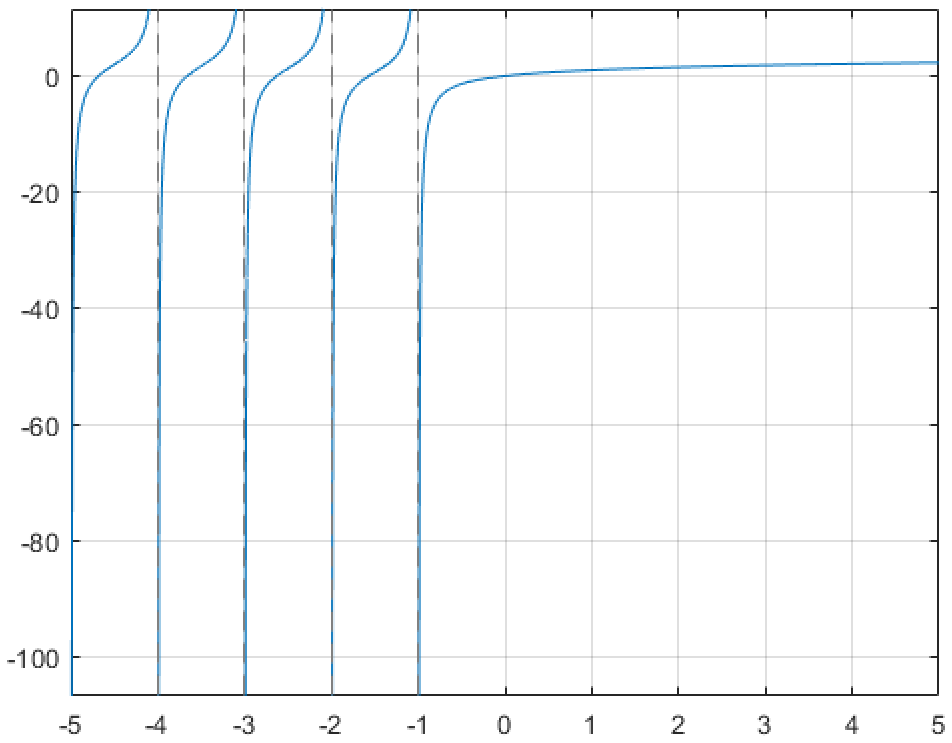
```
syms x
V = [x sin(x) 3*i];
M = [exp(i*x) 2; -6 Inf];
harmonic(V)
harmonic(M)
```

```
ans =
[ harmonic(x), harmonic(sin(x)), harmonic(3i)]
ans =
[ harmonic(exp(x*1i)), 3/2]
[                 Inf, Inf]
```

Plot Harmonic Function

Plot the harmonic function from `x` = -5 to `x` = 5. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(harmonic(x),[-5,5]), grid on
```



Differentiate and Find Limit of Harmonic Function

The functions `diff` and `limit` handle expressions containing `harmonic`.

Find the second derivative of `harmonic(x^2+1)`.

```
syms x
diff(harmonic(x^2+1),x,2)
```

```
ans =
2*psi(1, x^2 + 2) + 4*x^2*psi(2, x^2 + 2)
```

Find the limit of `harmonic(x)` as `x` tends to ∞ and of `(x+1)*harmonic(x)` as `x` tends to -1.

```
syms x
limit(harmonic(x),Inf)
limit((x+1)*harmonic(x),-1)
```

```
ans =
Inf
ans =
-1
```

Taylor Series Expansion of Harmonic Function

Use `taylor` to expand the harmonic function in terms of the Taylor series.

```
syms x
taylor(harmonic(x))
```

```
ans =
(pi^6*x^5)/945 - zeta(5)*x^4 + (pi^4*x^3)/90...
 - zeta(3)*x^2 + (pi^2*x)/6
```

Expand Harmonic Function

Use `expand` to expand the harmonic function.

```
syms x
expand(harmonic(2*x+3))
```

```
ans =
harmonic(x + 1/2)/2 + log(2) + harmonic(x)/2 - 1/(2*(x + 1/2))...
 + 1/(2*x + 1) + 1/(2*x + 2) + 1/(2*x + 3)
```

Input Arguments

x — Input
number | vector | matrix | multidimensional array | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic N-D array

Input, specified as number, vector, matrix, or as a multidimensional array or symbolic variable, expression, function, vector, matrix, or multidimensional array.

Definitions

Harmonic Function

The harmonic function for x is defined as

$$\mathrm{harmonic}(x) = \sum_{k=1}^{x} \frac{1}{k}$$

It is also defined as

$$\mathrm{harmonic}(x) = \Psi(x+1) + \gamma$$

where $\Psi(x)$ is the polygamma function and γ is the Euler-Mascheroni constant.

Algorithms

The harmonic function is defined for all complex arguments z except for negative integers -1, -2,... where a singularity occurs.

If `x` has denominator 1, 2, 3, 4, or 6, then an explicit result is computed and returned. For other rational numbers, `harmonic` uses the functional equation

$\mathrm{harmonic}(x+1) = \mathrm{harmonic}(x) + \frac{1}{x}$ to obtain a result with an argument x from the interval [0, 1].

`expand` expands `harmonic` using the equations $\mathrm{harmonic}(x+1) = \mathrm{harmonic}(x) + \frac{1}{x}$,

$\mathrm{harmonic}(-x) = \mathrm{harmonic}(x) - \frac{1}{x} + \pi\cot(\pi x)$, and the Gauss multiplication formula for `harmonic(kx)`, where k is an integer.

`harmonic` implements the following explicit formulae:

$$\text{harmonic}\left(\frac{-1}{2}\right) = -2\ln(2)$$

$$\text{harmonic}\left(-\frac{2}{3}\right) = -\frac{3}{2}\ln(3) - \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(-\frac{1}{3}\right) = -\frac{3}{2}\ln(3) + \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{-3}{4}\right) = -3\ln(2) - \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{-1}{4}\right) = -3\ln(2) + \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{-5}{6}\right) = -2\ln(2) - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}\left(\frac{-1}{6}\right) = -2\ln(2) - \frac{3}{2}\ln(3) + \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}(0) = 0$$

$$\text{harmonic}\left(\frac{1}{2}\right) = 2 - 2\ln(2)$$

$$\text{harmonic}\left(\frac{1}{3}\right) = 3 - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{2}{3}\right) = \frac{3}{2} - \frac{3}{2}\ln(3) + \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{1}{4}\right) = 4 - 3\ln(2) - \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{3}{4}\right) = \frac{4}{3} - 3\ln(2) + \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{1}{6}\right) = 6 - 2\ln(2) - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{2}\pi$$

$$\mathrm{harmonic}\left(\frac{5}{6}\right)=\frac{6}{5}-2\ln(2)-\frac{3}{2}\ln(3)+\frac{\sqrt{3}}{2}\pi$$

$$\mathrm{harmonic}(1)=1$$

$$\mathrm{harmonic}(\infty)=\infty$$

$$\mathrm{harmonic}(-\infty)=NaN$$

See Also

`beta` | `factorial` | `gamma` | `gammaln` | `nchoosek` | `zeta`

Introduced in R2014a

has

Check if expression contains particular subexpression

Syntax

```
has(expr,subexpr)
```

Description

`has(expr,subexpr)` returns logical `1` (true) if `expr` contains `subexpr`. Otherwise, it returns logical `0` (false).

- If `expr` is an array, `has(expr,subexpr)` returns an array of the same size as `expr`. The returned array contains logical `1s` (true) where the elements of `expr` contain `subexpr`, and logical `0s` (false) where they do not.
- If `subexpr` is an array, `has(expr,subexpr)` checks if `expr` contains any element of `subexpr`.

Examples

Check If Expression Contains Particular Subexpression

Use the `has` function to check if an expression contains a particular variable or subexpression.

Check if these expressions contain variable `z`.

```
syms x y z
has(x + y + z, z)
```

```
ans =
  logical
   1
```

```
has(x + y, z)

ans =
  logical
   0
```

Check if `x + y + z` contains the following subexpressions. Note that `has` finds the subexpression `x + z` even though the terms `x` and `z` do not appear next to each other in the expression.

```
has(x + y + z, x + y)
has(x + y + z, y + z)
has(x + y + z, x + z)

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

Check if the expression `(x + 1)^2` contains `x^2`. Although `(x + 1)^2` is mathematically equivalent to the expression `x^2 + 2*x + 1`, the result is a logical `0` because `has` typically does not transform expressions to different forms when testing for subexpressions.

```
has((x + 1)^2, x^2)

ans =
  logical
   0
```

Expand the expression and then call `has` to check if the result contains `x^2`. Because `expand((x + 1)^2)` transforms the original expression to `x^2 + 2*x + 1`, the `has` function finds the subexpression `x^2` and returns logical `1`.

```
has(expand((x + 1)^2), x^2)

ans =
  logical
   1
```

Check If Expression Contains Any of Specified Subexpressions

Check if a symbolic expression contains any of subexpressions specified as elements of a vector.

If an expression contains one or more of the specified subexpressions, `has` returns logical `1`.

```
syms x
has(sin(x) + cos(x) + x^2, [tan(x), cot(x), sin(x), exp(x)])
```

```
ans =
  logical
   1
```

If an expression does not contain any of the specified subexpressions, `has` returns logical `0`.

```
syms x
has(sin(x) + cos(x) + x^2, [tan(x), cot(x), exp(x)])
```

```
ans =
  logical
   0
```

Find Matrix Elements Containing Particular Subexpression

Using `has`, find those elements of a symbolic matrix that contain a particular subexpression.

First, create a matrix.

```
syms x y
M = [sin(x)*sin(y), cos(x*y) + 1; cos(x)*tan(x), 2*sin(x)^2]
```

```
M =
[ sin(x)*sin(y), cos(x*y) + 1]
[ cos(x)*tan(x),   2*sin(x)^2]
```

Use `has` to check which elements of `M` contain `sin(x)`. The result is a matrix of the same size as `M`, with `1`s and `0`s as its elements. For the elements of `M` containing the specified expression, `has` returns logical `1`s. For the elements that do not contain that subexpression, `has` returns logical `0`s.

```
T = has(M, sin(x))

T =
  2×2 logical array
     1     0
     0     1
```

Return only the elements that contain `sin(x)` and replace all other elements with `0` by multiplying `M` by `T` elementwise.

```
M.*T

ans =
[ sin(x)*sin(y),          0]
[             0, 2*sin(x)^2]
```

To check if any of matrix elements contain a particular subexpression, use `any`.

```
any(has(M(:), sin(x)))

ans =
  logical
   1

any(has(M(:), cos(y)))

ans =
  logical
   0
```

Find Vector Elements Containing Any of Specified Subexpressions

Using `has`, find those elements of a symbolic vector that contain any of the specified subexpressions.

```
syms x y z
T = has([x + 1, cos(y) + 1, y + z, 2*x*cos(y)], [x, cos(y)])

T =
  1×4 logical array
     1     1     0     1
```

Return only the elements of the original vector that contain `x` or `cos(y)` or both, and replace all other elements with `0` by multiplying the original vector by `T` elementwise.

```
[x + 1, cos(y) + 1, y + z, 2*x*cos(y)].*T
```

```
ans =
[ x + 1, cos(y) + 1, 0, 2*x*cos(y)]
```

Use `has` for Symbolic Functions

If `expr` or `subexpr` is a symbolic function, `has` uses `formula(expr)` or `formula(subexpr)`. This approach lets the `has` function check if an expression defining the symbolic function `expr` contains an expression defining the symbolic function `subexpr`.

Create a symbolic function.

```
syms x
f(x) = sin(x) + cos(x);
```

Here, `sin(x) + cos(x)` is an expression defining the symbolic function `f`.

```
formula(f)
```

```
ans =
cos(x) + sin(x)
```

Check if `f` and `f(x)` contain `sin(x)`. In both cases `has` checks if the expression `sin(x) + cos(x)` contains `sin(x)`.

```
has(f, sin(x))
has(f(x), sin(x))
```

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

Check if `f(x^2)` contains `f`. For these arguments, `has` returns logical `0` (false) because it does not check if the expression `f(x^2)` contains the letter `f`. This call is equivalent to `has(f(x^2), formula(f))`, which, in turn, resolves to `has(cos(x^2) + sin(x^2), cos(x) + sin(x))`.

```
has(f(x^2), f)
```

```
ans =
  logical
   0
```

Check for Calls to Particular Function

Check for calls to a particular function by specifying the function name as the second argument. Check for calls to any one of multiple functions by specifying the multiple functions as a cell array of character vectors.

Integrate `tan(x^7)`. Determine if the integration is successful by checking the result for calls to `int`. Because `has` finds the `int` function and returns logical `1` (`true`), the integration is not successful.

```
syms x
f = int(tan(x^7), x);
has(f, 'int')

ans =
  logical
   1
```

Check if the solution to a differential equation contains calls to either `sin` or `cos` by specifying the second argument as `{'sin','cos'}`. The `has` function returns logical `0` (`false`), which means the solution does not contain calls to either `sin` or `cos`.

```
syms y(x) a
sol = dsolve(diff(y,x) == a*y);
has(sol, {'sin' 'cos'})

ans =
  logical
   0
```

Input Arguments

`expr` — Expression to test

symbolic expression | symbolic function | symbolic equation | symbolic inequality | symbolic vector | symbolic matrix | symbolic array

Expression to test, specified as a symbolic expression, function, equation, or inequality. Also it can be a vector, matrix, or array of symbolic expressions, functions, equations, and inequalities.

`subexpr` — Subexpression to check for

symbolic variable | symbolic expression | symbolic function | symbolic equation | symbolic inequality | symbolic vector | symbolic matrix | symbolic array | character vector | cell array of character vectors

Subexpression to test for, specified as a symbolic variable, expression, function, equation, or inequality, or a character vector, or a cell array of character vectors. `subexpr` can also be a vector, matrix, or array of symbolic variables, expressions, functions, equations, and inequalities.

Tips

- `has` does not transform or simplify expressions. This is why it does not find subexpressions like `x^2` in expressions like `(x + 1)^2`. However, in some cases `has` might find that an expression or subexpression can be represented in a form other than its original form. For example, `has` finds that the expression `-x - 1` can be represented as `-(x + 1)`. Thus, the call `has(-x - 1, x + 1)` returns `1`.
- If `expr` is an empty symbolic array, `has` returns an empty logical array of the same size as `expr`.

See Also

`subexpr` | `subs` | `times`

Introduced in R2015b

heaviside

Heaviside step function

Syntax

```
heaviside(x)
```

Description

`heaviside(x)` returns the value `0` for `x < 0`, `1` for `x > 0`, and `1/2` for `x = 0`.

Examples

Evaluate Heaviside Function for Numeric and Symbolic Arguments

Depending on the argument value, `heaviside` returns one of these values: `0`, `1`, or `1/2`. If the argument is a floating-point number (not a symbolic object), then `heaviside` returns floating-point results.

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
```

Use Assumptions on Variables

`heaviside` takes into account assumptions on variables.

```
syms x
assume(x < 0)
heaviside(x)
```

```
ans =
0
```

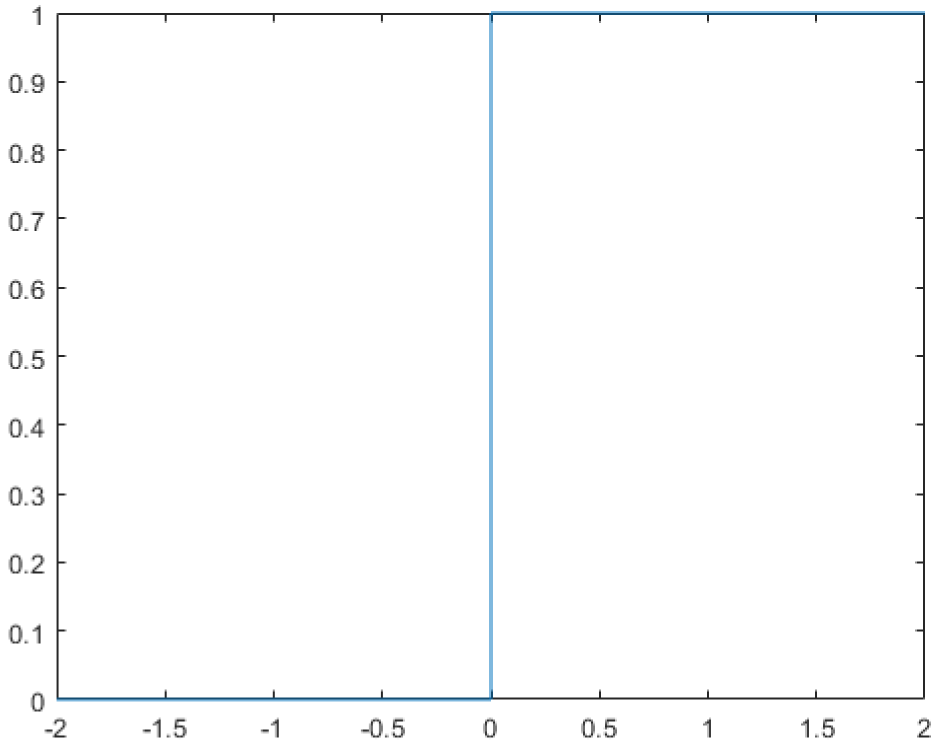
For further computations, clear the assumptions:

```
syms x clear
```

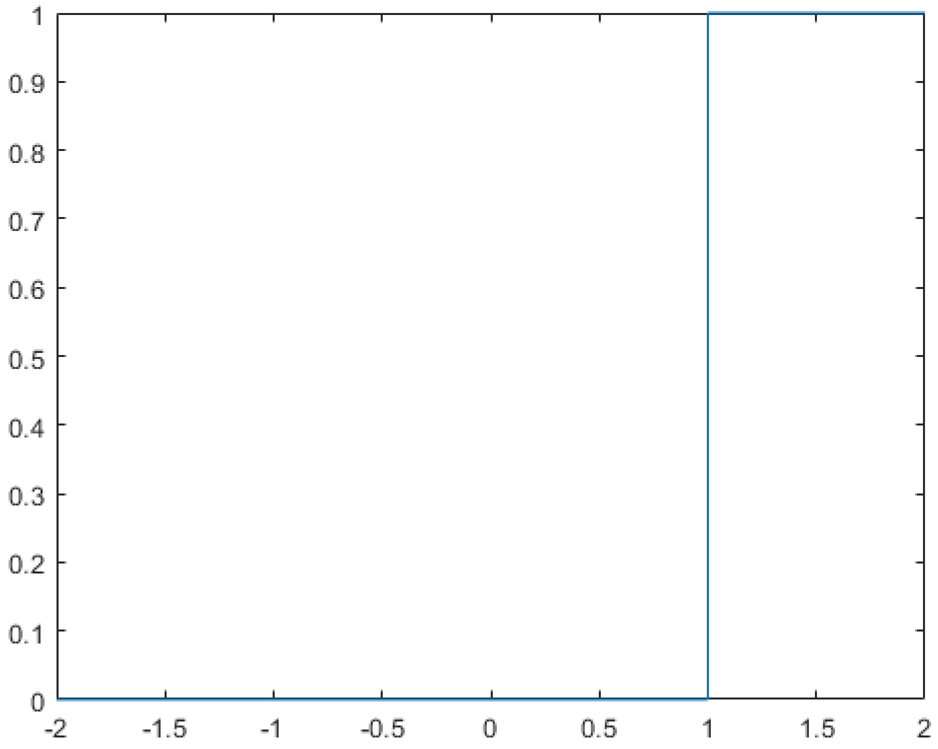
Plot Heaviside Function

Plot the Heaviside step function for `x` and `x - 1`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(heaviside(x), [-2, 2])
```



```
fplot(heaviside(x - 1), [-2, 2])
```



Evaluate Heaviside Function for Symbolic Matrix

Call `heaviside` for this symbolic matrix. When the input argument is a matrix, `heaviside` computes the Heaviside function for each element.

```
syms x
heaviside(sym([-1 0; 1/2 x]))
```

```
ans =
[ 0,          1/2]
[ 1, heaviside(x)]
```

Differentiate and Integrate Expressions Involving Heaviside Function

Compute derivatives and integrals of expressions involving the Heaviside function.

Find the first derivative of the Heaviside function. The first derivative of the Heaviside function is the Dirac delta function.

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

ans =
dirac(x)
```

Find the integral of the expression involving the Heaviside function:

```
syms x
int(exp(-x)*heaviside(x), x, -Inf, Inf)

ans =
1
```

Change Value of Heaviside Function at Origin

`heaviside` assumes that the value of the Heaviside function at the origin is `1/2`.

```
heaviside(sym(0))

ans =
1/2
```

Other common values for the Heaviside function at the origin are `0` and `1`. To change the value of `heaviside` at the origin, use the `'HeavisideAtOrigin'` preference of `sympref`. Store the previous parameter value returned by `sympref`, so that you can restore it later.

```
oldparam = sympref('HeavisideAtOrigin',1);
```

Check the new value of `heaviside` at `0`.

```
heaviside(sym(0))

ans =
1
```

The preferences set by `sympref` persist throughout your current and future MATLAB sessions. To restore the previous value of `heaviside` at the origin, use the value stored in `oldparam`.

```
sympref('HeavisideAtOrigin',oldparam);
```

Alternatively, you can restore the default value of `'HeavisideAtOrigin'` by using the `'default'` setting.

```
sympref('HeavisideAtOrigin','default');
```

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.

See Also

`dirac` | `sympref`

Introduced before R2006a

hermiteForm

Hermite form of matrix

Syntax

```
H = hermiteForm(A)
[U,H] = hermiteForm(A)

___ = hermiteForm(A,var)
```

Description

`H = hermiteForm(A)` returns the Hermite normal form on page 4-828 of a matrix `A`. The elements of `A` must be integers or polynomials in a variable determined by `symvar(A,1)`. The Hermite form `H` is an upper triangular matrix.

`[U,H] = hermiteForm(A)` returns the Hermite normal form of `A` and a unimodular transformation matrix `U`, such that `H = U*A`.

`___ = hermiteForm(A,var)` assumes that the elements of `A` are univariate polynomials in the specified variable `var`. If `A` contains other variables, `hermiteForm` treats those variables as symbolic parameters.

You can use the input argument `var` in any of the previous syntaxes.

If `A` does not contain `var`, then `hermiteForm(A)` and `hermiteForm(A,var)` return different results.

Examples

Hermite Form for Matrix of Integers

Find the Hermite form of an inverse Hilbert matrix.

```
A = sym(invhilb(5))
H = hermiteForm(A)

A =
[    25,   -300,    1050,   -1400,    630]
[  -300,   4800,  -18900,   26880, -12600]
[  1050, -18900,   79380, -117600,  56700]
[ -1400,  26880, -117600,  179200, -88200]
[   630, -12600,   56700,  -88200,  44100]

H =
[ 5,  0, -210, -280,  630]
[ 0, 60,    0,    0,    0]
[ 0,  0,  420,    0,    0]
[ 0,  0,    0,  840,    0]
[ 0,  0,    0,    0, 2520]
```

Hermite Form for Matrix of Univariate Polynomials

Create a 2-by-2 matrix, the elements of which are polynomials in the variable x.

```
syms x
A = [x^2 + 3, (2*x - 1)^2; (x + 2)^2, 3*x^2 + 5]

A =
[   x^2 + 3, (2*x - 1)^2]
[ (x + 2)^2,   3*x^2 + 5]
```

Find the Hermite form of this matrix.

```
H = hermiteForm(A)

H =
[ 1, (4*x^3)/49 + (47*x^2)/49 - (76*x)/49 + 20/49]
[ 0,            x^4 + 12*x^3 - 13*x^2 - 12*x - 11]
```

Hermite Form for Matrix of Multivariate Polynomials

Create a 2-by-2 matrix that contains two variables: x and y.

```
syms x y
A = [2/x + y, x^2 - y^2; 3*sin(x) + y, x]
```

```
A =
[      y + 2/x, x^2 - y^2]
[ y + 3*sin(x),         x]
```

Find the Hermite form of this matrix. If you do not specify the polynomial variable, `hermiteForm` uses `symvar(A,1)` and thus determines that the polynomial variable is `x`. Because `3*sin(x) + y` is not a polynomial in `x`, `hermiteForm` throws an error.

```
H = hermiteForm(A)
```

```
Error using mupadengine/feval (line 163)
Cannot convert the matrix entries to integers or univariate polynomials.
```

Find the Hermite form of `A` specifying that all elements of `A` are polynomials in the variable `y`.

```
H = hermiteForm(A,y)
```

```
H =
[ 1, (x*y^2)/(3*x*sin(x) - 2) + (x*(x - x^2))/(3*x*sin(x) - 2)]
[ 0,     3*y^2*sin(x) - 3*x^2*sin(x) + y^3 + y*(- x^2 + x) + 2]
```

Hermite Form and Transformation Matrix

Find the Hermite form and the corresponding transformation matrix for an inverse Hilbert matrix.

```
A = sym(invhilb(3));
[U,H] = hermiteForm(A)
```

```
U =
[ 13,  9,  7]
[  6,  4,  3]
[ 20, 15, 12]

H =
[ 3,  0, 30]
[ 0, 12,  0]
[ 0,  0, 60]
```

Verify that `H = U*A`.

```
isAlways(H == U*A)
```

```
ans =
  3×3 logical array
     1     1     1
     1     1     1
     1     1     1
```

Find the Hermite form and the corresponding transformation matrix for a matrix of polynomials.

```
syms x y
A = [2*(x - y), 3*(x^2 - y^2);
     4*(x^3 - y^3), 5*(x^4 - y^4)];
[U,H] = hermiteForm(A,x)
```

```
U =
[                   1/2,  0]
[ 2*x^2 + 2*x*y + 2*y^2, -1]

H =
[ x - y,         (3*x^2)/2 - (3*y^2)/2]
[     0, x^4 + 6*x^3*y - 6*x*y^3 - y^4]
```

Verify that `H = U*A`.

```
isAlways(H == U*A)
```

```
ans =
  2×2 logical array
     1     1
     1     1
```

If You Specify Variable for Integer Matrix

If a matrix does not contain a particular variable, and you call `hermiteForm` specifying that variable as the second argument, then the result differs from what you get without specifying that variable. For example, create a matrix that does not contain any variables.

```
A = [9 -36 30; -36 192 -180; 30 -180 180]
```

```
A =
     9   -36    30
   -36   192  -180
    30  -180   180
```

Call `hermiteForm` specifying variable `x` as the second argument. In this case, `hermiteForm` assumes that the elements of `A` are univariate polynomials in `x`.

```
syms x
hermiteForm(A,x)

ans =
     1     0     0
     0     1     0
     0     0     1
```

Call `hermiteForm` without specifying variables. In this case, `hermiteForm` treats `A` as a matrix of integers.

```
hermiteForm(A)

ans =
     3     0    30
     0    12     0
     0     0    60
```

Input Arguments

`A` — Input matrix
symbolic matrix

Input matrix, specified as a symbolic matrix, the elements of which are integers or univariate polynomials. If the elements of `A` contain more than one variable, use the `var` argument to specify a polynomial variable, and treat all other variables as symbolic parameters. If `A` is multivariate, and you do not specify `var`, then `hermiteForm` uses `symvar(A,1)` to determine a polynomial variable.

`var` — Polynomial variable
symbolic variable

Polynomial variable, specified as a symbolic variable.

Output Arguments

H — Hermite normal form of input matrix
symbolic matrix

Hermite normal form of input matrix, returned as a symbolic matrix. The Hermite form of a matrix is an upper triangular matrix.

U — Transformation matrix
unimodular symbolic matrix

Transformation matrix, returned as a unimodular symbolic matrix. If elements of `A` are integers, then elements of `U` are also integers, and `det(U) = 1` or `det(U) = -1`. If elements of `A` are polynomials, then elements of `U` are univariate polynomials, and `det(U)` is a constant.

Definitions

Hermite Normal Form

For any square n-by-n matrix A with integer coefficients, there exists an n-by-n matrix H and an n-by-n unimodular matrix U, such that $A*U = H$, where H is the Hermite normal form of A. A unimodular matrix is a real square matrix, such that its determinant equals `1` or `-1`. If A is a matrix of polynomials, then the determinant of U is a constant.

`hermiteForm` returns the Hermite normal form of a nonsingular integer square matrix

A as an upper triangular matrix H, such that $H_{jj} \geq 0$ and $-\frac{H_{jj}}{2} < H_{ij} \leq \frac{H_{jj}}{2}$ for $j > i$. If A is not a square matrix or a singular matrix, the matrix H is simply an upper triangular matrix.

See Also

`jordan` | `smithForm`

Introduced in R2015b

hermiteH

Hermite polynomials

Syntax

```
hermiteH(n,x)
```

Description

`hermiteH(n,x)` represents the `n`th-degree Hermite polynomial at the point `x`.

Examples

First Five Hermite Polynomials

Find the first five Hermite polynomials of the second kind for the variable `x`.

```
syms x
hermiteH([0, 1, 2, 3, 4], x)
```

```
ans =
[ 1, 2*x, 4*x^2 - 2, 8*x^3 - 12*x, 16*x^4 - 48*x^2 + 12]
```

Hermite Polynomials for Numeric and Symbolic Arguments

Depending on its arguments, `hermiteH` returns floating-point or exact symbolic results.

Find the value of the fifth-degree Hermite polynomial at these points. Because these numbers are not symbolic objects, `hermiteH` returns floating-point results.

```
hermiteH(5, [1/6, 1/3, 1/2, 2/3, 3/4])
```

```
ans =
   19.2634   34.2058   41.0000   36.8066   30.0938
```

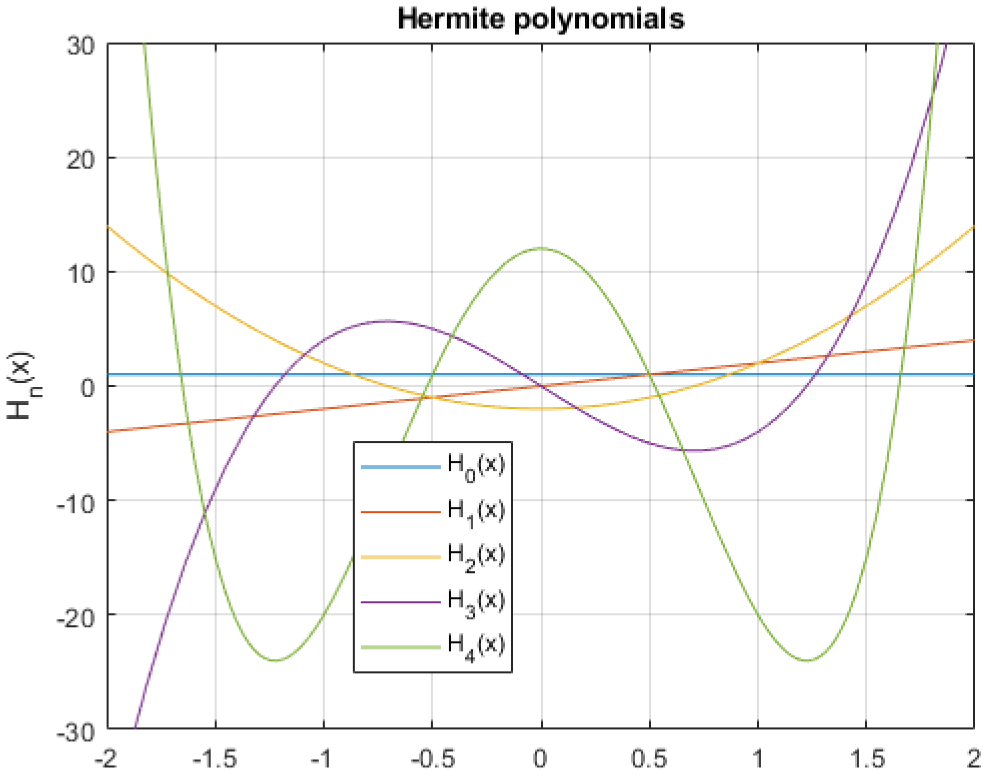
Find the value of the fifth-degree Hermite polynomial for the same numbers converted to symbolic objects. For symbolic numbers, `hermiteH` returns exact symbolic results.

```
hermiteH(5, sym([1/6, 1/3, 1/2, 2/3, 3/4]))

ans =
[ 4681/243, 8312/243, 41, 8944/243, 963/32]
```

Plot Hermite Polynomials

Plot the first five Hermite polynomials. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(hermiteH(0:4,x))
axis([-2 2 -30 30])
grid on

ylabel('H_n(x)')
legend('H_0(x)', 'H_1(x)', 'H_2(x)', 'H_3(x)', 'H_4(x)', 'Location', 'Best')
title('Hermite polynomials')
```



Input Arguments

`n` — Degree of polynomial

nonnegative integer | symbolic variable | symbolic expression | symbolic function | vector | matrix

Degree of the polynomial, specified as a nonnegative integer, symbolic variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

x — Evaluation point

number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Evaluation point, specified as a number, symbolic number, variable, expression, or function, or as a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Definitions

Hermite Polynomials

Hermite polynomials are defined by this recursion formula:

$$H(0,x)=1, \quad H(1,x)=2x, \quad H(n,x)=2xH(n-1,x)-2(n-1)H(n-2,x)$$

Hermite polynomials are orthogonal on the real line with respect to the weight function

$$w(x)=e^{-x^2}$$

Tips

- `hermiteH` returns floating-point results for numeric arguments that are not symbolic objects.
- `hermiteH` acts element-wise on nonscalar inputs.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, then `hermiteH` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

References

[1] Hochstrasser, U. W. "Orthogonal Polynomials." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`chebyshevT` | `chebyshevU` | `gegenbauerC` | `jacobiP` | `laguerreL` | `legendreP`

Introduced in R2014b

hessian

Hessian matrix of scalar function

Syntax

```
hessian(f,v)
```

Description

`hessian(f,v)` finds the Hessian matrix on page 4-835 of the scalar function `f` with respect to vector `v` in Cartesian coordinates.

If you do not specify `v`, then `hessian(f)` finds 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`.

Examples

Find Hessian Matrix of Scalar Function

Find the Hessian matrix of a function by using `hessian`. Then find the Hessian matrix of the same function as the Jacobian of the gradient of the function.

Find the Hessian matrix of this function of three variables:

```
syms x y z
f = x*y + 2*z*x;
hessian(f,[x,y,z])
```

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

Alternatively, compute the Hessian matrix of this function as the Jacobian of the gradient of that function:

```
jacobian(gradient(f))
```

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

Input Arguments

`f` — Scalar function

symbolic expression | symbolic function

Scalar function, specified as symbolic expression or symbolic function.

`v` — Vector with respect to which you find Hessian matrix

symbolic vector

Vector with respect to which you find Hessian matrix, specified as a symbolic vector. By default, `v` is a vector constructed from all symbolic variables found in `f`. The order of variables in this vector is defined by `symvar`.

If `v` is an empty symbolic object, such as `sym([])`, then `hessian` returns an empty symbolic object.

Definitions

Hessian Matrix

The Hessian matrix of $f(x)$ is the square matrix of the second partial derivatives of $f(x)$.

$$H(f) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

See Also

`curl` | `diff` | `divergence` | `gradient` | `jacobian` | `laplacian` | `potential` | `vectorPotential`

Introduced in R2011b

horner

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)
```

```
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])
```

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

See Also

`collect` | `combine` | `expand` | `factor` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

Introduced before R2006a

horzcat

Concatenate symbolic arrays horizontally

Syntax

```
horzcat(A1,...,AN)
[A1 ... AN]
```

Description

`horzcat(A1,...,AN)` horizontally concatenates the symbolic arrays `A1,...,AN`. For vectors and matrices, all inputs must have the same number of rows. For multidimensional arrays, `horzcat` concatenates inputs along the second dimension. The remaining dimensions must match.

`[A1 ... AN]` is a shortcut for `horzcat(A1,...,AN)`.

Examples

Concatenate Two Symbolic Matrices Horizontally

Create matrices `A` and `B`.

```
A = sym('a%d%d',[2 2])
B = sym('b%d%d',[2 2])
```

```
A =
[ a11, a12]
[ a21, a22]
B =
[ b11, b12]
[ b21, b22]
```

Concatenate `A` and `B`.

```
horzcat(A,B)

ans =
[ a11, a12, b11, b12]
[ a21, a22, b21, b22]
```

Alternatively, use the shortcut `[A B]`.

```
[A B]

ans =
[ a11, a12, b11, b12]
[ a21, a22, b21, b22]
```

Concatenate Multiple Symbolic Arrays Horizontally

```
A = sym('a%d',[3 1]);
B = sym('b%d%d',[3 3]);
C = sym('c%d%d',[3 2]);
horzcat(C,A,B)

ans =
[ c11, c12, a1, b11, b12, b13]
[ c21, c22, a2, b21, b22, b23]
[ c31, c32, a3, b31, b32, b33]
```

Alternatively, use the shortcut `[C A B]`.

```
[C A B]

ans =
[ c11, c12, a1, b11, b12, b13]
[ c21, c22, a2, b21, b22, b23]
[ c31, c32, a3, b31, b32, b33]
```

Concatenate Multidimensional Arrays Horizontally

Create the 3-D symbolic arrays `A` and `B`.

```
A = sym('a%d%d',[2 3]);
A(:,:,2) = -A
B = sym('b%d%d', [2 2]);
B(:,:,2) = -B
```

```
A(:,:,1) =
[ a11, a12, a13]
[ a21, a22, a23]
A(:,:,2) =
[ -a11, -a12, -a13]
[ -a21, -a22, -a23]

B(:,:,1) =
[ b11, b12]
[ b21, b22]
B(:,:,2) =
[ -b11, -b12]
[ -b21, -b22]
```

Use `horzcat` to concatenate `A` and `B`.

```
horzcat(A,B)
```

```
ans(:,:,1) =
[ a11, a12, a13, b11, b12]
[ a21, a22, a23, b21, b22]
ans(:,:,2) =
[ -a11, -a12, -a13, -b11, -b12]
[ -a21, -a22, -a23, -b21, -b22]
```

Alternatively, use the shortcut `[A B]`.

```
[A B]
```

```
ans(:,:,1) =
[ a11, a12, a13, b11, b12]
[ a21, a22, a23, b21, b22]
ans(:,:,2) =
[ -a11, -a12, -a13, -b11, -b12]
[ -a21, -a22, -a23, -b21, -b22]
```

Input Arguments

`A1,...,AN` — Input arrays

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array

Input arrays, specified as symbolic variables, vectors, matrices, or multidimensional arrays.

See Also

`cat` | `vertcat`

Introduced before R2006a

hypergeom

Hypergeometric function

Syntax

```
hypergeom(a,b,z)
```

Description

`hypergeom(a,b,z)` represents the generalized hypergeometric function on page 4-846.

Examples

Hypergeometric Function for Numeric and Symbolic Arguments

Depending on its arguments, `hypergeom` can return floating-point or exact symbolic results.

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

```
A = [hypergeom([1, 2], 2.5, 2),
hypergeom(1/3, [2, 3], pi),
hypergeom([1, 1/2], 1/3, 3*i)]
```

```
A =
  -1.2174 - 0.8330i
   1.2091 + 0.0000i
  -0.2028 + 0.2405i
```

Compute the hypergeometric function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `hypergeom` returns unresolved symbolic calls.

```
symA = [hypergeom([1, 2], 2.5, sym(2)),
hypergeom(1/3, [2, 3], sym(pi)),
hypergeom([1, 1/2], sym(1/3), 3*i)]

symA =
     hypergeom([1, 2], 5/2, 2)
    hypergeom(1/3, [2, 3], pi)
 hypergeom([1/2, 1], 1/3, 3i)
```

Use `vpa` to approximate symbolic results with the required number of digits:

```
vpa(symA,10)

ans =
   - 1.21741893 - 0.8330405509i
                    1.209063189
 - 0.2027516975 + 0.2405013423i
```

Special Values

The hypergeometric function has special values for some parameters:

```
syms a b c d x
hypergeom([], [], x)
hypergeom([a, b, c, d], [a, b, c, d], x)
hypergeom(a, [], x)

ans =
exp(x)

ans =
exp(x)

ans =
1/(1 - x)^a
```

Any hypergeometric function, evaluated at 0, has the value 1:

```
syms a b c d
hypergeom([a, b], [c, d], 0)

ans =
1
```

If, after cancelling identical parameters in the first two arguments, the list of upper parameters contains 0, the resulting hypergeometric function is constant with the value 1. For details, see "Algorithms" on page 4-847.

```
hypergeom([0, 0, 2, 3], [a, 0, 4], x)
```

```
ans =
1
```

If, after canceling identical parameters in the first two arguments, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, the hypergeometric function is a polynomial. If all parameters as well as the argument `x` are numeric, a corresponding explicit value is returned.

```
hypergeom([(-4), -2 , 3], [-3, 1, 4], x*pi*sqrt(2))
```

```
ans =
(6*pi^2*x^2)/5 - 2*2^(1/2)*pi*x + 1
```

Hypergeometric functions also reduce to other special functions for some parameters:

```
hypergeom([1], [a], x)
hypergeom([a], [a, b], x)
```

```
ans =
(exp(x/2)*whittakerM(1 - a/2, a/2 - 1/2, -x))/(-x)^(a/2)

ans =
 x^(1/2 - b/2)*gamma(b)*besseli(b - 1, 2*x^(1/2))
```

Handling Expressions That Contain Hypergeometric Functions

Many functions, such as `diff` and `taylor`, can handle expressions containing `hypergeom`.

Differentiate this expression containing hypergeometric function:

```
syms a b c d x
diff(1/x*hypergeom([a, b], [c, d], x), x)
```

```
ans =
(a*b*hypergeom([a + 1, b + 1], [c + 1, d + 1], x))/(c*d*x)...
 - hypergeom([a, b], [c, d], x)/x^2
```

Compute the Taylor series of this hypergeometric function:

```
taylor(hypergeom([1, 2], [3], x), x)

ans =
(2*x^5)/7 + x^4/3 + (2*x^3)/5 + x^2/2 + (2*x)/3 + 1
```

Input Arguments

a — Upper parameters of hypergeometric function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Upper parameters of hypergeometric function, specified as a number, variable, symbolic expression, symbolic function, or vector.

b — Lower parameters of hypergeometric function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Lower parameters of hypergeometric function, specified as a number, variable, symbolic expression, symbolic function, or vector.

z — Argument of hypergeometric function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Argument of hypergeometric function, specified as a number, variable, symbolic expression, symbolic function, or vector. If `z` is a vector, `hypergeom(a,b,z)` is evaluated element-wise.

Definitions

Generalized Hypergeometric Function

The generalized hypergeometric function of order p, q is defined as follows:

$$ {}_pF_q\left(a;b;z\right) = {}_pF_q\left(a_1,\ldots,a_j,\ldots,a_p;b_1,\ldots,b_k,\ldots,b_q;z\right) = \sum_{n=0}^{\infty}\left(\frac{\left(a_1\right)_n\cdots\left(a_j\right)_n\cdots\left(a_p\right)_n}{\left(b_1\right)_n\cdots\left(b_k\right)_n\cdots\left(b_q\right)_n}\right)\left(\frac{z^n}{n!}\right). $$

Here $a = [a_1,a_2,...,a_p]$ and $b = [b_1,b_2,...,b_q]$ are vectors of lengths p and q, respectively.

$(a)_k$ and $(b)_k$ are Pochhammer symbols.

For empty vectors a and b, `hypergeom` is defined as follows:

$$_0F_q\left(;b;z\right) = \sum_{k=0}^{\infty} \frac{1}{(b_1)_k\,(b_2)_k \cdots (b_q)_k}\left(\frac{z^k}{k!}\right)$$

$$_pF_0\left(a;;z\right) = \sum_{k=0}^{\infty} (a_1)_k\,(a_2)_k \cdots (a_p)_k\left(\frac{z^k}{k!}\right)$$

$$_0F_0\left(;;z\right) = \sum_{k=0}^{\infty}\left(\frac{z^k}{k!}\right) = e^z.$$

Pochhammer Symbol

The Pochhammer symbol is defined as follows:

$$(x)_n = \frac{\Gamma(x+n)}{\Gamma(x)}.$$

If n is a positive integer, then $(x)_n = x(x + 1)...(x + n - 1)$.

Algorithms

The hypergeometric function is

$$_pF_q\left(a;b;z\right) = {}_pF_q\left(a_1,\ldots,a_j,\ldots,a_p;b_1,\ldots,b_k,\ldots,b_q;z\right) = \sum_{n=0}^{\infty}\left(\frac{(a_1)_n\cdots(a_j)_n\cdots(a_p)_n}{(b_1)_n\cdots(b_k)_n\cdots(b_q)_n}\right)\left(\frac{z^n}{n!}\right).$$

- The hypergeometric function has convergence criteria:
  - Converges if $p \le q$ and $|z| < \infty$.
  - Converges if $p = q + 1$ and $|z| < 1$. For $|z| >= 1$, the series diverges, and is defined by analytic continuation.
  - Diverges if $p > q + 1$ and $z \neq 0$. Here, the series is defined by an asymptotic expansion of $_pF_q(a;b;z)$ around $z = 0$. The branch cut is the positive real axis.

- The function is a polynomial, called the hypergeometric polynomial, if any a_j is a nonpositive integer.
- The function is undefined:
  - If any b_k is a nonpositive integer such that $b_k > a_j$ where a_j is also a nonpositive integer, because division by 0 occurs
  - If any b_k is a nonpositive integer and no a_j is a nonpositive integer
- The function has reduced order when upper and lower parameter values are equal and cancel. If r values of the upper and lower parameters are equal (that is, $a = [a_1, \ldots, a_{p-r}, c_1, \ldots, c_r]$, $b = [b_1, \ldots, b_{q-r}, c_1, \ldots, c_r]$), then the order (p, q) of ${}_pF_q(a;b;z)$ is reduced to $(p - r, q - r)$:

  $${}_pF_q\left(a_1,\ldots,a_{p-r},c_1,\ldots,c_r;b_1,\ldots,b_{q-r},c_1,\ldots,c_r;z\right) =$$
  $${}_{p-r}F_{q-r}\left(a_1,\ldots,a_{p-r};b_1,\ldots,b_{q-r};z\right)$$

  This rule applies even if any c_i is zero or a negative integer [2].
- ${}_pF_q(a;b;z)$ is symmetric. That is, it does not depend on the order $a_1, a_2, \ldots$ in a or $b_1, b_2, \ldots$ in b.
-

  $U(z) = {}_pF_q(a;b;z)$ satisfies the differential equation in z

  $$\left[\delta(\delta+b-1) - z(\delta+a)\right]U(z) = 0, \quad \delta = z\frac{\partial}{\partial z}.$$

  Here, $(\delta + a)$ represents

  $$\prod_{i=1}^{p}(\delta + a_i).$$

  And $(\delta + b)$ represents

  $$\prod_{j=1}^{q}(\delta + b_j).$$

  Thus, the order of this differential equation is $max(p, q + 1)$, and the hypergeometric function is only one of its solutions. If $p < q + 1$, this differential equation has a regular singularity at $z = 0$ and an irregular singularity at $z = \infty$. If $p = q + 1$, the points $z = 0$, $z = 1$, and $z = \infty$ are regular singularities, which explains the convergence properties of the hypergeometric series.

- The hypergeometric function has these special values:
  - ${}_pF_p(a;a;z) = {}_0F_0(;;z) = \mathrm{e}^z$.
  - ${}_pF_q(a;b;z) = 1$ if the list of upper parameters a contains more `0`s than the list of lower parameters b.
  - ${}_pF_q(a;b;0) = 1$.

References

[1] 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.

[2] Luke, Y.L. "The Special Functions and Their Approximations", Vol. 1, Academic Press, New York, 1969.

[3] Prudnikov, A.P., Yu.A. Brychkov, and O.I. Marichev, "Integrals and Series", Vol. 3: More Special Functions, Gordon and Breach, 1990.

See Also

`kummerU` | `meijerG` | `whittakerM` | `whittakerW`

Introduced before R2006a

ifourier

Inverse Fourier transform

Syntax

```
ifourier(F)
ifourier(F,transVar)
ifourier(F,var,transVar)
```

Description

`ifourier(F)` returns the “Inverse Fourier Transform” on page 4-854 of `F`. By default, the independent variable is `w` and the transformation variable is `x`. If `F` does not contain `w`, `ifourier` uses the function `symvar`.

`ifourier(F,transVar)` uses the transformation variable `transVar` instead of `x`.

`ifourier(F,var,transVar)` uses the independent variable `var` and the transformation variable `transVar` instead of `w` and `x`, respectively.

Examples

Inverse Fourier Transform of Symbolic Expression

Compute the inverse Fourier transform of `exp(-w^2/4)`. By default, the inverse transform is in terms of `x`.

```
syms w
F = exp(-w^2/4);
ifourier(F)
```

```
ans =
exp(-x^2)/pi^(1/2)
```

Default Independent Variable and Transformation Variable

Compute the inverse Fourier transform of `exp(-w^2-a^2)`. By default, the independent and transformation variables are `w` and `x`, respectively.

```
syms a w t
F = exp(-w^2-a^2);
ifourier(F)
```

```
ans =
exp(- a^2 - x^2/4)/(2*pi^(1/2))
```

Specify the transformation variable as `t`. If you specify only one variable, that variable is the transformation variable. The independent variable is still `w`.

```
ifourier(F,t)
```

```
ans =
exp(- a^2 - t^2/4)/(2*pi^(1/2))
```

Inverse Fourier Transforms Involving Dirac and Heaviside Functions

Compute the inverse Fourier transform of expressions in terms of Dirac and Heaviside functions.

```
syms t w
ifourier(dirac(w), w, t)
```

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

```
f = 2*exp(-abs(w))-1;
ifourier(f,w,t)
```

```
ans =
-(2*pi*dirac(t) - 4/(t^2 + 1))/(2*pi)
```

```
f = exp(-w)*heaviside(w);
ifourier(f,w,t)
```

```
ans =
-1/(2*pi*(- 1 + t*1i))
```

Specify Parameters of Inverse Fourier Transform

Specify parameters of the inverse Fourier transform.

Compute the inverse Fourier transform of this expression using the default values of the Fourier parameters `c = 1`, `s = -1`. For details, see “Inverse Fourier Transform” on page 4-854.

```
syms t w
f = -(sqrt(sym(pi))*w*exp(-w^2/4)*i)/2;
ifourier(f,w,t)

ans =
t*exp(-t^2)
```

Change the Fourier parameters to `c = 1`, `s = 1` by using `sympref`, and compute the transform again. The sign of the result changes.

```
sympref('FourierParameters',[1 1]);
ifourier(f,w,t)

ans =
-t*exp(-t^2)
```

Change the Fourier parameters to `c = 1/(2*pi)`, `s = 1`. The result changes.

```
sympref('FourierParameters', [1/(2*sym(pi)) 1]);
ifourier(f,w,t)

ans =
-2*t*pi*exp(-t^2)
```

Preferences set by `sympref` persist through your current and future MATLAB sessions. Restore the default values of `c` and `s` by setting `FourierParameters` to `'default'`.

```
sympref('FourierParameters','default');
```

Inverse Fourier Transform of Array Inputs

Find the inverse Fourier transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `ifourier` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x), 1; sin(y), i*z];
vars = [w, x; y, z];
transVars = [a, b; c, d];
ifourier(M,vars,transVars)

ans =
[                         exp(x)*dirac(a),    dirac(b)]
[ (dirac(c - 1)*1i)/2 - (dirac(c + 1)*1i)/2, dirac(1, d)]
```

If `ifourier` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
ifourier(x,vars,transVars)

ans =
[ x*dirac(a), -dirac(1, b)*1i]
[ x*dirac(c),      x*dirac(d)]
```

If Inverse Fourier Transform Cannot Be Found

If `ifourier` cannot transform the input, then it returns an unevaluated call to `fourier`.

```
syms F(w) t
f = ifourier(F,w,t)

f =
fourier(F(w), w, -t)/(2*pi)
```

Input Arguments

`F` — Input
symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

`x` (default) | symbolic variable

Independent variable, specified as a symbolic variable. This variable is often called the "frequency variable." If you do not specify the variable, then `ifourier` uses `w`. If `F` does not contain `w`, then `ifourier` uses the function `symvar` to determine the independent variable.

`transVar` — Transformation variable

`x` (default) | `t` | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. It is often called the "time variable" or "space variable." By default, `ifourier` uses `x`. If `x` is the independent variable of `F`, then `ifourier` uses `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

$$f(x) = \frac{|s|}{2\pi c} \int_{-\infty}^{\infty} F(w) e^{-iswx} dw.$$

c and s are parameters of the inverse Fourier transform. The `ifourier` function uses $c = 1$, $s = -1$.

Tips

- If any argument is an array, then `ifourier` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.

- The toolbox computes the inverse Fourier transform via the 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, then it returns results in terms of the Fourier transform.
- To compute the Fourier transform, use `fourier`.

References

[1] Oberhettinger, F. "Tables of Fourier Transforms and Fourier Transforms of Distributions." Springer, 1990.

See Also

`fourier` | `ilaplace` | `iztrans` | `laplace` | `sympref` | `ztrans`

Topics

"Fourier and Inverse Fourier Transforms" on page 2-220

Introduced before R2006a

igamma

Incomplete gamma function

Syntax

```
igamma(nu,z)
```

Description

`igamma(nu,z)` returns the incomplete gamma function.

`igamma` uses the definition of the upper incomplete gamma function on page 4-858. The MATLAB `gammainc` function uses the definition of the lower incomplete gamma function on page 4-858, `gammainc(z, nu) = 1 - igamma(nu, z)/gamma(nu)`. The order of input arguments differs between these functions.

Examples

Compute Incomplete Gamma Function for Numeric and Symbolic Arguments

Depending on its arguments, `igamma` returns floating-point or exact symbolic results.

Compute the incomplete gamma function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
A = [igamma(0, 1), igamma(3, sqrt(2)), igamma(pi, exp(1)), igamma(3, Inf)]
```

```
A =
    0.2194    1.6601    1.1979         0
```

Compute the incomplete gamma function for the numbers converted to symbolic objects:

```
symA = [igamma(sym(0), 1), igamma(3, sqrt(sym(2))),...
igamma(sym(pi), exp(sym(1))), igamma(3, sym(Inf))]
```

```
symA =
[ -ei(-1), exp(-2^(1/2))*(2*2^(1/2) + 4), igamma(pi, exp(1)), 0]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.21938393439552027367716377546012,...
1.6601049038903044104826564373576,...
1.1979302081330828196865548471769,...
0]
```

Compute Lower Incomplete Gamma Function

`igamma` is implemented according to the definition of the upper incomplete gamma function. If you want to compute the lower incomplete gamma function, convert results returned by `igamma` as follows.

Compute the lower incomplete gamma function for these arguments using the MATLAB `gammainc` function:

```
A = [-5/3, -1/2, 0, 1/3];
gammainc(A, 1/3)

ans =
   1.1456 + 1.9842i   0.5089 + 0.8815i   0.0000 + 0.0000i   0.7175 + 0.0000i
```

Compute the lower incomplete gamma function for the same arguments using `igamma`:

```
1 - igamma(1/3, A)/gamma(1/3)

ans =
   1.1456 + 1.9842i   0.5089 + 0.8815i   0.0000 + 0.0000i   0.7175 + 0.0000i
```

If one or both arguments are complex numbers, use `igamma` to compute the lower incomplete gamma function. `gammainc` does not accept complex arguments.

```
1 - igamma(1/2, i)/gamma(1/2)

ans =
   0.9693 + 0.4741i
```

Input Arguments

nu — Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

z — Input
symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Upper Incomplete Gamma Function

The following integral defines the upper incomplete gamma function:

$$\Gamma(\eta,z)=\int_z^\infty t^{\eta-1}e^{-t}\,dt$$

Lower Incomplete Gamma Function

The following integral defines the lower incomplete gamma function:

$$\gamma(\eta,z)=\int_0^z t^{\eta-1}e^{-t}\,dt$$

Tips

- The MATLAB `gammainc` function does not accept complex arguments. For complex arguments, use `igamma`.

- `gammainc(z, nu) = 1 - igamma(nu, z)/gamma(nu)` represents the lower incomplete gamma function in terms of the upper incomplete gamma function.
- `igamma(nu,z) = gamma(nu)(1 - gammainc(z, nu))` represents the upper incomplete gamma function in terms of the lower incomplete gamma function.
- `gammainc(z, nu, 'upper') = igamma(nu, z)/gamma(nu).`

See Also

`ei` | `erfc` | `factorial` | `gamma` | `gammainc` | `int`

Introduced in R2014a

ilaplace

Inverse Laplace transform

Syntax

```
ilaplace(F)
ilaplace(F,transVar)
ilaplace(F,var,transVar)
```

Description

`ilaplace(F)` returns the “Inverse Laplace Transform” on page 4-864 of `F`. By default, the independent variable is `s` and the transformation variable is `t`. If `F` does not contain `s`, `ilaplace` uses the function `symvar`.

`ilaplace(F,transVar)` uses the transformation variable `transVar` instead of `t`.

`ilaplace(F,var,transVar)` uses the independent variable `var` and transformation variable `transVar` instead of `s` and `t`, respectively.

Examples

Inverse Laplace Transform of Symbolic Expression

Compute the inverse Laplace transform of `1/s^2`. By default, the inverse transform is in terms of `t`.

```
syms s
F = 1/s^2;
ilaplace(F)
```

```
ans =
t
```

Default Independent Variable and Transformation Variable

Compute the inverse Laplace transform of `1/(s-a)^2`. By default, the independent and transformation variables are `s` and `t`, respectively.

```
syms a s
F = 1/(s-a)^2;
ilaplace(F)

ans =
t*exp(a*t)
```

Specify the transformation variable as `x`. If you specify only one variable, that variable is the transformation variable. The independent variable is still `s`.

```
syms x
ilaplace(F,x)

ans =
x*exp(a*x)
```

Specify both the independent and transformation variables as `a` and `x` in the second and third arguments, respectively.

```
ilaplace(F,a,x)

ans =
x*exp(s*x)
```

Inverse Laplace Transforms Involving Dirac and Heaviside Functions

Compute the following inverse Laplace transforms that involve the Dirac and Heaviside functions:

```
syms s t
ilaplace(1,s,t)

ans =
dirac(t)
```

```
F = exp(-2*s)/(s^2+1);
ilaplace(F,s,t)

ans =
heaviside(t - 2)*sin(t - 2)
```

Inverse Laplace Transform of Array Inputs

Find the inverse Laplace transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `ilaplace` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x) 1; sin(y) i*z];
vars = [w x; y z];
transVars = [a b; c d];
ilaplace(M,vars,transVars)

ans =
[        exp(x)*dirac(a),       dirac(b)]
[ ilaplace(sin(y), y, c), dirac(1, d)*1i]
```

If `ilaplace` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
syms w x y z a b c d
ilaplace(x,vars,transVars)

ans =
[ x*dirac(a), dirac(1, b)]
[ x*dirac(c),  x*dirac(d)]
```

If Inverse Laplace Transform Cannot Be Found

If `ilaplace` cannot compute the inverse transform, then it returns an unevaluated call to `ilaplace`.

```
syms F(s) t
F(s) = exp(s);
f = ilaplace(F,s,t)
```

```
f =
ilaplace(exp(s), s, t)
```

Return the original expression by using `laplace`.

```
laplace(f,t,s)
```

```
ans =
exp(s)
```

Inverse Laplace Transform of Symbolic Function

Compute the Inverse Laplace transform of symbolic functions. When the first argument contains symbolic functions, then the second argument must be a scalar.

```
syms f1(x) f2(x) a b
f1(x) = exp(x);
f2(x) = x;
ilaplace([f1 f2],x,[a b])
```

```
ans =
[ ilaplace(exp(x), x, a), dirac(1, b)]
```

Input Arguments

`F` — Input

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

s (default) | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Independent variable, specified as a symbolic variable, expression, vector, or matrix. This variable is often called the "complex frequency variable." If you do not specify the variable, then `ilaplace` uses `s`. If `F` does not contain `s`, then `ilaplace` uses the function `symvar` to determine the independent variable.

`transVar` — Transformation variable

`t` (default) | `x` | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. It is often called the "time variable" or "space variable." By default, `ilaplace` uses `t`. If `t` is the independent variable of `F`, then `ilaplace` uses `x`.

Definitions

Inverse Laplace Transform

The inverse Laplace transform $f = f(t)$ of $F = F(s)$ is:

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

Here, c is a suitable complex number.

Tips

- If any argument is an array, then `ilaplace` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.
- To compute the direct Laplace transform, use `laplace`.

See Also

`fourier` | `ifourier` | `iztrans` | `laplace` | `ztrans`

Topics

“Solve Differential Equations Using Laplace Transform” on page 2-225

Introduced before R2006a

imag

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`.

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(2*exp(1 + sym(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`:

```
syms x
A = [-1 + sym(i), sinh(x); exp(10 + sym(7)*i), exp(sym(pi)*i)];
imag(A)
```

```
ans =
[              1, imag(sinh(x))]
[ exp(10)*sin(7),             0]
```

Tips

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

Alternatives

You can compute the imaginary part of `z` via the conjugate: `imag(z)= (z - conj(z))/2i`.

See Also

`conj` | `in` | `real` | `sign` | `signIm`

Introduced before R2006a

in

Numeric type of symbolic input

Syntax

`in(x,type)`

Description

`in(x,type)` expresses the logical condition that `x` is of the specified `type`.

Examples

Express Condition on Symbolic Variable or Expression

The syntax `in(x,type)` expresses the condition that `x` is of the specified `type`. Express the condition that `x` is of type `Real`.

```
syms x
cond = in(x,'real')
```

```
cond =
in(x, 'real')
```

Evaluate the condition using `isAlways`. Because `isAlways` cannot determine the condition, it issues a warning and returns logical `0` (`false`).

```
isAlways(cond)
```

```
Warning: Unable to prove 'in(x, 'real')'.

ans =
  logical
     0
```

Assume the condition `cond` is true using `assume`, and evaluate the condition again. The `isAlways` function returns logical `1` (`true`) indicating that the condition is true.

```
assume(cond)
isAlways(cond)
```

```
ans =
  logical
     1
```

Clear the assumption on `x` to use it in further computations.

```
syms x clear
```

Express Conditions in Output

Functions such as `solve` use `in` in output to express conditions.

Solve the equation `sin(x) == 0` using `solve`. Set the option `ReturnConditions` to `true` to return conditions on the solution. The `solve` function uses `in` to express the conditions.

```
syms x
[solx, params, conds] = solve(sin(x) == 0,'ReturnConditions',true)
```

```
solx =
pi*k

params =
k

conds =
in(k, 'integer')
```

The solution is `pi*k` with parameter `k` under the condition `in(k,'integer')`. You can use this condition to set an assumption for further computations. Under the assumption, `solve` returns only integer values of `k`.

```
assume(conds)
k = solve(solx > 0, solx < 5*pi, params)
```

```
k =
 1
 2
 3
 4
```

To find the solutions corresponding to these values of k, use subs to substitute for k in solx.

```
subs(solx,k)
```

```
ans =
   pi
 2*pi
 3*pi
 4*pi
```

Clear the assumption on k to use it in further computations.

```
assume(params, 'clear')
```

Test if Elements of Symbolic Matrix Are Rational

Create symbolic matrix M.

```
syms x y z
M = sym([1.22 i x; sin(y) 3*x 0; Inf sqrt(3) sym(22/7)])
```

```
M =
[  61/50,      1i,    x]
[ sin(y),     3*x,    0]
[    Inf, 3^(1/2), 22/7]
```

Use isAlways to test if the elements of M are rational numbers. The in function acts on M element-by-element. Note that isAlways returns logical 0 (false) for statements that cannot be decided and issues a warning for those statements.

```
in(M,'rational')
```

```
ans =
[  in(61/50, 'rational'),      in(1i, 'rational'),    in(x, 'rational')]
[ in(sin(y), 'rational'),     in(3*x, 'rational'),    in(0, 'rational')]
[    in(Inf, 'rational'), in(3^(1/2), 'rational'), in(22/7, 'rational')]
```

```
isAlways(in(M,'rational'))
```

```
Warning: Unable to prove 'in(sin(y), 'rational')'.
Warning: Unable to prove 'in(3*x, 'rational')'.
```

```
Warning: Unable to prove 'in(x, 'rational')'.
ans =
  3×3 logical array
   1   0   0
   0   0   1
   0   0   1
```

Input Arguments

`x` — Input

symbolic number | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic expression | symbolic function

Input, specified as a symbolic number, vector, matrix, multidimensional array, expression, or function.

`type` — Type of input

`'real'` | `'positive'` | `'integer'` | `'rational'`

Type of input, specified as `'real'`, `'positive'`, `'integer'`, or `'rational'`.

See Also

`assume` | `assumeAlso` | `false` | `imag` | `isalways` | `isequal` | `isequaln` | `isfinite` | `isinf` | `piecewise` | `real` | `true`

Introduced in R2014b

incidenceMatrix

Find incidence matrix of system of equations

Syntax

```
A = incidenceMatrix(eqs,vars)
```

Description

`A = incidenceMatrix(eqs,vars)` for `m` equations `eqs` and `n` variables `vars` returns an `m`-by-`n` matrix `A`. Here, `A(i,j) = 1` if `eqs(i)` contains `vars(j)` or any derivative of `vars(j)`. All other elements of `A` are `0`s.

Examples

Incidence Matrix

Find the incidence matrix of a system of five equations in five variables.

Create the following symbolic vector `eqs` containing five symbolic differential equations.

```
syms y1(t) y2(t) y3(t) y4(t) y5(t) c1 c3
eqs = [diff(y1(t),t) == y2(t),...
       diff(y2(t),t) == c1*y1(t) + c3*y3(t),...
       diff(y3(t),t) == y2(t) + y4(t),...
       diff(y4(t),t) == y3(t) + y5(t),...
       diff(y5(t),t) == y4(t)];
```

Create the vector of variables. Here, `c1` and `c3` are symbolic parameters (not variables) of the system.

```
vars = [y1(t), y2(t), y3(t), y4(t), y5(t)];
```

Find the incidence matrix `A` for the equations `eqs` and with respect to the variables `vars`.

```
A = incidenceMatrix(eqs, vars)

A =
     1     1     0     0     0
     1     1     1     0     0
     0     1     1     1     0
     0     0     1     1     1
     0     0     0     1     1
```

Input Arguments

`eqs` — Equations
vector of symbolic equations | vector of symbolic expressions

Equations, specified as a vector of symbolic equations or expressions.

`vars` — Variables
vector of symbolic variables | vector of symbolic functions | vector of symbolic function calls

Variables, specified as a vector of symbolic variables, symbolic functions, or function calls, such as `x(t)`.

Output Arguments

`A` — Incidence matrix
matrix of double-precision values

Incidence matrix, returned as a matrix of double-precision values.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies` | `spy`

Introduced in R2014b

int

Definite and indefinite integrals

Syntax

```
int(expr,var)
int(expr,var,a,b)
int(___,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`. If `expr` is a constant, then the default variable is `x`.

`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`. If `expr` is a constant, then the default variable is `x`.

`int(expr,var,[a,b])`, `int(expr,var,[a b])`, and `int(expr,var,[a;b])` are equivalent to `int(expr,var,a,b)`.

`int(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Indefinite Integral of Univariate Expression

Find an indefinite integral of this univariate expression:

```
syms x
int(-2*x/(1 + x^2)^2)
```

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

Indefinite Integrals of Multivariate Expression

Find indefinite integrals of this multivariate expression with respect to the variables `x` and `z`:

```
syms x z
int(x/(1 + z^2), x)
int(x/(1 + z^2), z)

ans =
x^2/(2*(z^2 + 1))

ans =
x*atan(z)
```

If you do not specify the integration variable, `int` uses the variable returned by `symvar`. For this expression, `symvar` returns `x`:

```
symvar(x/(1 + z^2), 1)

ans =
x
```

Definite Integrals of Univariate Expressions

Integrate this expression from `0` to `1`:

```
syms x
int(x*log(1 + x), 0, 1)

ans =
1/4
```

Integrate this expression from `sin(t)` to `1` specifying the integration range as a vector:

```
syms x t
int(2*x, [sin(t), 1])

ans =
cos(t)^2
```

Integrals of Matrix Elements

Find indefinite integrals for the expressions listed as the elements of a matrix:

```
syms a x t z
int([exp(t), exp(a*t); sin(t), cos(t)])
```

```
ans =
[  exp(t), exp(a*t)/a]
[ -cos(t),     sin(t)]
```

Apply IgnoreAnalyticConstraints

Compute this indefinite integral. By default, `int` uses strict mathematical rules. These rules do not let `int` rewrite `asin(sin(x))` and `acos(cos(x))` as `x`.

```
syms x
int(acos(sin(x)), x)
```

```
ans =
x*acos(sin(x)) + x^2/(2*sign(cos(x)))
```

If you want a simple practical solution, try `IgnoreAnalyticConstraints`:

```
int(acos(sin(x)), x, 'IgnoreAnalyticConstraints', true)
```

```
ans =
-(x*(x - pi))/2
```

Ignore Special Cases

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)
```

Find Cauchy Principal Value

Compute this definite integral, where the integrand has a pole in the interior of the interval of integration. Mathematically, this integral is not defined.

```
syms x
int(1/(x - 1), x, 0, 2)
```

```
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)
```

```
ans =
0
```

Approximate Indefinite Integrals

If `int` cannot compute a closed form of an integral, it returns an unresolved integral:

```
syms x
F = sin(sinh(x));
int(F, x)
```

```
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
```

Approximate Definite Integrals

Compute this definite integral:

```
syms x
F = int(cos(x)/sqrt(1 + x^2), x, 0, 10)
```

```
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 five significant digits:

```
vpa(F, 5)
```

```
ans =
0.37571
```

Input Arguments

`expr` — Integrand
symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic number

Integrand, specified as a symbolic expression or function, a constant, or a vector or matrix of symbolic expressions, functions, or constants.

`var` — Integration variable
symbolic variable

Integration variable, specified as a symbolic variable. If you do not specify this variable, `int` uses the default variable determined by `symvar(expr,1)`. If `expr` is a constant, then the default variable is `x`.

`a` — Lower bound
number | symbolic number | symbolic variable | symbolic expression | symbolic function

Lower bound, specified as a number, symbolic number, variable, expression or function (including expressions and functions with infinities).

`b` — Upper bound
number | symbolic number | symbolic variable | symbolic expression | symbolic function

Upper bound, specified as a number, symbolic number, variable, expression or function (including expressions and functions 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`.

Example: `'IgnoreAnalyticConstraints',true` specifies that `int` applies purely algebraic simplifications to the integrand.

`IgnoreAnalyticConstraints` — Indicator for applying purely algebraic simplifications to integrand
`false` (default) | `true`

Indicator for applying purely algebraic simplifications to integrand, specified as `true` or `false`. 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.

`IgnoreSpecialCases` — Indicator for ignoring special cases
`false` (default) | `true`

Indicator for ignoring special cases, specified as `true` or `false`. 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.

`PrincipalValue` — Indicator for returning principal value
`false` (default) | `true`

Indicator for returning principal value, specified as `true` or `false`. If the value is `true`, compute the Cauchy principal value of the integral.

Tips

- In contrast to differentiation, symbolic integration is a more complicated task. If `int` cannot compute an integral of an expression, one of these 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 returns an unresolved integral.

  Try approximating such integrals by using one of these 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.
- For indefinite integrals, `int` implicitly assumes that the integration variable `var` is real. For definite integrals, `int` restricts the integration variable `var` to the specified integration interval. If one or both integration bounds `a` and `b` are not numeric, `int` assumes that `a <= b` unless you explicitly specify otherwise.

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, then $f(g(x)) = x$ is assumed to be valid for all complex values x. In particular:
  - $\log(e^x) = x$
  - $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
  - $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
  - $\text{lambertW}_k(x \cdot e^x) = x$ for all values of k

See Also

`diff` | `functionalDerivative` | `symprod` | `symsum` | `symvar` | `vpaintegral`

Topics

"Integration" on page 2-56

Introduced before R2006a

int8int16int32int64

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`

Function	Output Range	Output Type	Bytes per Element	Output Class
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

See Also

`double` | `single` | `sym` | `uint16` | `uint32` | `uint64` | `uint8` | `vpa`

Introduced before R2006a

inv

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)

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)

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)))
```

```
ans =
[   16,  -120,   240,  -140]
[ -120,  1200, -2700,  1680]
[  240, -2700,  6480, -4200]
[ -140,  1680, -4200,  2800]
```

See Also

`det` | `eig` | `rank`

Introduced before R2006a

isAlways

Check whether equation or inequality holds for all values of its variables

---

Note `isAlways` issues a warning when returning false for undecidable inputs. To suppress the warning, set the `Unknown` option to `false` as `isAlways(cond,'Unknown','false')`. For details, see “Handle Output for Undecidable Conditions”.

Syntax

```
isAlways(cond)
isAlways(cond,Name,Value)
```

Description

`isAlways(cond)` checks if the condition `cond` is valid for all possible values of the symbolic variables in `cond`. When verifying `cond`, the `isAlways` function considers all assumptions 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.

Examples

Test Conditions

Check if this inequality is valid for all values of `x`.

```
syms x
isAlways(abs(x) >= 0)
```

```
ans =
  logical
   1
```

`isAlways` returns logical `1` (`true`) indicating that the inequality `abs(x) >= 0` is valid for all values of `x`.

Check if this equation is valid for all values of `x`.

```
isAlways(sin(x)^2 + cos(x)^2 == 1)
```

```
ans =
  logical
   1
```

`isAlways` returns logical `1` (`true`) indicating that the inequality is valid for all values of `x`.

Test if One of Several Conditions Is Valid

Check if at least one of these two conditions is valid. To check if at least one of several conditions is valid, combine them using the logical operator `or` or its shortcut `|`.

```
syms x
isAlways(sin(x)^2 + cos(x)^2 == 1 | x^2 > 0)
```

```
ans =
  logical
   1
```

Check if both conditions are valid. To check if several conditions are valid, combine them using the logical operator `and` or its shortcut `&`.

```
isAlways(sin(x)^2 + cos(x)^2 == 1 & abs(x) > 2*abs(x))
```

```
ans =
  logical
   0
```

Handle Output for Undecidable Conditions

Test this condition. When `isAlways` cannot determine if the condition is valid, it returns logical `0` (`false`) and issues a warning by default.

```
syms x
isAlways(2*x >= x)

Warning: Unable to prove 'x <= 2*x'.
ans =
  logical
   0
```

To change this default behavior, use `Unknown`. For example, specify `Unknown` as `false` to suppress the warning and make `isAlways` return logical `0` (`false`) if it cannot determine the validity of the condition.

```
isAlways(2*x >= x,'Unknown','false')

ans =
  logical
   0
```

Instead of `false`, you can also specify `error` to return an error, and `true` to return logical `1` (`true`).

Test Conditions with Assumptions

Check this inequality under the assumption that `x` is negative. When `isAlways` tests 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 =
  logical
   1
```

For further computations, clear the assumption on `x`.

```
syms x clear
```

Input Arguments

`cond` — Condition to check
symbolic condition | vector of symbolic conditions | matrix of symbolic conditions | multidimensional array of symbolic conditions

Condition to check, specified as a symbolic condition, or a vector, matrix, or multidimensional array of symbolic conditions.

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`.

Example: `isAlways(cond,'Unknown',true)` makes `isAlways` return logical `1` (`true`) when the specified condition cannot be decided.

`Unknown` — Return value for undecidable condition
falseWithWarning (default) | false | true | error

Return value for an undecidable condition, specified as the comma-separated pair of `'Unknown'` and one of these values.

`falseWithWarning` (default)	On undecidable inputs, return logical `0` (`false`) and a warning that the condition cannot be proven.
`false`	On undecidable inputs, return logical `0` (`false`).
`true`	On undecidable inputs, return logical `1` (`true`).
`error`	On undecidable inputs, return an error.

See Also

`assume` | `assumeAlso` | `assumptions` | `in` | `isequal` | `isequaln` | `isfinite` | `isinf` | `isnan` | `piecewise` | `sym` | `syms`

Topics

“Use Assumptions on Symbolic Variables” on page 1-28
“Clear Assumptions and Reset the Symbolic Engine” on page 3-67

Introduced in R2012a

isequal

Test equality of symbolic inputs

Syntax

```
isequal(a,b)
isequal(a1,a2,...,aN)
```

Description

`isequal(a,b)` returns logical `1` (`true`) if `A` and `B` are the same size and their contents are of equal value. Otherwise, `isequal` returns logical `0` (`false`). `isequal` does not consider `NaN` (not a number) values equal. `isequal` recursively compares the contents of symbolic data structures and the properties of objects. If all contents in the respective locations are equal, `isequal` returns logical `1` (`true`).

`isequal(a1,a2,...,aN)` returns logical `1` (`true`) if all the inputs `a1,a2,...,aN` are equal.

Examples

Test Numbers for Equality

Test numeric or symbolic inputs for equality using `isequal`. If you compare numeric inputs against symbolic inputs, `isequal` returns `0` (`false`) because double and symbolic are distinct data types.

Test if `2` and `5` are equal. Because you are comparing doubles, the MATLAB `isequal` function is called. `isequal` returns `0` (`false`) as expected.

```
isequal(2,5)
```

```
ans =
  logical
   0
```

Test if the solution of the equation `cos(x) == -1` is `pi`. The `isequal` function returns `1` (`true`) meaning the solution is equal to `pi`.

```
syms x
sol = solve(cos(x) == -1, x);
isequal(sol,sym(pi))

ans =
  logical
   1
```

Compare the double and symbolic representations of `1`. `isequal` returns `0` (`false`) because double and symbolic are distinct data types. To return `1` (`true`) in this case, use `logical` instead.

```
usingIsEqual = isequal(pi,sym(pi))
usingLogical = logical(pi == sym(pi))

usingIsEqual =
  logical
   0
usingLogical =
  logical
   1
```

Test Symbolic Expressions for Equality

Test if `rewrite` correctly rewrites `tan(x)` as `sin(x)/cos(x)`. The `isequal` function returns `1` (`true`) meaning the rewritten result equals the test expression.

```
syms x
f = rewrite(tan(x),'sincos');
testf = sin(x)/cos(x);
isequal(f,testf)

ans =
  logical
   1
```

Test Symbolic Vectors and Matrices for Equality

Test vectors and matrices for equality using `isequal`.

Test if solutions of the quadratic equation found by `solve` are equal to the expected solutions. `isequal` function returns `1` (`true`) meaning the inputs are equal.

```
syms a b c x
eqn = a*x^2 + b*x + c;
Sol = solve(eqn, x);
testSol = [-(b+(b^2-4*a*c)^(1/2))/(2*a); -(b-(b^2-4*a*c)^(1/2))/(2*a)];
isequal(Sol,testSol)

ans =
  logical
   1
```

The Hilbert matrix is a special matrix that is difficult to invert accurately. If the inverse is accurately computed, then multiplying the inverse by the original Hilbert matrix returns the identity matrix.

Use this condition to symbolically test if the inverse of `hilb(20)` is correctly calculated. `isequal` returns `1` (`true`) meaning that the product of the inverse and the original Hilbert matrix is equal to the identity matrix.

```
H = sym(hilb(20));
prod = H*inv(H);
eye20 = sym(eye(20));
isequal(prod,eye20)

ans =
  logical
   1
```

Compare Inputs Containing `NaN`

Compare three vectors containing `NaN` (not a number). `isequal` returns logical `0` (`false`) because `isequal` does not treat `NaN` values as equal to each other.

```
syms x
A1 = [x NaN NaN];
A2 = [x NaN NaN];
```

```
A3 = [x NaN NaN];
isequal(A1, A2, A3)

ans =
  logical
   0
```

Input Arguments

`a,b` — Inputs to compare

numbers | vectors | matrices | multidimensional arrays | symbolic numbers | symbolic variables | symbolic vectors | symbolic matrices | symbolic multidimensional arrays | symbolic functions | symbolic expressions

Inputs to compare, specified as numbers, vectors, matrices, or multidimensional arrays or symbolic numbers, variables, vectors, matrices, multidimensional arrays, functions, or expressions.

`a1,a2,...,aN` — Several inputs to compare

numbers | vectors | matrices | multidimensional arrays | symbolic numbers | symbolic variables | symbolic vectors | symbolic matrices | symbolic multidimensional arrays | symbolic functions | symbolic expressions

Several inputs to compare, specified as numbers, vectors, matrices, or multidimensional arrays or symbolic numbers, variables, vectors, matrices, multidimensional arrays, functions, or expressions.

Tips

- When your inputs are not symbolic objects, the MATLAB `isequal` function is called. If one of the arguments is symbolic, then all other arguments are converted to symbolic objects before comparison, and the symbolic `isequal` function is called.

See Also

`in` | `isAlways` | `isequaln` | `isfinite` | `isinf` | `isnan` | `logical`

Introduced before R2006a

isequaln

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.

Examples

Compare Two Expressions

Use `isequaln` to compare these two expressions:

```
syms x
isequaln(abs(x), x)
```

```
ans =
  logical
   0
```

For positive `x`, these expressions are identical:

```
assume(x > 0)
isequaln(abs(x), x)
```

```
ans =
  logical
   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 =
  logical
   0
```

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 =
  logical
   1
```

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.

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.

See Also

`in` | `isAlways` | `isequal` | `isequaln` | `isfinite` | `isinf` | `isnan`

Introduced in R2013a

isfinite

Check whether symbolic array elements are finite

Syntax

```
isfinite(A)
```

Description

`isfinite(A)` returns an array of the same size as `A` containing logical `1`s (true) where the elements of `A` are finite, and logical `0`s (false) where they are not. For a complex number, `isfinite` returns `1` if both the real and imaginary parts of that number are finite. Otherwise, it returns `0`.

Examples

Determine Which Elements of Symbolic Array Are Finite Values

Using `isfinite`, determine which elements of this symbolic matrix are finite values:

```
isfinite(sym([pi NaN Inf; 1 + i Inf + i NaN + i]))
```

```
ans =
  2×3 logical array
     1     0     0
     1     0     0
```

Determine if Exact and Approximated Values Are Finite

Approximate these symbolic values with the 50-digit accuracy:

```
V = sym([pi, 2*pi, 3*pi, 4*pi]);
V_approx = vpa(V, 50);
```

The cotangents of the exact values are not finite:

```
cot(V)
isfinite(cot(V))

ans =
[ Inf, Inf, Inf, Inf]

ans =
  1×4 logical array
     0     0     0     0
```

Nevertheless, the cotangents of the approximated values are finite due to the round-off errors:

```
isfinite(cot(V_approx))

ans =
  1×4 logical array
   1   1   1   1
```

Input Arguments

`A` — Input value

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic array | symbolic vector | symbolic matrix

Input value, specified as a symbolic number, variable, expression, or function, or as an array, vector, or matrix of symbolic numbers, variables, expressions, functions.

Tips

- For any `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, or `isnan(A)` is `1` for each element.
- Elements of `A` are recognized as finite if they are
  - Not symbolic `NaN`
  - Not symbolic `Inf` or `-Inf`
  - Not sums or products containing symbolic infinities `Inf` or `-Inf`

See Also

`in` | `isAlways` | `isequal` | `isequaln` | `isinf` | `isnan`

Introduced in R2013b

isinf

Check whether symbolic array elements are infinite

Syntax

```
isinf(A)
```

Description

`isinf(A)` returns an array of the same size as `A` containing logical `1`s (true) where the elements of `A` are infinite, and logical `0`s (false) where they are not. For a complex number, `isinf` returns `1` if the real or imaginary part of that number is infinite or both real and imaginary parts are infinite. Otherwise, it returns `0`.

Examples

Determine Which Elements of Symbolic Array Are Infinite

Using `isinf`, determine which elements of this symbolic matrix are infinities:

```
isinf(sym([pi NaN Inf; 1 + i Inf + i NaN + i]))
```

```
ans =
  2×3 logical array
     0     0     1
     0     1     0
```

Determine if Exact and Approximated Values Are Infinite

Approximate these symbolic values with the 50-digit accuracy:

```
V = sym([pi, 2*pi, 3*pi, 4*pi]);
V_approx = vpa(V, 50);
```

The cotangents of the exact values are infinite:

```
cot(V)
isinf(cot(V))

ans =
[ Inf, Inf, Inf, Inf]

ans =
  1×4 logical array
     1     1     1     1
```

Nevertheless, the cotangents of the approximated values are not infinite due to the round-off errors:

```
isinf(cot(V_approx))

ans =
  1×4 logical array
     0     0     0     0
```

Input Arguments

`A` — Input value

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic array | symbolic vector | symbolic matrix

Input value, specified as a symbolic number, variable, expression, or function, or as an array, vector, or matrix of symbolic numbers, variables, expressions, functions.

Tips

- For any `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, or `isnan(A)` is `1` for each element.
- The elements of `A` are recognized as infinite if they are
  - Symbolic `Inf` or `-Inf`
  - Sums or products containing symbolic `Inf` or `-Inf` and not containing the value `NaN`.

See Also

`in` | `isAlways` | `isequal` | `isequaln` | `isfinite` | `isnan`

Introduced in R2013b

isLowIndexDAE

Check if differential index of system of equations is lower than 2

Syntax

`isLowIndexDAE(eqs,vars)`

Description

`isLowIndexDAE(eqs,vars)` checks if the system `eqs` of first-order semilinear differential algebraic equations (DAEs) has a low differential index. If the differential index of the system is `0` or `1`, then `isLowIndexDAE` returns logical `1` (true). If the differential index of `eqs` is higher than `1`, then `isLowIndexDAE` returns logical `0` (false).

The number of equations `eqs` must match the number of variables `vars`.

Examples

Check Differential Index of DAE System

Check if a system of first-order semilinear DAEs has a low differential index (`0` or `1`).

Create the following system of two differential algebraic equations. Here, `x(t)` and `y(t)` are the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x(t) y(t)
eqs = [diff(x(t),t) == x(t) + y(t), x(t)^2 + y(t)^2 == 1];
vars = [x(t), y(t)];
```

Use `isLowIndexDAE` to check the differential order of the system. The differential order of this system is `1`. For systems of index `0` and `1`, `isLowIndexDAE` returns `1` (`true`).

```
isLowIndexDAE(eqs, vars)

ans =
  logical
   1
```

Reduce Differential Index of DAE System

Check if the following DAE system has a low or high differential index. If the index is higher than `1`, then use `reduceDAEIndex` to reduce it.

Create the following system of two differential algebraic equations. Here, `x(t)`, `y(t)`, and `z(t)` are the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms  x(t) y(t) z(t) f(t)
eqs = [diff(x(t),t) == x(t) + z(t),...
       diff(y(t),t) == f(t), x(t) == y(t)];
vars = [x(t), y(t), z(t)];
```

Use `isLowIndexDAE` to check the differential index of the system. For this system `isLowIndexDAE` returns `0` (`false`). This means that the differential index of the system is `2` or higher.

```
isLowIndexDAE(eqs, vars)

ans =
  logical
   0
```

Use `reduceDAEIndex` to rewrite the system so that the differential index is `1`. Calling this function with four output arguments also shows the differential index of the original system. The new system has one additional state variable, `Dyt(t)`.

```
[newEqs, newVars, ~, oldIndex] = reduceDAEIndex(eqs, vars)

newEqs =
 diff(x(t), t) - z(t) - x(t)
              Dyt(t) - f(t)
                x(t) - y(t)
      diff(x(t), t) - Dyt(t)
```

```
newVars =
   x(t)
   y(t)
   z(t)
 Dyt(t)

oldIndex =
     2
```

Check if the differential order of the new system is lower than `2`.

```
isLowIndexDAE(newEqs, newVars)
```

```
ans =
  logical
   1
```

Input Arguments

`eqs` — System of first-order semilinear differential algebraic equations

vector of symbolic equations | vector of symbolic expressions

System of first-order semilinear differential algebraic equations, specified as a vector of symbolic equations or expressions.

`vars` — State variables

vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]`

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

“Solve Differential Algebraic Equations (DAEs)” on page 2-193

Introduced in R2014b

isnan

Check whether symbolic array elements are `NaN`s

Syntax

```
isnan(A)
```

Description

`isnan(A)` returns an array of the same size as `A` containing logical `1`s (true) where the elements of `A` are symbolic `NaN`s, and logical `0`s (false) where they are not.

Examples

Determine Which Elements of Symbolic Array Are `NaN`s

Using `isnan`, determine which elements of this symbolic matrix are `NaN`s:

```
isnan(sym([pi NaN Inf; 1 + i Inf + i NaN + i]))
```

```
ans =
  2×3 logical array
     0     1     0
     0     0     1
```

Input Arguments

`A` — Input value

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic array | symbolic vector | symbolic matrix

Input value, specified as a symbolic number, variable, expression, or function, or as an array, vector, or matrix of symbolic numbers, variables, expressions, functions.

Tips

- For any `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, or `isnan(A)` is `1` for each element.
- Symbolic expressions and functions containing `NaN` evaluate to `NaN`. For example, `sym(NaN + i)` returns symbolic `NaN`.

See Also

`isAlways` | `isequal` | `isequaln` | `isfinite` | `isinf`

Introduced in R2013b

isolate

Isolate variable or expression in equation

Syntax

```
isolate(eqn,expr)
```

Description

`isolate(eqn,expr)` rearranges the equation `eqn` so that the expression `expr` appears on the left side. The result is similar to solving `eqn` for `expr`. If `isolate` cannot isolate `expr`, it moves all terms containing `expr` to the left side. The output of `isolate` lets you eliminate `expr` from `eqn` by using `subs`.

Examples

Isolate Variable in Equation

Isolate `x` in the equation `a*x^2 + b*x + c == 0`.

```
syms x a b c
eqn = a*x^2 + b*x + c == 0;
xSol = isolate(eqn, x)
```

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

You can use the output of `isolate` to eliminate the variable from the equation using `subs`.

Eliminate `x` from `eqn` by substituting `lhs(xSol)` for `rhs(xSol)`.

```
eqn2 = subs(eqn, lhs(xSol), rhs(xSol))
```

```
eqn2 =
c + (b + (b^2 - 4*a*c)^(1/2))^2/(4*a) - (b*(b + (b^2 - 4*a*c)^(1/2)))/(2*a) == 0
```

Isolate Expression in Equation

Isolate `y(t)` in the following equation.

```
syms y(t)
eqn = a*y(t)^2 + b*c == 0;
isolate(eqn, y(t))
```

```
ans =
y(t) == ((-b)^(1/2)*c^(1/2))/a^(1/2)
```

Isolate `a*y(t)` in the same equation.

```
isolate(eqn, a*y(t))
```

```
ans =
a*y(t) == -(b*c)/y(t)
```

`isolate` Returns Simplest Solution

For equations with multiple solutions, `isolate` returns the simplest solution.

Demonstrate this behavior by isolating `x` in `sin(x) == 0`, which has multiple solutions at `0`, `pi`, `3*pi/2`, and so on.

```
isolate(sin(x) == 0, x)
```

```
ans =
x == 0
```

`isolate` does not consider special cases when returning the solution. Instead, `isolate` returns a general solution that is not guaranteed to hold for all values of the variables in the equation.

Isolate `x` in the equation `a*x^2/(x-a) == 1`. The returned value of `x` does not hold in the special case `a = 0`.

```
syms a x
isolate(a*x^2/(x-a) == 1, x)
```

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

`isolate` Follows Assumptions on Variables

`isolate` returns only results that are consistent with the assumptions on the variables in the equation.

First, assume `x` is negative, and then isolate `x` in the equation `x^4 == 1`.

```
syms x
assume(x < 0)
eqn = x^4 == 1;
isolate(x^4 == 1, x)

ans =
x == -1
```

Remove the assumption. `isolate` chooses a different solution to return.

```
assume(x, 'clear')
isolate(x^4 == 1, x)

ans =
x == 1
```

Tips

- If `eqn` has no solution, `isolate` errors. `isolate` also ignores special cases. If the only solutions to `eqn` are special cases, then `isolate` ignores those special cases and errors.
- The returned solution is not guaranteed to hold for all values of the variables in the solution.
- `expr` cannot be a mathematical constant such as `pi`.

Input Arguments

`eqn` — Input equation
symbolic equation

Input equation, specified as a symbolic equation.

Example: `a*x^2 + b*x + c == 0`

`expr` — Variable or expression to isolate
symbolic variable | symbolic expression

Variable or expression to isolate, specified as a symbolic variable or expression.

See Also

`lhs` | `linsolve` | `rhs` | `solve` | `subs`

Topics

"Solve Algebraic Equation" on page 2-145
"Solve System of Algebraic Equations" on page 2-153
"Solve Equations Numerically" on page 2-172
"Solve System of Linear Equations" on page 2-169

Introduced in R2017a

isUnit

Determine if input is a symbolic unit

Syntax

```
tf = isUnit(expr)
```

Description

`tf = isUnit(expr)` returns logical `1` (`true`) if `expr` is a unit, or a product of powers of units, and logical `0` (`false`) if it is not.

Examples

Determine if Input is a Unit

Determine if an expression is a symbolic unit by using `isUnit`.

Test if `3*u.m` is a symbolic unit, where `u = symunit`. The `isUnit` function returns logical `0` (`false`) because `3*u.m` contains the symbolic number `3`.

```
u = symunit;
isUnit(3*u.m)
```

```
ans =
  logical
   0
```

Check if `u.m`, `u.mW`, and `x*u.Hz` are units, where `u = symunit`. The `isUnit` function returns the array `[1 1 0]`, meaning that the first two expressions are units but the third expression is not.

```
syms x
units = [u.m u.mW x*u.Hz];
isUnit(units)
```

```
ans =
  1×3 logical array
   1   1   0
```

Input Arguments

`expr` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Tips

- `1` represents a dimensionless unit. Hence, `isUnit(sym(1))` returns logical `1` (`true`).

See Also

`checkUnits` | `findUnits` | `newUnit` | `separateUnits` | `str2symunit` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

iztrans

Inverse Z-transform

Syntax

```
iztrans(F)
iztrans(F,transVar)
iztrans(F,var,transVar)
```

Description

`iztrans(F)` returns the “Inverse Z-Transform” on page 4-921 of `F`. By default, the independent variable is `z` and the transformation variable is `n`. If `F` does not contain `z`, `iztrans` uses the function `symvar`.

`iztrans(F,transVar)` uses the transformation variable `transVar` instead of `n`.

`iztrans(F,var,transVar)` uses the independent variable `var` and transformation variable `transVar` instead of `z` and `n` respectively.

Examples

Inverse Z-Transform of Symbolic Expression

Compute the inverse Z-transform of `2*z/(z-2)^2`. By default, the inverse transform is in terms of `n`.

```
syms z
F = 2*z/(z-2)^2;
iztrans(F)
```

```
ans =
2^n + 2^n*(n - 1)
```

Specify Independent Variable and Transformation Variable

Compute the inverse Z-transform of `1/(a*z)`. By default, the independent and transformation variables are `z` and `n`, respectively.

```
syms z a
F = 1/(a*z);
iztrans(F)
```

```
ans =
kroneckerDelta(n - 1, 0)/a
```

Specify the transformation variable as `m`. If you specify only one variable, that variable is the transformation variable. The independent variable is still `z`.

```
syms m
iztrans(F,m)
```

```
ans =
kroneckerDelta(m - 1, 0)/a
```

Specify both the independent and transformation variables as `a` and `m` in the second and third arguments, respectively.

```
iztrans(F,a,m)
```

```
ans =
kroneckerDelta(m - 1, 0)/z
```

Inverse Z-Transforms Involving Kronecker Delta Function

Compute the inverse Z-transforms of these expressions. The results involve the Kronecker Delta function.

```
syms n z
iztrans(1/z,z,n)
```

```
ans =
kroneckerDelta(n - 1, 0)
```

```
f = (z^3 + 3*z^2)/z^5;
iztrans(f,z,n)
```

```
ans =
kroneckerDelta(n - 2, 0) + 3*kroneckerDelta(n - 3, 0)
```

Inverse Z-Transform of Array Inputs

Find the inverse Z-transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `iztrans` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x) 1; sin(y) i*z];
vars = [w x; y z];
transVars = [a b; c d];
iztrans(M,vars,transVars)
```

```
ans =
[ exp(x)*kroneckerDelta(a, 0), kroneckerDelta(b, 0)]
[       iztrans(sin(y), y, c),   iztrans(z, z, d)*1i]
```

If `iztrans` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
syms w x y z a b c d
iztrans(x,vars,transVars)
```

```
ans =
[ x*kroneckerDelta(a, 0),       iztrans(x, x, b)]
[ x*kroneckerDelta(c, 0), x*kroneckerDelta(d, 0)]
```

Inverse Z-Transform of Symbolic Function

Compute the Inverse Z-transform of symbolic functions. When the first argument contains symbolic functions, then the second argument must be a scalar.

```
syms f1(x) f2(x) a b
f1(x) = exp(x);
f2(x) = x;
iztrans([f1, f2],x,[a, b])
```

```
ans =
[ iztrans(exp(x), x, a), iztrans(x, x, b)]
```

If Inverse Z-Transform Cannot Be Found

If `iztrans` cannot compute the inverse transform, it returns an unevaluated call.

```
syms F(z) n
F(z) = exp(z);
f = iztrans(F,z,n)
```

```
f =
iztrans(exp(z), z, n)
```

Return the original expression by using `ztrans`.

```
ztrans(f,n,z)
```

```
ans =
exp(z)
```

Input Arguments

`F` — Input

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

`x` (default) | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Independent variable, specified as a symbolic variable, expression, vector, or matrix. This variable is often called the "complex frequency variable." If you do not specify the variable, then `iztrans` uses `z`. If `F` does not contain `z`, then `iztrans` uses the function `symvar`.

`transVar` — Transformation variable

`x` (default) | `t` | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. It is often called the"time variable" or "space variable." By default, `iztrans` uses `n`. If `n` is the independent variable of `F`, then `iztrans` uses `k`.

Definitions

Inverse Z-Transform

Where R is a positive number, such that the function $F = F(z)$ is analytic on and outside the circle $|z| = R$, the inverse Z-transform is

$$f(n) = \frac{1}{2\pi i} \oint_{|z|=R} F(z) z^{n-1} dz, \quad n = 0,1,2...$$

Tips

- If any argument is an array, then `iztrans` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.
- To compute the direct Z-transform, use `ztrans`.

See Also

`fourier` | `ifourier` | `ilaplace` | `kroneckerDelta` | `laplace` | `ztrans`

Topics

"Solve Difference Equations Using Z-Transform" on page 2-233

Introduced before R2006a

jacobiAM

Jacobi amplitude function

Syntax

```
jacobiAM(u,m)
```

Description

`jacobiAM(u,m)` returns the “Jacobi Amplitude Function” on page 4-926 of `u` and `m`. If `u` or `m` is an array, then `jacobiAM` acts element-wise.

Examples

Calculate Jacobi Amplitude Function for Numeric Inputs

```
jacobiAM(2,1)
```

```
ans =
    1.3018
```

Call `jacobiAM` on array inputs. `jacobiAM` acts element-wise when `u` or `m` is an array.

```
jacobiAM([2 1 -3],[1 2 3])
```

```
ans =
    1.3018    0.7370   -3.7571
```

Calculate Jacobi Amplitude Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi amplitude function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiAM` returns exact symbolic output.

```
jacobiAM(sym(2),sym(1))

ans =
2*atan(exp(2)) - pi/2
```

Show that for other values of u or m, jacobiAM returns an unevaluated function call.

```
jacobiAM(sym(2),sym(3))

ans =
jacobiAM(2, 3)
```

Find Jacobi Amplitude Function for Symbolic Variables or Expressions

For symbolic variables or expressions, jacobiAM returns the unevaluated function call.

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

f =
jacobiAM(x, y)
```

Substitute values for the variables by using subs, and convert values to double by using double.

```
f = subs(f, [x y], [3 5])

f =
jacobiAM(3, 5)

fVal = double(f)

fVal =
    3.1104
```

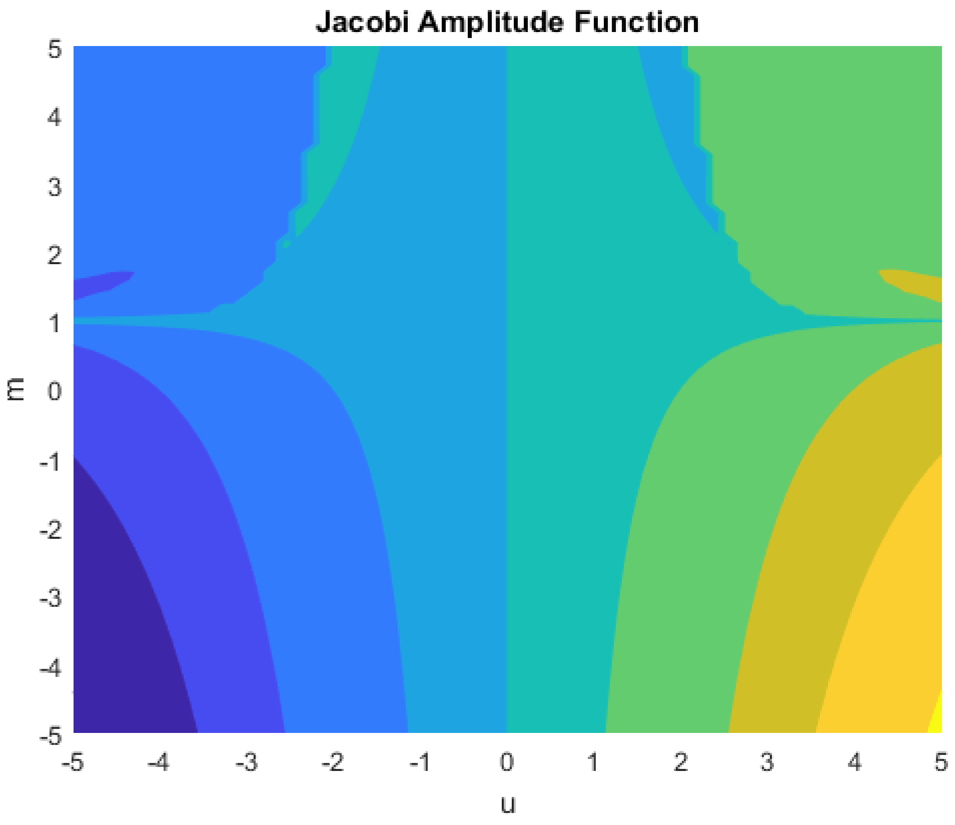
Calculate f to higher precision using vpa.

```
fVal = vpa(f)
fVal =
3.1104428381773623934754349126446
```

Plot Jacobi Amplitude Function

Plot the Jacobi amplitude function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiAM(u,m);
fcontour(f,'Fill','on')
title('Jacobi Amplitude Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi Amplitude Function

The Jacobi amplitude function $\mathrm{am}(u,m)$ is defined by $\mathrm{am}(u,m) = \varphi$ where $\mathrm{F}(\varphi,m) = u$ and F represents the incomplete elliptic integral of the first kind. F is implemented as `ellipticF`.

See Also

`ellipticF` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobian

Jacobian matrix

Syntax

```
jacobian(f,v)
```

Description

`jacobian(f,v)` computes the Jacobian matrix on page 4-929 of `f` with respect to `v`.

The (i,j) element of the result is $\dfrac{\partial f(i)}{\partial \mathrm{v}(j)}$.

Examples

Jacobian of Vector Function

The Jacobian of a vector function is a matrix of the partial derivatives of that function.

Compute the Jacobian matrix of `[x*y*z, y^2, x + z]` with respect to `[x, y, z]`.

```
syms x y z
jacobian([x*y*z, y^2, x + z], [x, y, z])

ans =
[ y*z, x*z, x*y]
[   0, 2*y,   0]
[   1,   0,   1]
```

Now, compute the Jacobian of `[x*y*z, y^2, x + z]` with respect to `[x; y; z]`.

```
jacobian([x*y*z, y^2, x + z], [x; y; z])
```

Jacobian of Scalar Function

The Jacobian of a scalar function is the transpose of its gradient.

Compute the Jacobian of `2*x + 3*y + 4*z` with respect to `[x, y, z]`.

```
syms x y z
jacobian(2*x + 3*y + 4*z, [x, y, z])
```

```
ans =
[ 2, 3, 4]
```

Now, compute the gradient of the same expression.

```
gradient(2*x + 3*y + 4*z, [x, y, z])
```

```
ans =
 2
 3
 4
```

Jacobian with Respect to Scalar

The Jacobian of a function with respect to a scalar is the first derivative of that function. For a vector function, the Jacobian with respect to a scalar is a vector of the first derivatives.

Compute the Jacobian of `[x^2*y, x*sin(y)]` with respect to `x`.

```
syms x y
jacobian([x^2*y, x*sin(y)], x)
```

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

Now, compute the derivatives.

```
diff([x^2*y, x*sin(y)], x)
```

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

Input Arguments

`f` — Scalar or vector function

symbolic expression | symbolic function | symbolic vector

Scalar or vector function, specified as a symbolic expression, function, or vector. If `f` is a scalar, then the Jacobian matrix of `f` is the transposed gradient of `f`.

`v` — Vector of variables with respect to which you compute Jacobian

symbolic variable | symbolic vector

Vector of variables with respect to which you compute Jacobian, specified as a symbolic variable or vector of symbolic variables. If `v` is a scalar, then the result is equal to the transpose of `diff(f,v)`. If `v` is an empty symbolic object, such as `sym([])`, then `jacobian` returns an empty symbolic object.

Definitions

Jacobian Matrix

The Jacobian matrix of the vector function $f = (f_1(x_1,...,x_n),...,f_n(x_1,...,x_n))$ is the matrix of the derivatives of f:

$$J(x_1,\ldots x_n) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `laplacian` | `potential` | `vectorPotential`

Introduced before R2006a

jacobiCD

Jacobi CD elliptic function

Syntax

```
jacobiCD(u,m)
```

Description

`jacobiCD(u,m)` returns the “Jacobi CD Elliptic Function” on page 4-934 of `u` and `m`. If `u` or `m` is an array, then `jacobiCD` acts element-wise.

Examples

Calculate Jacobi CD Elliptic Function for Numeric Inputs

```
jacobiCD(2,1)
```

```
ans =
    1
```

Call `jacobiCD` on array inputs. `jacobiCD` acts element-wise when `u` or `m` is an array.

```
jacobiCD([2 1 -3],[1 2 3])
```

```
ans =
    1.0000    2.3829 -178.6290
```

Calculate Jacobi CD Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi CD elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiCD` returns exact symbolic output.

```
jacobiCD(sym(2),sym(1))
```

```
ans =
1
```

Show that for other values of `u` or `m`, `jacobiCD` returns an unevaluated function call.

```
jacobiCD(sym(2),sym(3))
```

```
ans =
jacobiCD(2, 3)
```

Find Jacobi CD Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiCD` returns the unevaluated function call.

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

```
f =
jacobiCD(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiCD(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    1.0019
```

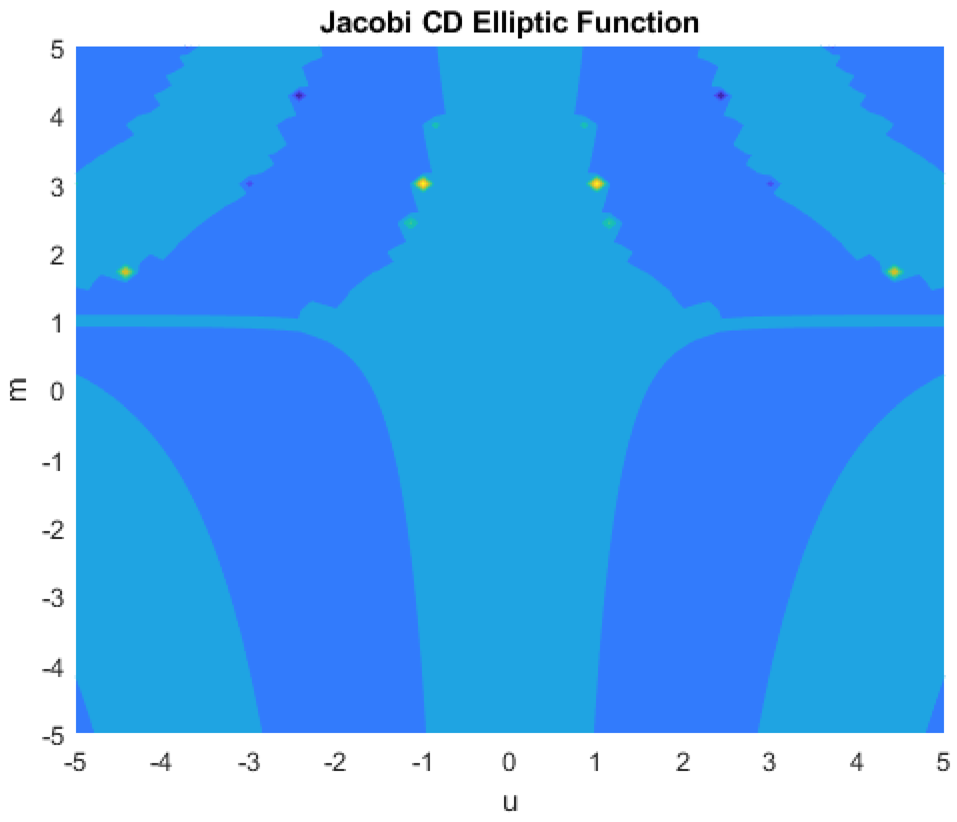
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
1.0019475527333315357888731083364
```

Plot Jacobi CD Elliptic Function

Plot the Jacobi CD elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiCD(u,m);
fcontour(f,'Fill','on')
title('Jacobi CD Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi CD Elliptic Function

The Jacobi CD elliptic function is
<mathphrase>$\mathrm{cd}(u,m) = \mathrm{cn}(u,m)/\mathrm{dn}(u,m)$</mathphrase>

where cn and dn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4i\mathrm{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiCN

Jacobi CN elliptic function

Syntax

```
jacobiCN(u,m)
```

Description

`jacobiCN(u,m)` returns the "Jacobi CN Elliptic Function" on page 4-939 of `u` and `m`. If `u` or `m` is an array, then `jacobiCN` acts element-wise.

Examples

Calculate Jacobi CN Elliptic Function for Numeric Inputs

```
jacobiCN(2,1)
```

```
ans =
    0.2658
```

Call `jacobiCN` on array inputs. `jacobiCN` acts element-wise when `u` or `m` is an array.

```
jacobiCN([2 1 -3],[1 2 3])
```

```
ans =
    0.2658    0.7405    0.8165
```

Calculate Jacobi CN Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi CN elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiCN` returns exact symbolic output.

```
jacobiCN(sym(2),sym(1))
```

```
ans =
1/cosh(2)
```

Show that for other values of `u` or `m`, `jacobiCN` returns an unevaluated function call.

```
jacobiCN(sym(2),sym(3))
```

```
ans =
jacobiCN(2, 3)
```

Find Jacobi CN Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiCN` returns the unevaluated function call.

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

```
f =
jacobiCN(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiCN(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.9995
```

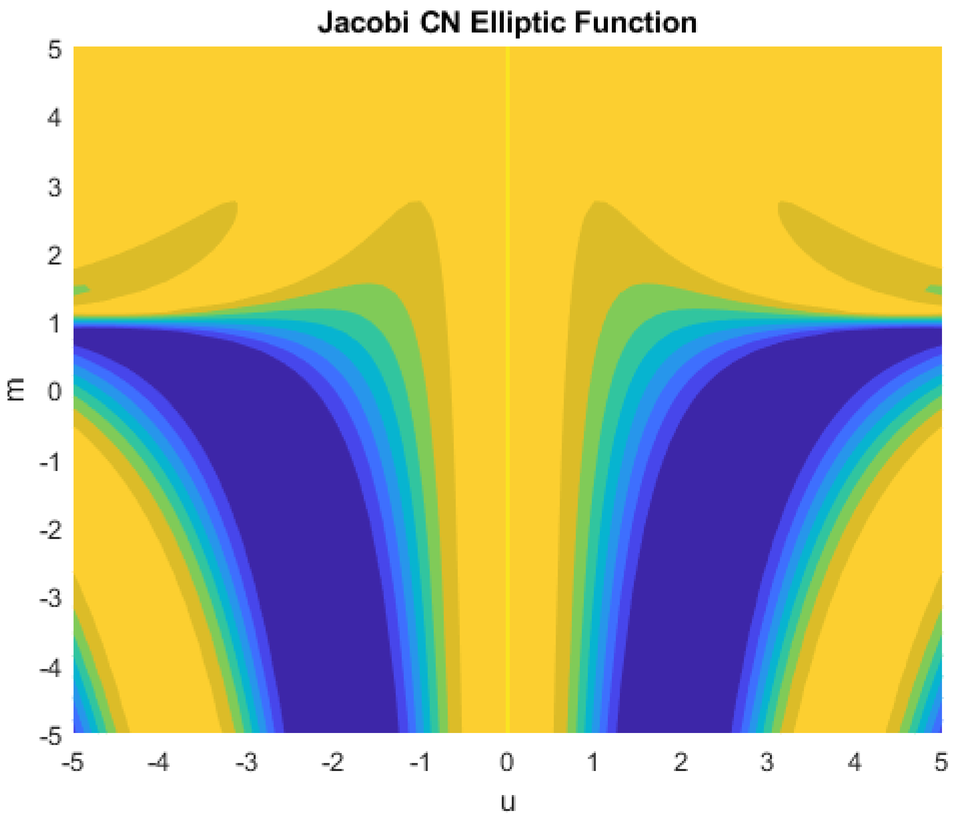
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.9995148837279268257000709197021
```

Plot Jacobi CN Elliptic Function

Plot the Jacobi CN elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiCN(u,m);
fcontour(f,'Fill','on')
title('Jacobi CN Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi CN Elliptic Function

The Jacobi CN elliptic function is $\text{cn}(u,m) = \cos(\text{am}(u,m))$ where am is the Jacobi amplitude function.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\text{K}(m)$ and $4i\text{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiCS

Jacobi CS elliptic function

Syntax

```
jacobiCS(u,m)
```

Description

`jacobiCS(u,m)` returns the “Jacobi CS Elliptic Function” on page 4-944 of `u` and `m`. If `u` or `m` is an array, then `jacobiCS` acts element-wise.

Examples

Calculate Jacobi CS Elliptic Function for Numeric Inputs

```
jacobiCS(2,1)
```

```
ans =
    0.2757
```

Call `jacobiCS` on array inputs. `jacobiCS` acts element-wise when `u` or `m` is an array.

```
jacobiCS([2 1 -3],[1 2 3])
```

```
ans =
    0.2757    1.1017    1.4142
```

Calculate Jacobi CS Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi CS elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiCS` returns exact symbolic output.

```
jacobiCS(sym(2),sym(1))
```

```
ans =
1/sinh(2)
```

Show that for other values of `u` or `m`, `jacobiCS` returns an unevaluated function call.

```
jacobiCS(sym(2),sym(3))
```

```
ans =
jacobiCS(2, 3)
```

Find Jacobi CS Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiCS` returns the unevaluated function call.

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

```
f =
jacobiCS(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiCS(3, 5)
```

```
fVal = double(f)
```

```
fVal =
   32.0925
```

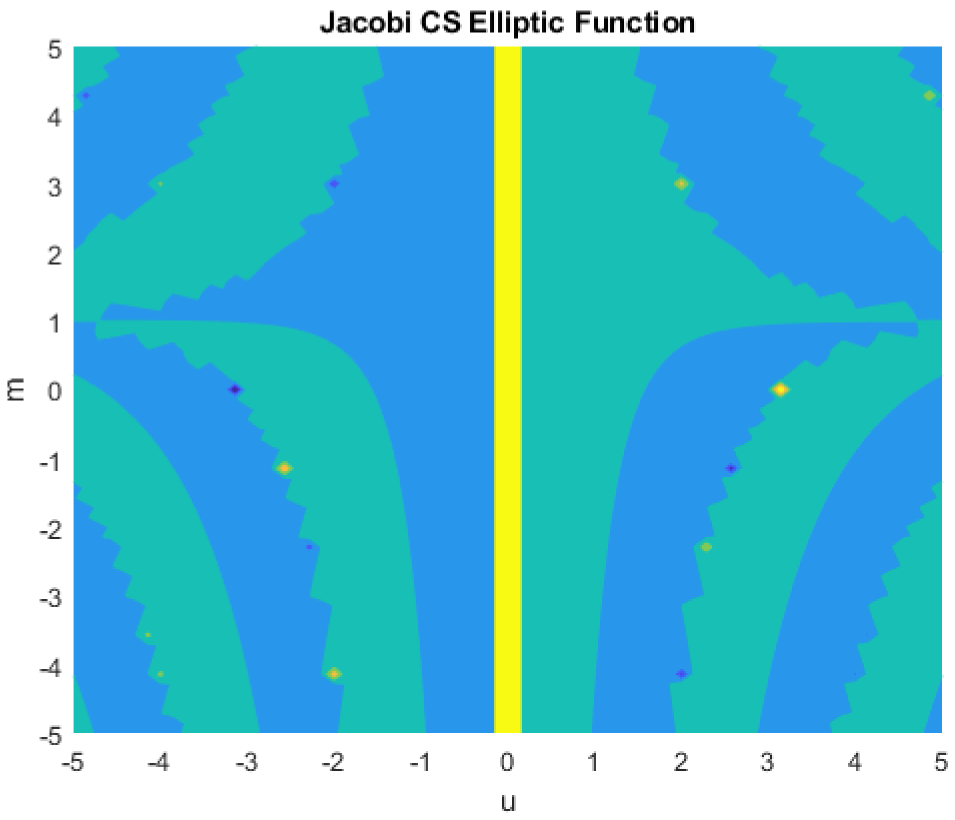
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
32.092535022751828816106562829547
```

Plot Jacobi CS Elliptic Function

Plot the Jacobi CS elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiCS(u,m);
fcontour(f,'Fill','on')
title('Jacobi CS Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

m — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi CS Elliptic Function

The Jacobi CS elliptic function is
<mathphrase>$\mathrm{cs}(u,m) = \mathrm{cn}(u,m)/\mathrm{sn}(u,m)$</mathphrase>

where cn and sn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4i\mathrm{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiDC

Jacobi DC elliptic function

Syntax

```
jacobiDC(u,m)
```

Description

`jacobiDC(u,m)` returns the “Jacobi DC Elliptic Function” on page 4-949 of `u` and `m`. If `u` or `m` is an array, then `jacobiDC` acts element-wise.

Examples

Calculate Jacobi DC Elliptic Function for Numeric Inputs

```
jacobiDC(2,1)
```

```
ans =
     1
```

Call `jacobiDC` on array inputs. `jacobiDC` acts element-wise when `u` or `m` is an array.

```
jacobiDC([2 1 -3],[1 2 3])
```

```
ans =
    1.0000    0.4197   -0.0056
```

Calculate Jacobi DC Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi DC elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`,`jacobiDC` returns exact symbolic output.

```
jacobiDC(sym(2),sym(1))
```

```
ans =
1
```

Show that for other values of `u` or `m`, `jacobiDC` returns an unevaluated function call.

```
jacobiDC(sym(2),sym(3))
```

```
ans =
jacobiDC(2, 3)
```

Find Jacobi DC Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiDC` returns the unevaluated function call.

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

```
f =
jacobiDC(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiDC(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.9981
```

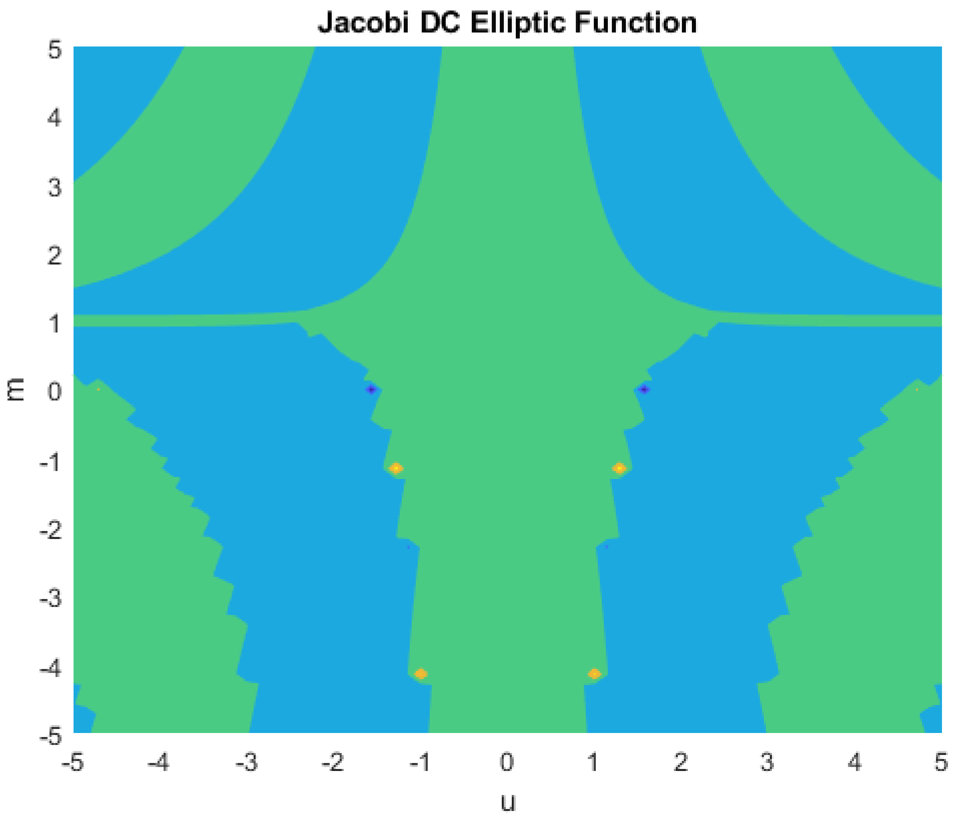
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.99805623285568333815968501058428
```

Plot Jacobi DC Elliptic Function

Plot the Jacobi DC elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiDC(u,m);
fcontour(f,'Fill','on')
title('Jacobi DC Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi DC Elliptic Function

The Jacobi DC elliptic function is
<mathphrase>$\mathrm{dc}(u,m) = \mathrm{dn}(u,m)/\mathrm{cn}(u,m)$</mathphrase>

where dn and cn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods 4K(m) and 4iK'(m), where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiDN

Jacobi DN elliptic function

Syntax

```
jacobiDN(u,m)
```

Description

`jacobiDN(u,m)` returns the “Jacobi DN Elliptic Function” on page 4-954 of `u` and `m`. If `u` or `m` is an array, then `jacobiDN` acts element-wise.

Examples

Calculate Jacobi DN Elliptic Function for Numeric Inputs

```
jacobiDN(2,1)
```

```
ans =
    0.2658
```

Call `jacobiDN` on array inputs. `jacobiDN` acts element-wise when `u` or `m` is an array.

```
jacobiDN([2 1 -3],[1 2 3])
```

```
ans =
    0.2658    0.3107   -0.0046
```

Calculate Jacobi DN Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi DN elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiDN` returns exact symbolic output.

```
jacobiDN(sym(2),sym(1))
```

```
ans =
1/cosh(2)
```

Show that for other values of `u` or `m`, `jacobiDN` returns an unevaluated function call.

```
jacobiDN(sym(2),sym(3))
```

```
ans =
jacobiDN(2, 3)
```

Find Jacobi DN Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiDN` returns the unevaluated function call.

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

```
f =
jacobiDN(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiDN(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.9976
```

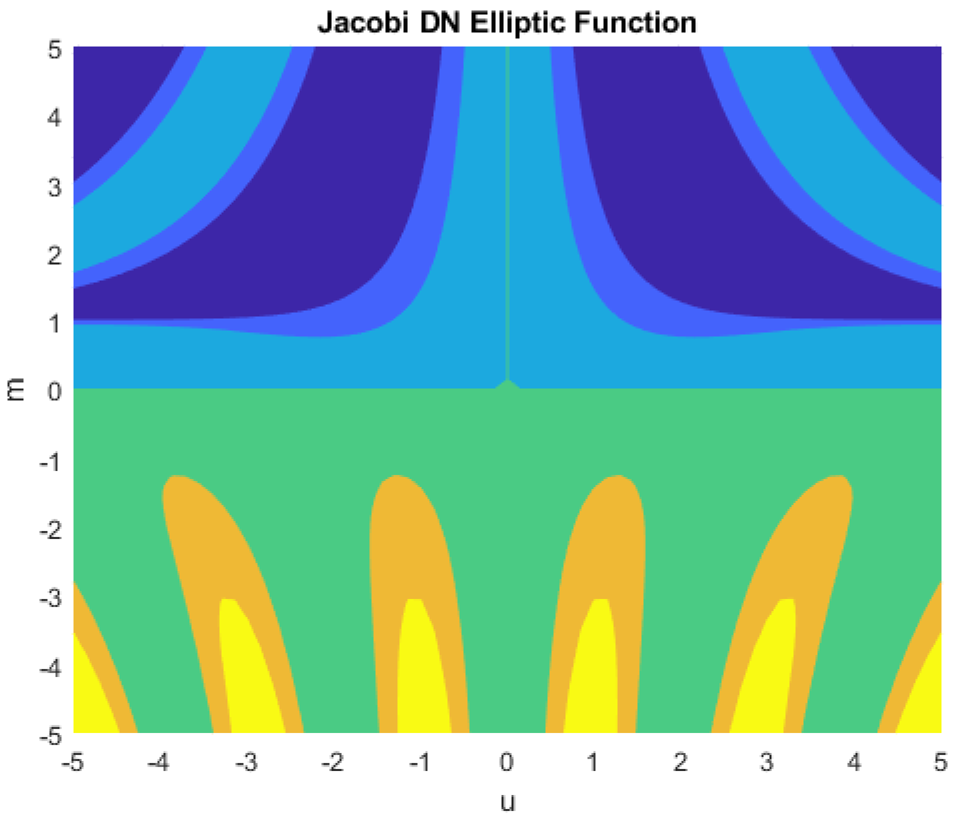
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.99757205953668099307853539907267
```

Plot Jacobi DN Elliptic Function

Plot the Jacobi DN elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiDN(u,m);
fcontour(f,'Fill','on')
title('Jacobi DN Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

m — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi DN Elliptic Function

The Jacobi DN elliptic function is

$$\mathrm{dn}(u \mid m) = \sqrt{1 - m \sin(\varphi)^2}$$

where φ is such that $\mathrm{F}(\varphi,m) = \mathrm{u}$.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4\mathrm{iK}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiDS

Jacobi DS elliptic function

Syntax

```
jacobiDS(u,m)
```

Description

`jacobiDS(u,m)` returns the “Jacobi DS Elliptic Function” on page 4-959 of `u` and `m`. If `u` or `m` is an array, then `jacobiDS` acts element-wise.

Examples

Calculate Jacobi DS Elliptic Function for Numeric Inputs

```
jacobiDS(2,1)
```

```
ans =
    0.2757
```

Call `jacobiDS` on array inputs. `jacobiDS` acts element-wise when `u` or `m` is an array.

```
jacobiDS([2 1 -3],[1 2 3])
```

```
ans =
    0.2757    0.4623   -0.0079
```

Calculate Jacobi DS Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi DS elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiDS` returns exact symbolic output.

```
jacobiDS(sym(2),sym(1))
```

```
ans =
1/sinh(2)
```

Show that for other values of `u` or `m`, `jacobiDS` returns an unevaluated function call.

```
jacobiDS(sym(2),sym(3))
```

```
ans =
jacobiDS(2, 3)
```

Find Jacobi DS Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiDS` returns the unevaluated function call.

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

```
f =
jacobiDS(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiDS(3, 5)
```

```
fVal = double(f)
```

```
fVal =
   32.0302
```

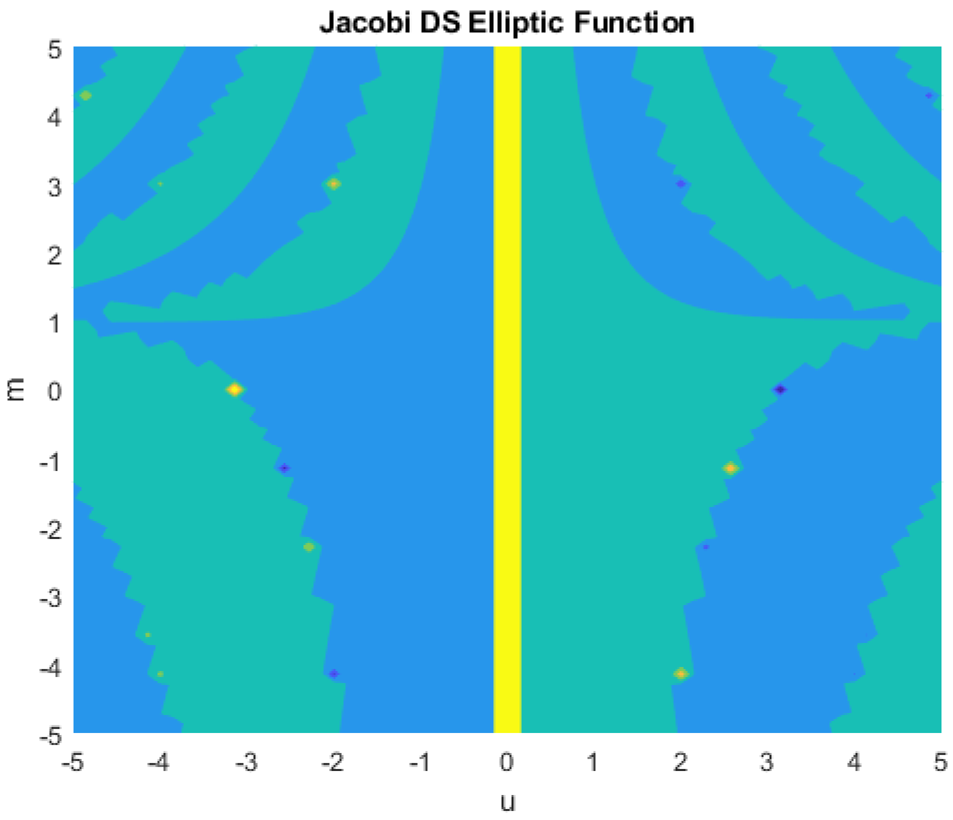
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
32.030154607596772037587224629884
```

Plot Jacobi DS Elliptic Function

Plot the Jacobi DS elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiDS(u,m);
fcontour(f,'Fill','on')
title('Jacobi DS Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi DS Elliptic Function

The Jacobi DS elliptic function is
<mathphrase>$\mathrm{ds}(u,m) = \mathrm{dn}(u,m)/\mathrm{sn}(u,m)$</mathphrase>

where dn and sn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4i\mathrm{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiNC

Jacobi NC elliptic function

Syntax

```
jacobiNC(u,m)
```

Description

`jacobiNC(u,m)` returns the “Jacobi NC Elliptic Function” on page 4-964 of `u` and `m`. If `u` or `m` is an array, then `jacobiNC` acts element-wise.

Examples

Calculate Jacobi NC Elliptic Function for Numeric Inputs

```
jacobiNC(2,1)
```

```
ans =
    3.7622
```

Call `jacobiNC` on array inputs. `jacobiNC` acts element-wise when `u` or `m` is an array.

```
jacobiNC([2 1 -3],[1 2 3])
```

```
ans =
    3.7622    1.3505    1.2247
```

Calculate Jacobi NC Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi NC elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiNC` returns exact symbolic output.

```
jacobiNC(sym(2),sym(1))
```

```
ans =
cosh(2)
```

Show that for other values of `u` or `m`, `jacobiNC` returns an unevaluated function call.

```
jacobiNC(sym(2),sym(3))
```

```
ans =
jacobiNC(2, 3)
```

Find Jacobi NC Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiNC` returns the unevaluated function call.

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

```
f =
jacobiNC(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiNC(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    1.0005
```

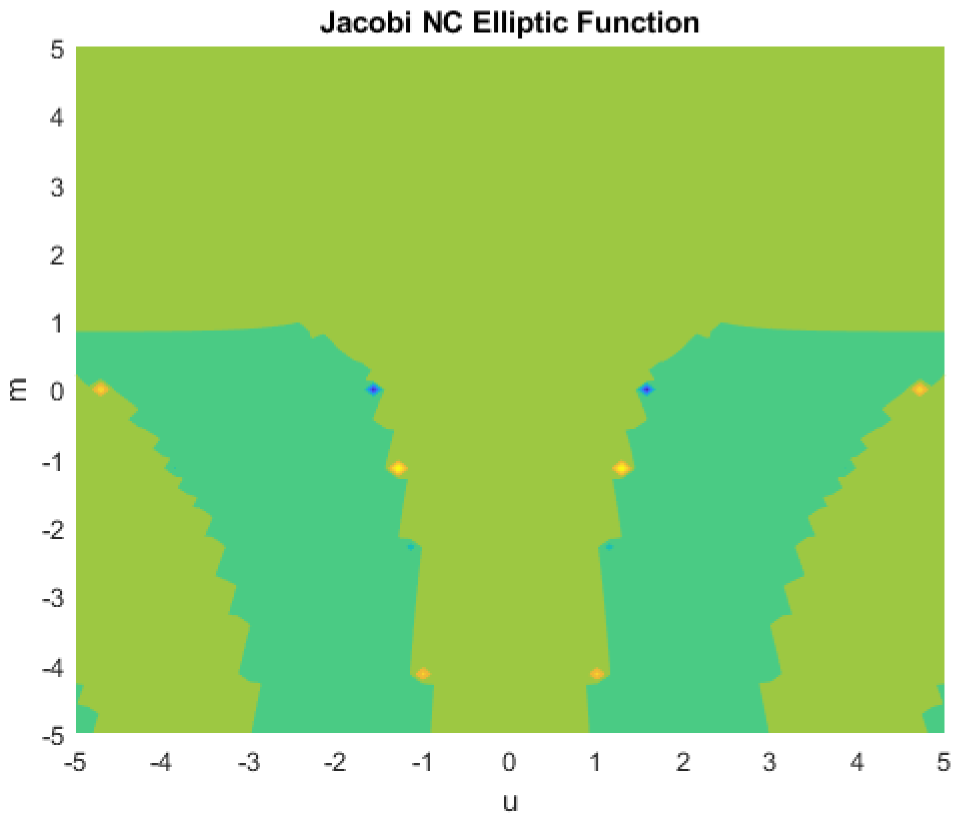
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
1.0004853517240922102007985618873
```

Plot Jacobi NC Elliptic Function

Plot the Jacobi NC elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiNC(u,m);
fcontour(f,'Fill','on')
title('Jacobi NC Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi NC Elliptic Function

The Jacobi NC elliptic function is
<mathphrase>$\mathrm{nc}(u,m) = 1/\mathrm{cn}(u,m)$</mathphrase>

where cn is the respective Jacobi elliptic function.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4i\mathrm{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiND

Jacobi ND elliptic function

Syntax

```
jacobiND(u,m)
```

Description

`jacobiND(u,m)` returns the “Jacobi ND Elliptic Function” on page 4-969 of `u` and `m`. If `u` or `m` is an array, then `jacobiND` acts element-wise.

Examples

Calculate Jacobi ND Elliptic Function for Numeric Inputs

```
jacobiND(2,1)
```

```
ans =
    3.7622
```

Call `jacobiND` on array inputs. `jacobiND` acts element-wise when `u` or `m` is an array.

```
jacobiND([2 1 -3],[1 2 3])
```

```
ans =
    3.7622    3.2181 -218.7739
```

Calculate Jacobi ND Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi ND elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiND` returns exact symbolic output.

```
jacobiND(sym(2),sym(1))
```

```
ans =
cosh(2)
```

Show that for other values of `u` or `m`, `jacobiND` returns an unevaluated function call.

```
jacobiND(sym(2),sym(3))
```

```
ans =
jacobiND(2, 3)
```

Find Jacobi ND Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiND` returns the unevaluated function call.

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

```
f =
jacobiND(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiND(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    1.0024
```

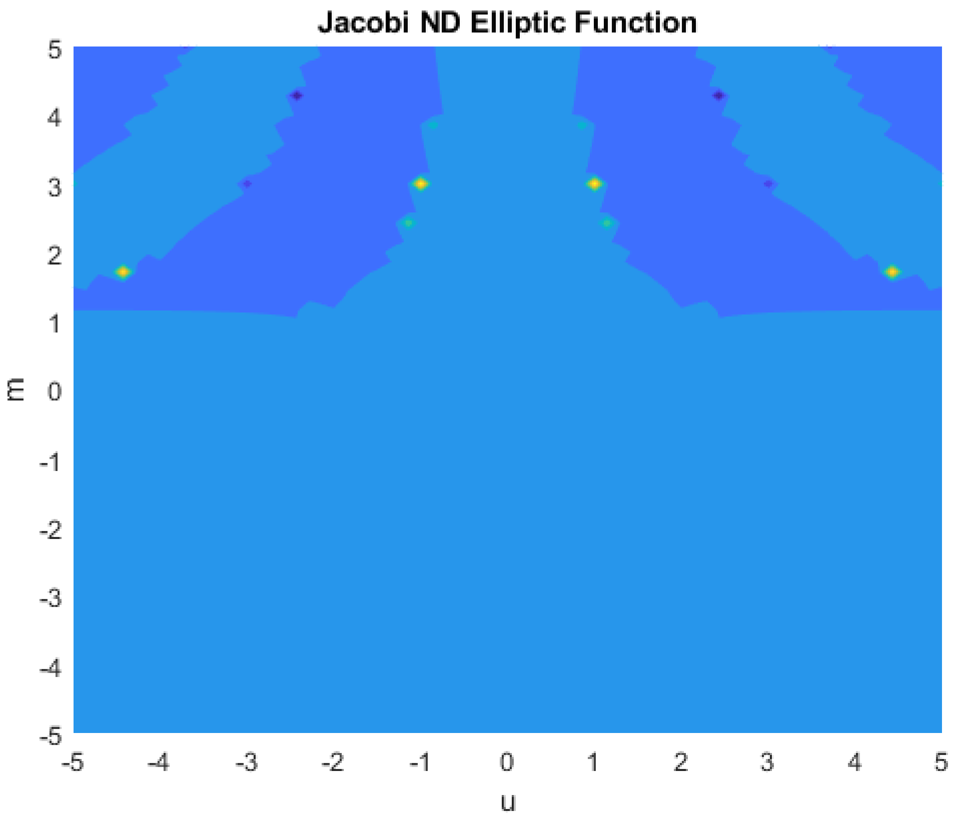
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
1.0024338497055006289470589737758
```

Plot Jacobi ND Elliptic Function

Plot the Jacobi ND elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiND(u,m);
fcontour(f,'Fill','on')
title('Jacobi ND Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi ND Elliptic Function

The Jacobi ND elliptic function is
<mathphrase>nd(u,m) = 1/dn(u,m)</mathphrase>

where dn is the respective Jacobi elliptic function.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods 4K(m) and 4iK'(m), where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiNS

Jacobi NS elliptic function

Syntax

```
jacobiNS(u,m)
```

Description

`jacobiNS(u,m)` returns the “Jacobi NS Elliptic Function” on page 4-974 of `u` and `m`. If `u` or `m` is an array, then `jacobiNS` acts element-wise.

Examples

Calculate Jacobi NS Elliptic Function for Numeric Inputs

```
jacobiNS(2,1)
```

```
ans =
    1.0373
```

Call `jacobiNS` on array inputs. `jacobiNS` acts element-wise when `u` or `m` is an array.

```
jacobiNS([2 1 -3],[1 2 3])
```

```
ans =
    1.0373    1.4879    1.7321
```

Calculate Jacobi NS Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi NS elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiNS` returns exact symbolic output.

```
jacobiNS(sym(2),sym(1))
```

```
ans =
coth(2)
```

Show that for other values of `u` or `m`, `jacobiNS` returns an unevaluated function call.

```
jacobiNS(sym(2),sym(3))
```

```
ans =
jacobiNS(2, 3)
```

Find Jacobi NS Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiNS` returns the unevaluated function call.

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

```
f =
jacobiNS(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiNS(3, 5)
```

```
fVal = double(f)
```

```
fVal =
   32.1081
```

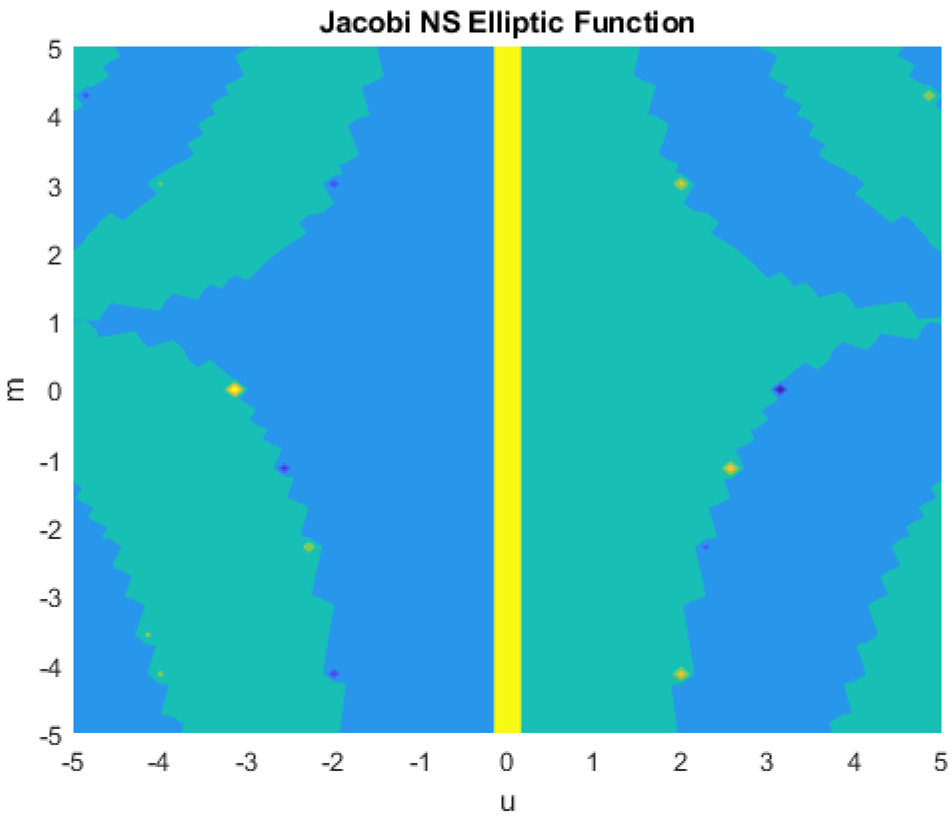
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
32.108111189955611054545195854805
```

Plot Jacobi NS Elliptic Function

Plot the Jacobi NS elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiNS(u,m);
fcontour(f,'Fill','on')
title('Jacobi NS Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi NS Elliptic Function

The Jacobi NS elliptic function is
<mathphrase>ns(u,m) = 1/ds(u,m)</mathphrase>

where ds is the respective Jacobi elliptic function.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods 4K(m) and 4iK'(m), where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiSC` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiP

Jacobi polynomials

Syntax

```
jacobiP(n,a,b,x)
```

Description

`jacobiP(n,a,b,x)` returns the `n`th degree Jacobi polynomial on page 4-980 with parameters `a` and `b` at `x`.

Examples

Find Jacobi Polynomials for Numeric and Symbolic Inputs

Find the Jacobi polynomial of degree `2` for numeric inputs.

```
jacobiP(2,0.5,-3,6)

ans =
    7.3438
```

Find the Jacobi polynomial for symbolic inputs.

```
syms n a b x
jacobiP(n,a,b,x)

ans =
jacobiP(n, a, b, x)
```

If the degree of the Jacobi polynomial is not specified, `jacobiP` cannot find the polynomial and returns the function call.

Specify the degree of the Jacobi polynomial as `1` to return the form of the polynomial.

```
J = jacobiP(1,a,b,x)

J =
a/2 - b/2 + x*(a/2 + b/2 + 1)
```

To find the numeric value of a Jacobi polynomial, call `jacobiP` with the numeric values directly. Do not substitute into the symbolic polynomial because the result can be inaccurate due to round-off. Test this by using `subs` to substitute into the symbolic polynomial, and compare the result with a numeric call.

```
J = jacobiP(300, -1/2, -1/2, x);
subs(J,x,vpa(1/2))
jacobiP(300, -1/2, -1/2, vpa(1/2))

ans =
101573673381249394050.64541318209
ans =
0.032559931334979678350422392588404
```

When `subs` is used to substitute into the symbolic polynomial, the numeric result is subject to round-off error. The direct numerical call to `jacobiP` is accurate.

Find Jacobi Polynomial with Vector and Matrix Inputs

Find the Jacobi polynomials of degrees `1` and `2` by setting `n = [1 2]` for `a = 3` and `b = 1`.

```
syms x
jacobiP([1 2],3,1,x)

ans =
[ 3*x + 1, 7*x^2 + (7*x)/2 - 1/2]
```

`jacobiP` acts on `n` element-wise to return a vector with two entries.

If multiple inputs are specified as a vector, matrix, or multidimensional array, these inputs must be the same size. Find the Jacobi polynomials for `a = [1 2;3 1]`, `b = [2 2;1 3]`, `n = 1` and `x`.

```
a = [1 2;3 1];
b = [2 2;1 3];
J = jacobiP(1,a,b,x)
```

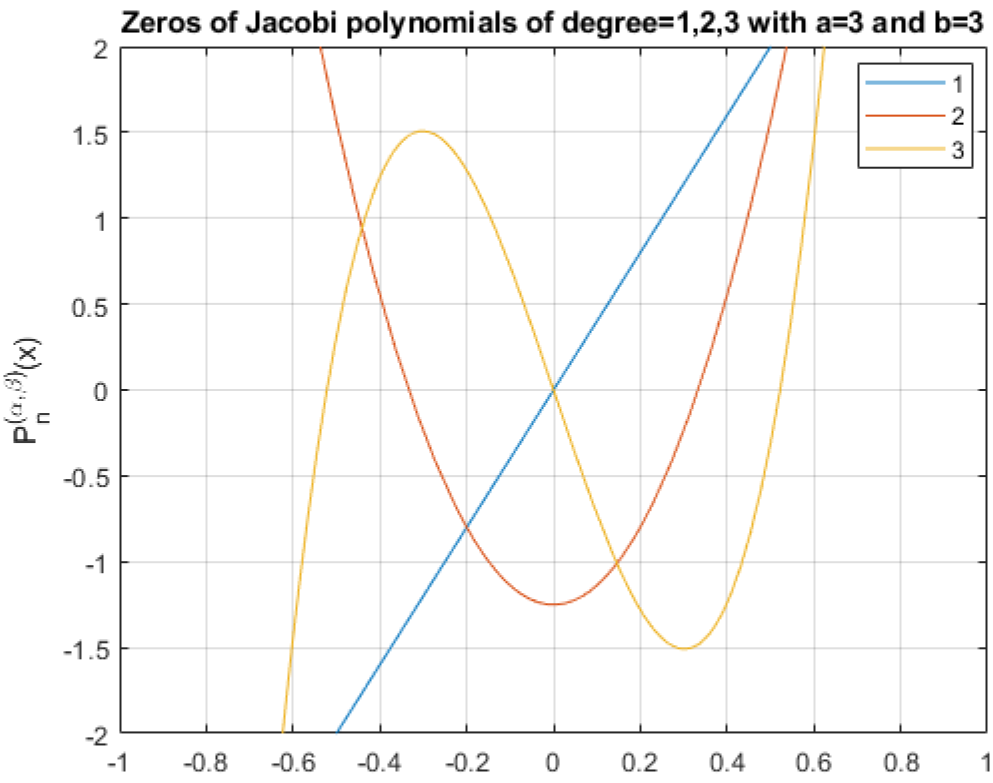
```
J =
[ (5*x)/2 - 1/2,      3*x]
[       3*x + 1, 3*x - 1]
```

`jacobiP` acts element-wise on `a` and `b` to return a matrix of the same size as `a` and `b`.

Visualize Zeros of Jacobi Polynomials

Plot Jacobi polynomials of degree `1`, `2`, and `3` for `a = 3`, `b = 3`, and `-1<x<1`. To better view the plot, set axis limits by using `axis`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(jacobiP(1:3,3,3,x))
axis([-1 1 -2 2])
grid on

ylabel('P_n^{(\alpha,\beta)}(x)')
title('Zeros of Jacobi polynomials of degree=1,2,3 with a=3 and b=3');
legend('1','2','3','Location','best')
```



Prove Orthogonality of Jacobi Polynomials with Respect to Weight Function

The Jacobi polynomials P(n,a,b,x) are orthogonal with respect to the weight function $(1-x)^a(1-x)^b$ on the interval `[-1,1]`.

Prove P(3,a,b,x) and P(5,a,b,x) are orthogonal with respect to the weight function $(1-x)^a(1-x)^b$ by integrating their product over the interval `[-1,1]`, where `a = 3.5` and `b = 7.2`.

```
syms x
a = 3.5;
b = 7.2;
P3 = jacobiP(3, a, b, x);
P5 = jacobiP(5, a, b, x);
w = (1-x)^a*(1+x)^b;
int(P3*P5*w, x, -1, 1)

ans =
0
```

Input Arguments

`n` — Degree of Jacobi polynomial

nonnegative integer | vector of nonnegative integers | matrix of nonnegative integers | multidimensional array of nonnegative integers | symbolic nonnegative integer | symbolic variable | symbolic vector | symbolic matrix | symbolic function | symbolic expression | symbolic multidimensional array

Degree of Jacobi polynomial, specified as a nonnegative integer, or a vector, matrix, or multidimensional array of nonnegative integers, or a symbolic nonnegative integer, variable, vector, matrix, function, expression, or multidimensional array.

`a` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic expression | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, expression, or multidimensional array.

`b` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic expression | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, expression, or multidimensional array.

x — Evaluation point
number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic expression | symbolic multidimensional array

Evaluation point, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, expression, or multidimensional array.

Definitions

Jacobi Polynomials

The Jacobi polynomials are given by the recursion formula

$$\begin{aligned}2nc_n c_{2n-2} P(n,a,b,x) &= c_{2n-1}\left(c_{2n-2}c_{2n}x + a^2 - b^2\right)P(n-1,a,b,x)\\ &\quad -2(n-1+a)(n-1+b)c_{2n}P(n-2,a,b,x),\end{aligned}$$

where

$$c_n = n + a + b$$

$$P(0,a,b,x) = 1$$

$$P(1,a,b,x) = \frac{a-b}{2} + \left(1 + \frac{a+b}{2}\right)x.$$

For fixed real a > -1 and b > -1, the Jacobi polynomials are orthogonal on the interval

`[-1,1]` with respect to the weight function $w(x) = (1-x)^a (1+x)^b$.

For a = 0 and b = 0, the Jacobi polynomials P(n,0,0,x) reduce to the Legendre polynomials P(n, x).

The relation between Jacobi polynomials P(n,a,b,x) and Chebyshev polynomials of the first kind T(n,x) is

$$T(n,x) = \frac{2^{2n}(n!)^2}{(2n)!} P\left(n, -\frac{1}{2}, -\frac{1}{2}, x\right).$$

The relation between Jacobi polynomials P(n,a,b,x) and Chebyshev polynomials of the second kind $U(n,x)$ is

$$U(n,x) = \frac{2^{2n} n!(n+1)!}{(2n+1)!} P\left(n, \frac{1}{2}, \frac{1}{2}, x\right).$$

The relation between Jacobi poynomials P(n,a,b,x) and Gegenbauer polynomials G(n,a,x) is

$$G(n,a,x) = \frac{\Gamma\left(a+\frac{1}{2}\right)\Gamma(n+2a)}{\Gamma(2a)\Gamma\left(n+a+\frac{1}{2}\right)} P\left(n, a-\frac{1}{2}, a-\frac{1}{2}, x\right).$$

See Also

`chebyshevT` | `chebyshevU` | `gegenbauerC` | `hermiteH` | `hypergeom` | `laguerreL` | `legendreP`

Introduced in R2014b

jacobiSC

Jacobi SC elliptic function

Syntax

```
jacobiSC(u,m)
```

Description

`jacobiSC(u,m)` returns the “Jacobi SC Elliptic Function” on page 4-986 of `u` and `m`. If `u` or `m` is an array, then `jacobiSC` acts element-wise.

Examples

Calculate Jacobi SC Elliptic Function for Numeric Inputs

```
jacobiSC(2,1)
```

```
ans =
    3.6269
```

Call `jacobiSC` on array inputs. `jacobiSC` acts element-wise when `u` or `m` is an array.

```
jacobiSC([2 1 -3],[1 2 3])
```

```
ans =
    3.6269    0.9077    0.7071
```

Calculate Jacobi SC Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi SC elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiSC` returns exact symbolic output.

```
jacobiSC(sym(2),sym(1))
```

```
ans =
sinh(2)
```

Show that for other values of `u` or `m`, `jacobiSC` returns an unevaluated function call.

```
jacobiSC(sym(2),sym(3))
```

```
ans =
jacobiSC(2, 3)
```

Find Jacobi SC Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiSC` returns the unevaluated function call.

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

```
f =
jacobiSC(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiSC(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.0312
```

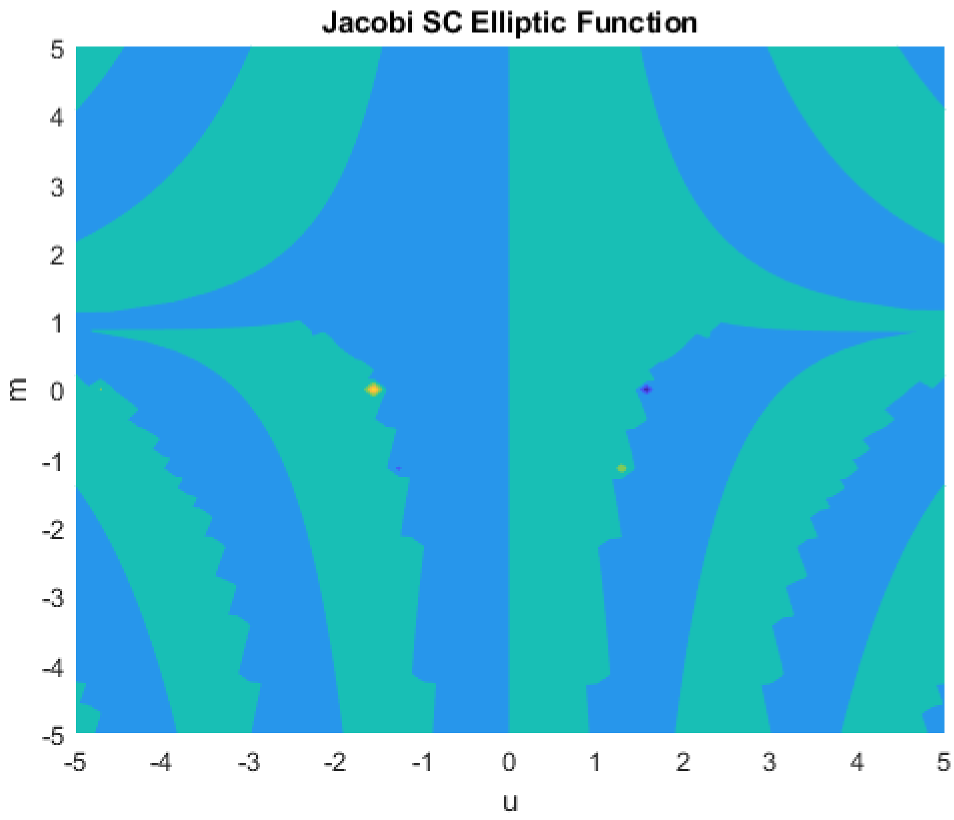
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.031159894327171581127518352857409
```

Plot Jacobi SC Elliptic Function

Plot the Jacobi SC elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiSC(u,m);
fcontour(f,'Fill','on')
title('Jacobi SC Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi SC Elliptic Function

The Jacobi SC elliptic function is
<mathphrase>$\mathrm{sc}(u,m) = \mathrm{sn}(u,m)/\mathrm{cn}(u,m)$</mathphrase>

where sn and cn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods 4K(m) and 4iK'(m), where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSD` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiSD

Jacobi SD elliptic function

Syntax

```
jacobiSD(u,m)
```

Description

`jacobiSD(u,m)` returns the “Jacobi SD Elliptic Function” on page 4-991 of `u` and `m`. If `u` or `m` is an array, then `jacobiSD` acts element-wise.

Examples

Calculate Jacobi SD Elliptic Function for Numeric Inputs

```
jacobiSD(2,1)
```

```
ans =
    3.6269
```

Call `jacobiSD` on array inputs. `jacobiSD` acts element-wise when `u` or `m` is an array.

```
jacobiSD([2 1 -3],[1 2 3])
```

```
ans =
    3.6269    2.1629 -126.3078
```

Calculate Jacobi SD Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi SD elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiSD` returns exact symbolic output.

```
jacobiSD(sym(2),sym(1))
```

```
ans =
sinh(2)
```

Show that for other values of `u` or `m`, `jacobiSD` returns an unevaluated function call.

```
jacobiSD(sym(2),sym(3))
```

```
ans =
jacobiSD(2, 3)
```

Find Jacobi SD Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiSD` returns the unevaluated function call.

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

```
f =
jacobiSD(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiSD(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.0312
```

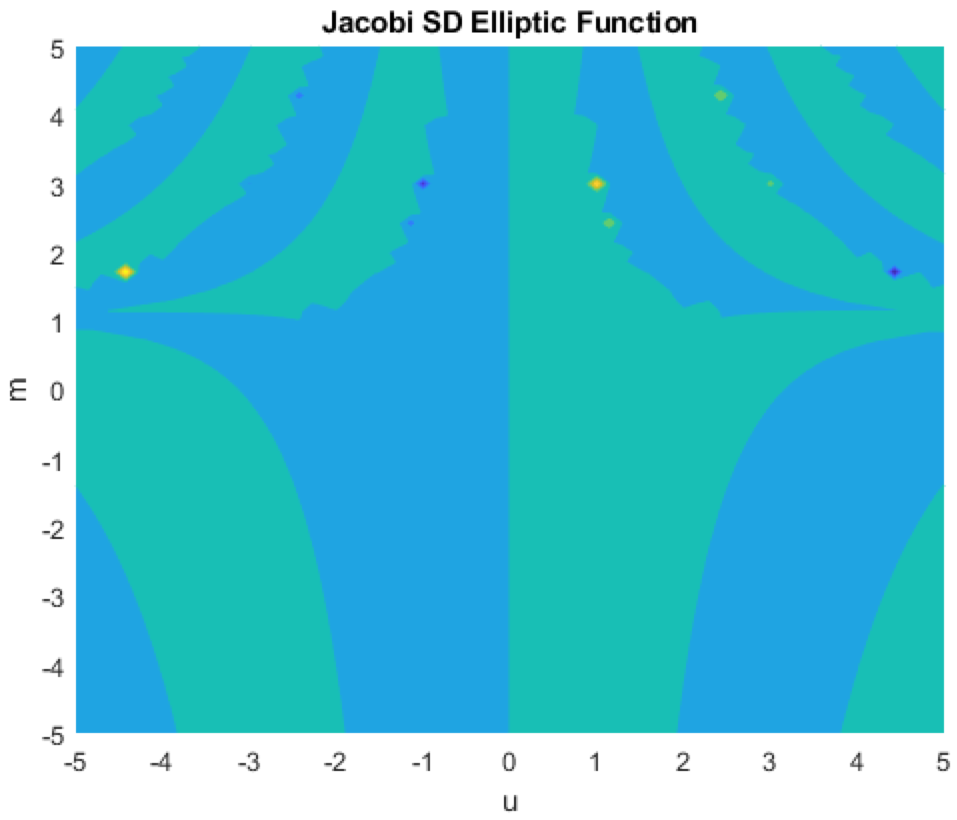
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.031220579864538785956650143970485
```

Plot Jacobi SD Elliptic Function

Plot the Jacobi SD elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiSD(u,m);
fcontour(f,'Fill','on')
title('Jacobi SD Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi SD Elliptic Function

The Jacobi SD elliptic function is
<mathphrase>sd(u,m) = sn(u,m)/dn(u,m)</mathphrase>

where sn and dn are the respective Jacobi elliptic functions.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods 4K(m) and 4iK'(m), where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSN` | `jacobiZeta`

Introduced in R2017b

jacobiSN

Jacobi SN elliptic function

Syntax

```
jacobiSN(u,m)
```

Description

`jacobiSN(u,m)` returns the "Jacobi SN Elliptic Function" on page 4-996 of `u` and `m`. If `u` or `m` is an array, then `jacobiSN` acts element-wise.

Examples

Calculate Jacobi SN Elliptic Function for Numeric Inputs

```
jacobiSN(2,1)
```

```
ans =
    0.9640
```

Call `jacobiSN` on array inputs. `jacobiSN` acts element-wise when `u` or `m` is an array.

```
jacobiSN([2 1 -3],[1 2 3])
```

```
ans =
    0.9640    0.6721    0.5773
```

Calculate Jacobi SN Elliptic Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi SN elliptic function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiSN` returns exact symbolic output.

```
jacobiSN(sym(2),sym(1))
```

```
ans =
tanh(2)
```

Show that for other values of `u` or `m`, `jacobiSN` returns an unevaluated function call.

```
jacobiSN(sym(2),sym(3))
```

```
ans =
jacobiSN(2, 3)
```

Find Jacobi SN Elliptic Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiSN` returns the unevaluated function call.

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

```
f =
jacobiSN(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiSN(3, 5)
```

```
fVal = double(f)
```

```
fVal =
    0.0311
```

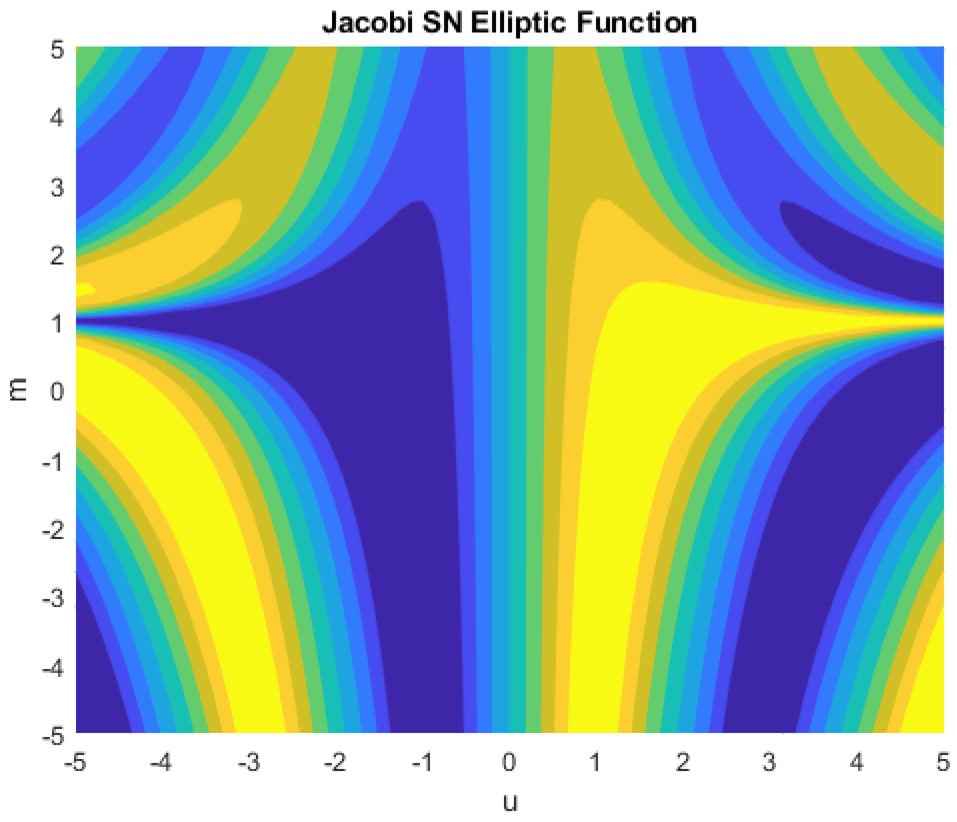
Calculate `f` to higher precision using `vpa`.

```
fVal = vpa(f)
fVal =
0.031144778155397389598324170696454
```

Plot Jacobi SN Elliptic Function

Plot the Jacobi SN elliptic function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiSN(u,m);
fcontour(f,'Fill','on')
title('Jacobi SN Elliptic Function')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi SN Elliptic Function

The Jacobi SN elliptic function is $\mathrm{sn}(u,m) = \sin(\mathrm{am}(u,m))$ where am is the Jacobi amplitude function.

The Jacobi elliptic functions are meromorphic and doubly periodic in their first argument with periods $4\mathrm{K}(m)$ and $4i\mathrm{K}'(m)$, where K is the complete elliptic integral of the first kind, implemented as `ellipticK`.

See Also

`ellipticK` | `jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiZeta`

Introduced in R2017b

jacobiZeta

Jacobi zeta function

Syntax

```
jacobiZeta(u,m)
```

Description

`jacobiZeta(u,m)` returns the “Jacobi Zeta Function” on page 4-1001 of `u` and `m`. If `u` or `m` is an array, then `jacobiZeta` acts element-wise.

Examples

Calculate Jacobi Zeta Function for Numeric Inputs

```
jacobiZeta(2,1)
```

```
ans =
    0.9640
```

Call `jacobiZeta` on array inputs. `jacobiZeta` acts element-wise when `u` or `m` is an array.

```
jacobiZeta([2 1 -3],[1 2 3])
```

```
ans =
   0.9640 + 0.0000i   0.5890 - 0.4569i  -2.3239 + 1.9847i
```

Calculate Jacobi Zeta Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Jacobi zeta function. For symbolic input where `u = 0` or `m = 0` or `1`, `jacobiZeta` returns exact symbolic output.

```
jacobiZeta(sym(2),sym(1))
```

```
ans =
tanh(2)
```

Show that for other values of `u` or `m`, `jacobiZeta` returns an unevaluated function call.

```
jacobiZeta(sym(2),sym(3))
```

```
ans =
jacobiZeta(2, 3)
```

Find Jacobi Zeta Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `jacobiZeta` returns the unevaluated function call.

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

```
f =
jacobiZeta(x, y)
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
f = subs(f, [x y], [3 5])
```

```
f =
jacobiZeta(3, 5)
```

```
fVal = double(f)
```

```
fVal =
   4.0986 - 3.0018i
```

Calculate `f` to arbitrary precision using `vpa`.

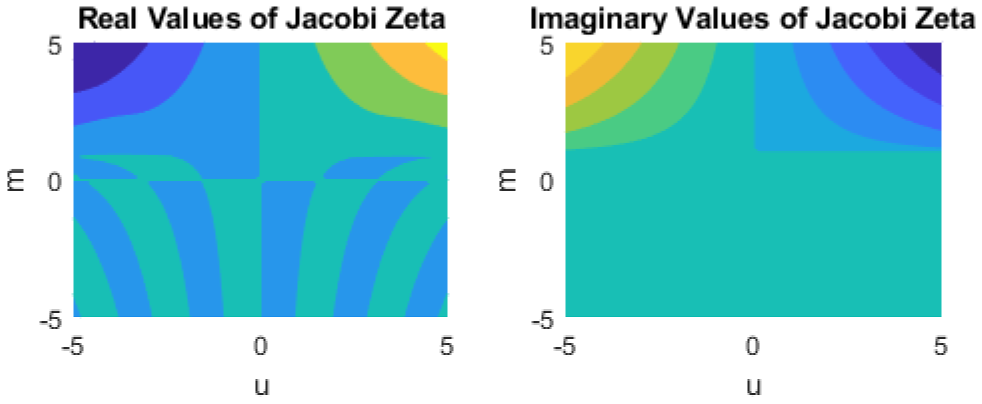
```
fVal = vpa(f)
```

```
fVal =
4.0986033838332279126523721581432 - 3.0017792319714320747021938869936i
```

Plot Jacobi Zeta Function

Plot real and imaginary values of the Jacobi zeta function using `fcontour`. Set `u` on the x-axis and `m` on the y-axis by using the symbolic function `f` with the variable order `(u,m)`. Fill plot contours by setting `Fill` to `on`.

```
syms f(u,m)
f(u,m) = jacobiZeta(u,m);

subplot(2,2,1)
fcontour(real(f),'Fill','on')
title('Real Values of Jacobi Zeta')
xlabel('u')
ylabel('m')

subplot(2,2,2)
fcontour(imag(f),'Fill','on')
title('Imaginary Values of Jacobi Zeta')
xlabel('u')
ylabel('m')
```



Input Arguments

u — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Jacobi Zeta Function

The Jacobi zeta function $\mathrm{Z}(u,m)$ is defined as

$$\mathrm{Z}(u,m)=\frac{2\pi}{\mathrm{K}(m)}\left(\sum_{i=1}^{\infty}\frac{\mathrm{q}(m)^i}{1-\mathrm{q}(m)^{2i}}\sin\left(\frac{2\pi}{\mathrm{K}(m)}iu\right)\right).$$

$\mathrm{K}(m)$ is the complete integral of the first kind, implemented as `ellipticK`. $\mathrm{q}(m)$ is the elliptic nome, implemented as `ellipticNome`.

See Also

`jacobiAM` | `jacobiCD` | `jacobiCN` | `jacobiCS` | `jacobiDC` | `jacobiDN` | `jacobiDS` | `jacobiNC` | `jacobiND` | `jacobiNS` | `jacobiSC` | `jacobiSD` | `jacobiSN`

Introduced in R2017b

jordan

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
```

```
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` | `hermiteForm` | `inv` | `smithForm`

Introduced before R2006a

kroneckerDelta

Kronecker delta function

Syntax

```
kroneckerDelta(m)
kroneckerDelta(m,n)
```

Description

`kroneckerDelta(m)` returns `1` if `m == 0` and `0` if `m ~= 0`.

`kroneckerDelta(m,n)` returns `1` if `m == n` and `0` if `m ~= n`.

Examples

Compare Two Symbolic Variables

Note For `kroneckerDelta` with numeric inputs, use the `eq` function instead.

Set symbolic variable `m` equal to symbolic variable `n` and test their equality using `kroneckerDelta`.

```
syms m n
m = n;
kroneckerDelta(m,n)
```

```
ans =
1
```

`kroneckerDelta` returns `1` indicating that the inputs are equal.

Compare symbolic variables `p` and `q`.

```
syms p q
kroneckerDelta(p,q)

ans =
kroneckerDelta(p - q, 0)
```

`kroneckerDelta` cannot decide if `p == q` and returns the function call with the undecidable input. Note that `kroneckerDelta(p, q)` is equal to `kroneckerDelta(p - q, 0)`.

To force a logical result for undecidable inputs, use `isAlways`. The `isAlways` function issues a warning and returns logical `0` (`false`) for undecidable inputs. Set the `Unknown` option to `false` to suppress the warning.

```
isAlways(kroneckerDelta(p, q), 'Unknown', 'false')

ans =
  logical
   0
```

Compare Symbolic Variable with Zero

Set symbolic variable `m` to `0` and test `m` for equality with `0`. The `kroneckerDelta` function errors because it does not accept numeric inputs of type `double`.

```
m = 0;
kroneckerDelta(m)

Undefined function 'kroneckerDelta' for input arguments of type 'double'.
```

Use `sym` to convert `0` to a symbolic object before assigning it to `m`. This is because `kroneckerDelta` only accepts symbolic inputs.

```
syms m
m = sym(0);
kroneckerDelta(m)

ans =
   1
```

`kroneckerDelta` returns `1` indicating that `m` is equal to `0`. Note that `kroneckerDelta(m)` is equal to `kroneckerDelta(m, 0)`.

Compare Vector of Numbers with Symbolic Variable

Compare a vector of numbers `[1 2 3 4]` with symbolic variable `m`. Set `m` to `3`.

```
V = 1:4
syms m
m = sym(3)
sol = kroneckerDelta(V,m)
```

```
V =
     1     2     3     4
m =
3
sol =
[ 0, 0, 1, 0]
```

`kroneckerDelta` acts on `V` element-wise to return a vector, `sol`, which is the same size as `V`. The third element of `sol` is `1` indicating that the third element of `V` equals `m`.

Compare Two Matrices

Compare matrices `A` and `B`.

Declare matrices `A` and `B`.

```
syms m
A = [m m+1 m+2;m-2 m-1 m]
B = [m m+3 m+2;m-1 m-1 m+1]
```

```
A =
[     m, m + 1, m + 2]
[ m - 2, m - 1,     m]
B =
[     m, m + 3, m + 2]
[ m - 1, m - 1, m + 1]
```

Compare `A` and `B` using `kroneckerDelta`.

```
sol = kroneckerDelta(A,B)
```

```
sol =
[ 1, 0, 1]
[ 0, 1, 0]
```

`kroneckerDelta` acts on `A` and `B` element-wise to return the matrix `sol` which is the same size as `A` and `B`. The elements of `sol` that are `1` indicate that the corresponding elements of `A` and `B` are equal. The elements of `sol` that are `0` indicate that the corresponding elements of `A` and `B` are not equal.

Use kroneckerDelta in Inputs to Other Functions

`kroneckerDelta` appears in the output of `iztrans`.

```
syms z n
sol = iztrans(1/(z-1), z, n)
```

```
sol =
1 - kroneckerDelta(n, 0)
```

Use this output as input to `ztrans` to return the initial input expression.

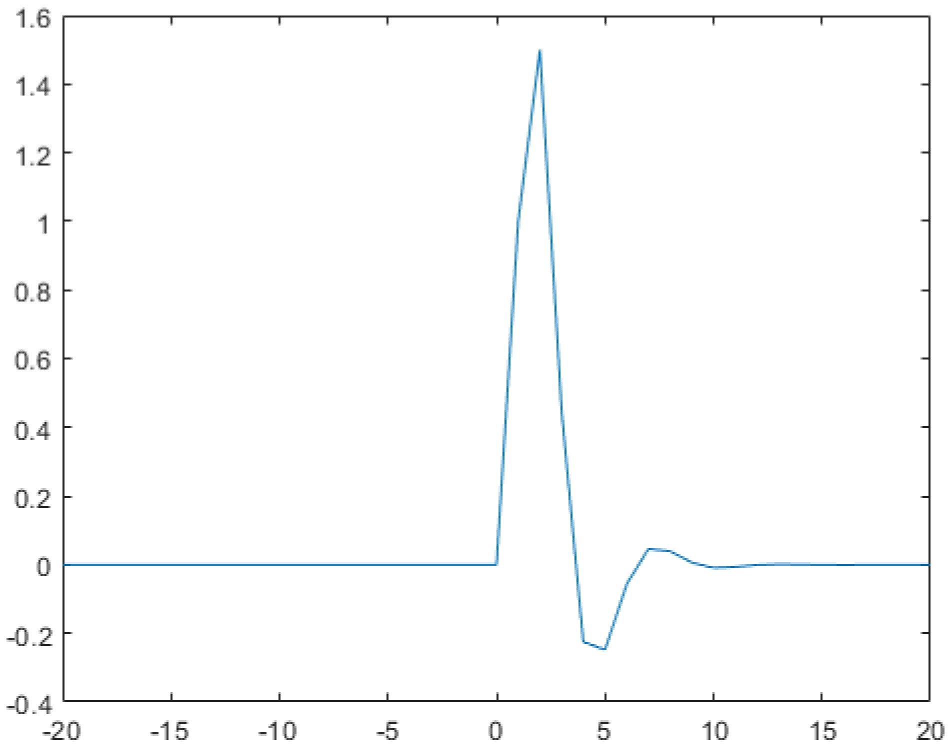
```
ztrans(sol, n, z)
```

```
ans =
z/(z - 1) - 1
```

Filter Response to Kronecker Delta Input

Use `filter` to find the response of a filter when the input is the Kronecker Delta function. Convert `k` to a symbolic vector using `sym` because `kroneckerDelta` only accepts symbolic inputs, and convert it back to double using `double`. Provide arbitrary filter coefficients `a` and `b` for simplicity.

```
b = [0 1 1];
a = [1 -0.5 0.3];
k = -20:20;
x = double(kroneckerDelta(sym(k)));
y = filter(b,a,x);
plot(k,y)
```



Input Arguments

`m` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array. At least one of the inputs, `m` or `n`, must be symbolic.

n — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array. At least one of the inputs, `m` or `n`, must be symbolic.

Definitions

Kronecker Delta Function

The Kronecker delta function is defined as

$$\delta(m,n) = \begin{cases} 0 & \text{if m} \neq n \\ 1 & \text{if m} = n \end{cases}$$

Tips

- When `m` or `n` is `NaN`, the `kroneckerDelta` function returns `NaN`.

See Also

`iztrans` | `ztrans`

Introduced in R2014b

kummerU

Confluent hypergeometric Kummer U function

Syntax

`kummerU(a,b,z)`

Description

`kummerU(a,b,z)` computes the value of confluent hypergeometric function, `U(a,b,z)`. If the real parts of `z` and `a` are positive values, then the integral representations of the Kummer U function is as follows:

$$U(a,b,z)=\frac{1}{\Gamma(a)}\int_0^\infty e^{-zt}t^{a-1}(1+t)^{b-a-1}\,dt$$

Examples

Equation Returning the Kummer U Function as Its Solution

`dsolve` can return solutions of second-order ordinary differential equations in terms of the Kummer U function.

Solve this equation. The solver returns the results in terms of the Kummer U function and another hypergeometric function.

```
syms t z y(z)
dsolve(z^3*diff(y,2) + (z^2 + t)*diff(y) + z*y)
```

```
ans =
(C4*hypergeom(1i/2, 1 + 1i, t/(2*z^2)))/z^1i +...
(C3*kummerU(1i/2, 1 + 1i, t/(2*z^2)))/z^1i
```

Kummer U Function for Numeric and Symbolic Arguments

Depending on its arguments, `kummerU` can return floating-point or exact symbolic results.

Compute the Kummer U function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
A = [kummerU(-1/3, 2.5, 2)
kummerU(1/3, 2, pi)
kummerU(1/2, 1/3, 3*i)]
```

```
A =
   0.8234 + 0.0000i
   0.7284 + 0.0000i
   0.4434 - 0.3204i
```

Compute the Kummer U function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `kummerU` returns unresolved symbolic calls.

```
symA = [kummerU(-1/3, 2.5, sym(2))
kummerU(1/3, 2, sym(pi))
kummerU(1/2, sym(1/3), 3*i)]
```

```
symA =
  kummerU(-1/3, 5/2, 2)
    kummerU(1/3, 2, pi)
 kummerU(1/2, 1/3, 3i)
```

Use `vpa` to approximate symbolic results with the required number of digits.

```
vpa(symA,10)
```

```
ans =
                 0.8233667846
                 0.7284037305
 0.4434362538 - 0.3204327531i
```

Some Special Values of Kummer U

The Kummer U function has special values for some parameters.

If `a` is a negative integer, the Kummer U function reduces to a polynomial.

```
syms a b z
[kummerU(-1, b, z)
kummerU(-2, b, z)
kummerU(-3, b, z)]
```

```
ans =
                                                                       z - b
                                                   b - 2*z*(b + 1) + b^2 + z^2
 6*z*(b^2/2 + (3*b)/2 + 1) - 2*b - 6*z^2*(b/2 + 1) - 3*b^2 - b^3 + z^3
```

If `b = 2*a`, the Kummer U function reduces to an expression involving the modified Bessel function of the second kind.

```
kummerU(a, 2*a, z)
```

```
ans =
(z^(1/2 - a)*exp(z/2)*besselk(a - 1/2, z/2))/pi^(1/2)
```

If `a = 1` or `a = b`, the Kummer U function reduces to an expression involving the incomplete gamma function.

```
kummerU(1, b, z)
```

```
ans =
z^(1 - b)*exp(z)*igamma(b - 1, z)
```

```
kummerU(a, a, z)
```

```
ans =
exp(z)*igamma(1 - a, z)
```

If `a = 0`, the Kummer U function is `1`.

```
kummerU(0, a, z)
```

```
ans =
1
```

Handle Expressions Containing the Kummer U Function

Many functions, such as `diff`, `int`, and `limit`, can handle expressions containing `kummerU`.

Find the first derivative of the Kummer U function with respect to `z`.

```
syms a b z
diff(kummerU(a, b, z), z)

ans =
(a*kummerU(a + 1, b, z)*(a - b + 1))/z - (a*kummerU(a, b, z))/z
```

Find the indefinite integral of the Kummer U function with respect to `z`.

```
int(kummerU(a, b, z), z)

ans =
((b - 2)/(a - 1) - 1)*kummerU(a, b, z) +...
(kummerU(a + 1, b, z)*(a - a*b + a^2))/(a - 1) -...
(z*kummerU(a, b, z))/(a - 1)
```

Find the limit of this Kummer U function.

```
limit(kummerU(1/2, -1, z), z, 0)

ans =
4/(3*pi^(1/2))
```

Input Arguments

`a` — Parameter of Kummer U function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Parameter of Kummer U function, specified as a number, variable, symbolic expression, symbolic function, or vector.

`b` — Parameter of Kummer U function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Parameter of Kummer U function, specified as a number, variable, symbolic expression, symbolic function, or vector.

`z` — Argument of Kummer U function

number | vector | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector

Argument of Kummer U function, specified as a number, variable, symbolic expression, symbolic function, or vector. If `z` is a vector, `kummerU(a,b,z)` is evaluated element-wise.

Definitions

Confluent Hypergeometric Function (Kummer U Function)

The confluent hypergeometric function (Kummer U function) is one of the solutions of the differential equation

$$z\frac{\partial^2}{\partial z^2}y+\left(b-z\right)\frac{\partial}{\partial z}y-ay=0$$

The other solution is the hypergeometric function ${}_1F_1(a,b,z)$.

The Whittaker W function can be expressed in terms of the Kummer U function:

$$W_{a,b}\left(z\right)=e^{-z/2}\ z^{b+1/2}\ U\left(b-a+\frac{1}{2},\,2b+1,\,z\right)$$

Tips

- `kummerU` returns floating-point results for numeric arguments that are not symbolic objects.
- `kummerU` acts element-wise on nonscalar inputs.
- All nonscalar arguments must have the same size. If one or two input arguments are nonscalar, then `kummerU` expands the scalars into vectors or matrices of the same size as the nonscalar arguments, with all elements equal to the corresponding scalar.

References

[1] Slater, L. J. “Confluent 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` | `whittakerW`

Introduced in R2014b

laguerreL

Generalized Laguerre Function and Laguerre Polynomials

Syntax

```
laguerreL(n,x)
laguerreL(n,a,x)
```

Description

`laguerreL(n,x)` returns the Laguerre polynomial of degree `n` if `n` is a nonnegative integer. When `n` is not a nonnegative integer, `laguerreL` returns the Laguerre function. For details, see "Generalized Laguerre Function" on page 4-1021.

`laguerreL(n,a,x)` returns the generalized Laguerre polynomial of degree `n` if `n` is a nonnegative integer. When `n` is not a nonnegative integer, `laguerreL` returns the generalized Laguerre function.

Examples

Find Laguerre Polynomials for Numeric and Symbolic Inputs

Find the Laguerre polynomial of degree `3` for input `4.3`.

```
laguerreL(3,4.3)
```

```
ans =
    2.5838
```

Find the Laguerre polynomial for symbolic inputs. Specify degree `n` as `3` to return the explicit form of the polynomial.

```
syms x
laguerreL(3,x)
```

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

If the degree of the Laguerre polynomial `n` is not specified, `laguerreL` cannot find the polynomial. When `laguerreL` cannot find the polynomial, it returns the function call.

```
syms n x
laguerreL(n,x)

ans =
laguerreL(n, x)
```

Find Generalized Laguerre Polynomial

Find the explicit form of the generalized Laguerre polynomial `L(n,a,x)` of degree `n = 2`.

```
syms a x
laguerreL(2,a,x)

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

Return Generalized Laguerre Function

When `n` is not a nonnegative integer, `laguerreL(n,a,x)` returns the generalized Laguerre function.

```
laguerreL(-2.7,3,2)

ans =
    0.2488
```

`laguerreL` is not defined for certain inputs and returns an error.

```
syms x
laguerreL(-5/2, -3/2, x)

Error using symengine (line 60)
The function 'laguerreL' is not defined for parameter values
'-5/2' and '-3/2'.
```

Find Laguerre Polynomial with Vector and Matrix Inputs

Find the Laguerre polynomials of degrees `1` and `2` by setting `n = [1 2]`.

```
syms x
laguerreL([1 2],x)

ans =
[ 1 - x, x^2/2 - 2*x + 1]
```

`laguerreL` acts element-wise on `n` to return a vector with two elements.

If multiple inputs are specified as a vector, matrix, or multidimensional array, the inputs must be the same size. Find the generalized Laguerre polynomials where input arguments `n` and `x` are matrices.

```
syms a
n = [2 3; 1 2];
xM = [x^2 11/7; -3.2 -x];
laguerreL(n,a,xM)

ans =
[ a^2/2 - a*x^2 + (3*a)/2 + x^4/2 - 2*x^2 + 1,...
      a^3/6 + (3*a^2)/14 - (253*a)/294 - 676/1029]
[                                     a + 21/5,...
          a^2/2 + a*x + (3*a)/2 + x^2/2 + 2*x + 1]
```

`laguerreL` acts element-wise on `n` and `x` to return a matrix of the same size as `n` and `x`.

Differentiate and Find Limits of Laguerre Polynomials

Use `limit` to find the limit of a generalized Laguerre polynomial of degree `3` as `x` tends to ∞.

```
syms x
expr = laguerreL(3,2,x);
limit(expr,x,Inf)

ans =
-Inf
```

Use `diff` to find the third derivative of the generalized Laguerre polynomial `laguerreL(n,a,x)`.

```
syms n a
expr = laguerreL(n,a,x);
diff(expr,x,3)

ans =
-laguerreL(n - 3, a + 3, x)
```

Find Taylor Series Expansion of Laguerre Polynomials

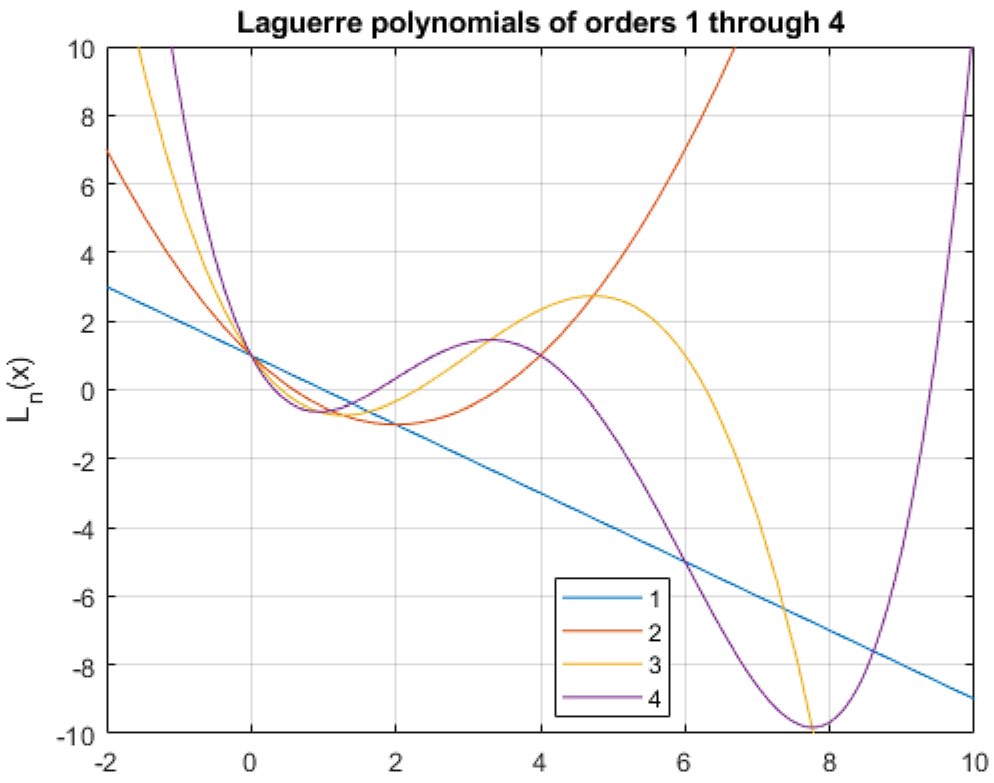
Use `taylor` to find the Taylor series expansion of the generalized Laguerre polynomial of degree `2` at `x = 0`.

```
syms a x
expr = laguerreL(2,a,x);
taylor(expr,x)

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

Plot Laguerre Polynomials

Plot the Laguerre polynomials of orders `1` through `4`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(laguerreL(1:4, x))
axis([-2  10 -10 10])
grid on

ylabel('L_n(x)')
title('Laguerre polynomials of orders 1 through 4')
legend('1','2','3','4','Location','best')
```



Input Arguments

`n` — Degree of polynomial

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Degree of polynomial, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array.

x — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array.

a — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array.

Definitions

Generalized Laguerre Function

The generalized Laguerre function is defined in terms of the hypergeometric function as

$$\text{laguerreL}(n,a,x) = \binom{n+a}{a} {}_1F_1(-n;a+1;x).$$

For nonnegative integer values of `n`, the function returns the generalized Laguerre polynomials that are orthogonal with respect to the scalar product

$$\langle f1, f2 \rangle = \int_0^\infty e^{-x} x^a f1(x) f2(x)\, dx.$$

In particular,

$$\langle \text{laguerreL}(n,a,x), \text{laguerreL}(m,a,x) \rangle = \begin{cases} 0 & \text{if } n \neq m \\ \dfrac{\Gamma(a+n+1)}{n!} & \text{if } n = m. \end{cases}$$

Algorithms

- The generalized Laguerre function is not defined for all values of parameters `n` and `a` because certain restrictions on the parameters exist in the definition of the hypergeometric functions. If the generalized Laguerre function is not defined for a particular pair of `n` and `a`, the `laguerreL` function returns an error message. See "Return Generalized Laguerre Function" on page 4-1017.
- The calls `laguerreL(n,x)` and `laguerreL(n,0,x)` are equivalent.
- If `n` is a nonnegative integer, the `laguerreL` function returns the explicit form of the corresponding Laguerre polynomial.
- The special values $\text{laguerreL}(n,a,0) = \binom{n+a}{a}$ are implemented for arbitrary values of `n` and `a`.
- If `n` is a negative integer and `a` is a numerical noninteger value satisfying $a \geq -n$, then `laguerreL` returns `0`.
- If `n` is a negative integer and `a` is an integer satisfying $a < -n$, the function returns an explicit expression defined by the reflection rule

  $$\text{laguerreL}(n,a,x) = (-1)^a \, e^x \, \text{laguerreL}(-n-a-1, a, -x)$$
- If all arguments are numerical and at least one argument is a floating-point number, then `laguerreL(x)` returns a floating-point number. For all other arguments, `laguerreL(n,a,x)` returns a symbolic function call.

See Also

`chebyshevT` | `chebyshevU` | `gegenbauerC` | `hermiteH` | `hypergeom` | `jacobiP` | `legendreP`

Introduced in R2014b

lambertw

Lambert W function

Syntax

```
lambertw(x)
lambertw(k,x)
```

Description

`lambertw(x)` is the Lambert W function on page 4-1031 of `x`, which returns the principal branch of the Lambert W function. Therefore, the syntax is equivalent to `lambertw(0,x)`.

`lambertw(k,x)` is the `k`th branch of the Lambert W function.

Examples

Equation Returning Lambert W Function as Its Solution

The Lambert W function `W(x)` is a set of solutions of the equation $x = W(x)e^{W(x)}$.

Solve this equation. The solutions is the Lambert W function.

```
syms x W
solve(x == W*exp(W), W)
```

```
ans =
lambertw(0, x)
```

Verify that various branches of the Lambert W function are valid solutions of the equation $x = W*e^{W}$:

```
k = -2:2
syms x
isAlways(x - subs(W*exp(W), W, lambertw(k,x)) == 0)
```

```
k =
    -2    -1     0     1     2

ans =
  1×5 logical array
     1     1     1     1     1
```

Lambert W Function for Numeric and Symbolic Arguments

Depending on its arguments, `lambertw` can return floating-point or exact symbolic results.

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

```
A = [0 -1/exp(1); pi i];
lambertw(A)
lambertw(-1, A)
```

```
ans =
   0.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0737 + 0.0000i   0.3747 + 0.5764i

ans =
     -Inf + 0.0000i  -1.0000 + 0.0000i
  -0.3910 - 4.6281i  -1.0896 - 2.7664i
```

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

```
A = [0 -1/exp(sym(1)); pi i];
W0 = lambertw(A)
Wmin1 = lambertw(-1, A)
```

```
W0 =
[              0,              -1]
[ lambertw(0, pi), lambertw(0, 1i)]

Wmin1 =
[             -Inf,               -1]
[ lambertw(-1, pi), lambertw(-1, 1i)]
```

Use `vpa` to approximate symbolic results with the required number of digits:

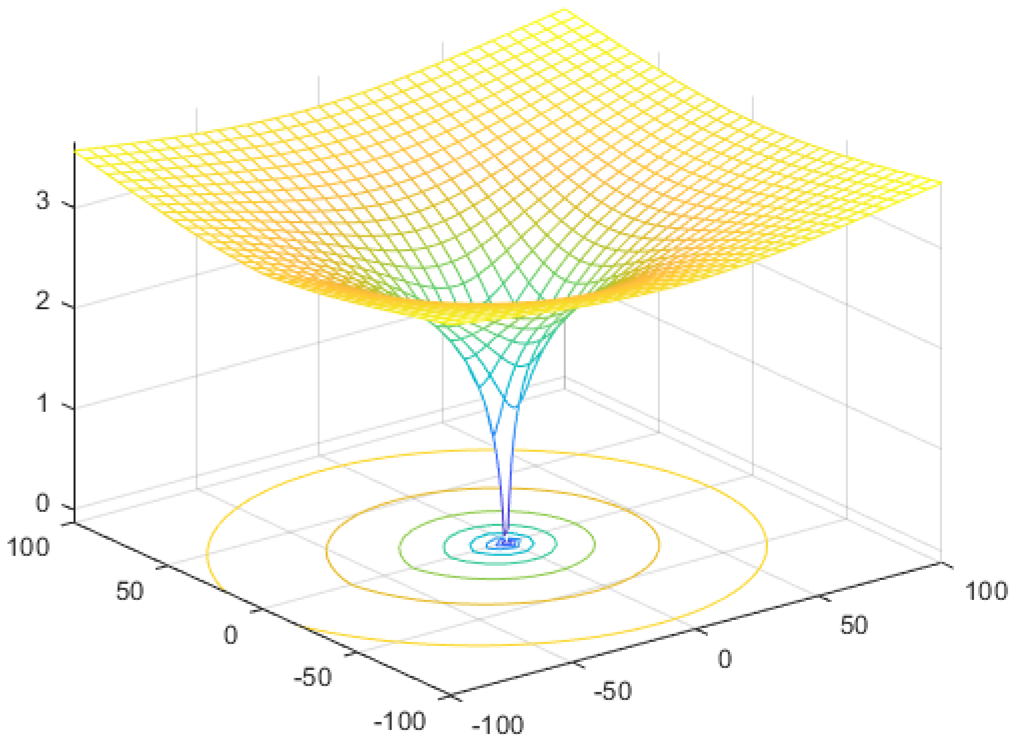
```
vpa(W0, 10)
vpa(Wmin1, 5)

ans =
[           0,                         -1.0]
[ 1.073658195, 0.3746990207 + 0.576412723i]

ans =
[                 -Inf,                -1.0]
[ - 0.39097 - 4.6281i, - 1.0896 - 2.7664i]
```

Lambert W Function Plot on Complex Plane

Plot the principal branch of the Lambert W function on the complex plane.

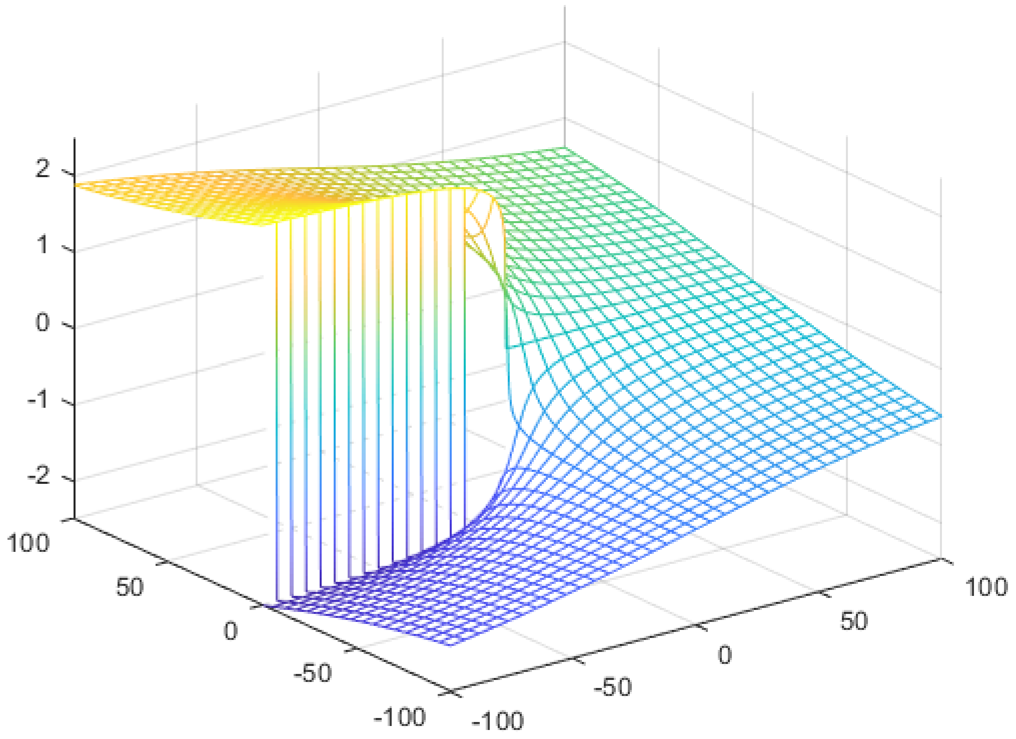
Create the combined mesh and contour plot of the real value of the Lambert W function on the complex plane.

```
syms x y real
fmesh(real(lambertw(x + i*y)), [-100, 100, -100, 100], 'ShowContours', 'on')
```

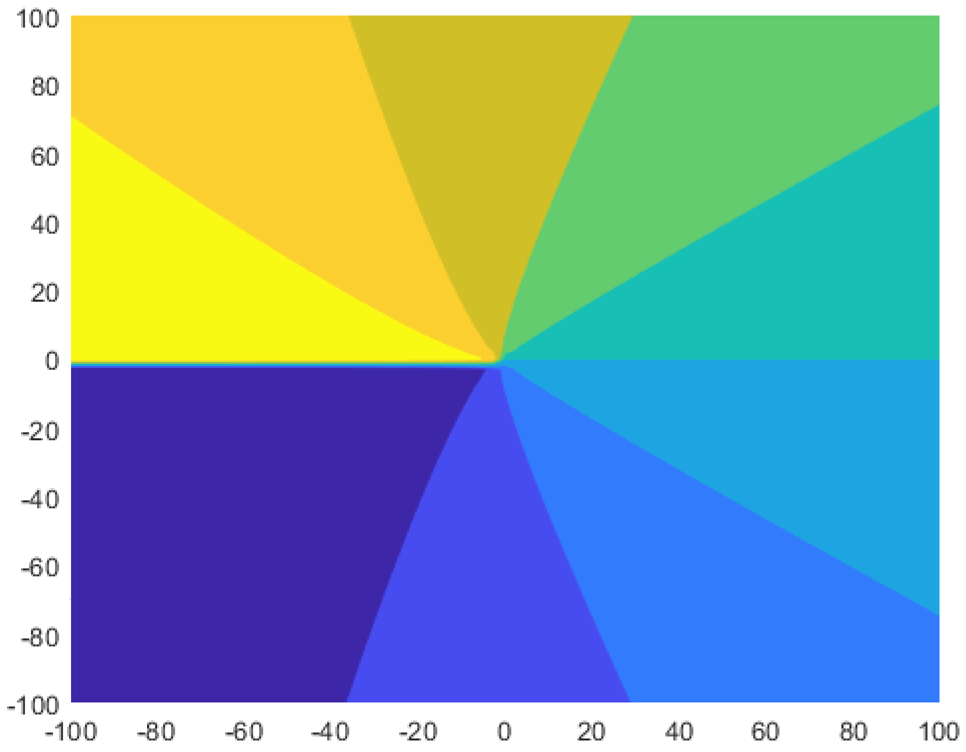


Now, plot the imaginary value of the Lambert W function on the complex plane. This function has a branch cut along the negative real axis. For better perspective, create the mesh and contour plots separately.

```
fmesh(imag(lambertw(x + i*y)), [-100, 100, -100, 100])
```

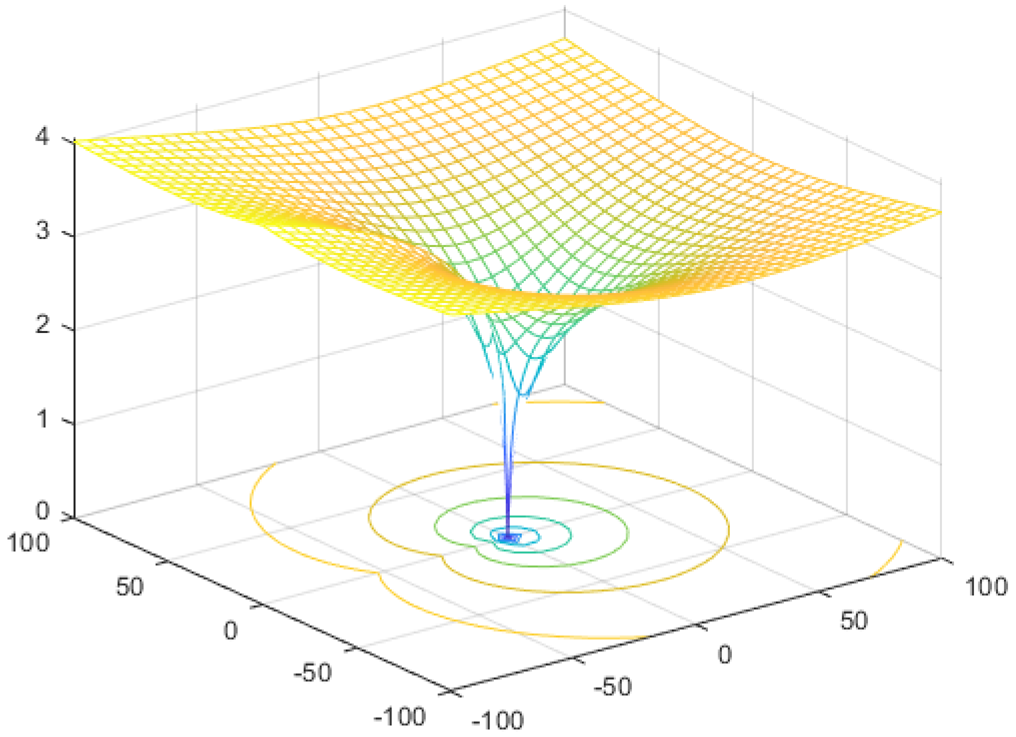


```
fcontour(imag(lambertw(x + i*y)), [-100, 100, -100, 100], 'Fill', 'on')
```



Plot the absolute value of the Lambert W function on the complex plane.

```
fmesh(abs(lambertw(x + i*y)), [-100, 100, -100, 100], 'ShowContours', 'on')
```



For further computations, clear the assumptions on `x` and `y`:

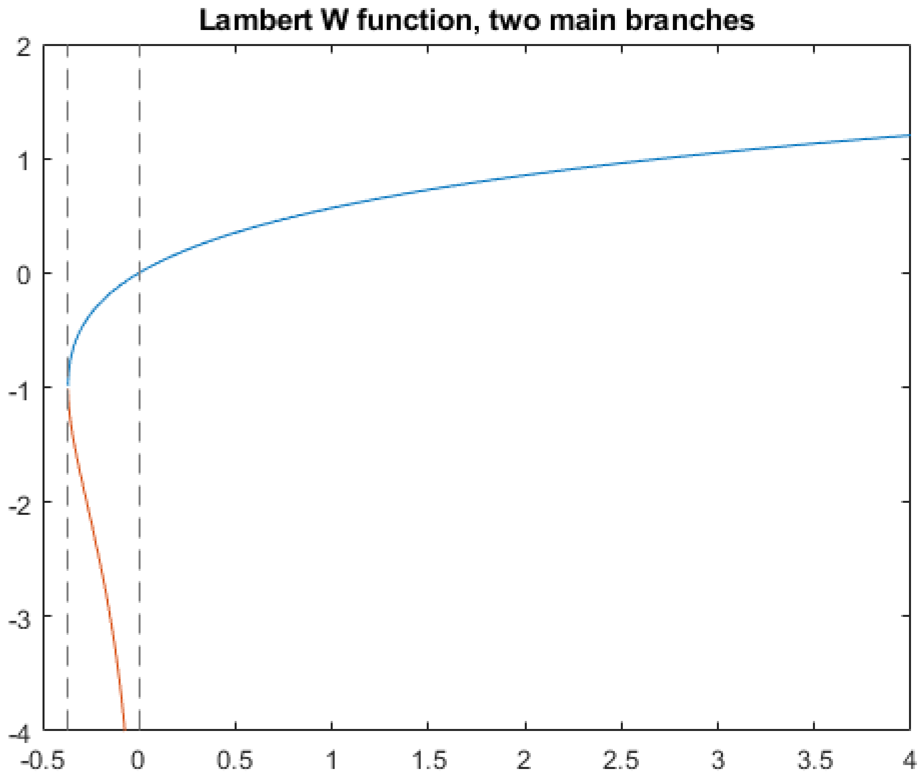
```
syms x y clear
```

Plot Two Main Branches of `Lambert W` function

Plot the two main branches, $W_0(x)$ and $W_{-1}(x)$, of the Lambert W function. Before R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(lambertw(x))
hold on
```

```
fplot(lambertw(-1, x))
axis([-0.5, 4, -4, 2])
title('Lambert W function, two main branches')
```



Input Arguments

x — Argument of Lambert W function

number | symbolic number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Argument of Lambert W function, specified as a number, symbolic number, variable, expression, function, or vector or matrix of numbers, symbolic numbers, variables, expressions, or functions. If `x` is a vector or matrix, `lambertW` returns the Lambert W function for each element of `x`.

`k` — Branch of Lambert W function
integer | vector | matrix

Branch of Lambert W function, specified as an integer or a vector or matrix of integers. If `k` is a vector or matrix, `lambertW` returns the Lambert W function for each element of `k`.

Definitions

Lambert W Function

The Lambert W function W(x) represents the solutions y of the equation $ye^y = x$ for any complex number `x`.

- For complex x, the equation has an infinite number of solutions y = lambertW(k,x) where k ranges over all integers.
- For real x where $x \geq 0$, the equation has exactly one real solution y = lambertW(x) = lambertW(0,x).
- For real x where $-e^{-1} < x < 0$, the equation has exactly two real solutions. The larger solution is represented by y = lambertW(x) and the smaller solution by y = lambertW(-1,x).
- For $x = -e^{-1}$, the equation has exactly one real solution y = -1 = lambertW(0, -exp(-1)) = lambertW(-1, -exp(-1)).

Algorithms

- The equation $x = w(x)\mathrm{e}^{w(x)}$ has infinitely many solutions on the complex plane. These solutions are represented by `w = lambertw(k,x)` with the *branch index* `k` ranging over the integers.
- For all real $x \geq 0$, the equation $x = w(x)\mathrm{e}^{w(x)}$ has exactly one real solution. It is represented by `w = lambertw(x)` or, equivalently, `w = lambertw(0,x)`.

- For all real x in the range -1/e < x < 0, there are exactly two distinct real solutions. The larger one is represented by `w = lambertw(x)`, and the smaller one is represented by `w = lambertw(-1,x)`.
- For x = -1/e, there is exactly one real solution `lambertw(0,-exp(-1)) = lambertw(-1,-exp(-1)) = -1`.
- `lambertw(k,x)` returns real values only if `k = 0` or `k = -1`.
- At least one input argument must be a scalar or both arguments must be vectors or matrices of the same size. If one input argument is a scalar and the other one is a vector or a matrix, `lambertw` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

References

[1] Corless, R.M, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, and D.E. Knuth “On the Lambert W Function” *Advances in Computational Mathematics*, vol.5, pp. 329–359, 1996.

See Also

Functions
`wrightOmega`

Introduced before R2006a

laplace

Laplace transform

Syntax

```
laplace(f)
laplace(f,transVar)
laplace(f,var,transVar)
```

Description

`laplace(f)` returns the “Laplace Transform” on page 4-1037 of `f`. By default, the independent variable is `t` and transformation variable is `s`.

`laplace(f,transVar)` uses the transformation variable `transVar` instead of `s`.

`laplace(f,var,transVar)` uses the independent variable `var` and the transformation variable `transVar` instead of `t` and `s`, respectively.

Examples

Laplace Transform of Symbolic Expression

Compute the Laplace transform of `1/sqrt(x)`. By default, the transform is in terms of `s`.

```
syms x y
f = 1/sqrt(x);
laplace(f)
```

```
ans =
pi^(1/2)/s^(1/2)
```

Specify Independent Variable and Transformation Variable

Compute the Laplace transform of `exp(-a*t)`. By default, the independent variable is `t`, and the transformation variable is `s`.

```
syms a t
f = exp(-a*t);
laplace(f)
```

```
ans =
1/(a + s)
```

Specify the transformation variable as `y`. If you specify only one variable, that variable is the transformation variable. The independent variable is still `t`.

```
laplace(f,y)
```

```
ans =
1/(a + y)
```

Specify both the independent and transformation variables as `a` and `y` in the second and third arguments, respectively.

```
laplace(f,a,y)
```

```
ans =
1/(t + y)
```

Laplace Transforms of Dirac and Heaviside Functions

Compute the Laplace transforms 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
```

Relation Between Laplace Transform of Function and it's Derivative

Show that the Laplace transform of the derivative of a function is expressed in terms of the Laplace transform of the function itself.

```
syms f(t) s
Df = diff(f(t),t);
laplace(Df,t,s)
```

```
ans =
s*laplace(f(t), t, s) - f(0)
```

Laplace Transform of Array Inputs

Find the Laplace transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `laplace` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x) 1; sin(y) i*z];
vars = [w x; y z];
transVars = [a b; c d];
laplace(M,vars,transVars)
```

```
ans =
[    exp(x)/a,   1/b]
[ 1/(c^2 + 1), 1i/d^2]
```

If `laplace` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
laplace(x,vars,transVars)
```

```
ans =
[ x/a, 1/b^2]
[ x/c,   x/d]
```

Laplace Transform of Symbolic Function

Compute the Laplace transform of symbolic functions. When the first argument contains symbolic functions, then the second argument must be a scalar.

```
syms f1(x) f2(x) a b
f1(x) = exp(x);
f2(x) = x;
laplace([f1 f2],x,[a b])

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

If Laplace Transform Cannot Be Found

If `laplace` cannot transform the input then it returns an unevaluated call.

```
syms f(t) s
f(t) = 1/t;
F = laplace(f,t,s)

F =
laplace(1/t, t, s)
```

Return the original expression by using `ilaplace`.

```
ilaplace(F,s,t)

ans =
1/t
```

Input Arguments

`f` — Input

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

`t` (default) | symbolic variable

Independent variable, specified as a symbolic variable. This variable is often called the "time variable" or the "space variable." If you do not specify the variable then, by default, `laplace` uses `t`. If `f` does not contain `t`, then `laplace` uses the function `symvar` to determine the independent variable.

`transVar` — Transformation variable
`s` (default) | `z` | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. This variable is often called the "complex frequency variable." If you do not specify the variable then, by default, `laplace` uses `s`. If `s` is the independent variable of `f`, then `laplace` uses `z`.

Definitions

Laplace Transform

The Laplace transform $F = F(s)$ of the expression $f = f(t)$ with respect to the variable t at the point s is

$$F(s) = \int_0^\infty f(t)\, e^{-st} dt.$$

Tips

- If any argument is an array, then `laplace` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.
- To compute the inverse Laplace transform, use `ilaplace`.

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `ztrans`

Topics

“Solve Differential Equations Using Laplace Transform” on page 2-225

Introduced before R2006a

laplacian

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`.

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`.

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
```

Definitions

Laplacian of Scalar Function

The Laplacian of the scalar function or functional expression *f* with respect to the vector $X = (X_1,\ldots,X_n)$ is the sum of the second derivatives of *f* with respect to $X_1,\ldots,X_n$:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f_i}{\partial x_i^2}$$

Tips

- If `x` is a scalar, `gradient(f, x) = diff(f, 2, x)`.

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`

Introduced in R2012a

latex

LaTeX form of symbolic expression

Syntax

```
latex(S)
```

Description

`latex(S)` returns the LaTeX form of the symbolic expression `S`.

Examples

LaTeX Form of Symbolic Expression

Find the LaTeX form of the symbolic expressions `x^2 + 1/x` and `sin(pi*x) + alpha`.

```
syms x phi
latex(x^2 + 1/x)
latex(sin(pi*x) + phi)
```

```
ans =
    '\frac{1}{x}+x^2'

ans =
    '\varphi +\sin\left(\pi \,x\right)'
```

LaTeX Form of Symbolic Matrix

Find the LaTeX form of the symbolic matrix `M`.

```
syms x
M = [sym(1)/3 x; exp(x) x^2]
latexM = latex(M)
```

```
M =
[    1/3,    x]
[ exp(x), x^2]

latexM =
    '\left(\begin{array}{cc} \frac{1}{3} & x\\ {\mathrm{e}}^x & x^2 \end{array}\right)'
```

Use LaTeX to Format Title, Axis Labels, and Ticks

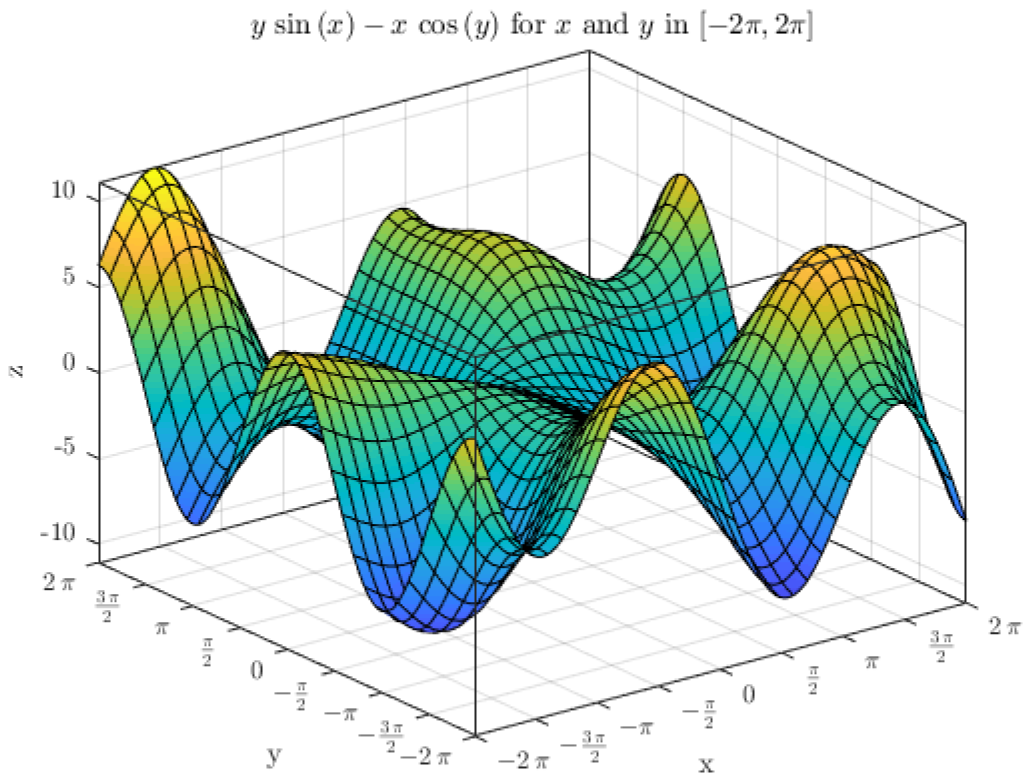
For x and y from -2π to 2π, plot the 3-D surface $y\sin(x) - x\cos(y)$. Store the axes handle in `a` by using `gca`. Display the axes box by using `a.Box` and set the tick label interpreter to `latex`.

Create the x-axis ticks by spanning the x-axis limits at intervals of `pi/2`. Convert the axis limits to precise multiples of `pi/2` using `round` and get the symbolic tick values in `S`. Display the ticks by setting the `XTick` property of `a` to `S`. Create the LaTeX labels for the x-axis by using `arrayfun` to apply `latex` to `S` and then concatenating `$`. Display the labels by assigning them to the `XTickLabel` property of `a`.

Repeat these steps for the y-axis. Set the x- and y-axes labels and the title using the `latex` interpreter.

```
syms x y
f = y.*sin(x)-x.*cos(y);
fsurf(f,[-2*pi 2*pi])
a = gca;
a.TickLabelInterpreter = 'latex';
a.Box = 'on';
a.BoxStyle = 'full';

S = sym(a.XLim(1):pi/2:a.XLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
a.XTick = double(S);
a.XTickLabel = strcat('$',arrayfun(@latex, S, 'UniformOutput', false),'$');

S = sym(a.YLim(1):pi/2:a.YLim(2));
S = sym(round(vpa(S/pi*2))*pi/2);
a.YTick = double(S);
a.YTickLabel = strcat('$',arrayfun(@latex, S, 'UniformOutput', false),'$');

xlabel('x','Interpreter','latex');
ylabel('y','Interpreter','latex');
```

```
zlabel('z','Interpreter','latex');
title(['$' latex(f) '$ for $x$ and $y$ in $[-2\pi,2\pi]$'],'Interpreter','latex')
```



Input Arguments

s — Input

symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

See Also

`ccode` | `fortran` | `pretty` | `texlabel`

Introduced before R2006a

lcm

Least common multiple

Syntax

```
lcm(A)
lcm(A,B)
```

Description

`lcm(A)` finds the least common multiple of all elements of `A`.

`lcm(A,B)` finds the least common multiple of `A` and `B`.

Examples

Least Common Multiple of Four Integers

To find the least common multiple of three or more values, specify those values as a symbolic vector or matrix.

Find the least common multiple of these four integers, specified as elements of a symbolic vector.

```
A = sym([4420, -128, 8984, -488])
lcm(A)
```

```
A =
[ 4420, -128, 8984, -488]

ans =
9689064320
```

Alternatively, specify these values as elements of a symbolic matrix.

```
A = sym([4420, -128; 8984, -488])
lcm(A)

A =
[ 4420, -128]
[ 8984, -488]

ans =
9689064320
```

Least Common Multiple of Rational Numbers

`lcm` lets you find the least common multiple of symbolic rational numbers.

Find the least common multiple of these rational numbers, specified as elements of a symbolic vector.

```
lcm(sym([3/4, 7/3, 11/2, 12/3, 33/4]))

ans =
924
```

Least Common Multiple of Complex Numbers

`lcm` lets you find the least common multiple of symbolic complex numbers.

Find the least common multiple of these complex numbers, specified as elements of a symbolic vector.

```
lcm(sym([10 - 5*i, 20 - 10*i, 30 - 15*i]))

ans =
- 60 + 30i
```

Least Common Multiple of Elements of Matrices

For vectors and matrices, `lcm` finds the least common multiples element-wise. Nonscalar arguments must be the same size.

Find the least common multiples for the elements of these two matrices.

```
A = sym([309, 186; 486, 224]);
B = sym([558, 444; 1024, 1984]);
lcm(A,B)

ans =
[  57474, 13764]
[ 248832, 13888]
```

Find the least common multiples for the elements of matrix `A` and the value `99`. Here, `lcm` expands `99` into the `2`-by-`2` matrix with all elements equal to `99`.

```
lcm(A,99)

ans =
[ 10197,  6138]
[  5346, 22176]
```

Least Common Multiple of Polynomials

Find the least common multiple of univariate and multivariate polynomials.

Find the least common multiple of these univariate polynomials.

```
syms x
lcm(x^3 - 3*x^2 + 3*x - 1, x^2 - 5*x + 4)

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

Find the least common multiple of these multivariate polynomials. Because there are more than two polynomials, specify them as elements of a symbolic vector.

```
syms x y
lcm([x^2*y + x^3, (x + y)^2, x^2 + x*y^2 + x*y + x + y^3 + y])

ans =
(x^3 + y*x^2)*(x^2 + x*y^2 + x*y + x + y^3 + y)
```

Input Arguments

`A` — Input value

number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input value, specified as a number, symbolic number, variable, expression, function, or a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

`B` — Input value

number | symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input value, specified as a number, symbolic number, variable, expression, function, or a vector or matrix of numbers, symbolic numbers, variables, expressions, or functions.

Tips

- Calling `lcm` for numbers that are not symbolic objects invokes the MATLAB `lcm` function.
- The MATLAB `lcm` function does not accept rational or complex arguments. To find the least common multiple of rational or complex numbers, convert these numbers to symbolic objects by using `sym`, and then use `lcm`.
- Nonscalar arguments must have the same size. If one input arguments is nonscalar, then `lcm` expands the scalar into a vector or matrix of the same size as the nonscalar argument, with all elements equal to the corresponding scalar.

See Also

`gcd`

Introduced in R2014b

ldivide.\

Symbolic array left division

Syntax

```
B.\A
ldivide(B,A)
```

Description

`B.\A` divides `A` by `B`.

`ldivide(B,A)` is equivalent to `B.\A`.

Examples

Divide Scalar by Matrix

Create a `2`-by-`3` matrix.

```
B = sym('b', [2 3])
```

```
B =
[ b1_1, b1_2, b1_3]
[ b2_1, b2_2, b2_3]
```

Divide the symbolic expression `sin(a)` by each element of the matrix `B`.

```
syms a
B.\sin(a)
```

```
ans =
[ sin(a)/b1_1, sin(a)/b1_2, sin(a)/b1_3]
[ sin(a)/b2_1, sin(a)/b2_2, sin(a)/b2_3]
```

Divide Matrix by Matrix

Create a 3-by-3 symbolic Hilbert matrix and a 3-by-3 diagonal matrix.

```
H = sym(hilb(3))
d = diag(sym([1 2 3]))
```

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

d =
[ 1, 0, 0]
[ 0, 2, 0]
[ 0, 0, 3]
```

Divide d by H by using the elementwise left division operator .\. This operator divides each element of the first matrix by the corresponding element of the second matrix. The dimensions of the matrices must be the same.

```
H.\d
```

```
ans =
[ 1, 0,  0]
[ 0, 6,  0]
[ 0, 0, 15]
```

Divide Expression by Symbolic Function

Divide a symbolic expression by a symbolic function. The result is a symbolic function.

```
syms f(x)
f(x) = x^2;
f1 = f.\(x^2 + 5*x + 6)
```

```
f1(x) =
(x^2 + 5*x + 6)/x^2
```

Input Arguments

`A` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

`B` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

See Also

`ctranspose` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

le

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`.

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
```

Tips

- Calling <= or `le` for non-symbolic `A` and `B` invokes the MATLAB `le` function. This function returns a logical array with elements set to logical `1 (true)` where `A` is less than or equal to `B`; otherwise, it returns logical `0 (false)`.
- 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`.

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` | `ne`

Topics

"Set Assumptions" on page 1-28

Introduced in R2012a

legendreP

Legendre polynomials

Syntax

```
legendreP(n,x)
```

Description

`legendreP(n,x)` returns the `n`th degree Legendre polynomial on page 4-1060 at `x`.

Examples

Find Legendre Polynomials for Numeric and Symbolic Inputs

Find the Legendre polynomial of degree `3` at `5.6`.

```
legendreP(3,5.6)
```

```
ans =
  430.6400
```

Find the Legendre polynomial of degree `2` at `x`.

```
syms x
legendreP(2,x)
```

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

If you do not specify a numerical value for the degree `n`, the `legendreP` function cannot find the explicit form of the polynomial and returns the function call.

```
syms n
legendreP(n,x)
```

```
ans =
legendreP(n, x)
```

Find Legendre Polynomial with Vector and Matrix Inputs

Find the Legendre polynomials of degrees `1` and `2` by setting `n = [1 2]`.

```
syms x
legendreP([1 2],x)

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

`legendreP` acts element-wise on `n` to return a vector with two elements.

If multiple inputs are specified as a vector, matrix, or multidimensional array, the inputs must be the same size. Find the Legendre polynomials where input arguments `n` and `x` are matrices.

```
n = [2 3; 1 2];
xM = [x^2 11/7; -3.2 -x];
legendreP(n,xM)

ans =
[ (3*x^4)/2 - 1/2,        2519/343]
[           -16/5, (3*x^2)/2 - 1/2]
```

`legendreP` acts element-wise on `n` and `x` to return a matrix of the same size as `n` and `x`.

Differentiate and Find Limits of Legendre Polynomials

Use `limit` to find the limit of a Legendre polynomial of degree `3` as `x` tends to -∞.

```
syms x
expr = legendreP(4,x);
limit(expr,x,-Inf)

ans =
Inf
```

Use `diff` to find the third derivative of the Legendre polynomial of degree `5`.

```
syms n
expr = legendreP(5,x);
diff(expr,x,3)

ans =
(945*x^2)/2 - 105/2
```

Find Taylor Series Expansion of Legendre Polynomial

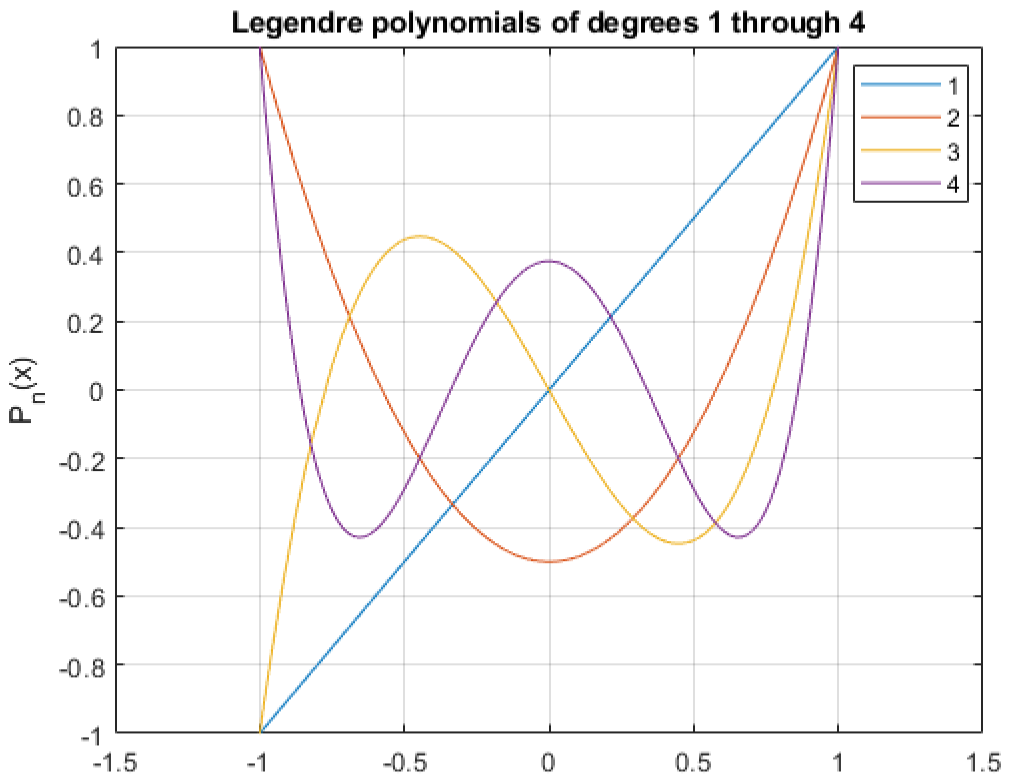
Use `taylor` to find the Taylor series expansion of the Legendre polynomial of degree `2` at `x = 0`.

```
syms x
expr = legendreP(2,x);
taylor(expr,x)

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

Plot Legendre Polynomials

Plot Legendre polynomials of orders `1` through `4`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x y
fplot(legendreP(1:4, x))
axis([-1.5 1.5 -1 1])
grid on

ylabel('P_n(x)')
title('Legendre polynomials of degrees 1 through 4')
legend('1','2','3','4','Location','best')
```



Find Roots of Legendre Polynomial

Use `vpasolve` to find the roots of the Legendre polynomial of degree `7`.

```
syms x
roots = vpasolve(legendreP(7,x) == 0)
```

```
roots =
 -0.94910791234275852452618968404785
 -0.74153118559939443986386477328079
 -0.40584515137739716690660641207696
                                   0
  0.40584515137739716690660641207696
```

```
   0.74153118559939443986386477328079
   0.94910791234275852452618968404785
```

Input Arguments

n — Degree of polynomial

nonnegative number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Degree of polynomial, specified as a nonnegative number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array. All elements of nonscalar inputs should be nonnegative integers or symbols.

x — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic function | symbolic multidimensional array

Input, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, function, or multidimensional array.

Definitions

Legendre Polynomial

The Legendre polynomials are defined as

$$P(n,x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left(x^2 - 1\right)^n.$$

They satisfy the recursion formula

$$P(n,x) = \frac{2n-1}{n}xP(n-1,x) - \frac{n-1}{n}P(n-2,x),$$

where

$$P(0,x) = 1$$

$$P(1,x) = x.$$

The Legendre polynomials are orthogonal on the interval [-1,1] with respect to the weight function $w(x) = 1$.

The relation with Gegenbauer polynomials G(n,a,x) is

$$P(n,x) = G\left(n,\frac{1}{2},x\right).$$

The relation with Jacobi polynomials P(n,a,b,x) is

$$P(n,x) = P(n,0,0,x).$$

See Also

`chebyshevT` | `chebyshevU` | `gegenbauerC` | `hermiteH` | `hypergeom` | `jacobiP` | `laguerreL`

Introduced in R2014b

lhs

Left side (LHS) of equation

Syntax

```
lhs(eqn)
```

Description

`lhs(eqn)` returns the left side of the symbolic equation `eqn`. The value of `eqn` also can be a symbolic condition, such as `x > 0`. If `eqn` is an array, then `lhs` returns an array of the left sides of the equations in `eqn`.

Examples

Find Left Side of Equation

Find the left side of the equation `2*y == x^2` by using `lhs`.

First, declare the equation.

```
syms x y
eqn = 2*y == x^2
```

```
eqn =
2*y == x^2
```

Find the left side of `eqn` by using `lhs`.

```
lhsEqn = lhs(eqn)
```

```
lhsEqn =
2*y
```

Find Left Side of Condition

Find the left side of the condition `x + y < 1` by using `lhs`.

First, declare the condition.

```
syms x y
cond = x + y < 1
```

```
cond =
 x + y < 1
```

Find the left side of `cond` by using `lhs`.

```
lhsCond = lhs(cond)
```

```
lhsCond =
 x + y
```

Note Conditions that use the > operator are internally rewritten using the < operator. Therefore, `lhs` returns the original right side. For example, `lhs(x > a)` returns `a`.

Find Left Side of Equations in Array

For an array that contains equations and conditions, `lhs` returns an array of the left sides of those equations or conditions. The output array is the same size as the input array.

Find the left side of the equations and conditions in the vector `V`.

```
syms x y
V = [y^2 == x^2, x ~= 0, x*y >= 1]
```

```
V =
[ y^2 == x^2, x ~= 0, 1 <= x*y]
```

```
lhsV = lhs(V)
```

```
lhsV =
[ y^2, x, 1]
```

Because any condition using the >= operator is internally rewritten using the <= operator, the sides of the last condition in `V` are exchanged.

Input Arguments

`eqn` — Equation or condition
symbolic equation | symbolic condition | vector of symbolic equations or conditions | matrix of symbolic equations or conditions | multidimensional array of symbolic equations or conditions

Equation or condition, specified as a symbolic equation or condition, or a vector, matrix, or multidimensional array of symbolic equations or conditions.

See Also

`assume` | `children` | `rhs` | `subs`

Introduced in R2017a

limit

Limit of symbolic expression

Syntax

```
limit(f,var,a)
limit(f,a)
limit(f)

limit(f,var,a,'left')

limit(f,var,a,'right')
```

Description

`limit(f,var,a)` returns the “Bidirectional Limit” on page 4-1067 of the symbolic expression `f` when `var` approaches `a`.

`limit(f,a)` uses the default variable found by `symvar`.

`limit(f)` returns the limit at `0`.

`limit(f,var,a,'left')` returns the “Left Side Limit” on page 4-1067 of `f` as `var` approaches `a`.

`limit(f,var,a,'right')` returns the “Right Side Limit” on page 4-1067.

Examples

Limit of Symbolic Expression

Calculate the bidirectional limit of this symbolic expression as `x` approaches `0`.

```
syms x h
f = sin(x)/x;
limit(f,x,0)

ans =
1
```

Calculate the limit of this expression as `h` approaches `0`.

```
f = (sin(x+h)-sin(x))/h;
limit(f,h,0)

ans =
cos(x)
```

Right and Left Limits of Symbolic Expression

Calculate the right and left limits of symbolic expressions.

```
syms x
f = 1/x;
limit(f,x,0,'right')

ans =
Inf

limit(f,x,0,'left')

ans =
-Inf
```

Limit of Expressions in Symbolic Vector

Calculate the limit of expressions in a symbolic vector. `limit` acts element-wise on the vector.

```
syms x a
V = [(1+a/x)^x exp(-x)];
limit(V,x,Inf)
```

```
ans =
[ exp(a), 0]
```

Input Arguments

`f` — Input
symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable
x (default) | symbolic variable

Independent variable, specified as a symbolic variable. If you do not specify `var`, then `symvar` determines the independent variable.

`a` — Limit point
number | symbolic number | symbolic variable | symbolic expression

Limit point, specified as a number, or a symbolic number, variable, or expression.

Definitions

Bidirectional Limit

$$L = \lim_{x \to a} f(x), x - a \in \mathbb{R} \setminus \{0\}.$$

Left Side Limit

$$L = \lim_{x \to a^-} f(x), x - a < 0.$$

Right Side Limit

$$L = \lim_{x \to a^+} f(x), x - a > 0.$$

See Also

`diff` | `poles` | `taylor`

Introduced before R2006a

linsolve

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.

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.

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: Solution is not unique because the system is rank-deficient.
  In sym.linsolve at 67
X =
              x/a
 -(x - a*y)/(a*b)
                0
R =
2
```

Definitions

Matrix Representation of System of Linear Equations

A system of linear equations

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\cdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m$$

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:

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

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`.

See Also

`cond` | `dsolve` | `equationsToMatrix` | `inv` | `norm` | `odeToVectorField` | `rank` | `solve` | `symvar` | `vpasolve`

Topics

"Solve System of Algebraic Equations" on page 2-153

Introduced in R2012b

log

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

`log10` | `log2`

Introduced before R2006a

log10

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`

Introduced before R2006a

log2

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`

logical

Check validity of equation or inequality

Syntax

`logical(cond)`

Description

`logical(cond)` checks whether the condition `cond` is valid. To test conditions that require assumptions or simplifications, use `isAlways` instead of `logical`.

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 if `3/5` is less than `2/3`:

```
logical(sym(3)/5 < sym(2)/3)
```

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

Note that `logical` typically ignores assumptions on variables.

```
syms x
assume(x == 5)
logical(x == 5)
```

```
ans =
  logical
   0
```

To compare expressions taking into account assumptions on their variables, use `isAlways`:

```
isAlways(x == 5)
```

```
ans =
  logical
   1
```

For further computations, clear the assumption on `x`:

```
syms x clear
```

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 =
  logical
   1
```

Check this inequality. Note that `logical` evaluates the left side of the inequality.

```
logical(sym(11)/4 - sym(1)/2 > 2)
```

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

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 =
  logical
   0
```

`logical` also does not realize that this inequality is invalid:

```
logical(sin(x)/cos(x) ~= tan(x))
```

```
ans =
  logical
   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 =
  logical
     1

isAlways(sin(x)/cos(x) ~= tan(x))

Warning: Unable to prove 'sin(x)/cos(x) ~= tan(x)'.
ans =
  logical
   0
```

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.

See Also

`assume` | `assumeAlso` | `assumptions` | `in` | `isAlways` | `isequal` | `isequaln` | `isfinite` | `isinf` | `isnan` | `sym` | `syms`

Topics

"Use Assumptions on Symbolic Variables" on page 1-28
"Clear Assumptions and Reset the Symbolic Engine" on page 3-67

Introduced in R2012a

logint

Logarithmic integral function

Syntax

```
logint(X)
```

Description

`logint(X)` represents the logarithmic integral function on page 4-1085 (integral logarithm).

Examples

Integral Logarithm for Numeric and Symbolic Arguments

Depending on its arguments, `logint` returns floating-point or exact symbolic results.

Compute integral logarithms for these numbers. Because these numbers are not symbolic objects, `logint` returns floating-point results.

```
A = logint([-1, 0, 1/4, 1/2, 1, 2, 10])
```

```
A =
   0.0737 + 3.4227i   0.0000 + 0.0000i  -0.1187 + 0.0000i  -0.3787 + 0.0000i...
     -Inf + 0.0000i   1.0452 + 0.0000i   6.1656 + 0.0000i
```

Compute integral logarithms for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `logint` returns unresolved symbolic calls.

```
symA = logint(sym([-1, 0, 1/4, 1/2, 1, 2, 10]))
```

```
symA =
[ logint(-1), 0, logint(1/4), logint(1/2), -Inf, logint(2), logint(10)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

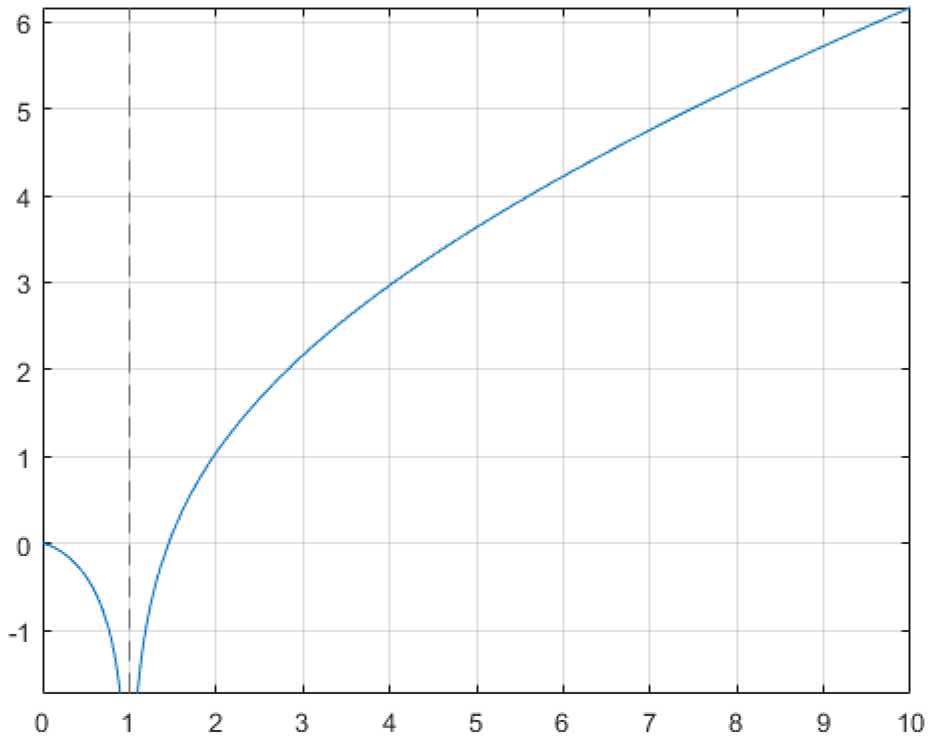
```
vpa(symA)
```

```
ans =
[ 0.07366791204642548599010096523015...
 + 3.4227333787773627895923750617977i,...
0,...
-0.11866205644712310530509570647204,...
-0.37867104306108797672720718463656,...
-Inf,...
1.0451637801174927848445888891946,...
6.1655995047872979375229817526695]
```

Plot Integral Logarithm

Plot the integral logarithm function on the interval from 0 to 10. Before R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(logint(x), [0, 10])
grid on
```



Handle Expressions Containing Integral Logarithm

Many functions, such as `diff` and `limit`, can handle expressions containing `logint`.

Find the first and second derivatives of the integral logarithm:

```
syms x
diff(logint(x), x)
diff(logint(x), x, x)
```

```
ans =
1/log(x)
```

```
ans =
-1/(x*log(x)^2)
```

Find the right and left limits of this expression involving `logint`:

```
limit(exp(1/x)/logint(x + 1), x, 0, 'right')

ans =
Inf

limit(exp(1/x)/logint(x + 1), x, 0, 'left')

ans =
0
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Logarithmic Integral Function

The logarithmic integral function, also called the integral logarithm, is defined as follows:

$$\operatorname{logint}(x) = \operatorname{Li}(x) = \int_0^x \frac{1}{\ln(t)}\,dt$$

Tips

- `logint(sym(0))` returns `1`.

- `logint(sym(1))` returns `-Inf`.
- `logint(z) = ei(log(z))` for all complex `z`.

References

[1] Gautschi, W., and W. F. Cahill. "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

`coshint` | `cosint` | `ei` | `expint` | `int` | `log` | `sinhint` | `sinint` | `ssinint`

Introduced in R2014a

logm

Matrix logarithm

Syntax

```
R = logm(A)
```

Description

`R = logm(A)` computes the matrix logarithm of the square matrix `A`.

Examples

Matrix Logarithm

Compute the matrix logarithm for the 2-by-2 matrix.

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

```
ans =
[ log(x), log(x)/(2*x) - log(-x)/(2*x)]
[      0,                      log(-x)]
```

Input Arguments

`A` — Input matrix
square matrix

Input matrix, specified as a square symbolic matrix.

Output Arguments

`R` — Resulting matrix

symbolic matrix

Resulting function, returned as a symbolic matrix.

See Also

`eig` | `expm` | `funm` | `jordan` | `sqrtm`

Introduced in R2014b

lt

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`.

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
```

Tips

- Calling < or `lt` for non-symbolic `A` and `B` invokes the MATLAB `lt` function. This function returns a logical array with elements set to logical `1 (true)` where `A` is less than `B`; otherwise, it returns logical `0 (false)`.
- 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`.

See Also

`eq` | `ge` | `gt` | `isAlways` | `le` | `ne`

Topics

"Set Assumptions" on page 1-28

Introduced in R2012a

lu

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`. The syntax `lu(A,'matrix')` is identical.

`[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`. The syntax `lu(A,'matrix')` is identical.

`[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.

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.

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 =
  3×3 logical array
     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 =
  3×3 logical array
     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 =
  4×2 logical array
     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 =
  4×2 logical array
     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]
```

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:

$$P_{ij} = \delta_{p_i,j} = \begin{cases} 1 \text{ if } j = p_i \\ 0 \text{ if } j \neq p_i \end{cases}$$

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.

See Also

`chol` | `eig` | `isAlways` | `lu` | `qr` | `svd` | `vpa`

Introduced in R2013a

massMatrixForm

Extract mass matrix and right side of semilinear system of differential algebraic equations

Syntax

```
[M,F] = massMatrixForm(eqs,vars)
```

Description

`[M,F] = massMatrixForm(eqs,vars)` returns the mass matrix `M` and the right side of equations `F` of a semilinear system of first-order differential algebraic equations (DAEs). Algebraic equations in `eqs` that do not contain any derivatives of the variables in `vars` correspond to empty rows of the mass matrix `M`.

The mass matrix `M` and the right side of equations `F` refer to this form.

$$M\left(t,x(t)\right)\dot{x}(t) = F\left(t,x(t)\right).$$

Examples

Convert DAE System to Mass Matrix Form

Convert a semilinear system of differential algebraic equations to mass matrix form.

Create the following system of differential algebraic equations. Here, the functions `x1(t)` and `x2(t)` represent state variables of the system. The system also contains symbolic parameters `r` and `m`, and the function `f(t, x1, x2)`. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x1(t) x2(t) f(t, x1, x2) r m;
eqs = [m*x2(t)*diff(x1(t), t) + m*t*diff(x2(t), t) == f(t,x1(t),x2(t)),...
```

```
       x1(t)^2 + x2(t)^2 == r^2];
vars = [x1(t) x2(t)];
```

Find the mass matrix form of this system.

```
[M,F] = massMatrixForm(eqs, vars)
```

```
M =
[ m*x2(t), m*t]
[       0,   0]

F =
      f(t, x1(t), x2(t))
 r^2 - x2(t)^2 - x1(t)^2
```

Solve this system using the numerical solver `ode15s`. Before you use `ode15s`, assign the following values to symbolic parameters of the system: `m = 100`, `r = 1`, `f(t, x1, x2) = t + x1*x2`. Also, replace the state variables `x1(t)`, `x2(t)` by variables `Y1`, `Y2` acceptable by `matlabFunction`.

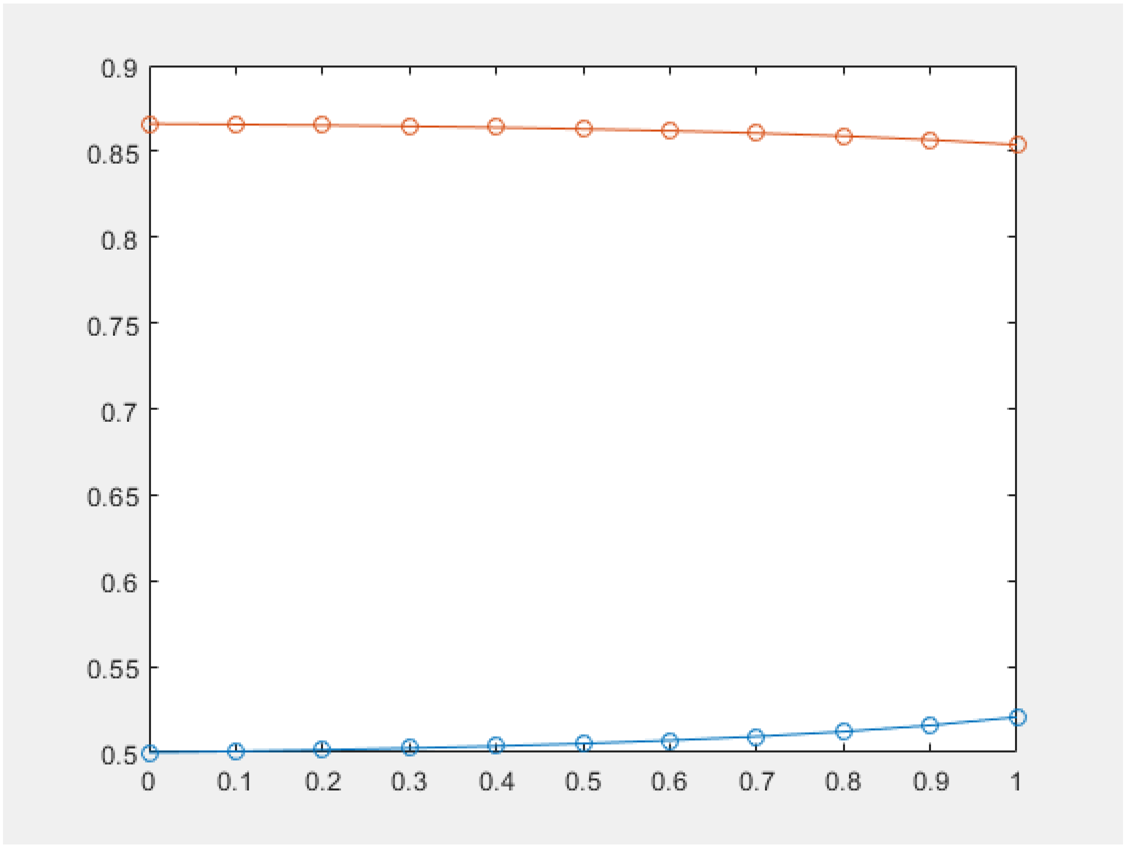
```
syms Y1 Y2;
M = subs(M, [vars,m,r,f], [Y1,Y2,100,1,@(t,x1,x2) t+x1*x2]);
F = subs(F, [vars,m,r,f], [Y1,Y2,100,1,@(t,x1,x2) t+x1*x2]);
```

Create the following function handles `MM` and `FF`. You can use these function handles as input arguments for `odeset` and `ode15s`. These functions require state variables to be specified as column vectors.

```
MM = matlabFunction(M,'vars',{t,[Y1;Y2]});
FF = matlabFunction(F,'vars',{t,[Y1;Y2]});
```

Solve the system using `ode15s`.

```
opt = odeset('Mass', MM, 'InitialSlope', [0.005;0]);
ode15s(FF, [0,1], [0.5; 0.5*sqrt(3)], opt)
```



Input Arguments

`eqs` — System of semilinear first-order DAEs

vector of symbolic equations | vector of symbolic expressions

System of semilinear first-order DAEs, specified as a vector of symbolic equations or expressions.

`vars` — State variables

vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

Output Arguments

`M` — Mass matrix

symbolic matrix

Mass matrix of the system, returned as a symbolic matrix. The number of rows is the number of equations in `eqs`, and the number of columns is the number of variables in `vars`.

`F` — Right sides of equations

symbolic column vector of symbolic expressions

Right sides of equations, returned as a column vector of symbolic expressions. The number of elements in this vector is equal to the number of equations in `eqs`.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `matlabFunction` | `ode15s` | `odeFunction` | `odeset` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

"Solve DAEs Using Mass Matrix Solvers" on page 2-213

Introduced in R2014b

matlabFunction

Convert symbolic expression to function handle or file

Syntax

```
g = matlabFunction(f)
g = matlabFunction(f1,...,fN)
g = matlabFunction(___,Name,Value)
```

Description

`g = matlabFunction(f)` converts the symbolic expression or function `f` to a MATLAB function with handle `g`. The function can be used without Symbolic Math Toolbox.

`g = matlabFunction(f1,...,fN)` converts `f1,...,fN` to a MATLAB function with `N` outputs. The function handle is `g`. Each element of `f1,...,fN` can be a symbolic expression, function, or a vector of symbolic expressions or functions.

`g = matlabFunction(___,Name,Value)` converts symbolic expressions, functions, or vectors of symbolic expressions or functions to a MATLAB function using additional options specified by one or more `Name,Value` pair arguments. You can specify `Name,Value` after the input arguments used in the previous syntaxes.

Examples

Convert Symbolic Expression to Anonymous Function

Convert the symbolic expression `r` to a MATLAB function with the handle `ht`. The function can be used without Symbolic Math Toolbox.

```
syms x y
r = sqrt(x^2 + y^2);
ht = matlabFunction(r)
ht =
  function_handle with value:
    @(x,y)sqrt(x.^2+y.^2)
```

Convert multiple symbolic expressions using comma-separated input.

```
ht = matlabFunction(r, r^2)
```

```
ht =
  function_handle with value:
    @(x,y)deal(sqrt(x.^2+y.^2),x.^2+y.^2)
```

Convert Symbolic Function to Anonymous Function

Create a symbolic function and convert it to a MATLAB function with the handle `ht`.

```
syms x y
f(x,y) = x^3 + y^3;
ht = matlabFunction(f)
```

```
ht =
  function_handle with value:
    @(x,y)x.^3+y.^3
```

Write MATLAB Function to File with Comments

Write the generated MATLAB function to a file by specifying the `File` option. Existing files are overwritten. When writing to a file, `matlabFunction` optimizes the code using intermediate variables named `t0`, `t1`, ... Include comments in the file by using the `Comments` option.

Write the MATLAB function generated from `f` to the file `myfile`.

```
syms x
f = x^2 + log(x^2);
matlabFunction(f,'File','myfile');
```

```
function f = myfile(x)
%MYFILE
%    F = MYFILE(X)
%    This function was generated by the Symbolic Math Toolbox version 7.3.
%    01-Jan-2017 00:00:00

t2 = x.^2;
f = t2+log(t2);
```

Include the comment `Version: 1.1` in the file.

```
matlabFunction(f,'File','myfile','Comments','Version: 1.1')
```

```
function f = myfile(x)
...
%Version: 1.1
t2 = x.^2;
...
```

Disable Code Optimization

When you convert a symbolic expression to a MATLAB function and write the resulting function to a file, `matlabFunction` optimizes the code by default. This approach can help simplify and speed up further computations that use the file. However, generating the optimized code from some symbolic expressions and functions can be very time consuming. Use `Optimize` to disable code optimization.

Create a symbolic expression.

```
syms x
r = x^2*(x^2 + 1);
```

Convert `r` to a MATLAB function and write the function to the file `myfile`. By default, `matlabFunction` creates a file containing the optimized code.

```
f =  matlabFunction(r,'File','myfile');
```

```
function r = myfile(x)
%MYFILE
%    R = MYFILE(X)
t2 = x.^2;
r = t2.*(t2+1.0);
```

Disable the code optimization by setting the value of `Optimize` to `false`.

```
f =  matlabFunction(r,'File','myfile','Optimize',false);
```

```
function r = myfile(x)
%MYFILE
%    R = MYFILE(X)
r = x.^2.*(x.^2+1.0);
```

Generate Sparse Matrices

When you convert a symbolic matrix to a MATLAB function, `matlabFunction` represents it by a dense matrix by default. If most of the elements of the input symbolic matrix are zeros, the more efficient approach is to represent it by a sparse matrix.

Create a 3-by-3 symbolic diagonal matrix:

```
syms x
A = diag(x*ones(1,3))
```

```
A =
[ x, 0, 0]
[ 0, x, 0]
[ 0, 0, x]
```

Convert `A` to a MATLAB function representing a numeric matrix, and write the result to the file `myfile1`. By default, the generated MATLAB function creates the dense numeric matrix specifying each element of the matrix, including all zero elements.

```
f1 = matlabFunction(A,'File','myfile1');
```

```
function A = myfile1(x)
%MYFILE1
%    A = MYFILE1(X)
A = reshape([x,0.0,0.0,0.0,x,0.0,0.0,0.0,x],[3,3]);
```

Convert `A` to a MATLAB function setting `Sparse` to `true`. Now, the generated MATLAB function creates the sparse numeric matrix specifying only nonzero elements and assuming that all other elements are zeros.

```
f2 = matlabFunction(A,'File','myfile2','Sparse',true);
```

```
function A = myfile2(x)
%MYFILE2
```

```
%    A = MYFILE2(X)
A = sparse([1,2,3],[1,2,3],[x,x,x],3,3);
```

Specify Input Arguments for Generated Function

When converting an expression to a MATLAB function, you can specify the order of the input arguments of the resulting function. You also can specify that some input arguments are vectors instead of single variables.

Create a symbolic expression.

```
syms x y z
r = x + y/2 + z/3;
```

Convert `r` to a MATLAB function and write this function to the file `myfile`. By default, `matlabFunction` uses alphabetical order of input arguments when converting symbolic expressions.

```
matlabFunction(r,'File','myfile');
```

```
function r = myfile(x,y,z)
%MYFILE
%    R = MYFILE(X,Y,Z)
r = x+y.*(1.0./2.0)+z.*(1.0./3.0);
```

Use the `Vars` argument to specify the order of input arguments for the generated MATLAB function.

```
matlabFunction(r,'File','myfile','Vars',[y z x]);
```

```
function r = myfile(y,z,x)
%MYFILE
%    R = MYFILE(Y,Z,X)
r = x+y.*(1.0./2.0)+z.*(1.0./3.0);
```

Now, convert an expression `r` to a MATLAB function whose second input argument is a vector.

```
syms x y z t
r = (x + y/2 + z/3)*exp(-t);
matlabFunction(r,'File','myfile','Vars',{t,[x y z]});
```

```
function r = myfile(t,in2)
%MYFILE
```

```
%    R = MYFILE(T,IN2)
x = in2(:,1);
y = in2(:,2);
z = in2(:,3);
r = exp(-t).*(x+y.*(1.0./2.0)+z.*(1.0./3.0));
```

Specify Output Variables

When converting a symbolic expression to a MATLAB function, you can specify the names of the output variables. Note that `matlabFunction` without the `File` argument (or with a file path specified by an empty character vector) creates a function handle and ignores the `Outputs` flag.

Create symbolic expressions `r` and `q`.

```
syms x y z
r = x^2 + y^2 + z^2;
q = x^2 - y^2 - z^2;
```

Convert `r` and `q` to a MATLAB function and write the resulting function to a file `myfile`, which returns a vector of two elements, `name1` and `name2`.

```
f = matlabFunction(r,q,'File','myfile',...
                   'Outputs',{'name1','name2'});
```

```
function [name1,name2] = myfile(x,y,z)
%MYFILE
%    [NAME1,NAME2] = MYFILE(X,Y,Z)
t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
name1 = t2+t3+t4;
if nargout > 1
    name2 = t2-t3-t4;
end
```

- “Generate MATLAB Functions from Symbolic Expressions” on page 2-254
- “Create MATLAB Functions from MuPAD Expressions” on page 3-71

Input Arguments

`f` — Symbolic input to be converted to MATLAB function

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Symbolic input to be converted to a MATLAB function, specified as a symbolic expression, function, vector, or matrix. When converting sparse symbolic vectors or matrices, use the name-value pair argument `'Sparse',true`.

`f1,...,fN` — Symbolic input to be converted to MATLAB function with `N` outputs

several symbolic expressions | several symbolic functions | several symbolic vectors | several symbolic matrices

Symbolic input to be converted to MATLAB function with `N` outputs, specified as several symbolic expressions, functions, vectors, or matrices, separated by comma.

`matlabFunction` does not create a separate output argument for each element of a symbolic vector or matrix. For example, `g = matlabFunction([x + 1, y + 1])` creates a MATLAB function with one output argument, while `g = matlabFunction(x + 1, y + 1)` creates a MATLAB function with two output arguments.

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`.

Example: `matlabFunction(f,'File','myfile','Optimize',false)`

`Comments` — Comments to include in file header

character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector.

`File` — Path to file containing generated MATLAB function

character vector

Path to the file containing the generated MATLAB function, specified as a character vector. The generated function accepts arguments of type `double`, and can be used without Symbolic Math Toolbox. If `File` is empty, `matlabFunction` generates an anonymous function. If `File` does not end in `.m`, the function appends `.m`.

When writing to a file, `matlabFunction` optimizes the code using intermediate variables named `t0`, `t1`, ... To disable code optimization, use the `Optimize` argument.

See “Write MATLAB Function to File with Comments” on page 4-1106.

`Optimize` — Flag preventing optimization of code written to function file

`true` (default) | `false`

Flag preventing optimization of code written to a function file, specified as `false` or `true`.

When writing to a file, `ccode` optimizes the code using intermediate variables named `t0`, `t1`, ...

`matlabFunction` without the `File` argument (or with a file path specified by an empty character vector) creates a function handle. In this case, the code is not optimized. If you try to enforce code optimization by setting `Optimize` to `true`, then `matlabFunction` throws an error.

See “Disable Code Optimization” on page 4-1107.

`Sparse` — Flag that switches between sparse and dense matrix generation

`false` (default) | `true`

Flag that switches between sparse and dense matrix generation, specified as `true` or `false`. When you specify `'Sparse',true`, the generated MATLAB function represents symbolic matrices by sparse numeric matrices. Use `'Sparse',true` when you convert symbolic matrices containing many zero elements. Often, operations on sparse matrices are more efficient than the same operations on dense matrices.

See “Generate Sparse Matrices” on page 4-1108.

`Vars` — Order of input variables or vectors in generated MATLAB function

character vector | vector of symbolic variables | one-dimensional cell array of character vectors | one-dimensional cell array of symbolic variables | one-dimensional cell array of vectors of symbolic variables

Order of input variables or vectors in a generated MATLAB function, specified as a character vector, a vector of symbolic variables, or a one-dimensional cell array of character vectors, symbolic variables, or vectors of symbolic variables.

The number of specified input variables must equal or exceed the number of free variables in `f`. Do not use the same names for the input variables specified by `Vars` and the output variables specified by `Outputs`.

By default, when you convert symbolic expressions, the order is alphabetical. When you convert symbolic functions, their input arguments appear in front of other variables, and all other variables are sorted alphabetically.

See “Specify Output Variables” on page 4-1110.

`Outputs` — Names of output variables
one-dimensional cell array of character vectors

Names of output variables, specified as a one-dimensional cell array of character vectors.

If you do not specify the output variable names, then they coincide with the names you use when 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`.

Do not use the same names for the input variables specified by `Vars` and the output variables specified by `Outputs`.

`matlabFunction` without the `File` argument (or with a file path specified by an empty character vector) creates a function handle. In this case, `matlabFunction` ignores the `Outputs` flag.

See “Specify Output Variables” on page 4-1110.

Output Arguments

`g` — Function handle that can serve as input argument to numerical functions
MATLAB function handle

Function handle that can serve as an input argument to numerical functions, returned as a MATLAB function handle.

Tips

- 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.
- To convert a MuPAD expression or function to a MATLAB symbolic expression, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine,'MuPAD_Function',x1,...,xn)`. Then you can convert the resulting symbolic expression to a MATLAB function.

  `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 “Differences Between MATLAB and MuPAD Syntax” on page 3-49 list, always check the conversion results. To verify the results, execute the resulting function.
- If the `File` option is empty, then an anonymous function is returned.

See Also

`ccode` | `daeFunction` | `fortran` | `matlabFunctionBlock` | `odeFunction` | `rehash` | `simscapeEquation` | `subs` | `sym2poly`

Topics

“Generate MATLAB Functions from Symbolic Expressions” on page 2-254
“Create MATLAB Functions from MuPAD Expressions” on page 3-71

Introduced in R2008b

matlabFunctionBlock

Convert symbolic expression to MATLAB function block

Syntax

```
matlabFunctionBlock(block,f)
matlabFunctionBlock(block,f1,...,fN)
matlabFunctionBlock(___,Name,Value)
```

Description

`matlabFunctionBlock(block,f)` converts `f` to a MATLAB function block that you can use in Simulink models. Here, `f` can be a symbolic expression, function, or a vector of symbolic expressions or functions.

`block` specifies the name of the block that you create or modify.

`matlabFunctionBlock(block,f1,...,fN)` converts symbolic expressions or functions `f1,...,fN` to a MATLAB function block with `N` outputs. Each element of `f1,...,fN` can be a symbolic expression, function, or a vector of symbolic expressions or functions.

`matlabFunctionBlock(___,Name,Value)` converts a symbolic expression, function, or a vector of symbolic expressions or functions to a MATLAB function block using additional options specified by one or more `Name,Value` pair arguments. You can specify `Name,Value` after the input arguments used in the previous syntaxes.

Examples

Convert Symbolic Expression to MATLAB Function Block

Create a new model and convert a symbolic expression to a MATLAB function block. Include comments in the block by specifying the `Comments` option.

Create a new model and open it.

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

Create a symbolic expression.

```
syms x y z
f = x^2 + y^2 + z^2;
```

Use `matlabFunctionBlock` to create the block `my_block` containing the symbolic expression. `matlabFunctionBlock` overwrites existing blocks. Double-click the generated block to open and edit the function defining the block.

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

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

%    This function was generated by the Symbolic Math Toolbox version 7.3.
%    01-Jan-2017 00:00:00

f = x.^2+y.^2+z.^2;
```

Include the comment `Version 1.1` in the block.

```
matlabFunctionBlock('my_system/my_block',f,'Comments','Version: 1.1')
```

```
function f = my_block(x,y,z)
...
%Version: 1.1
f = x.^2+y.^2+z.^2;
```

Save and close `my_system`.

```
save_system('my_system')
close_system('my_system')
```

Convert Symbolic Function MATLAB Function Block

Create a new model and convert a symbolic function to a MATLAB function block.

Create a new empty model and open it.

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

Create a symbolic function.

```
syms x y z
f(x, y, z) = x^2 + y^2 + z^2;
```

Convert `f` to a MATLAB function block. Double-click the block to see the function.

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

```
function f = my_block(x,y,z)
%#codegen
f = x.^2+y.^2+z.^2;
```

Create Blocks with Multiple Outputs

Convert several symbolic expressions to a MATLAB function block with multiple output ports.

Create a new empty model and open it.

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

Create three symbolic expressions.

```
syms x y z
f = x^2;
g = y^2;
h = z^2;
```

Convert them to a MATLAB function block. `matlabFunctionBlock` creates a block with three output ports. Double-click the block to see the function.

```
matlabFunctionBlock('my_system/my_block',f,g,h)
```

```
function [f,g,h] = my_block(x,y,z)
%#codegen
f = x.^2;
if nargout > 1
    g = y.^2;
```

```
end
if nargout > 2
    h = z.^2;
end
```

Specify Function Name for Generated Function

Specifying the name of the function defining the generated MATLAB function block.

Create a new empty model and open it.

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

Create a symbolic expression.

```
syms x y z
f = x^2 + y^2 + z^2;
```

Generate a block and set the function name to `my_function`. Double-click the block to see the function.

```
matlabFunctionBlock('my_system/my_block',f,...
                    'FunctionName', 'my_function')
```

```
function f = my_function(x,y,z)
%#codegen
f = x.^2+y.^2+z.^2;
```

Disable Code Optimization

When you convert a symbolic expression to a MATLAB function block, `matlabFunctionBlock` optimizes the code by default. This approach can help simplify and speed up further computations that use the file. Nevertheless, generating the optimized code from some symbolic expressions and functions can be very time-consuming. Use `Optimize` to disable code optimization.

Create a new empty model and open it.

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

Create a symbolic expression.

```
syms x
r = x^2*(x^2 + 1);
```

Use `matlabFunctionBlock` to create the block `my_block` containing the symbolic expression. Double-click the block to see the function defining the block. By default, `matlabFunctionBlock` creates a file containing the optimized code.

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

```
function r = my_block(x)
%#codegen
t2 = x.^2;
r = t2.*(t2+1.0);
```

Disable the code optimization by setting the value of `Optimize` to `false`.

```
matlabFunctionBlock('my_system/my_block',r,...
                    'Optimize',false)
```

```
function r = my_block(x)
%#codegen
r = x.^2.*(x.^2+1.0);
```

Specify Input Ports for Generated Block

Specify the order of the input variables that form the input ports in a generated block.

Create a new empty model and open it.

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

Create a symbolic expression.

```
syms x y z
f = x^2 + y^2 + z^2;
```

Convert the expression to a MATLAB function block. By default, `matlabFunctionBlock` uses alphabetical order of input arguments when converting symbolic expressions.

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

```
function f = my_block(x,y,z)
%#codegen
f = x.^2+y.^2+z.^2;
```

Use the `Vars` argument to specify the order of the input ports.

```
matlabFunctionBlock('my_system/my_block',f,...
                    'Vars', [y z x])
```

```
function f = my_block(y,z,x)
%#codegen
f = x.^2+y.^2+z.^2;
```

Specify Output Ports

When generating a block, rename the output variables and the corresponding ports.

Create a new empty model and open it.

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

Create a symbolic expression.

```
syms x y z
f = x^2 + y^2 + z^2;
```

Convert the expression to a MATLAB function block and specify the names of the output variables and ports. Double-click the block to see the function defining the block.

```
matlabFunctionBlock('my_system/my_block',f,f + 1,f + 2,...
                    'Outputs', {'name1','name2','name3'})
```

```
function [name1,name2,name3] = my_block(x,y,z)
%#codegen
t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
name1 = t2+t3+t4;
if nargout > 1
    name2 = t2+t3+t4+1.0;
```

```
end
if nargout > 2
    name3 = t2+t3+t4+2.0;
end
```

Specify Function Name, Input and Output Ports

Call `matlabFunctionBlock` using several name-value pair arguments simultaneously.

Create a new empty model and open it.

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

Create a symbolic expression.

```
syms x y z
f = x^2 + y^2 + z^2;
```

Call `matlabFunctionBlock` using the name-value pair arguments to specify the function name, the order of the input ports, and the names of the output ports. Double-click the block to see the function defining the block.

```
matlabFunctionBlock('my_system/my_block',f,f + 1,f + 2,...
                    'FunctionName', 'my_function','Vars',[y z x],...
                    'Outputs',{'name1','name2','name3'})
```

```
function [name1,name2,name3] = my_function(y,z,x)
%#codegen
t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
name1 = t2+t3+t4;
if nargout > 1
    name2 = t2+t3+t4+1.0;
end
if nargout > 2
    name3 = t2+t3+t4+2.0;
end
```

- “Generate MATLAB Function Blocks from Symbolic Expressions” on page 2-259
- “Create MATLAB Function Blocks from MuPAD Expressions” on page 3-75

Input Arguments

`block` — Block to create of modify
character vector

Block to create of modify, specified as a character vector.

`f` — Symbolic input to be converted to MATLAB function block
symbolic expression | symbolic function | symbolic vector | symbolic matrix

Symbolic input to be converted to MATLAB function block, specified as a symbolic expression, function, vector, or matrix

`f1,...,fN` — Symbolic input to be converted to MATLAB function block with `N` outputs
several symbolic expressions | several symbolic functions | several symbolic vectors | several symbolic matrices

Symbolic input to be converted to MATLAB function block with `N` outputs, specified as several symbolic expressions, functions, vectors, or matrices, separated by comma.

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`.

Example: `matlabFunctionBlock('my_system/my_block',f,'FunctionName','myfun')`

`Comments` — Comments to include in file header
character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector.

`FunctionName` — Name of function
coincides with the input argument `block` (default) | character vector

Name of the function, specified as a character vector. By default, `matlabFunction(block,…)` uses `block` as the function name.

See “Specify Function Name for Generated Function” on page 4-1118.

`Optimize` — Flag preventing code optimization

`true` (default) | `false`

Flag preventing code optimization, specified as `false` or `true`.

When writing to a file, `matlabFunctionBlock` optimizes the code using intermediate variables named `t0`, `t1`, ...

See “Disable Code Optimization” on page 4-1118.

`Vars` — Order of input variables and corresponding input ports of generated block

character vector | one-dimensional cell array of character vectors | one-dimensional cell array of symbolic variables | one-dimensional cell array of vectors of symbolic variables | vector of symbolic variables

Order of input variables and corresponding input ports of generated block, specified as a character vector, a vector of symbolic variables, or a one-dimensional cell array of character vectors, symbolic variables, or vectors of symbolic variables.

The number of specified input ports must equal or exceed the number of free variables in `f`. Do not use the same names for the input ports specified by `Vars` and the output ports specified by `Outputs`.

By default, when you convert symbolic expressions, the order is alphabetical. When you convert symbolic functions, their input arguments appear in front of other variables, and all other variables are sorted alphabetically.

See “Specify Input Ports for Generated Block” on page 4-1119.

`Outputs` — Names of output ports

`out` followed by output port numbers (default) | one-dimensional cell array of character vectors

Names of output ports, specified as a one-dimensional cell array of character vectors. If you do not specify the output port names, `matlabFunctionBlock` uses names that consist of the word `out` followed by output port numbers, for example, `out3`.

Do not use the same names for the input ports specified by `Vars` and the output ports specified by `Outputs`. See “Specify Output Ports” on page 4-1120.

Tips

- To convert a MuPAD expression or function to a MATLAB symbolic expression, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine,'MuPAD_Function',x1,...,xn)`. Then you can convert the resulting symbolic expression to a MATLAB function block. `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 on page 3-49 list, always check the conversion results. To verify the results, you can run the simulation containing the resulting block.

See Also

`ccode` | `fortran` | `matlabFunction` | `simscapeEquation` | `subs` | `sym2poly`

Topics

"Generate MATLAB Function Blocks from Symbolic Expressions" on page 2-259
"Create MATLAB Function Blocks from MuPAD Expressions" on page 3-75

Introduced in R2009a

max

Largest elements

Syntax

```
C = max(A)
C = max(A,[],dim)
[C,I] = max(___)

C = max(A,B)
```

Description

`C = max(A)` returns the largest element of `A` if `A` is a vector. If `A` is a matrix, this syntax treats the columns of `A` as vectors, returning a row vector containing the largest element from each column.

`C = max(A,[],dim)` returns the largest elements of matrix `A` along the dimension `dim`. Thus, `max(A,[],1)` returns a row vector containing the largest elements of each column of `A`, and `max(A,[],2)` returns a column vector containing the largest elements of each row of `A`.

Here, the required argument `[]` serves as a divider. If you omit it, `max(A,dim)` compares elements of `A` with the value `dim`.

`[C,I] = max(___)` finds the indices of the largest elements, and returns them in output vector `I`. If there are several identical largest values, this syntax returns the index of the first largest element that it finds.

`C = max(A,B)` compares each element of `A` with the corresponding element of `B` and returns `C` containing the largest elements of each pair.

Examples

Maximum of Vector of Numbers

Find the largest of these numbers. Because these numbers are not symbolic objects, you get a floating-point result.

```
max([-pi, pi/2, 1, 1/3])
```

```
ans =
    1.5708
```

Find the largest of the same numbers converted to symbolic objects.

```
max(sym([-pi, pi/2, 1, 1/3]))
```

```
ans =
pi/2
```

Maximum of Each Column in Symbolic Matrix

Create matrix `A` containing symbolic numbers, and call `max` for this matrix. By default, `max` returns the row vector containing the largest elements of each column.

```
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
max(A)
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

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

Maximum of Each Row in Symbolic Matrix

Create matrix `A` containing symbolic numbers, and find the largest elements of each row of the matrix. In this case, `max` returns the result as a column vector.

```
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
max(A,[],2)
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

ans =
 2
 5
 3
```

Indices of Largest Elements

Create matrix A. Find the largest element in each column and its index.

```
A = 1./sym(magic(3))
[Cc,Ic] = max(A)
```

```
A =
[ 1/8,   1, 1/6]
[ 1/3, 1/5, 1/7]
[ 1/4, 1/9, 1/2]

Cc =
[ 1/3, 1, 1/2]

Ic =
     2     1     3
```

Now, find the largest element in each row and its index.

```
[Cr,Ir] = max(A,[],2)
```

```
Cr =
   1
 1/3
 1/2

Ir =
     2
     1
     3
```

If `dim` exceeds the number of dimensions of `A`, then the syntax `[C,I] = max(A,[],dim)` returns `C = A` and `I = ones(size(A))`.

```
[C,I] = max(A,[],3)

C =
[ 1/8,   1, 1/6]
[ 1/3, 1/5, 1/7]
[ 1/4, 1/9, 1/2]

I =
     1     1     1
     1     1     1
     1     1     1
```

Largest Elements of Two Symbolic Matrices

Create matrices `A` and `B` containing symbolic numbers. Use `max` to compare each element of `A` with the corresponding element of `B`, and return the matrix containing the largest elements of each pair.

```
A = sym(pascal(3))
B = toeplitz(sym([pi/3 pi/2 pi]))
maxAB = max(A,B)

A =
[ 1, 1, 1]
[ 1, 2, 3]
[ 1, 3, 6]

B =
[ pi/3, pi/2,   pi]
[ pi/2, pi/3, pi/2]
[   pi, pi/2, pi/3]

maxAB =
[ pi/3, pi/2, pi]
[ pi/2,    2,  3]
[   pi,    3,  6]
```

Maximum of Complex Numbers

When finding the maximum of these complex numbers, `max` chooses the number with the largest complex modulus.

```
modulus = abs([-1 - i, 1 + 1/2*i])
maximum = max(sym([1 - i, 1/2 + i]))

modulus =
    1.4142    1.1180

maximum =
1 - 1i
```

If the numbers have the same complex modulus, `min` chooses the number with the largest phase angle.

```
modulus = abs([1 - 1/2*i, 1 + 1/2*i])
phaseAngle = angle([1 - 1/2*i, 1 + 1/2*i])
maximum = max(sym([1 - 1/2*i, 1/2 + i]))

modulus =
    1.1180    1.1180

phaseAngle =
   -0.4636    0.4636

maximum =
1/2 + 1i
```

Input Arguments

`A` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, vector, or matrix. All elements of `A` must be convertible to floating-point numbers. If `A` is a scalar, then `max(A)` returns `A`. `A` cannot be a multidimensional array.

`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. The default value is `1`. If `dim` exceeds the number of dimensions of `A`, then `max(A,[],dim)` returns `A`, and `[C,I] = max(A,[],dim)` returns `C = A` and `I = ones(size(A))`.

`B` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, vector, or matrix. All elements of `B` must be convertible to floating-point numbers. If `A` and `B` are scalars, then `max(A,B)` returns the largest of `A` and `B`.

If one argument is a vector or matrix, the other argument must either be a scalar or have the same dimensions as the first one. If one argument is a scalar and the other argument is a vector or matrix, then `max` expands the scalar into a vector or a matrix of the same length with all elements equal to that scalar.

`B` cannot be a multidimensional array.

Output Arguments

`C` — Largest elements

symbolic number | symbolic vector

Largest elements, returned as a symbolic number or vector of symbolic numbers.

`I` — Indices of largest elements

symbolic number | symbolic vector | symbolic matrix

Indices of largest elements, returned as a symbolic number or vector of symbolic numbers. `[C,I] = max(A,[],dim)` also returns matrix `I = ones(size(A))` if the value `dim` exceeds the number of dimensions of `A`.

Tips

- Calling `max` for numbers (or vectors or matrices of numbers) that are not symbolic objects invokes the MATLAB `max` function.
- For complex input `A`, `max` returns the complex number with the largest complex modulus (magnitude), computed with `max(abs(A))`. If complex numbers have the same modulus, `max` chooses the number with the largest phase angle, `max(angle(A))`.
- `max` ignores `NaN`s.

See Also

`abs` | `angle` | `max` | `min` | `sort`

Introduced in R2014a

meijerG

Meijer G-function

Syntax

```
meijerG(a,b,c,d,z)
```

Description

`meijerG(a,b,c,d,z)` returns the “Meijer G-Function” on page 4-1137. `meijerG` is element-wise in `z`. The input parameters `a`, `b`, `c`, and `d` are vectors that can be empty, as in `meijerG([], [], 3.2, [], 1)`.

Examples

Calculate Meijer G-Function for Numeric Inputs

```
syms x
meijerG(3, [], [], 2, 5)

ans =
    25
```

Call `meijerG` when `z` is an array. `meijerG` acts element-wise.

```
a = 2;
z = [1 2 3];
meijerG(a, [], [], [], z)

ans =
    0.3679    1.2131    2.1496
```

Calculate Meijer G-Function for Symbolic Numbers

Convert numeric input to symbolic form using `sym`, and find the Meijer G-function. For certain symbolic inputs, `meijerG` returns exact symbolic output using other functions.

```
meijerG(sym(2), [], [], [], sym(3))
```

```
ans =
3*exp(-1/3)
```

```
meijerG(sym(2/5), [], sym(1/2), [], sym(3))
```

```
ans =
(2^(4/5)*3^(1/2)*gamma(1/10))/80
```

Find Meijer G-Function for Symbolic Variables or Expressions

For symbolic variables or expressions, `meijerG` returns an output in terms of simple or special functions.

```
syms a b c d z
f = meijerG(a,b,c,d,z)
```

```
f =
(gamma(c - a + 1)*(1/z)^(1 - a)*hypergeom([c - a + 1, d - a + 1],...
 b - a + 1, 1/z))/(gamma(b - a + 1)*gamma(a - d))
```

Substitute values for the variables by using `subs`, and convert values to double by using `double`.

```
fVal = subs(f, [a b c d z], [1.2 3 5 7 9])
```

```
fVal =
(266*9^(1/5)*hypergeom([24/5, 34/5], 14/5, 1/9))/(25*gamma(-29/5))
```

```
double(fVal)
```

```
ans =
   5.7586e+03
```

Calculate `fVal` to higher precision using `vpa`.

```
vpa(fVal)
```

```
ans =
5758.5946416377834597597497022199
```

Differentiate Meijer G-Function

Differentiate the Meijer G-function by using `diff`.

```
syms a b c d z
mG = meijerG(a, b, c, d, z);
diffmG = diff(mG)
```

```
diffmG =
(gamma(c - a + 1)*(a - 1)*hypergeom([c - a + 1, d - a + 1],...
 b - a + 1, 1/z))/(z^2*gamma(b - a + 1)*gamma(a - d)*(1/z)^a)...
 - (gamma(c - a + 1)*(1/z)^(1 - a)*(c - a + 1)*(d - a + 1)...
*hypergeom([c - a + 2, d - a + 2], b - a + 2, 1/z))...
/(z^2*gamma(b - a + 1)*gamma(a - d)*(b - a + 1))
```

Relations Between Meijer G-Function and Other Functions

Show relations between `meijerG` and simpler functions for given parameter values.

Show that when `a`, `b`, and `d` are empty, and `c = 0`, then `meijerG` reduces to `exp(-z)`.

```
syms z
meijerG([], [], 0, [], z)
```

```
ans =
exp(-z)
```

Show that when `a`, `b`, and `d` are empty, and `c = [1/2 -1/2]`, then `meijerG` reduces to $2K_v(1,2z^{1/2})$.

```
meijerG([], [], [1/2 -1/2], [], z)
```

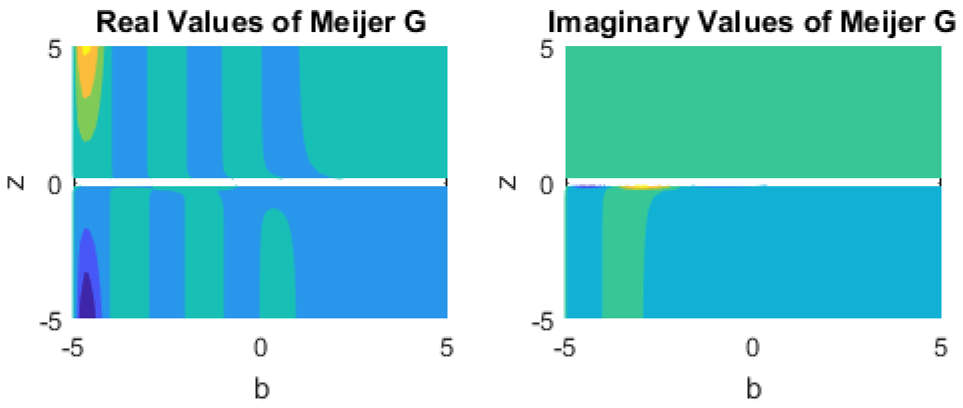
```
ans =
2*besselk(1, 2*z^(1/2))
```

Plot Meijer G-Function

Plot the real and imaginary values of the Meijer G-function for values of `b` and `z`, where `a = [-2 2]` and `c` and `d` are empty. Fill the contours by setting `Fill` to `on`.

```
syms b z
f = meijerG([-2 2], b, [], [], z);
```

```
subplot(2,2,1)
fcontour(real(f),'Fill','on')
title('Real Values of Meijer G')
xlabel('b')
ylabel('z')

subplot(2,2,2)
fcontour(imag(f),'Fill','on')
title('Imag. Values of Meijer G')
xlabel('b')
ylabel('z')
```



Input Arguments

a — Input

number | vector | symbolic number | symbolic variable | symbolic vector | symbolic function | symbolic expression

Input, specified as a number or vector, or a symbolic number, variable, vector, function, or expression.

b — Input

number | vector | symbolic number | symbolic variable | symbolic vector | symbolic function | symbolic expression

Input, specified as a number or vector, or a symbolic number, variable, vector, function, or expression.

`c` — Input

number | vector | symbolic number | symbolic variable | symbolic vector | symbolic function | symbolic expression

Input, specified as a number or vector, or a symbolic number, variable, vector, function, or expression.

`d` — Input

number | vector | symbolic number | symbolic variable | symbolic vector | symbolic function | symbolic expression

Input, specified as a number or vector, or a symbolic number, variable, vector, function, or expression.

`z` — Input

number | vector | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number or vector, or a symbolic number, variable, vector, function, or expression.

Definitions

Meijer G-Function

The Meijer G-function meijerG([a_1,...,a_n],[a_{n+1},...,a_p],[b_1,...,b_m],[b_{m+1},...,b_q],z) is a general function that includes other special functions as particular cases, and is defined as

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1,\ldots,a_p \\ b_1,\ldots,b_p \end{matrix}\middle| z\right)=\frac{1}{2\pi \mathrm{i}}\int \frac{\left(\prod_{j=1}^{m}\Gamma\left(b_j-s\right)\right)\left(\prod_{j=1}^{n}\Gamma\left(1-a_j+s\right)\right)}{\left(\prod_{j=m+1}^{q}\Gamma\left(1-b_j+s\right)\right)\left(\prod_{j=n+1}^{p}\Gamma\left(a_j-s\right)\right)}z^s ds.$$

Algorithms

For the Meijer G-function meijerG($[a_1,\dots,a_n]$,$[a_{n+1},\dots,a_p]$,$[b_1,\dots,b_m]$,$[b_{m+1},\dots,b_q]$,z), for $a_i \in (a_1,\dots,a_n)$ and $b_j \in (b_1,\dots,b_m)$, no difference $a_i - b_j$ should be a positive integer.

The Meijer G-function involves a complex contour integral with one of the following types of integration paths:

- The contour goes from - i ∞ to i ∞ so that all poles of $\Gamma\left(b_j - s\right)$, $j = 1, \dots, m$ lie to the right of the path, and all poles of $\Gamma\left(1 - a_k + s\right)$, $k = 1, \dots, n$ lie to the left of the path.

  The integral converges if $c = m + n - \frac{p+q}{2} > 0$, $|arg(z)| < c\,\pi$. If $|arg(z)| = c\,\pi$, $c \geq 0$, the integral converges absolutely when $p = q$ and $\Re(\psi) < -1$, where

  $\Psi = \left(\sum_{j=1}^{q} b_j\right) - \left(\sum_{i=1}^{p} a_i\right)$. When $p \neq q$, the integral converges if you choose the contour so that the contour points near i ∞ and - i ∞ have a real part σ satisfying

  $(q-p)\sigma > \Re(\psi) + 1 - \frac{q-p}{2}$.
- The contour is a loop beginning and ending at *infinity* and encircling all poles of $\Gamma\left(b_j - s\right)$, $j = 1, \dots, m$ moving in the negative direction, but none of the poles of $\Gamma\left(1 - a_k + s\right)$, $k = 1, \dots, n$. The integral converges if $q \geq 1$ and either $p < q$ or $p = q$ and $|z| < 1$.
- The contour is a loop beginning and ending at - ∞ and encircling all poles of $\Gamma\left(1 - a_k + s\right)$, $k = 1, \dots, n$ moving in the positive direction, but none of the poles of $\Gamma\left(b_j + s\right)$, $j = 1, \dots, m$. The integral converges if $p \geq 1$ and either $p > q$ or $p = q$ and $|z| > 1$.

The integral represents an inverse Laplace transform or, more specifically, a Mellin-Barnes type of integral.

For a given set of parameters, the contour chosen in the definition of the Meijer G-function is the one for which the integral converges. If the integral converges for several contours, all contours lead to the same function.

The Meijer G-function satisfies a differential equation of order *max*(p, q) with respect to a variable z:

$$\left((-1)^{m+n-p}\, z\left(\prod_{i=1}^{p}\left(z\frac{d}{dz}-a_i-1\right)\right)-\prod_{j=1}^{q}\left(z\frac{d}{dz}-b_j\right)\right)G_{p,q}^{m,n}\left(\begin{matrix}a_1,\dots,a_p\\ b_1,\dots,b_p\end{matrix}\middle|\, z\right)=0.$$

If $p < q$, this differential equation has a regular singularity at $z = 0$ and an irregular singularity at $z = \infty$. If $p = q$, the points $z = 0$ and $z = \infty$ are regular singularities, and there is an additional regular singularity at $z = (-1)^{m+n-p}$.

The Meijer G-function represents an analytic continuation of the hypergeometric function [1]. For particular choices of parameters, you can express the Meijer G-function through the hypergeometric function. For example, if no two of the b_h terms, $h = 1, \dots, m$, differ by an integer or zero and all poles are simple, then

$$G_{p,q}^{m,n}\left(\begin{matrix}a_1,\dots,a_p\\ b_1,\dots,b_p\end{matrix}\middle|\, z\right)=\sum_{h=1}^{m}\frac{\left(\prod_{j=1\dots m,\, j\neq h}\Gamma\left(b_j-b_h\right)\right)\left(\prod_{j=1}^{n}\Gamma\left(1+b_h-a_j\right)\right)}{\left(\prod_{j=m+1}^{q}\Gamma\left(1+b_h-b_j\right)\right)\left(\prod_{j=n+1}^{p}\Gamma\left(a_j-b_h\right)\right)}z^{b}h_p\, F_{q-1}\left(A_h;B_h;(-1)^{p-m-n}\, z\right).$$

Here $p < q$ or $p = q$ and $|z| < 1$. A_h denotes

$$A_h=1+b_h-a_1,\dots,1+b_h-a_p.$$

B_h denotes

$$B_h=1+b_h-b_1,\dots,1+b_h-b_{(h-1)},1+b_h-b_{h+1},\dots,1+b_h-b_q.$$

Meijer G-functions with different parameters can represent the same function.

- The Meijer G-function is symmetric with respect to the parameters. Changing the order inside each of the following lists of vectors does not change the resulting Meijer G-function: $[a_1, \ldots, a_n]$, $[a_{n+1}, \ldots, a_p]$, $[b_1, \ldots, b_m]$, $[b_{m+1}, \ldots, b_q]$.
- If z is not a negative real number and $z \neq 0$, the function satisfies the following identity:

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1,\ldots,a_p \\ b_1,\ldots,b_q \end{matrix}\middle| z\right) = G_{q,p}^{n,m}\left(\begin{matrix} 1-b_1,\ldots,1-b_p \\ 1-a_1,\ldots,1-a_p \end{matrix}\middle| \frac{1}{z}\right).$$

.

- If $0 < n < p$ and $r = a_1 - a_p$ is an integer, the function satisfies the following identity:

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1,a_2,\ldots,a_{p-1},a_p \\ b_1,b_2,\ldots,b_{q-1},b_q \end{matrix}\middle| z\right) = G_{p,q}^{m,n}\left(\begin{matrix} a_p,a_2,\ldots,a_{p-1},a_1 \\ b_1,b_2,\ldots,b_{q-1},b_q \end{matrix}\middle| z\right).$$

.

- If $0 < m < q$ and $r = b_1 - b_q$ is an integer, the function satisfies the following identity:

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1,a_2,\ldots,a_{p-1},a_p \\ b_1,b_2,\ldots,b_{q-1},b_q \end{matrix}\middle| z\right) = (-1)^{\gamma}\, G_{p,q}^{m,n}\left(\begin{matrix} a_1,a_2,\ldots,a_{p-1},a_p \\ b_q,b_2,\ldots,b_{q-1},b_1 \end{matrix}\middle| z\right).$$

.

According to these rules, the `meijerG` function call can return `meijerG` with modified input parameters.

References

[1] Luke, Y. L., *The Special Functions and Their Approximations*. Vol. 1. New York: Academic Press, 1969.

[2] Prudnikov, A. P., Yu. A. Brychkov, and O. I. Marichev, *Integrals and Series*. Vol 3: More Special Functions. Gordon and Breach, 1990.

[3] Abramowitz, M., I. A. Stegun, Handbook of Mathematical Functions. 9th printing. New York: Dover Publications, 1970.

See Also

`hypergeom`

Introduced in R2017b

mfun

Numeric evaluation of special mathematical function

Note `mfun` will be removed in a future release. Instead, use the appropriate special function syntax listed in `mfunlist`. For example, use `bernoulli(n)` instead of `mfun('bernoulli',n)`.

Syntax

```
mfun('function',par1,par2,par3,par4)
```

Description

`mfun('function',par1,par2,par3,par4)` numerically evaluates one of the special mathematical functions listed in `mfunlist`. 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 `mfunlist`.

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`.

See Also

`mfunlist`

Introduced before R2006a

mfunlist

List special functions for use with `mfun`

Note `mfun` will be removed in a future release. Instead, use the appropriate special function syntax listed below. For example, use `bernoulli(n)` instead of `mfun('bernoulli',n)`.

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	Special Function Syntax	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)`
			`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)`
			`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)`	`beta(x,y)`	
Binomial coefficients	$$\binom{m}{n}=\frac{m!}{n!(m-n)!}$$ $$=\frac{\Gamma(m+1)}{\Gamma(n+1)\Gamma(m-n+1)}$$	`binomial(m,n)`	`nchoosek(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)`
			`ellipticK(k)`
			
			`ellipticE(k)`
			
			`ellipticPi(a,k)`
				`a` is real, $-\infty < a < \infty$.
				
				`k` is real, $0 < k < 1$.

Function Name	Definition	mfun Name	Special Function Syntax	Arguments

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)`
			`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)`
			`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)`	`dawson(x)`	
Digamma function	$\Psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$	`Psi(x)`	`psi(x)`	
Dilogarithm integral	$f(x) = \int_1^x \frac{\ln(t)}{1-t} dt$	`dilog(x)`	`dilog(x)`	$x > 1$
Error function	$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	`erf(z)`	`erf(z)`	

Function Name	Definition	mfun Name	Special Function Syntax	Arguments
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)`	`euler(n)` `euler(n,z)`	$n \geq 0$ $\lvert t \rvert < \frac{\pi}{2}$
Exponential integrals	$Ei(n,z) = \int_1^{\infty} \frac{e^{-zt}}{t^n} dt$ $Ei(x) = PV\left(-\int_{-\infty}^{x} \frac{e^t}{t}\right)$	`Ei(n,z)` `Ei(x)`	`expint(n,x)` `ei(x)`	$n \geq 0$ $\text{Real}(z) > 0$
Fresnel sine and cosine integrals	$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt$ $S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt$	`FresnelC(x)` `FresnelS(x)`	`fresnelc(x)` `fresnels(x)`	
Gamma function	$\Gamma(z) = \int_0^{\infty} t^{z-1}e^{-t}dt$	`GAMMA(z)`	`gamma(z)`	
Harmonic function	$h(n) = \sum_{k=1}^{n} \frac{1}{k} = \Psi(n+1) + \gamma$	`harmonic(n)`	`harmonic(n)`	$n > 0$

Function Name	Definition	mfun Name	Special Function Syntax	Arguments

Hyperbolic sine and cosine integrals	$Shi(z)=\int_0^z \frac{\sinh(t)}{t}dt$
	
	$Chi(z)=\gamma+\ln(z)+\int_0^z \frac{\cosh(t)-1}{t}dt$
		`Shi(z)`
		
		`Chi(z)`
			`sinhint(z)`
			
			`coshint(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,...]`
			`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)`
			`ellipticF(x,k)`
			
			`ellipticF(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` $= z$	`igamma(z1,z2)` `z1` $= a$ `z2` $= z$	
Logarithm of the gamma function	$\text{lnGAMMA}(z)=\ln(\Gamma(z))$	`lnGAMMA(z)`	`gammaln(z)`	

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

Chebyshev of the first and second kind	`T(n,x)`
	`U(n,x)`
		`chebyshevT(n,x)`
		`chebyshevU(n,x)`	

Gegenbauer	`G(n,a,x)`	`gegenbauerC(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1/2`.
Hermite	`H(n,x)`	`hermiteH(n,x)`	
Jacobi	`P(n,a,b,x)`	`jacobiP(n,a,b,x)`	`a`, `b` are nonrational algebraic expressions or rational numbers greater than `-1`.
Laguerre	`L(n,x)`	`laguerreL(n,x)`	
Generalized Laguerre	`L(n,a,x)`	`laguerreL(n,a,x)`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	`legendreP(n,x)`	

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`

Introduced before R2006a

min

Smallest elements

Syntax

```
C = min(A)
C = min(A,[],dim)
[C,I] = min(___)

C = min(A,B)
```

Description

`C = min(A)` returns the smallest element of `A` if `A` is a vector. If `A` is a matrix, this syntax treats the columns of `A` as vectors, returning a row vector containing the smallest element from each column.

`C = min(A,[],dim)` returns the smallest elements of matrix `A` along the dimension `dim`. Thus, `min(A,[],1)` returns a row vector containing the smallest elements of each column of `A`, and `min(A,[],2)` returns a column vector containing the smallest elements of each row of `A`.

Here, the required argument `[]` serves as a divider. If you omit it, `min(A,dim)` compares elements of `A` with the value `dim`.

`[C,I] = min(___)` finds the indices of the smallest elements, and returns them in output vector `I`. If there are several identical smallest values, this syntax returns the index of the first smallest element that it finds.

`C = min(A,B)` compares each element of `A` with the corresponding element of `B` and returns `C` containing the smallest elements of each pair.

Examples

Minimum of Vector of Numbers

Find the smallest of these numbers. Because these numbers are not symbolic objects, you get a floating-point result.

```
min([-pi, pi/2, 1, 1/3])
```

```
ans =
   -3.1416
```

Find the smallest of the same numbers converted to symbolic objects.

```
min(sym([-pi, pi/2, 1, 1/3]))
```

```
ans =
-pi
```

Minimum of Each Column in Symbolic Matrix

Create matrix `A` containing symbolic numbers, and call `min` for this matrix. By default, `min` returns the row vector containing the smallest elements of each column.

```
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
min(A)
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

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

Minimum of Each Row in Symbolic Matrix

Create matrix `A` containing symbolic numbers, and find the smallest elements of each row of the matrix. In this case, `min` returns the result as a column vector.

```
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
min(A,[],2)
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

ans =
 0
 3
 1
```

Indices of Smallest Elements

Create matrix `A`. Find the smallest element in each column and its index.

```
A = 1./sym(magic(3))
[Cc,Ic] = min(A)
```

```
A =
[ 1/8,   1, 1/6]
[ 1/3, 1/5, 1/7]
[ 1/4, 1/9, 1/2]

Cc =
[ 1/8, 1/9, 1/7]

Ic =
     1     3     2
```

Now, find the smallest element in each row and its index.

```
[Cr,Ir] = min(A,[],2)
```

```
Cr=
 1/8
 1/7
 1/9

Ir =
     1
     3
     2
```

If `dim` exceeds the number of dimensions of `A`, then the syntax `[C,I] = min(A,[],dim)` returns `C = A` and `I = ones(size(A))`.

```
[C,I] = min(A,[],3)

C =
[ 1/8,   1, 1/6]
[ 1/3, 1/5, 1/7]
[ 1/4, 1/9, 1/2]

I =
     1     1     1
     1     1     1
     1     1     1
```

Smallest Elements of Two Symbolic Matrices

Create matrices `A` and `B` containing symbolic numbers. Use `min` to compare each element of `A` with the corresponding element of `B`, and return the matrix containing the smallest elements of each pair.

```
A = sym(pascal(3))
B = toeplitz(sym([pi/3 pi/2 pi]))
minAB = min(A,B)

A =
[ 1, 1, 1]
[ 1, 2, 3]
[ 1, 3, 6]

B =
[ pi/3, pi/2,   pi]
[ pi/2, pi/3, pi/2]
[   pi, pi/2, pi/3]

minAB =
[ 1,    1,    1]
[ 1, pi/3, pi/2]
[ 1, pi/2, pi/3]
```

Minimum of Complex Numbers

When finding the minimum of these complex numbers, `min` chooses the number with the smallest complex modulus.

```
modulus = abs([-1 - i, 1 + 1/2*i])
minimum = min(sym([1 - i, 1/2 + i]))

modulus =
    1.4142    1.1180

minimum =
1/2 + 1i
```

If the numbers have the same complex modulus, `min` chooses the number with the smallest phase angle.

```
modulus = abs([1 - 1/2*i, 1 + 1/2*i])
phaseAngle = angle([1 - 1/2*i, 1 + 1/2*i])
minimum = min(sym([1 - 1/2*i, 1/2 + i]))

modulus =
    1.1180    1.1180

phaseAngle =
   -0.4636    0.4636

minimum =
1 - 1i/2
```

Input Arguments

`A` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, vector, or matrix. All elements of `A` must be convertible to floating-point numbers. If `A` is a scalar, then `min(A)` returns `A`. `A` cannot be a multidimensional array.

`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. The default value is `1`. If `dim` exceeds the number of dimensions of `A`, then `min(A,[],dim)` returns `A`, and `[C,I] = min(A,[],dim)` returns `C = A` and `I = ones(size(A))`.

`B` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, vector, or matrix. All elements of `B` must be convertible to floating-point numbers. If `A` and `B` are scalars, then `min(A,B)` returns the smallest of `A` and `B`.

If one argument is a vector or matrix, the other argument must either be a scalar or have the same dimensions as the first one. If one argument is a scalar and the other argument is a vector or matrix, then `min` expands the scalar into a vector or a matrix of the same length with all elements equal to that scalar.

`B` cannot be a multidimensional array.

Output Arguments

`C` — Smallest elements

symbolic number | symbolic vector

Smallest elements, returned as a symbolic number or vector of symbolic numbers.

`I` — Indices of smallest elements

symbolic number | symbolic vector | symbolic matrix

Indices of smallest elements, returned as a symbolic number or vector of symbolic numbers. `[C,I] = min(A,[],dim)` also returns matrix `I = ones(size(A))` if the value `dim` exceeds the number of dimensions of `A`.

Tips

- Calling `min` for numbers (or vectors or matrices of numbers) that are not symbolic objects invokes the MATLAB `min` function.
- For complex input `A`, `min` returns the complex number with the smallest complex modulus (magnitude), computed with `min(abs(A))`. If complex numbers have the same modulus, `min` chooses the number with the smallest phase angle, `min(angle(A))`.
- `min` ignores `NaN`s.

See Also

`abs` | `angle` | `max` | `min` | `sort`

Introduced in R2014a

minpoly

Minimal polynomial of matrix

Syntax

```
minpoly(A)
minpoly(A,var)
```

Description

`minpoly(A)` returns a vector of the coefficients of the minimal polynomial on page 4-1159 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.

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
```

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 $p(A) = 0$.

See Also

`charpoly` | `eig` | `jordan` | `poly2sym` | `sym2poly`

Introduced in R2012b

minus-

Symbolic subtraction

Syntax

```
-A
A - B
minus(A,B)
```

Description

`-A` returns the negation of `A`.

`A - B` subtracts `B` from `A` and returns the result.

`minus(A,B)` is an alternate way to execute `A - B`.

Examples

Subtract Scalar from Array

Subtract `2` from array `A`.

```
syms x
A = [x 1;-2 sin(x)];
A - 2
```

```
ans =
[ x - 2,          -1]
[    -4, sin(x) - 2]
```

`minus` subtracts `2` from each element of `A`.

Subtract the identity matrix from matrix `M`:

```
syms x y z
M = [0 x; y z];
M - eye(2)

ans =
[ -1,     x]
[  y, z - 1]
```

Subtract Numeric and Symbolic Arguments

Subtract one number from another. Because these are not symbolic objects, you receive floating-point results.

```
11/6 - 5/4

ans =
    0.5833
```

Perform subtraction symbolically by converting the numbers to symbolic objects.

```
sym(11/6) - sym(5/4)

ans =
7/12
```

Alternatively, call `minus` to perform subtraction.

```
minus(sym(11/6),sym(5/4))

ans =
7/12
```

Subtract Matrices

Subtract matrices `B` and `C` from `A`.

```
A = sym([3 4; 2 1]);
B = sym([8 1; 5 2]);
C = sym([6 3; 4 9]);
Y = A - B - C

Y =
[ -11,   0]
[  -7, -10]
```

Use syntax `-Y` to negate the elements of `Y`.

```
-Y
```

```
ans =
[ 11,  0]
[  7, 10]
```

Subtract Functions

Subtract function `g` from function `f`.

```
syms f(x) g(x)
f = sin(x) + 2*x;
y = f - g
```

```
y(x) =
2*x - g(x) + sin(x)
```

Input Arguments

`A` — Input
symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression.

`B` — Input
symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression.

Tips

- All nonscalar arguments must have the same size. If one input argument is nonscalar, then `minus` expands the scalar into an array of the same size as the nonscalar argument, with all elements equal to the corresponding scalar.

See Also

`ctranspose` | `ldivide` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

mldivide\

Symbolic matrix left division

Syntax

```
X = A\B
X = mldivide(A,B)
```

Description

`X = A\B` solves the symbolic system of linear equations in matrix form, `A*X = B` for `X`.

If the solution does not exist or if it is not unique, the `\` operator issues a warning.

`A` can be a rectangular matrix, but the equations must be consistent. The symbolic operator `\` does not compute least-squares solutions.

`X = mldivide(A,B)` is equivalent to `x = A\B`.

Examples

System of Equations in Matrix Form

Solve a system of linear equations specified by a square matrix of coefficients and a vector of right sides of equations.

Create a matrix containing the coefficient of equation terms, and a vector containing the right sides of equations.

```
A = sym(pascal(4))
b = sym([4; 3; 2; 1])
```

```
A =
[ 1, 1,  1,  1]
[ 1, 2,  3,  4]
[ 1, 3,  6, 10]
[ 1, 4, 10, 20]

b =
 4
 3
 2
 1
```

Use the operator \ to solve this system.

```
X = A\b

X =
  5
 -1
  0
  0
```

Rank-Deficient System

Create a matrix containing the coefficients of equation terms, and a vector containing the right sides of equations.

```
A = sym(magic(4))
b = sym([0; 1; 1; 0])

A =
[ 16,  2,  3, 13]
[  5, 11, 10,  8]
[  9,  7,  6, 12]
[  4, 14, 15,  1]

b =
 0
 1
 1
 0
```

Find the rank of the system. This system contains four equations, but its rank is `3`. Therefore, the system is rank-deficient. This means that one variable of the system is not independent and can be expressed in terms of other variables.

```
rank(horzcat(A,b))
```

```
ans =
3
```

Try to solve this system using the symbolic \ operator. Because the system is rank-deficient, the returned solution is not unique.

```
A\b
```

```
Warning: Solution is not unique because the system is rank-deficient.

ans =
  1/34
 19/34
 -9/17
     0
```

Inconsistent System

Create a matrix containing the coefficient of equation terms, and a vector containing the right sides of equations.

```
A = sym(magic(4))
b = sym([0; 1; 2; 3])
```

```
A =
[ 16,  2,  3, 13]
[  5, 11, 10,  8]
[  9,  7,  6, 12]
[  4, 14, 15,  1]

b =
 0
 1
 2
 3
```

Try to solve this system using the symbolic \ operator. The operator issues a warning and returns a vector with all elements set to `Inf` because the system of equations is inconsistent, and therefore, no solution exists. The number of elements in the resulting vector equals the number of equations (rows in the coefficient matrix).

```
A\b
```

```
Warning: Solution does not exist because the system is inconsistent.

ans =
 Inf
 Inf
 Inf
 Inf
```

Find the reduced row echelon form of this system. The last row shows that one of the equations reduced to `0 = 1`, which means that the system of equations is inconsistent.

```
rref(horzcat(A,b))

ans =
[ 1, 0, 0,  1, 0]
[ 0, 1, 0,  3, 0]
[ 0, 0, 1, -3, 0]
[ 0, 0, 0,  0, 1]
```

Input Arguments

`A` — Coefficient matrix

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Coefficient matrix, specified as a symbolic number, variable, expression, function, vector, or matrix.

`B` — Right side

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Right side, specified as a symbolic number, variable, expression, function, vector, or matrix.

Output Arguments

`X` — Solution

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Solution, returned as a symbolic number, variable, expression, function, vector, or matrix.

Tips

- When dividing by zero, `mldivide` considers the numerator's sign and returns `Inf` or `-Inf` accordingly.

  ```
  syms x
  [sym(0)\sym(1), sym(0)\sym(-1), sym(0)\x]
  
  ans =
  [ Inf, -Inf, Inf*x]
  ```

See Also

`ctranspose` | `ldivide` | `minus` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

mod

Symbolic modulus after division

Syntax

```
mod(a,b)
```

Description

`mod(a,b)` finds the modulus on page 4-1172 after division. To find the remainder, use `rem`.

If `a` is a polynomial expression, then `mod(a,b)` finds the modulus for each coefficient.

Examples

Divide Integers by Integers

Find the modulus after division in case both the dividend and divisor are integers.

Find the modulus after division for these numbers.

```
[mod(sym(27), 4), mod(sym(27), -4), mod(sym(-27), 4), mod(sym(-27), -4)]
```

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

Divide Rationals by Integers

Find the modulus after division in case the dividend is a rational number, and divisor is an integer.

Find the modulus after division for these numbers.

```
[mod(sym(22/3), 5), mod(sym(1/2), 7), mod(sym(27/6), -11)]

ans =
[ 7/3, 1/2, -13/2]
```

Divide Polynomial Expressions by Integers

Find the modulus after division in case the dividend is a polynomial expression, and divisor is an integer. If the dividend is a polynomial expression, then `mod` finds the modulus for each coefficient.

Find the modulus after division for these polynomial expressions.

```
syms x
mod(x^3 - 2*x + 999, 10)

ans =
x^3 + 8*x + 9

mod(8*x^3 + 9*x^2 + 10*x + 11, 7)

ans =
x^3 + 2*x^2 + 3*x + 4
```

Divide Elements of Matrices

For vectors and matrices, `mod` finds the modulus after division element-wise. Nonscalar arguments must be the same size.

Find the modulus after division for the elements of these two matrices.

```
A = sym([27, 28; 29, 30]);
B = sym([2, 3; 4, 5]);
mod(A,B)

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

Find the modulus after division for the elements of matrix `A` and the value `9`. Here, `mod` expands `9` into the `2`-by-`2` matrix with all elements equal to `9`.

```
mod(A,9)
```

```
ans =
[ 0, 1]
[ 2, 3]
```

Input Arguments

`a` — Dividend (numerator)

number | symbolic number | symbolic variable | polynomial expression | vector | matrix

Dividend (numerator), specified as a number, symbolic number, variable, polynomial expression, or a vector or matrix of numbers, symbolic numbers, variables, or polynomial expressions.

`b` — Divisor (denominator)

number | symbolic number | vector | matrix

Divisor (denominator), specified as a number, symbolic number, or a vector or matrix of numbers or symbolic numbers.

Definitions

Modulus

The modulus of a and b is

$$\operatorname{mod}(a,b) = a - b * \operatorname{floor}\left(\frac{a}{b}\right),$$

where `floor` rounds (a/b) towards negative infinity. For example, the modulus of -8 and -3 is -2, but the modulus of -8 and 3 is 1.

If b = 0, then mod(a,0) = 0.

Tips

- Calling `mod` for numbers that are not symbolic objects invokes the MATLAB `mod` function.

- All nonscalar arguments must be the same size. If one input arguments is nonscalar, then `mod` expands the scalar into a vector or matrix of the same size as the nonscalar argument, with all elements equal to the corresponding scalar.

See Also

`quorem` | `rem`

Introduced before R2006a

mpower^

Symbolic matrix power

Syntax

```
A^B
mpower(A,B)
```

Description

`A^B` computes `A` to the `B` power.

`mpower(A,B)` is equivalent to `A^B`.

Examples

Matrix Base and Scalar Exponent

Create a `2`-by-`2` matrix.

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

```
A =
[ a11, a12]
[ a21, a22]
```

Find `A^2`.

```
A^2
```

```
ans =
[   a11^2 + a12*a21, a11*a12 + a12*a22]
[ a11*a21 + a21*a22,   a22^2 + a12*a21]
```

Scalar Base and Matrix Exponent

Create a 2-by-2 symbolic magic square.

```
A = sym(magic(2))
```

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

Find π^A.

```
sym(pi)^A
```

```
ans =
[   (3*pi^7 + 4)/(7*pi^2), (3*(pi^7 - 1))/(7*pi^2)]
[ (4*(pi^7 - 1))/(7*pi^2),   (4*pi^7 + 3)/(7*pi^2)]
```

Input Arguments

A — Base

number | symbolic number | symbolic variable | symbolic function | symbolic expression | square symbolic matrix

Base, specified as a number or a symbolic number, variable, expression, function, or square matrix. A and B must be one of the following:

- Both are scalars.
- A is a square matrix, and B is a scalar.
- B is a square matrix, and A is a scalar.

B — Exponent

number | symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic square matrix

Exponent, specified as a number or a symbolic number, variable, expression, function, or square matrix. A and B must be one of the following:

- Both are scalars.
- A is a square matrix, and B is a scalar.

- `B` is a square matrix, and `A` is a scalar.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

mrdivide/

Symbolic matrix right division

Syntax

```
X = B/A
X = mrdivide(B,A)
```

Description

`X = B/A` solves the symbolic system of linear equations in matrix form, `X*A = B` for `X`. The matrices `A` and `B` must contain the same number of columns. The right division of matrices `B/A` is equivalent to `(A'\B')'`.

If the solution does not exist or if it is not unique, the `/` operator issues a warning.

`A` can be a rectangular matrix, but the equations must be consistent. The symbolic operator `/` does not compute least-squares solutions.

`X = mrdivide(B,A)` is equivalent to `x = B/A`.

Examples

System of Equations in Matrix Form

Solve a system of linear equations specified by a square matrix of coefficients and a vector of right sides of equations.

Create a matrix containing the coefficient of equation terms, and a vector containing the right sides of equations.

```
A = sym(pascal(4))
b = sym([4 3 2 1])
```

```
A =
[ 1, 1,  1,  1]
[ 1, 2,  3,  4]
[ 1, 3,  6, 10]
[ 1, 4, 10, 20]

b =
[ 4, 3, 2, 1]
```

Use the operator / to solve this system.

```
X = b/A
```

```
X =
[ 5, -1, 0, 0]
```

Rank-Deficient System

Create a matrix containing the coefficient of equation terms, and a vector containing the right sides of equations.

```
A = sym(magic(4))'
b = sym([0 1 1 0])
```

```
A =
[ 16,  5,  9,  4]
[  2, 11,  7, 14]
[  3, 10,  6, 15]
[ 13,  8, 12,  1]

b =
[ 0, 1, 1, 0]
```

Find the rank of the system. This system contains four equations, but its rank is 3. Therefore, the system is rank-deficient. This means that one variable of the system is not independent and can be expressed in terms of other variables.

```
rank(vertcat(A,b))
```

```
ans =
3
```

Try to solve this system using the symbolic / operator. Because the system is rank-deficient, the returned solution is not unique.

```
b/A
```

```
Warning: Solution is not unique because the system is rank-deficient.

ans =
[ 1/34, 19/34, -9/17, 0]
```

Inconsistent System

Create a matrix containing the coefficient of equation terms, and a vector containing the right sides of equations.

```
A = sym(magic(4))'
b = sym([0 1 2 3])
```

```
A =
[ 16,  5,  9,  4]
[  2, 11,  7, 14]
[  3, 10,  6, 15]
[ 13,  8, 12,  1]

b =
[ 0, 1, 2, 3]
```

Try to solve this system using the symbolic / operator. The operator issues a warning and returns a vector with all elements set to `Inf` because the system of equations is inconsistent, and therefore, no solution exists. The number of elements equals the number of equations (rows in the coefficient matrix).

```
b/A
```

```
Warning: Solution does not exist because the system is inconsistent.

ans =
[ Inf, Inf, Inf, Inf]
```

Find the reduced row echelon form of this system. The last row shows that one of the equations reduced to `0 = 1`, which means that the system of equations is inconsistent.

```
rref(vertcat(A,b)')
```

```
ans =
[ 1, 0, 0,  1, 0]
[ 0, 1, 0,  3, 0]
[ 0, 0, 1, -3, 0]
[ 0, 0, 0,  0, 1]
```

Input Arguments

`A` — Coefficient matrix

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Coefficient matrix, specified as a symbolic number, variable, expression, function, vector, or matrix.

`B` — Right side

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Right side, specified as a symbolic number, variable, expression, function, vector, or matrix.

Output Arguments

`X` — Solution

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Solution, returned as a symbolic number, variable, expression, function, vector, or matrix.

Tips

- When dividing by zero, `mrdivide` considers the numerator's sign and returns `Inf` or `-Inf` accordingly.

  ```
  syms x
  [sym(1)/sym(0), sym(-1)/sym(0), x/sym(0)]
  ```

  ```
  ans =
  [ Inf, -Inf, Inf*x]
  ```

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mtimes` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

mtimes*

Symbolic matrix multiplication

Syntax

```
A*B
mtimes(A,B)
```

Description

A*B is the matrix product of A and B. If A is an m-by-p and B is a p-by-n matrix, then the result is an m-by-n matrix C defined as

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j)$$

For nonscalar A and B, the number of columns of A must equal the number of rows of B. Matrix multiplication is not universally commutative for nonscalar inputs. That is, typically A*B is not equal to B*A. If at least one input is scalar, then A*B is equivalent to A.*B and is commutative.

mtimes(A,B) is equivalent to A*B.

Examples

Multiply Two Vectors

Create a 1-by-5 row vector and a 5-by-1 column vector.

```
syms x
A = [x, 2*x^2, 3*x^3, 4*x^4]
B = [1/x; 2/x^2; 3/x^3; 4/x^4]
```

```
A =
[ x, 2*x^2, 3*x^3, 4*x^4]
```

```
B =
   1/x
 2/x^2
 3/x^3
 4/x^4
```

Find the matrix product of these two vectors.

```
A*B
```

```
ans =
30
```

Multiply Two Matrices

Create a 4-by-3 matrix and a 3-by-2 matrix.

```
A = sym('a%d%d', [4 3])
B = sym('b%d%d', [3 2])
```

```
A =
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
[ a41, a42, a43]

B =
[ b11, b12]
[ b21, b22]
[ b31, b32]
```

Multiply `A` by `B`.

```
A*B
```

```
ans =
[ a11*b11 + a12*b21 + a13*b31, a11*b12 + a12*b22 + a13*b32]
[ a21*b11 + a22*b21 + a23*b31, a21*b12 + a22*b22 + a23*b32]
[ a31*b11 + a32*b21 + a33*b31, a31*b12 + a32*b22 + a33*b32]
[ a41*b11 + a42*b21 + a43*b31, a41*b12 + a42*b22 + a43*b32]
```

Multiply Matrix by Scalar

Create a `4`-by-`4` Hilbert matrix `H`.

```
H = sym(hilb(4))
```

```
H =
[   1, 1/2, 1/3, 1/4]
[ 1/2, 1/3, 1/4, 1/5]
[ 1/3, 1/4, 1/5, 1/6]
[ 1/4, 1/5, 1/6, 1/7]
```

Multiply `H` by e^{π}.

```
C = H*exp(sym(pi))
```

```
C =
[   exp(pi), exp(pi)/2, exp(pi)/3, exp(pi)/4]
[ exp(pi)/2, exp(pi)/3, exp(pi)/4, exp(pi)/5]
[ exp(pi)/3, exp(pi)/4, exp(pi)/5, exp(pi)/6]
[ exp(pi)/4, exp(pi)/5, exp(pi)/6, exp(pi)/7]
```

Use `vpa` and `digits` to approximate symbolic results with the required number of digits. For example, approximate it with five-digit accuracy.

```
old = digits(5);
vpa(C)
digits(old)
```

```
ans =
[ 23.141,  11.57, 7.7136, 5.7852]
[  11.57, 7.7136, 5.7852, 4.6281]
[ 7.7136, 5.7852, 4.6281, 3.8568]
[ 5.7852, 4.6281, 3.8568, 3.3058]
```

Input Arguments

`A` — Input

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, function, vector, or matrix. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

`B` — Input

symbolic number | symbolic variable | symbolic function | symbolic expression | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, function, vector, or matrix. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `plus` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

mupad

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`. The file name must be a full path unless the file is in the current folder. 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:

```
reset(symengine);
if ~feature('ShowFigureWindows')
     disp('no display available, skipping test ....');
else mphandle = mupad; end
```

```
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`

Topics

"Create MuPAD Notebooks" on page 3-3
"Open MuPAD Notebooks" on page 3-6

Introduced in R2008b

mupadNotebookTitle

Window title of MuPAD notebook

Syntax

```
T = mupadNotebookTitle(nb)
```

Description

`T = mupadNotebookTitle(nb)` returns a cell array containing the window title of the MuPAD notebook with the handle `nb`. If `nb` is a vector of handles to notebooks, then `mupadNotebookTitle(nb)` returns a cell array of the same size as `nb`.

Examples

Find Titles of Particular Notebooks

Knowing the handles to notebooks, find the titles of these notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Find the titles of `myFile1.mn` and `myFile2.mn`:

```
mupadNotebookTitle([nb1; nb2])
```

```
ans =
    'myFile1'
    'myFile2'
```

List Titles of All Open Notebooks

Get a cell array containing titles of all currently open MuPAD notebooks.

Suppose that your current folder contains MuPAD notebooks named `myFile1.mn` and `myFile2.mn`. Open them keeping their handles in variables `nb1` and `nb2`, respectively. Also create a new notebook with the handle `nb3`:

```
nb1 = mupad('myFile1.mn')
nb2 = mupad('myFile2.mn')
nb3 = mupad
```

```
nb1 =
myFile1

nb2 =
myFile2

nb3 =
Notebook1
```

Suppose that there are no other open notebooks. Use `allMuPADNotebooks` to get a vector of handles to these notebooks:

```
allNBs = allMuPADNotebooks
```

```
allNBs =
myFile1
myFile2
Notebook1
```

List the titles of all open notebooks. The result is a cell array of character vectors.

```
mupadNotebookTitle(allNBs)
```

```
ans =
    'myFile1'
    'myFile2'
    'Notebook1
```

Return Single Notebook Title as Character Vector

`mupadNotebookTitle` returns a cell array of titles even if there is only one element in that cell array. If `mupadNotebookTitle` returns a cell array of one element, you can quickly convert it to a character vector by using `char`.

Create a new notebook with the handle `nb`:

```
nb = mupad;
```

Find the title of that notebook and convert it to a character vector:

```
titleAsStr = char(mupadNotebookTitle(nb));
```

Use the title the same way as any character vector:

```
disp(['The current notebook title is: ' titleAsStr])
```

```
The current notebook title is: Notebook1
```

- "Create MuPAD Notebooks" on page 3-3
- "Open MuPAD Notebooks" on page 3-6
- "Save MuPAD Notebooks" on page 3-12
- "Evaluate MuPAD Notebooks from MATLAB" on page 3-13
- "Copy Variables and Expressions Between MATLAB and MuPAD" on page 3-52
- "Close MuPAD Notebooks from MATLAB" on page 3-17

Input Arguments

`nb` — Pointer to MuPAD notebook
handle to notebook | vector of handles to notebooks

Pointer to MuPAD notebook, specified as a MuPAD notebook handle or a vector of handles. You create the notebook handle when opening a notebook with the `mupad` or `openmn` function.

You can get the list of all open notebooks using the `allMuPADNotebooks` function. `mupadNotebookTitle` accepts a vector of handles returned by `allMuPADNotebooks`.

Output Arguments

`T` — Window title of MuPAD notebook

cell array

Window title of MuPAD notebook, returned as a cell array. If `nb` is a vector of handles to notebooks, then `T` is a cell array of the same size as `nb`.

See Also

`allMuPADNotebooks` | `close` | `evaluateMuPADNotebook` | `getVar` | `mupad` | `openmn` | `setVar`

Topics

"Create MuPAD Notebooks" on page 3-3
"Open MuPAD Notebooks" on page 3-6
"Save MuPAD Notebooks" on page 3-12
"Evaluate MuPAD Notebooks from MATLAB" on page 3-13
"Copy Variables and Expressions Between MATLAB and MuPAD" on page 3-52
"Close MuPAD Notebooks from MATLAB" on page 3-17

Introduced in R2013b

mupadwelcome

Start MuPAD interfaces

Syntax

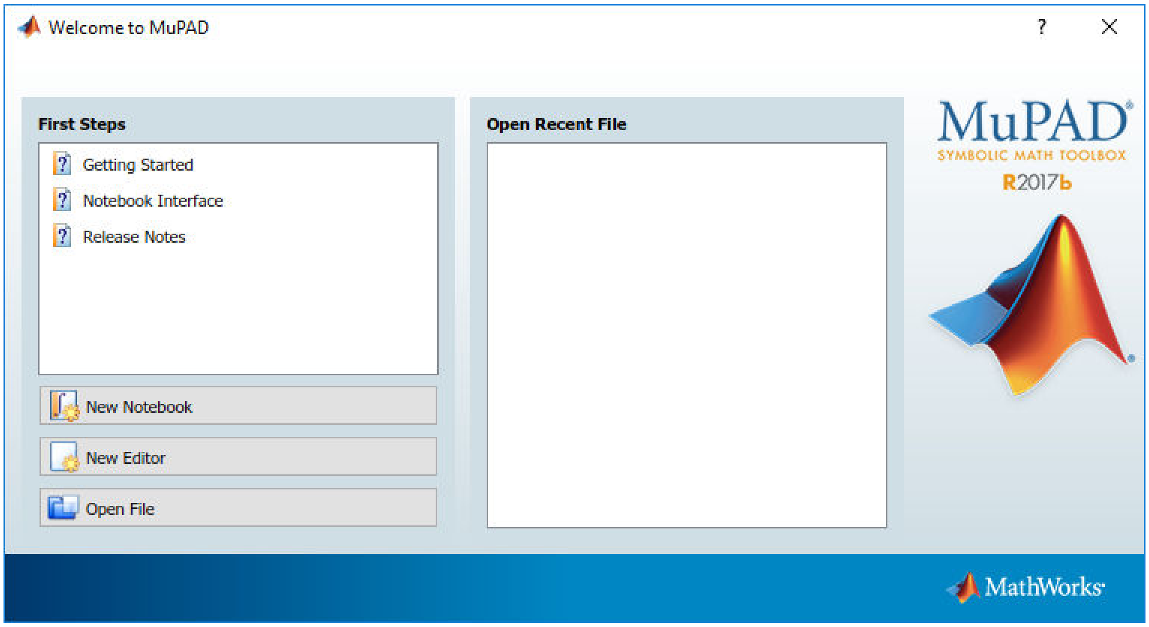
```
mupadwelcome
```

Description

`mupadwelcome` opens a window that enables you to start various interfaces:

- MuPAD Notebook, 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`

Topics

"Create MuPAD Notebooks" on page 3-3
"Open MuPAD Notebooks" on page 3-6

Introduced in R2008b

nchoosek

Binomial coefficient

Syntax

```
b = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`b = nchoosek(n,k)` returns the binomial coefficient of `n` and `k`, defined as `n!/(k!(n - k)!)`. This is the number of combinations of `n` items taken `k` at a time.

`C = nchoosek(v,k)` returns a matrix containing all possible combinations of the elements of vector `v` taken `k` at a time. Matrix `C` has `k` columns and `n!/(k!(n - k)!)` rows, where `n` is `length(v)`. In this syntax, `k` must be a nonnegative integer.

Examples

Binomial Coefficients for Numeric and Symbolic Arguments

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 + 1i/6, 1, 0]
```

Handle Expressions Containing Binomial Coefficients

Many functions, such as `diff` and `expand`, can handle expressions containing `nchoosek`.

Differentiate the binomial coefficient.

```
syms n k
diff(nchoosek(n, 2))
```

```
ans =
-(psi(n - 1) - psi(n + 1))*nchoosek(n, 2)
```

Expand the binomial coefficient.

```
expand(nchoosek(n, k))
```

```
ans =
-(n*gamma(n))/(k^2*gamma(k)*gamma(n - k) - k*n*gamma(k)*gamma(n - k))
```

Pascal Triangle

Use `nchoosek` to build the Pascal triangle.

```
m = 5;
for n = 0:m
    C = sym([]);
  for k = 0:n
    C = horzcat(C, nchoosek(n, k));
  end
  disp(C)
end
```

```
1
[ 1, 1]
[ 1, 2, 1]
[ 1, 3, 3, 1]
[ 1, 4, 6, 4, 1]
[ 1, 5, 10, 10, 5, 1]
```

All Combinations of Vector Elements

Find all combinations of elements of a `1`-by-`5` symbolic row vector taken three and four at a time.

Create a `1`-by-`5` symbolic vector with the elements `x1`, `x2`, `x3`, `x4`, and `x5`.

```
v = sym('x', [1, 5])
```

```
v =
[ x1, x2, x3, x4, x5]
```

Find all combinations of the elements of `v` taken three at a time.

```
C = nchoosek(v, 3)
```

```
C =
[ x1, x2, x3]
[ x1, x2, x4]
[ x1, x3, x4]
[ x2, x3, x4]
[ x1, x2, x5]
[ x1, x3, x5]
[ x2, x3, x5]
[ x1, x4, x5]
[ x2, x4, x5]
[ x3, x4, x5]
```

```
C = nchoosek(v, 4)
```

```
C =
[ x1, x2, x3, x4]
[ x1, x2, x3, x5]
[ x1, x2, x4, x5]
[ x1, x3, x4, x5]
[ x2, x3, x4, x5]
```

Input Arguments

`n` — Number of possible choices

symbolic number | symbolic variable | symbolic expression | symbolic function

Number of possible choices, specified as a symbolic number, variable, expression, or function.

`k` — Number of selected choices

symbolic number | symbolic variable | symbolic expression | symbolic function

Number of selected choices, specified as a symbolic number, variable, expression, or function. If the first argument is a symbolic vector `v`, then `k` must be a nonnegative integer.

`v` — Set of all choices

symbolic vector

Set of all choices, specified as a vector of symbolic numbers, variables, expressions, or functions.

Output Arguments

`b` — Binomial coefficient

nonnegative scalar value

Binomial coefficient, returned as a nonnegative scalar value.

`C` — All combinations of `v`

matrix

All combinations of `v`, returned as a matrix of the same type as `v`.

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:

$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}$$

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.

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` | `factorial` | `gamma` | `psi`

Introduced in R2012a

ne

Define inequality

Syntax

```
A ~= B
ne(A,B)
```

Description

`A ~= B` creates a symbolic inequality.

`ne(A,B)` is equivalent to `A ~= B`.

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
```

Tips

- Calling `~=` or `ne` for non-symbolic `A` and `B` invokes the MATLAB `ne` function. This function returns a logical array with elements set to logical `1 (true)` where `A` is not equal to `B`; otherwise, it returns logical `0 (false)`.
- 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`.

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` | `lt`

Topics

"Set Assumptions" on page 1-28

Introduced in R2012a

newUnit

Define new unit

Syntax

```
newUnit(name,definition)
```

Description

`newUnit(name,definition)` defines the new unit `name` using the expression `definition`. The definition must be in terms of existing symbolic units.

Examples

Define New Unit and Rewrite Unit

Define the new unit `speedOfLight` as `3e8` meters per second.

```
u = symunit;
c = newUnit('speedOfLight',3e8*u.m/u.s)
```

```
c =
[speedOfLight]
```

Define the famous equation $E = mc^2$ using the new unit.

```
syms mass
m = mass*u.kg;
E = m*c^2
```

```
E =
mass*[kg]*[speedOfLight]^2
```

Alternatively, you can specify the unit by using `u.SpeedOfLight`.

Rewrite `E` in terms of meters per second.

```
E = rewrite(E,u.m/u.s)

E =
90000000000000000*mass*(([kg]*[m]^2)/[s]^2)
```

Since the standard unit of energy is the Joule, rewrite `E` in terms of `Joule`.

```
E = rewrite(E,u.J)

E =
90000000000000000*mass*[J]
```

Input Arguments

`name` — Name of new unit
character vector | string

Name of the new unit, specified as a character vector or string.

`definition` — Definition of new unit
symbolic expression of units

Definition of the new unit, specified as a symbolic expression of units. The new unit must be defined in terms of existing symbolic units. For example, `newUnit('workday', 8*u.hour)` where `u = symunit`.

See Also

`checkUnits` | `isUnit` | `removeUnit` | `separateUnits` | `str2symunit` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

newUnitSystem

Define unit system

Syntax

```
newUnitSystem(name,baseUnits)
newUnitSystem(name,baseUnits,derivedUnits)
```

Description

`newUnitSystem(name,baseUnits)` defines a new “Unit System” on page 4-1207 with the name `name` and the base units `baseUnits`. Now, you can convert units into the new unit system by using `rewrite`. By default, the unit systems already available are `SI`, `CGS`, and `US`.

`newUnitSystem(name,baseUnits,derivedUnits)` additionally specifies the derived units `derivedUnits`.

Examples

Define New Unit System from Existing System

A unit system is a collection of units to express quantities. The easiest way to define a new unit system is to modify one of the default unit systems: `SI`, `CGS`, and `US`.

Modify `SI` to use kilometer for length and hour for time by getting the base units using `baseunits` and modifying them by using `subs`.

```
u = symunit;
SIUnits = baseUnits('SI')
```

```
SIUnits =
[ [kg], [s], [m], [A], [cd], [mol], [K]]
```

```
newUnits = subs(SIUnits,[u.m u.s],[u.km u.hr])

newUnits =
[ [kg], [h], [km], [A], [cd], [mol], [K]]
```

Note Do not define a variable called `baseUnits` because the variable will prevent access to the `baseUnits` function.

Define the new unit system `SI_km_hr` using the new base units.

```
newUnitSystem('SI_km_hr',newUnits)

ans =
    "SI_km_hr"
```

Rewrite 5 meter/second to the `SI_km_hr` unit system. As expected, the result is in terms of kilometers and hours.

```
rewrite(5*u.m/u.s,'SI_km_hr')

ans =
18*([km]/[h])
```

Specify Base and Derived Units Directly

Specify a new unit system by specifying the base and derived units directly. A unit system has up to 7 base units. For details, see “Unit System” on page 4-1207.

Define a new unit system with these base units: gram, hour, meter, ampere, candela, mol, and celsius. Specify these derived units: kilowatt, newton, and volt.

```
u = symunit;
sysName = 'myUnitSystem';
bunits = [u.g u.hr u.m u.A u.cd u.mol u.Celsius];
dunits = [u.kW u.N u.V];
newUnitSystem(sysName,bunits,dunits)

ans =
    "myUnitSystem"
```

Rewrite 2000 Watts to the new system. By default, `rewrite` uses base units, which can be hard to read.

```
rewrite(2000*u.W,sysName)

ans =
9331200000000000*(([g]*[m]^2)/[h]^3)
```

Instead, for readability, rewrite 2000 Watts to *derived* units of `myUnitSystem` by specifying `'Derived'` as the third argument. Converting to the derived units of a unit system attempts to select convenient units. The result uses the derived unit, kilowatt, instead of base units. For more information, see “Unit Conversions and Unit Systems” on page 2-30.

```
rewrite(2000*u.W,sysName,'Derived')

ans =
2*[kW]
```

Input Arguments

`name` — Name of unit system

string | character vector

Name of unit system, specified as a string or character vector.

`baseUnits` — Base units of unit system

vector of symbolic units

Base units of unit system, specified as a vector of symbolic units. The base units must be independent in terms of the dimensions mass, time, length, electric current, luminous intensity, amount of substance, and temperature. Thus, in a unit system, there are up to 7 base units.

`derivedUnits` — Derived units of unit system

vector of symbolic units

Derived units of unit system, specified as a vector of symbolic units. Derived units are optional and added for convenience of representation.

Definitions

Unit System

A unit system is a collection of base units and derived units that follows these rules:

- Base units must be independent in terms of the dimensions mass, time, length, electric current, luminous intensity, amount of substance, and temperature. Therefore, a unit system has up to 7 base units. As long as the independence is satisfied, any unit can be a base unit, including units such as newton or watt.
- A unit system can have less than 7 base units. For example, mechanical systems need base units only for the dimensions length, mass, and time.
- Derived units in a unit system must have a representation in terms of the products of powers of the base units for that system. Unlike base units, derived units do not have to be independent.
- Derived units are optional and added for convenience of representation. For example, kg m/s^2 is abbreviated by Newton.
- An example of a unit system is the SI unit system, which has 7 base units: kilogram, second, meter, ampere, candela, mol, and kelvin. There are 22 derived units found by calling `derivedUnits('SI')`.

See Also

`baseUnits` | `derivedUnits` | `removeUnitSystem` | `rewrite` | `symunit` | `unitSystems`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

nextprime

Next prime number

Syntax

```
nextprime(n)
```

Description

`nextprime(n)` returns the next prime number greater than or equal to `n`. If `n` is a vector or matrix, then `nextprime` acts element-wise on `n`.

Examples

Find Next Prime Number

Find the next prime number greater than `100`. Because `nextprime` only accepts symbolic input, wrap `100` with `sym`.

```
nextprime(sym(100))
```

```
ans =
101
```

Find the next prime numbers greater than `1000`, `10000`, and `100000` by specifying the input as a vector.

```
nextprime(sym([1000 10000 100000]))
```

```
ans =
[ 1009, 10007, 100003]
```

Find Large Prime Number

When finding large prime numbers, if your input has 15 or more digits, then use quotation marks to represent the number accurately. The best way to find an arbitrary large prime is to use powers of 10, which are accurately represented without requiring quotation marks. For more information, see “Numeric to Symbolic Conversion” on page 2-125.

Find a large prime number by using `10^sym(18)`.

```
nextprime(10^sym(18))
```

```
ans =
1000000000000000003
```

Find the next prime number to `823572345728582545` by using quotation marks.

```
nextprime(sym('823572345728582545'))
```

```
ans =
823572345728582623
```

Input Arguments

`n` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, or as a symbolic vector or matrix of symbolic numbers. `n` is rounded up.

Example: `sym(100)`

See Also

`isprime` | `prevprime` | `primes`

Introduced in R2016b

nnz

Number of nonzero elements

Syntax

```
nnz(X)
```

Description

`nnz(X)` computes the number of nonzero elements in `X`.

Examples

Number of Nonzero Elements and Matrix Density

Compute the number of nonzero elements of a 10-by-10 symbolic matrix and its density.

Create the following matrix as an element-wise product of a random matrix composed of `0`s and `1`s and the symbolic Hilbert matrix.

```
A = gallery('rando',10).*sym(hilb(10))
```

```
A =
[    0,  1/2,  1/3,    0,  1/5, 1/6,  1/7,    0,    0, 1/10]
[  1/2,  1/3,  1/4,    0,    0,   0,    0,  1/9,    0, 1/11]
[    0,  1/4,    0,    0,  1/7, 1/8,  1/9, 1/10,    0,    0]
[    0,  1/5,    0,    0,  1/8,   0, 1/10, 1/11,    0, 1/13]
[  1/5,    0,    0,  1/8,  1/9,   0, 1/11, 1/12,    0,    0]
[  1/6,    0,  1/8,    0,    0,   0,    0,    0, 1/14, 1/15]
[    0,  1/8,    0,    0, 1/11,   0,    0,    0,    0, 1/16]
[  1/8,    0, 1/10, 1/11,    0,   0,    0, 1/15, 1/16,    0]
[    0,    0, 1/11,    0, 1/13,   0, 1/15, 1/16, 1/17,    0]
[ 1/10, 1/11,    0,    0, 1/14,   0, 1/16,    0, 1/18,    0]
```

Compute the number of nonzero elements in the resulting matrix.

```
Number = nnz(A)

Number =
    48
```

Find the density of this sparse matrix.

```
Density = nnz(A)/prod(size(A)

Density =
    0.4800
```

Input Arguments

x — Input array
symbolic vector | symbolic matrix | symbolic multidimensional array

Input array, specified as a symbolic vector, matrix, or multidimensional array.

See Also

`nonzeros` | `rank` | `reshape` | `size`

Introduced in R2014b

nonzeros

Nonzero elements

Syntax

```
nonzeros(X)
```

Description

`nonzeros(X)` returns a column vector containing all nonzero elements of `X`.

Examples

List All Nonzero Elements of Symbolic Matrix

Find all nonzero elements of a 10-by-10 symbolic matrix.

Create the following 5-by-5 symbolic Toeplitz matrix.

```
T = toeplitz(sym([0 2 3 4 0]))
```

```
T =
[ 0, 2, 3, 4, 0]
[ 2, 0, 2, 3, 4]
[ 3, 2, 0, 2, 3]
[ 4, 3, 2, 0, 2]
[ 0, 4, 3, 2, 0]
```

Use the `triu` function to return a triangular matrix that retains only the upper part of `T`.

```
T1 = triu(T)
```

```
T1 =
[ 0, 2, 3, 4, 0]
[ 0, 0, 2, 3, 4]
[ 0, 0, 0, 2, 3]
[ 0, 0, 0, 0, 2]
[ 0, 0, 0, 0, 0]
```

List all nonzero elements of this matrix. `nonzeros` searches for nonzero elements of a matrix in the first column, then in the second one, and so on. It returns the column vector containing all nonzero elements. It retains duplicate elements.

```
nonzeros(T1)
```

```
ans =
 2
 3
 2
 4
 3
 2
 4
 3
 2
```

Input Arguments

x — Input array

symbolic vector | symbolic matrix | symbolic multidimensional array

Input array, specified as a symbolic vector, matrix, or multidimensional array.

See Also

`nnz` | `rank` | `reshape` | `size`

Introduced in R2014b

norm

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`. Because symbolic variables are assumed to be complex by default, the norm can contain unresolved calls to `conj` and `abs`.

`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`.

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`

Examples

Compute the `2`-norm of the inverse of the 3-by-3 magic square `A`:

```
A = inv(sym(magic(3)))
norm2 = norm(A)

A =
[  53/360, -13/90,  23/360]
[ -11/180,   1/45,  19/180]
[  -7/360,  17/90, -37/360]

norm2 =
3^(1/2)/6
```

Use `vpa` to approximate the result with 20-digit accuracy:

```
vpa(norm2, 20)

ans =
0.28867513459481288225
```

Compute the norm of `[x y]` and simplify the result. Because symbolic variables are assumed to be complex by default, the calls to `abs` do not simplify.

```
syms x y
simplify(norm([x y]))
```

```
ans =
(abs(x)^2 + abs(y)^2)^(1/2)
```

Assume `x` and `y` are real, and repeat the calculation. Now, the result is simplified.

```
assume([x y],'real')
simplify(norm([x y]))
```

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

Remove assumptions on `x` for further calculations. For details, see “Use Assumptions on Symbolic Variables” on page 1-28.

```
assume(x,'clear')
```

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)
```

```
A =
[  53/360, -13/90,  23/360]
[ -11/180,   1/45,  19/180]
[  -7/360,  17/90, -37/360]

norm1 =
16/45

normf =
391^(1/2)/60

normi =
16/45
```

Use `vpa` to approximate these results to 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)
```

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:

$$\|A\|_\infty = \max \left(\sum_{j=1}^{n} |A_{1j}|, \sum_{j=1}^{n} |A_{2j}|, \ldots, \sum_{j=1}^{n} |A_{mj}| \right)$$

P-norm of a Vector

The `P`-norm of a 1-by-n or n-by-1 vector V is defined as follows:

$$\|V\|_P = \left(\sum_{i=1}^{n} |V_i|^P \right)^{1/P}$$

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:

$$\|V\|_F = \sqrt{\sum_{i=1}^{n} |V_i|^2}$$

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:

$$\|V\|_\infty = \max(|V_i|), \text{ where } i = 1 \ldots n$$

The negative infinity norm of a 1-by-n or n-by-1 vector V is defined as follows:

$$\|V\|_{-\infty} = \min(|V_i|), \text{ where } i = 1 \ldots n$$

Tips

- Calling `norm` for a numeric matrix that is not a symbolic object invokes the MATLAB `norm` function.

See Also

`cond` | `equationsToMatrix` | `inv` | `linsolve` | `rank`

Introduced in R2012b

not

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`.

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 =
~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 & ~0 < 1/3
x2 =
2/3 < 1 & ~2/3 < 1/3
```

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

```
logical(x1)
isAlways(x2)

ans =
  logical
     0

ans =
  logical
     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 =
  logical
     0

ans =
  logical
     1
```

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`.

See Also

`all` | `and` | `any` | `isAlways` | `or` | `piecewise` | `xor`

Introduced in R2012a

null

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
```

```
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)
```

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

Z =
Empty sym: 1-by-0
```

See Also

`rank` | `rref` | `size` | `svd`

Introduced before R2006a

numden

Extract numerator and denominator

Syntax

```
[N,D] = numden(A)
```

Description

`[N,D] = numden(A)` converts `A` to a rational form where the numerator and denominator are relatively prime polynomials with integer coefficients. The function returns the numerator and denominator of the rational form of an expression.

If `A` is a symbolic or a numeric matrix, then `N` is the symbolic matrix of numerators, and `D` is the symbolic matrix of denominators. Both `N` and `D` are matrices of the same size as `A`.

Examples

Numerators and Denominators of Symbolic Numbers

Find the numerator and denominator of a symbolic number.

```
[n, d] = numden(sym(4/5))
```

```
n =
4

d =
5
```

Numerators and Denominators of Symbolic Expressions

Find the numerator and denominator of the symbolic expression.

```
syms x y
[n,d] = numden(x/y + y/x)

n =
x^2 + y^2

d =
x*y
```

Numerators and Denominators of Matrix Elements

Find the numerator and denominator of each element of a symbolic matrix.

```
syms a b
[n,d] = numden([a/b, 1/b; 1/a, 1/(a*b)])

n =
[ a, 1]
[ 1, 1]

d =
[ b,   b]
[ a, a*b]
```

Input Arguments

`A` — Input

symbolic number | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, expression, function, vector, or matrix.

Output Arguments

`N` — Numerator

symbolic number | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Numerator, returned as a symbolic number, expression, function, vector, or matrix.

`D` — Denominator
symbolic number | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Denominator, returned as a symbolic number, expression, function, vector, or matrix.

See Also
`divisors` | `partfrac` | `simplifyFraction`

Topics
"Extract Numerators and Denominators of Rational Expressions" on page 2-105

Introduced before R2006a

numel

Number of elements of symbolic array

Syntax

```
numel(A)
```

Description

`numel(A)` returns the number of elements in symbolic array `A`, equal to `prod(size(A))`.

Examples

Number of Elements in Vector

Find the number of elements in vector `V`.

```
syms x y
V = [x y 3];
numel(V)
```

```
ans =
     3
```

Number of Elements in 3-D Array

Create a 3-D symbolic array and find the number of elements in it.

Create the 3-D symbolic array `A`:

```
A = sym(magic(3));
A(:,:,2) = A'
```

```
A(:,:,1) =
[ 8, 1, 6]
[ 3, 5, 7]
[ 4, 9, 2]

A(:,:,2) =
[ 8, 3, 4]
[ 1, 5, 9]
[ 6, 7, 2]
```

Use `numel` to count the number of elements in `A`.

```
numel(A)
```

```
ans =
    18
```

Input Arguments

`A` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array

Input, specified as a symbolic variable, vector, matrix, or multidimensional array.

See Also

`prod` | `size`

Introduced in R2008b

odeFunction

Convert symbolic expressions to function handle for ODE solvers

Syntax

```
f = odeFunction(expr,vars)
f = odeFunction(expr,vars,p1,...,pN)
f = odeFunction(___,Name,Value)
```

Description

`f = odeFunction(expr,vars)` converts a system of symbolic algebraic expressions to a MATLAB function handle. This function handle can be used as input to the numerical MATLAB ODE solvers, except for `ode15i`. The argument `vars` specifies the state variables of the system.

`f = odeFunction(expr,vars,p1,...,pN)` specifies the symbolic parameters of the system as `p1,...,pN`.

`f = odeFunction(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Function Handle for ODE Solvers and Solve DAEs

Convert a system of symbolic differential algebraic equations to a function handle suitable for the MATLAB ODE solvers. Then solve the system by using the `ode15s` solver.

Create the following second-order differential algebraic equation.

```
syms y(t);
eqn = diff(y(t),t,2) == (1-y(t)^2)*diff(y(t),t) - y(t);
```

Use `reduceDifferentialOrder` to rewrite that equation as a system of two first-order differential equations. Here, `vars` is a vector of state variables of the system. The new variable `Dy(t)` represents the first derivative of `y(t)` with respect to `t`.

```
[eqs,vars] = reduceDifferentialOrder(eqn,y(t))
```

```
eqs =
 diff(Dyt(t), t) + y(t) + Dyt(t)*(y(t)^2 - 1)
                       Dyt(t) - diff(y(t), t)

vars =
   y(t)
 Dyt(t)
```

Set initial conditions for `y(t)` and its derivative `Dy(t)` to `2` and `0` respectively.

```
initConditions = [2 0];
```

Find the mass matrix `M` of the system and the right sides of the equations `F`.

```
[M,F] = massMatrixForm(eqs,vars)
```

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

F =
 - y(t) - Dyt(t)*(y(t)^2 - 1)
                      -Dyt(t)
```

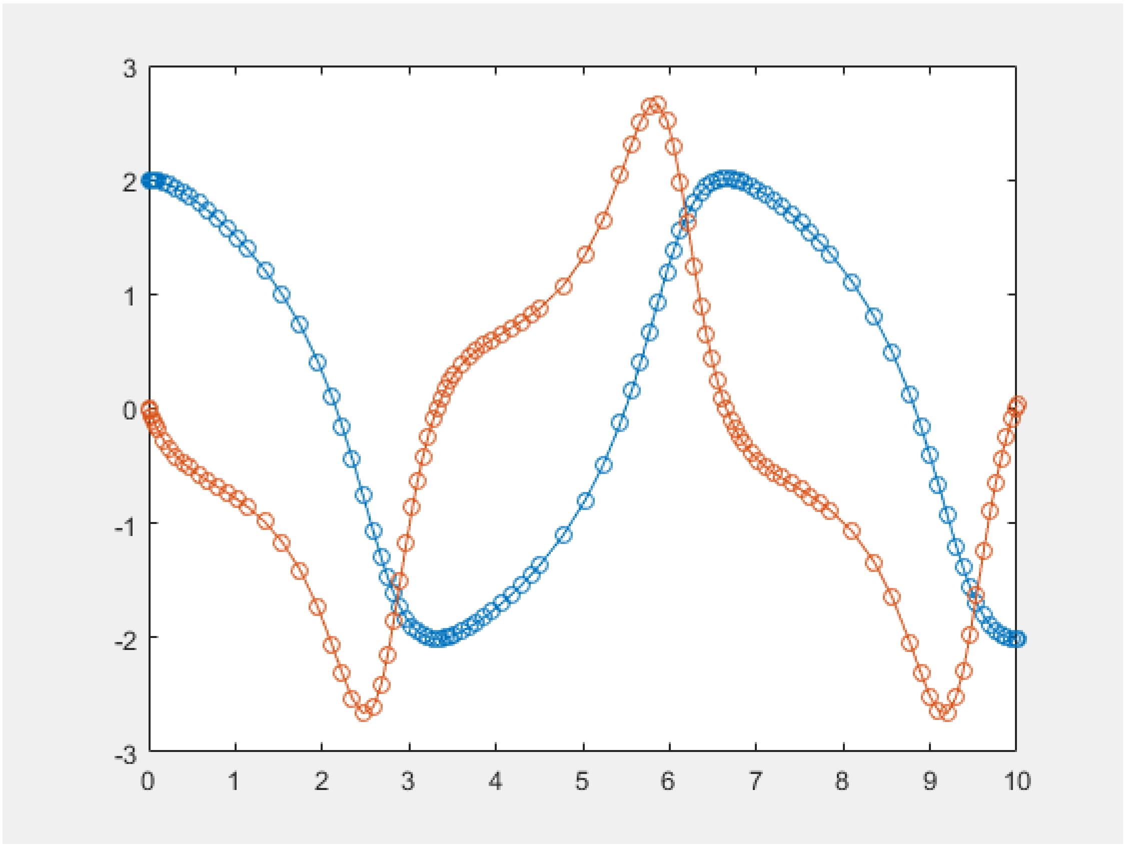
`M` and `F` refer to the form $\boldsymbol{M}\left(t,x(t)\right)\dot{x}(t)=\boldsymbol{F}\left(t,x(t)\right)$.. To simplify further computations, rewrite the system in the form $\dot{x}(t)=f\left(t,x(t)\right)$.

```
f = M\F
```

```
f =
                        Dyt(t)
 Dyt(t) - y(t) - Dyt(t)*y(t)^2
```

Convert `f` to a MATLAB function handle by using `odeFunction`. The resulting function handle is input to the MATLAB ODE solver `ode15s`.

```
odefun = odeFunction(f,vars);
ode15s(odefun, [0 10], initConditions)
```



Function Handles for System Containing Symbolic Parameters

Convert a system of symbolic differential equations containing both state variables and symbolic parameters to a function handle suitable for the MATLAB ODE solvers.

Create the system of differential algebraic equations. Here, the symbolic functions `x1(t)` and `x2(t)` represent the state variables of the system. The system also contains constant symbolic parameters `a`, `b`, and the parameter function `r(t)`. These parameters do not represent state variables. Specify the equations and state variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x1(t) x2(t) a b r(t)
eqs = [diff(x1(t),t) == a*x1(t) + b*x2(t)^2,...
       x1(t)^2 + x2(t)^2 == r(t)^2];
vars = [x1(t) x2(t)];
```

Find the mass matrix `M` and vector of the right side `F` for this system. `M` and `F` refer to the form $M(t,x(t))\dot{x}(t) = F(t,x(t))$..

```
[M,F] = massMatrixForm(eqs,vars)
```

```
M =
[ 1, 0]
[ 0, 0]

F =
        b*x2(t)^2 + a*x1(t)
 r(t)^2 - x1(t)^2 - x2(t)^2
```

Use `odeFunction` to generate MATLAB function handles from `M` and `F`. The function handle `F` contains symbolic parameters.

```
M = odeFunction(M,vars)
F = odeFunction(F,vars,a,b,r(t))
```

```
M =
  function_handle with value:
    @(t,in2)reshape([1.0,0.0,0.0,0.0],[2,2])

F =
  function_handle with value:
    @(t,in2,param1,param2,param3)[param1.*in2(1,:)+...
    param2.*in2(2,:).^2;param3.^2-in2(1,:).^2-in2(2,:).^2]
```

Specify the parameter values.

```
a = -0.6;
b = -0.1;
r = @(t) cos(t)/(1+t^2);
```

Create the reduced function handle `F`.

```
F = @(t,Y) F(t,Y,a,b,r(t));
```

Specify consistent initial conditions for the DAE system.

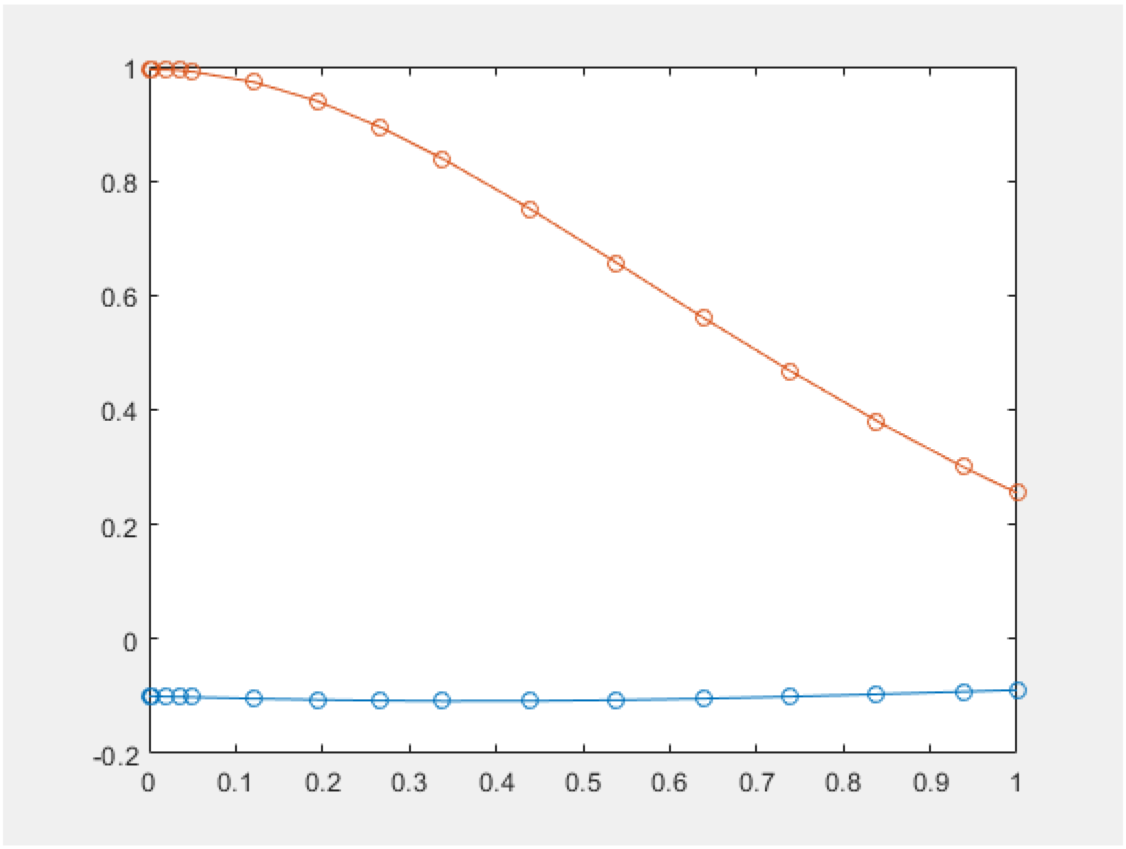
```
t0 = 0;
y0 = [-r(t0)*sin(0.1); r(t0)*cos(0.1)];
yp0 = [a*y0(1) + b*y0(2)^2; 1.234];
```

Create an option set that contains the mass matrix `M` of the system and vector `yp0` of initial conditions for the derivatives.

```
opt = odeset('mass',M,'InitialSlope',yp0);
```

Now, use `ode15s` to solve the system of equations.

```
ode15s(F, [t0, 1], y0, opt)
```



Write Function Handles to File with Comments

Write the generated function handles to files by using the `File` option. When writing to files, `odeFunction` optimizes the code using intermediate variables named `t0`, `t1`, ... Include comments the files by specifying the `Comments` option.

Define the system of differential equations. Find the mass matrix `M` and the right side `F`.

```
syms x(t) y(t)
eqs = [diff(x(t),t)+2*diff(y(t),t) == 0.1*y(t), ...
        x(t)-y(t) == cos(t)-0.2*t*sin(x(t))];
```

```
vars = [x(t) y(t)];
[M,F] = massMatrixForm(eqs,vars);
```

Write the MATLAB code for `M` and `F` to the files `myfileM` and `myfileF`. `odeFunction` overwrites existing files. Include the comment `Version: 1.1` in the files You can open and edit the output files.

```
M = odeFunction(M,vars,'File','myfileM','Comments','Version: 1.1');
```

```
function expr = myfileM(t,in2)
%MYFILEM
%    EXPR = MYFILEM(T,IN2)

%    This function was generated by the Symbolic Math Toolbox version 7.3.
%    01-Jan-2017 00:00:00

%Version: 1.1
expr = reshape([1.0,0.0,2.0,0.0],[2, 2]);
```

```
F = odeFunction(F,vars,'File','myfileF','Comments','Version: 1.1');
```

```
function expr = myfileF(t,in2)
%MYFILEF
%    EXPR = MYFILEF(T,IN2)

%    This function was generated by the Symbolic Math Toolbox version 7.3.
%    01-Jan-2017 00:00:00

%Version: 1.1
x = in2(1,:);
y = in2(2,:);
expr = [y.*(1.0./1.0e1);-x+y+cos(t)-t.*sin(x).*(1.0./5.0)];
```

Specify consistent initial values for `x(t)` and `y(t)` and their first derivatives.

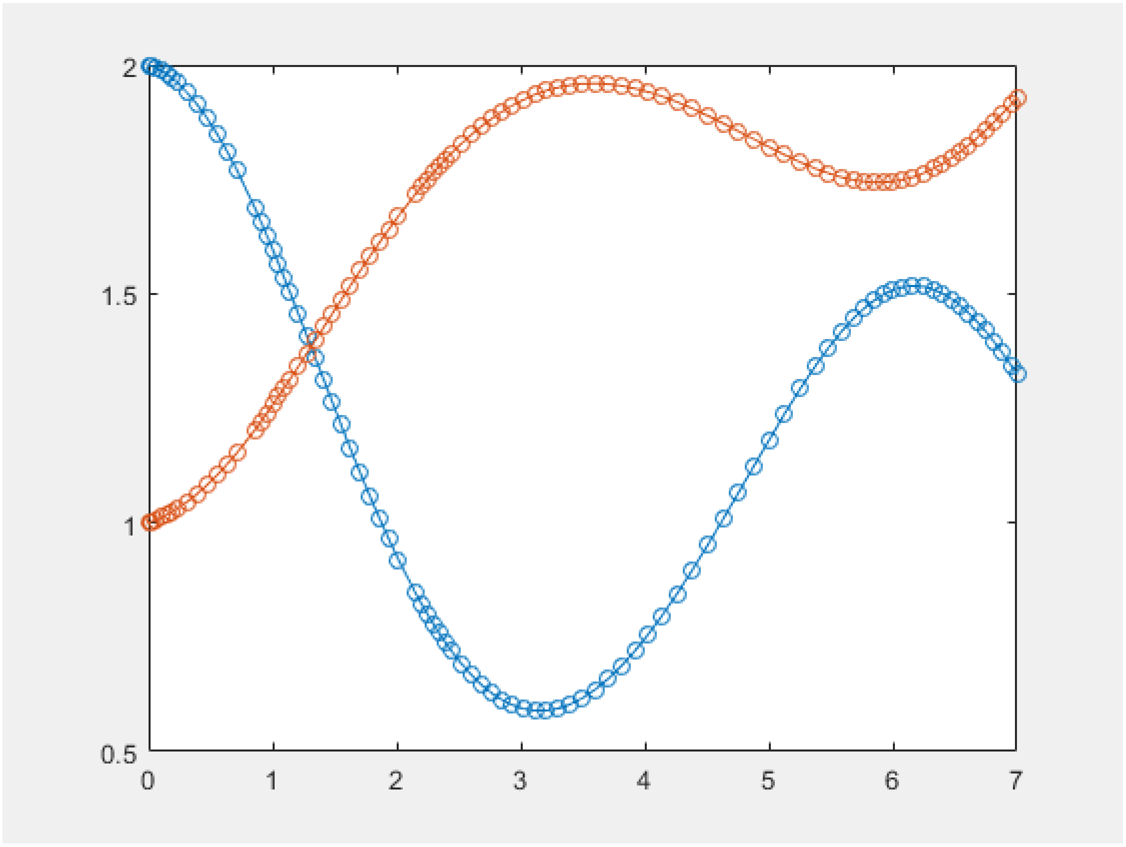
```
xy0 = [2; 1];    % x(t) and y(t)
xyp0 = [0; 0.05*xy0(2)];    % derivatives of x(t) and y(t)
```

Create an option set that contains the mass matrix `M`, initial conditions `xyp0`, and numerical tolerances for the numerical search.

```
opt = odeset('mass', M, 'RelTol', 10^(-6),...
                'AbsTol', 10^(-6), 'InitialSlope', xyp0);
```

Solve the system of equations by using `ode15s`.

```
ode15s(F, [0 7], xy0, opt)
```



Sparse Matrices

Use the name-value pair argument `'Sparse',true` when converting sparse symbolic matrices to MATLAB function handles.

Create the system of differential algebraic equations. Here, the symbolic functions `x1(t)` and `x2(t)` represent the state variables of the system. Specify the equations and state variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x1(t) x2(t)

a = -0.6;
b = -0.1;
r = @(t) cos(t)/(1 + t^2);

eqs = [diff(x1(t),t) == a*x1(t) + b*x2(t)^2,...
       x1(t)^2 + x2(t)^2 == r(t)^2];
vars = [x1(t) x2(t)];
```

Find the mass matrix `M` and vector of the right side `F` for this system. `M` and `F` refer to the form $M(t,x(t))\dot{x}(t) = F(t,x(t))$..

```
[M,F] = massMatrixForm(eqs,vars)

M =
[ 1, 0]
[ 0, 0]

F =
               - (3*x1(t))/5 - x2(t)^2/10
 cos(t)^2/(t^2 + 1)^2 - x1(t)^2 - x2(t)^2
```

Generate MATLAB function handles from `M` and `F`. Because most of the elements of the mass matrix `M` are zeros, use the `Sparse` argument when converting `M`.

```
M = odeFunction(M,vars,'Sparse',true)
F = odeFunction(F,vars)

M =
  function_handle with value:
    @(t,in2)sparse([1],[1],[1.0],2,2)

F =
  function_handle with value:
    @(t,in2)[in2(1,:).*(-3.0./5.0)-in2(2,:).^2.*(1.0./1.0e1);...
    cos(t).^2.*1.0./(t.^2+1.0).^2-in2(1,:).^2-in2(2,:).^2]
```

Specify consistent initial conditions for the DAE system.

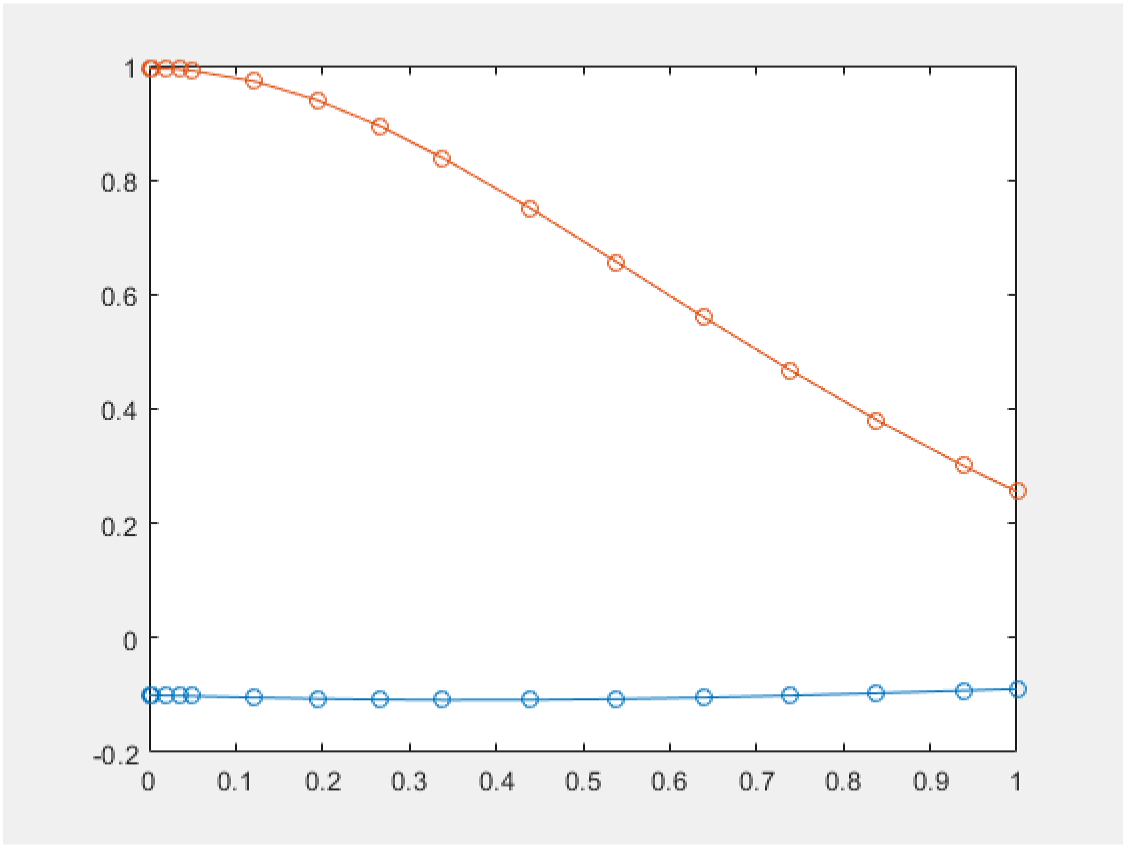
```
t0 = 0;
y0 = [-r(t0)*sin(0.1); r(t0)*cos(0.1)];
yp0= [a*y0(1) + b*y0(2)^2; 1.234];
```

Create an option set that contains the mass matrix `M` of the system and vector `yp0` of initial conditions for the derivatives.

```
opt = odeset('mass',M,'InitialSlope', yp0);
```

Solve the system of equations using `ode15s`.

```
ode15s(F, [t0, 1], y0, opt)
```



- "Solve DAEs Using Mass Matrix Solvers" on page 2-213

Input Arguments

`expr` — System of algebraic expressions

vector of symbolic expressions

System of algebraic expressions, specified as a vector of symbolic expressions.

`vars` — State variables

vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

`p1,...,pN` — Parameters of system

symbolic variables | symbolic functions | symbolic function calls | symbolic vector | symbolic matrix

Parameters of the system, specified as symbolic variables, functions, or function calls, such as `f(t)`. You can also specify parameters of the system as a vector or matrix of symbolic variables, functions, or function calls. If `expr` contains symbolic parameters other than the variables specified in `vars`, you must specify these additional parameters as `p1,...,pN`.

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`.

Example: `odeFunction(expr,vars,'File','myfile')`

`Comments` — Comments to include in file header

character vector | cell array of character vectors | string vector

Comments to include in the file header, specified as a character vector, cell array of character vectors, or string vector.

`File` — Path to file containing generated code

character vector

Path to the file containing generated code, specified as a character vector. The generated file accepts arguments of type `double`, and can be used without Symbolic Math Toolbox. If the value is empty, `odeFunction` generates an anonymous function. If the character vector does not end in `.m`, the function appends `.m`.

By default, `odeFunction` with the `File` argument generates a file containing optimized code. 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`. To disable code optimization, use the `Optimize` argument.

`Optimize` — Flag preventing optimization of code written to function file
`true` (default) | `false`

Flag preventing optimization of code written to a function file, specified as `false` or `true`.

By default, `odeFunction` with the `File` argument generates a file containing optimized code. 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`.

`odeFunction` without the `File` argument (or with a file path specified by an empty character vector) creates a function handle. In this case, the code is not optimized. If you try to enforce code optimization by setting `Optimize` to `true`, then `odeFunction` throws an error.

`Sparse` — Flag that switches between sparse and dense matrix generation
`false` (default) | `true`

Flag that switches between sparse and dense matrix generation, specified as `true` or `false`. When you specify `'Sparse',true`, the generated function represents symbolic matrices by sparse numeric matrices. Use `'Sparse',true` when you convert symbolic matrices containing many zero elements. Often, operations on sparse matrices are more efficient than the same operations on dense matrices. See “Sparse Matrices” on page 4-1237.

Output Arguments

`f` — Function handle that is input to numerical MATLAB ODE solvers, except `ode15i`
MATLAB function handle

Function handle that can serve as input argument to all numerical MATLAB ODE solvers, except for `ode15i`, returned as a MATLAB function handle.

`odeFunction` returns a function handle suitable for the ODE solvers such as `ode45`, `ode15s`, `ode23t`, and others. The only ODE solver that does not accept this function handle is the solver for fully implicit differential equations, `ode15i`. To convert the system of equations to a function handle suitable for `ode15i`, use `daeFunction`.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `matlabFunction` | `ode15i` | `ode15s` | `ode23t` | `ode45` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

"Solve DAEs Using Mass Matrix Solvers" on page 2-213

Introduced in R2015a

odeToVectorField

Reduce order of differential equations to 1

Note Character vector inputs will be removed in a future release. Instead, use `syms` to declare variables, and replace inputs such as `odeToVectorField('D2y = x')` with `syms y(x), odeToVectorField(diff(y,x,2) == x)`.

Syntax

```
V = odeToVectorField(eqn1,...,eqnN)
[V,S] = odeToVectorField(eqn1,...,eqnN)
```

Description

`V = odeToVectorField(eqn1,...,eqnN)` converts higher-order differential equations `eqn1,...,eqnN` to a system of first-order differential equations, returned as a symbolic vector.

`[V,S] = odeToVectorField(eqn1,...,eqnN)` converts `eqn1,...,eqnN` and returns two symbolic vectors. The first vector `V` is the same as the output of the previous syntax. The second vector `S` shows the substitutions made to obtain `V`.

Examples

Convert Higher-Order Equation to First-Order System

Convert this second-order differential equation to a system of first-order differential equations.

$$\frac{d^2y}{dt^2}+y^2t=3t.$$

```
syms y(t)
eqn = diff(y,2) + y^2*t == 3*t;
V = odeToVectorField(eqn)

V =
           Y[2]
 3*t - t*Y[1]^2
```

The elements of `V` represent the system of differential equations because $y = Y[1]$ and $Y[i]$ $' = V[i]$. Here, this particular output represents these equations:

- `diff(Y[1],t) = Y[2]`
- `diff(Y[2],t) = 3*t - t*Y[1]^2`

For details on the relation between the input and output, see “Algorithms” on page 4-1248.

Return Substitutions Made When Reducing Order

When reducing the order of differential equations, return the substitutions that `odeToVectorField` makes by specifying a second output argument.

```
syms f(t) g(t)
eqn1 = diff(g) == g-f;
eqn2 = diff(f,2) == g+f;
eqns = [eqn1 eqn2];
[V,S] = odeToVectorField(eqns)

V =
        Y[2]
 Y[1] + Y[3]
 Y[3] - Y[1]

S =
  f
 Df
  g
```

From `S`, we have `S[1] = Y[1] = f`, `S[2] = Y[2] = diff(f)`, and `S[3] = Y[3] = g`.

Numerically Solve Higher-Order Differential Equation

Numerically solve a higher-order differential equation by reducing the order of the equation, generating a MATLAB function handle, and then finding the numerical solution using the `ode45` function.

Convert this second-order differential equation to a system of first-order differential equations.

$$\frac{dy^2}{dx^2} = \left(1 - y^2\right)\frac{dy}{dx} - y.$$

```
syms y(t)
eqn = diff(y,2) == (1-y^2)*diff(y)-y;
V = odeToVectorField(eqn)
```

```
V =
                          Y[2]
 - (Y[1]^2 - 1)*Y[2] - Y[1]
```

Generate a MATLAB function handle from `V` by using `matlabFunction`.

```
M = matlabFunction(V,'vars', {'t','Y'})
```

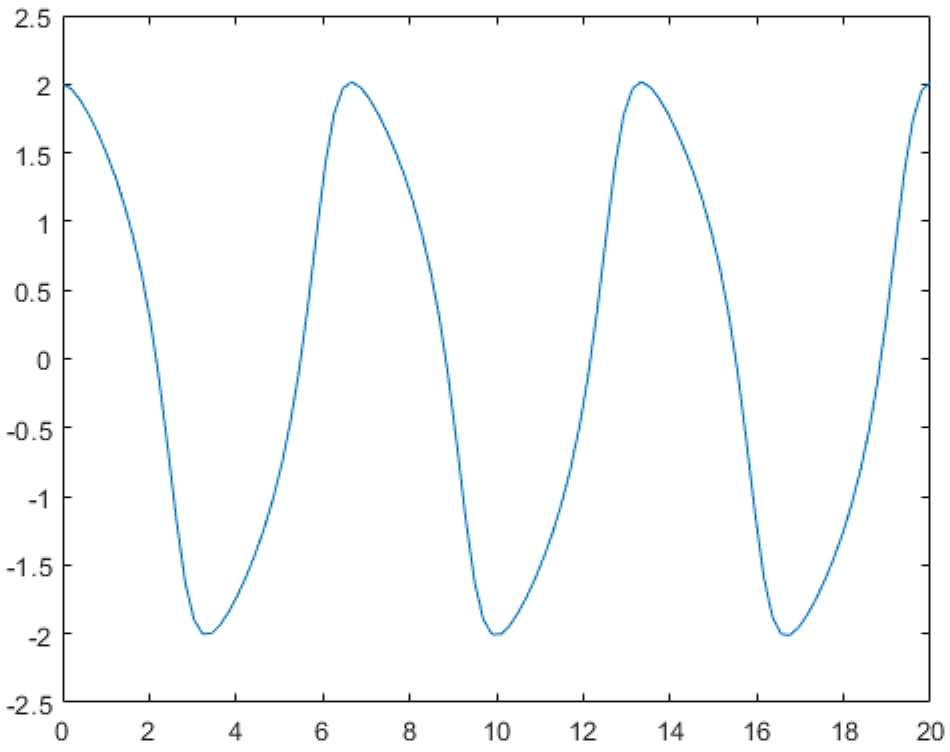
```
M =
  function_handle with value:
    @(t,Y)[Y(2);-(Y(1).^2-1.0).*Y(2)-Y(1)]
```

Solve this system over the interval `[0 20]` with initial conditions y’(0) = 2 and y”(0) = 0 by using the `ode45` function.

```
interval = [0 20];
y0 = [2 0];
ySol = ode45(M,interval,y0);
```

Generate values of `t` in the interval by using the `linspace` function. For these values, evaluate the solution for `y`, which is the first index in `ySol`, by calling the `deval` function with an index of `1`. Plot the solution.

```
tValues = linspace(0,20,100);
yValues = deval(ySol,tValues,1);
plot(tValues,yValues)
```



Convert Higher-Order System with Initial Condition

Convert the second-order differential equation $y''(x) = x$ with the initial condition $y(0) = a$ to a first-order system.

```
syms y(x) a
eqn = diff(y,x,2) == x;
cond = y(0) == a;
V = odeToVectorField(eqn,cond)
```

```
V =
 Y[2]
    x
```

Input Arguments

`eqn1,...,eqnN` — Higher-order differential equations

symbolic differential equation | array of symbolic differential equations | comma-separated list of symbolic differential equations

Higher-order differential equations, specified as a symbolic differential equation or an array or comma-separated list of symbolic differential equations. Use the == operator to create an equation. Use the `diff` function to indicate differentiation. For example, represent $d^2y(t)/dt^2 = t*y(t)$.

```
syms y(t)
eqn = diff(y,2) == t*y;
```

Output Arguments

`V` — First-order differential equations

symbolic expression | vector of symbolic expressions

First-order differential equations, returned as a symbolic expression or a vector of symbolic expressions. Each element of this vector is the right side of the first-order differential equation $Y[i]' = V[i]$.

`S` — Substitutions in first-order equations

vector of symbolic expressions

Substitutions in first-order equations, returned as a vector of symbolic expressions. The elements of the vector represent the substitutions, such that `S(1) = Y[1]`, `S(2) = Y[2]`,….

Tips

- To solve the resulting system of first-order differential equations, generate a MATLAB function handle using `matlabFunction` with `V` as an input. Then, use the generated MATLAB function handle as an input for the MATLAB numerical solver `ode23` or `ode45`.
- `odeToVectorField` can convert only quasi-linear differential equations. That is, the highest-order derivatives must appear linearly. For example, `odeToVectorField`

can convert $y*y''(t) = -t^2$ because it can be rewritten as $y''(t) = -t^2/y$. However, it cannot convert $y''(t)^2 = -t^2$ or $\sin(y''(t)) = -t^2$.

Algorithms

To convert an nth-order differential equation

$$a_n(t)y^{(n)} + a_{n-1}(t)y^{(n-1)} + \ldots + a_1(t)y' + a_0(t)y + r(t) = 0$$

into a system of first-order differential equations, `odetovectorfield` makes these substitutions.

$$Y_1 = y$$
$$Y_2 = y'$$
$$Y_3 = y''$$
$$\ldots$$
$$Y_{n-1} = y^{(n-2)}$$
$$Y_n = y^{(n-1)}$$

Using the new variables, it rewrites the equation as a system of n first-order differential equations:

$$Y_1{}' = y' = Y_2$$
$$Y_2{}' = y'' = Y_3$$
$$\ldots$$
$$Y_{n-1}{}' = y^{(n-1)} = Y_n$$
$$Y_n{}' = -\frac{a_{n-1}(t)}{a_n(t)}Y_n - \frac{a_{n-2}(t)}{a_n(t)}Y_{n-1} - \ldots - \frac{a_1(t)}{a_n(t)}Y_2 - \frac{a_0(t)}{a_n(t)}Y_1 + \frac{r(t)}{a_n(t)}$$

`odeToVectorField` returns the right sides of these equations as the elements of vector `V` and the substitutions made as the second output `S`.

See Also

`dsolve` | `matlabFunction` | `ode23` | `ode45`

Introduced in R2012a

openmn

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 file name must be a full path unless the file is in the current folder. 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`

Topics

“Create MuPAD Notebooks” on page 3-3
“Open MuPAD Notebooks” on page 3-6

Introduced in R2008b

openmu

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`

Topics

"Open MuPAD Notebooks" on page 3-6

Introduced in R2008b

openxvc

Open MuPAD uncompressed graphics file (XVC)

Syntax

`openxvc(file)`

Description

`openxvc(file)` opens the MuPAD XVC graphics file named `file`. The file name must be a full path unless the file is in the current folder.

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`

Topics

“Open MuPAD Notebooks” on page 3-6

Introduced in R2008b

openxvz

Open MuPAD compressed graphics file (XVZ)

Syntax

```
openxvz(file)
```

Description

`openxvz(file)` opens the MuPAD XVZ graphics file named `file`. The file name must be a full path unless the file is in the current folder.

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`

Topics

"Open MuPAD Notebooks" on page 3-6

Introduced in R2008b

or

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`.

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 `|`:

```
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 | 10 < -1
x2 =
0 < -1 | 1 < 0
```

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

```
isAlways(x1)
isAlways(x2)
```

```
ans =
  logical
     1

ans =
  logical
     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 `isAlways`:

```
isAlways(s1)
isAlways(s2)

ans =
  logical
     1

ans =
  logical
     0
```

Combine multiple conditions by applying `or` to the conditions using the `fold` function.

```
syms x
cond = fold(@or, x == 1:10);
assume(cond)
assumptions

ans =
x == 1 | x == 2 | x == 3 | x == 4 | x == 5 |...
 x == 6 | x == 7 | x == 8 | x == 9 | x == 10
```

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 `isAlways`.

See Also

`all` | `and` | `any` | `isAlways` | `not` | `piecewise` | `xor`

Introduced in R2012a

orth

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 on page 4-1263 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.

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.

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

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`.

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.

Algorithms

`orth` uses the classic Gram-Schmidt orthogonalization algorithm.

See Also

`norm` | `null` | `orth` | `rank` | `svd`

Introduced in R2013a

pade

Pade approximant

Syntax

```
pade(f,var)
pade(f,var,a)
pade(___,Name,Value)
```

Description

`pade(f,var)` returns the third-order Padé approximant of the expression `f` at `var = 0`. For details, see “Padé Approximant” on page 4-1271.

If you do not specify `var`, then `pade` uses the default variable determined by `symvar(f, 1)`.

`pade(f,var,a)` returns the third-order Padé approximant of expression `f` at the point `var = a`.

`pade(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. You can specify `Name,Value` after the input arguments in any of the previous syntaxes.

Examples

Find Padé Approximant for Symbolic Expressions

Find the Padé approximant of `sin(x)`. By default, `pade` returns a third-order Padé approximant.

```
syms x
pade(sin(x))
```

```
ans =
-(x*(7*x^2 - 60))/(3*(x^2 + 20))
```

Specify Expansion Variable

If you do not specify the expansion variable, `symvar` selects it. Find the Padé approximant of `sin(x) + cos(y)`. The `symvar` function chooses `x` as the expansion variable.

```
syms x y
pade(sin(x) + cos(y))
```

```
ans =
(- 7*x^3 + 3*cos(y)*x^2 + 60*x + 60*cos(y))/(3*(x^2 + 20))
```

Specify the expansion variable as `y`. The `pade` function returns the Padé approximant with respect to `y`.

```
pade(sin(x) + cos(y),y)
```

```
ans =
(12*sin(x) + y^2*sin(x) - 5*y^2 + 12)/(y^2 + 12)
```

Approximate Value of Function at Particular Point

Find the value of `tan(3*pi/4)`. Use `pade` to find the Padé approximant for `tan(x)` and substitute into it using `subs` to find `tan(3*pi/4)`.

```
syms x
f = tan(x);
P = pade(f);
y = subs(P,x,3*pi/4)
```

```
y =
(pi*((9*pi^2)/16 - 15))/(4*((9*pi^2)/8 - 5))
```

Use `vpa` to convert `y` into a numeric value.

```
vpa(y)
```

```
ans =
-1.2158518789569086447244881326842
```

Increase Accuracy of Padé Approximant

You can increase the accuracy of the Padé approximant by increasing the order. If the expansion point is a pole or a zero, the accuracy can also be increased by setting `OrderMode` to `relative`. The `OrderMode` option has no effect if the expansion point is not a pole or zero.

Find the Padé approximant of `tan(x)` using `pade` with an expansion point of `0` and `Order` of `[1 1]`. Find the value of `tan(1/5)` by substituting into the Padé approximant using `subs`, and use `vpa` to convert `1/5` into a numeric value.

```
syms x
p11 = pade(tan(x),x,0,'Order',[1 1])
p11 = subs(p11,x,vpa(1/5))
```

```
p11 =
x
p11 =
0.2
```

Find the approximation error by subtracting `p11` from the actual value of `tan(1/5)`.

```
y = tan(vpa(1/5));
error = y - p11
```

```
error =
0.0027100355086724833213582716475345
```

Increase the accuracy of the Padé approximant by increasing the order using `Order`. Set `Order` to `[2 2]`, and find the error.

```
p22 = pade(tan(x),x,0,'Order',[2 2])
p22 = subs(p22,x,vpa(1/5));
error = y - p22
```

```
p22 =
-(3*x)/(x^2 - 3)
error =
0.0000073328059697806186555689448317799
```

The accuracy increases with increasing order.

If the expansion point is a pole or zero, the accuracy of the Padé approximant decreases. Setting the `OrderMode` option to `relative` compensates for the decreased accuracy. For details, see “Padé Approximant” on page 4-1271. Because the `tan` function has a zero at `0`, setting `OrderMode` to `relative` increases accuracy. This option has no effect if the expansion point is not a pole or zero.

```
p22Rel = pade(tan(x),x,0,'Order',[2 2],'OrderMode','relative')
p22Rel = subs(p22Rel,x,vpa(1/5));
error = y - p22Rel
```

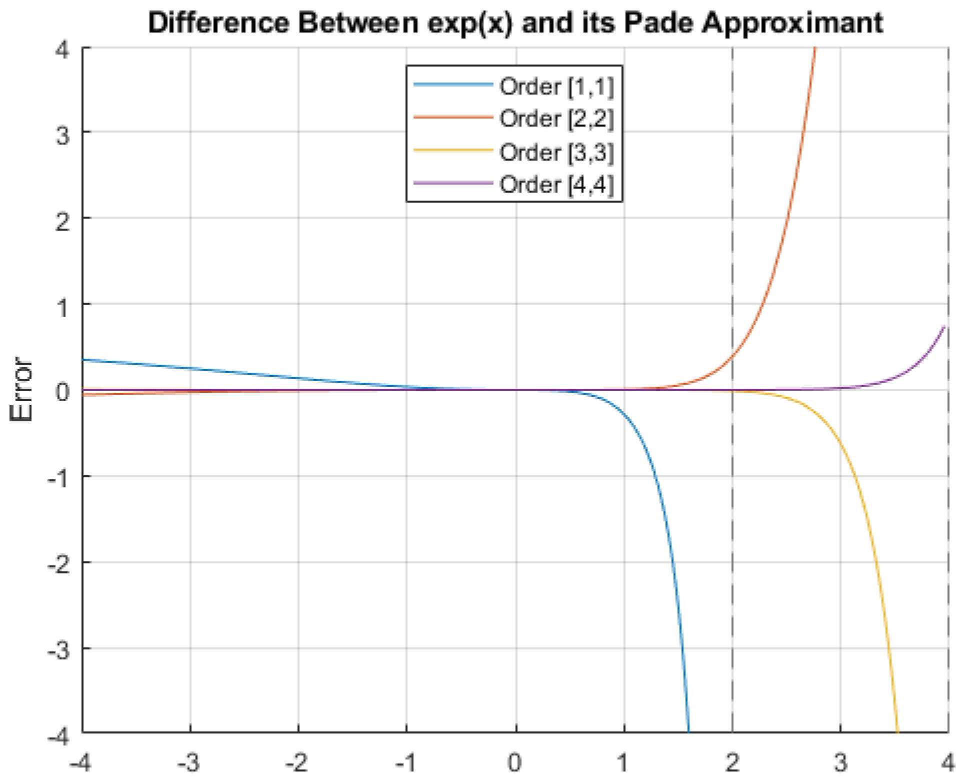
```
p22Rel =
(x*(x^2 - 15))/(3*(2*x^2 - 5))
error =
0.0000000084084014806113311713765317725998
```

The accuracy increases if the expansion point is a pole or zero and `OrderMode` is set to `relative`.

Plot Accuracy of Pade Approximant

Plot the difference between `exp(x)` and its Pade approximants of orders `[1 1]` through `[4 4]`. Use `axis` to focus on the region of interest. The plot shows that accuracy increases with increasing order of the Pade approximant. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
expr = exp(x);

hold on
grid on

for i = 1:4
    fplot(expr - pade(expr,'Order',i))
end

axis([-4 4 -4 4])
legend('Order [1,1]','Order [2,2]','Order [3,3]','Order [4,4]',...
                                            'Location','Best')
title('Difference Between exp(x) and its Pade Approximant')
ylabel('Error')
```



Input Arguments

`f` — Input to approximate

symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input to approximate, specified as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`var` — Expansion variable

symbolic variable

Expansion variable, specified as a symbolic variable. If you do not specify `var`, then `pade` uses the default variable determined by `symvar(f,1)`.

`a` — Expansion point
number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable. You also can specify the expansion point as a `Name,Value` pair argument. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

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`.

Example: `pade(f,'Order',[2 2])` returns the Padé approximant of `f` of order `m = 2` and `n = 2`.

`ExpansionPoint` — Expansion point
number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable. You can also specify the expansion point using the input argument `a`. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

`Order` — Order of Padé approximant
integer | vector of two integers | symbolic integer | symbolic vector of two integers

Order of the Padé approximant, specified as an integer, a vector of two integers, or a symbolic integer, or vector of two integers. If you specify a single integer, then the integer specifies both the numerator order m and denominator order n producing a Padé approximant with $m = n$. If you specify a vector of two integers, then the first integer specifies m and the second integer specifies n. By default, `pade` returns a Padé approximant with $m = n = 3$.

`OrderMode` — Flag that selects absolute or relative order for Padé approximant
character vector

Flag that selects absolute or relative order for Padé approximant, specified as a character vector. The default value of `absolute` uses the standard definition of the Padé approximant. If you set `OrderMode` to `relative`, it only has an effect when there is a pole or a zero at the expansion point `a`. In this case, to increase accuracy, `pade` multiplies the numerator by `(var - a)`$^{\mathtt{p}}$ where `p` is the multiplicity of the zero or pole at the expansion point. For details, see "Padé Approximant" on page 4-1271.

Definitions

Padé Approximant

By default, `pade` approximates the function *f*(*x*) using the standard form of the Padé approximant of order [*m*, *n*] around $x = x_0$ which is

$$\frac{a_0 + a_1(x - x_0) + \ldots + a_m(x - x_0)^m}{1 + b_1(x - x_0) + \ldots + b_n(x - x_0)^n}.$$

When `OrderMode` is `relative`, and a pole or zero exists at the expansion point $x = x_0$, the `pade` function uses this form of the Padé approximant

$$\frac{(x - x_0)^p\left(a_0 + a_1(x - x_0) + \ldots + a_m(x - x_0)^m\right)}{1 + b_1(x - x_0) + \ldots + b_n(x - x_0)^n}.$$

The parameters *p* and a_0 are given by the leading order term $f = a_0 (x - x_0)^p + O((x - x_0)^{p+1})$ of the series expansion of *f* around $x = x_0$. Thus, *p* is the multiplicity of the pole or zero at x_0.

Tips

- If you use both the third argument `a` and `ExpansionPoint` to specify the expansion point, the value specified via `ExpansionPoint` prevails.

Algorithms

- The parameters $a_1,\ldots,b_n$ are chosen such that the series expansion of the Pade approximant coincides with the series expansion of *f* to the maximal possible order.
- The expansion points ±∞ and ±i∞ are not allowed.
- When `pade` cannot find the Padé approximant, it returns the function call.
- For `pade` to return the Padé approximant, a Taylor or Laurent series expansion of *f* must exist at the expansion point.

See Also

`series` | `taylor`

Topics

"Padé Approximant" on page 2-70

Introduced in R2014b

partfrac

Partial fraction decomposition

Syntax

```
partfrac(expr,var)
partfrac(expr,var,Name,Value)
```

Description

`partfrac(expr,var)` finds the partial fraction decomposition of `expr` with respect to `var`. If you do not specify `var`, then `partfrac` uses the variable determined by `symvar`.

`partfrac(expr,var,Name,Value)` finds the partial fraction decomposition using additional options specified by one or more `Name,Value` pair arguments.

Examples

Partial Fraction Decomposition

Find partial fraction decomposition of univariate and multivariate expressions.

First, find partial fraction decomposition of univariate expressions. For expressions with one variable, you can omit specifying the variable.

```
syms x
partfrac(x^2/(x^3 - 3*x + 2))
```

```
ans =
5/(9*(x - 1)) + 1/(3*(x - 1)^2) + 4/(9*(x + 2))
```

For some expressions, `partfrac` returns visibly simpler forms.

```
partfrac((x^6 + 15*x^5 + 94*x^4 + 316*x^3 + 599*x^2 + 602*x + 247)/...
(x^6 + 14*x^5 + 80*x^4 + 238*x^3 + 387*x^2 + 324*x + 108))
```

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

Next, find partial fraction decomposition of a multivariate expression with respect to a particular variable.

```
syms a b
partfrac(a^2/(a^2 - b^2),a)
partfrac(a^2/(a^2 - b^2),b)
```

```
ans =
b/(2*(a - b)) - b/(2*(a + b)) + 1

ans =
a/(2*(a + b)) + a/(2*(a - b))
```

If you do not specify the variable, then `partfrac` computes partial fraction decomposition with respect to a variable determined by `symvar`.

```
symvar(a^2/(a^2 - b^2),1)
partfrac(a^2/(a^2 - b^2))
```

```
ans =
b

ans =
a/(2*(a + b)) + a/(2*(a - b))
```

Factorization Modes

Use the `FactorMode` argument to choose a particular factorization mode.

Find the partial fraction decomposition without specifying the factorization mode. By default, `partfrac` uses factorization over rational numbers. In this mode, `partfrac` keeps numbers in their exact symbolic form.

```
syms x
partfrac(1/(x^3 + 2), x)
```

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

Find the partial fraction decomposition of the same expression, but this time use numeric factorization over real numbers. In this mode, `partfrac` factors the denominator into linear and quadratic irreducible polynomials with real coefficients. This mode converts all numeric values to floating-point numbers.

```
partfrac(1/(x^3 + 2), x, 'FactorMode', 'real')
```

```
ans =
0.2099868416491455274612017678797/(x + 1.2599210498948731647672106072782) -...
(0.2099868416491455274612017678797*x - 0.52913368398939982491723521309077)/(x^2 -...
1.2599210498948731647672106072782*x + 1.5874010519681994747517056392723)
```

Find the partial fraction decomposition of this expression using factorization over complex numbers. In this mode, `partfrac` reduces quadratic polynomials in the denominator to linear expressions with complex coefficients. This mode converts all numeric values to floating-point numbers.

```
partfrac(1/(x^3 + 2), x, 'FactorMode', 'complex')
```

```
ans =
0.2099868416491455274612017678797/(x + 1.2599210498948731647672106072782) +...
(- 0.10499342082457276373060088393985 - 0.18185393932862023392667876903163i)/...
(x - 0.62996052494743658238360530363911 - 1.0911236359717214035600726141898i) +...
(- 0.10499342082457276373060088393985 + 0.18185393932862023392667876903163i)/...
(x - 0.62996052494743658238360530363911 + 1.0911236359717214035600726141898i)
```

Find the partial fraction decomposition of this expression using the full factorization mode. In this mode, `partfrac` factors the denominator into linear expressions, reducing quadratic polynomials to linear expressions with complex coefficients. This mode keeps numbers in their exact symbolic form.

```
partfrac(1/(x^3 + 2), x, 'FactorMode', 'full')
```

```
ans =
2^(1/3)/(6*(x + 2^(1/3))) +...
(2^(1/3)*((3^(1/2)*1i)/2 - 1/2))/(6*(x + 2^(1/3)*((3^(1/2)*1i)/2 - 1/2))) -...
(2^(1/3)*((3^(1/2)*1i)/2 + 1/2))/(6*(x - 2^(1/3)*((3^(1/2)*1i)/2 + 1/2)))
```

Approximate the result with floating-point numbers by using `vpa`. Because the expression does not contain any symbolic parameters besides the variable `x`, the result is the same as in complex factorization mode.

```
vpa(ans)
```

```
ans =
0.2099868416491455274612017678797/(x + 1.2599210498948731647672106072782) +...
(- 0.10499342082457276373060088393985 - 0.18185393932862023392667876903163i)/...
```

```
(x - 0.62996052494743658238360530363911 - 1.0911236359717214035600726141898i) +...
(- 0.10499342082457276373060088393985 + 0.18185393932862023392667876903163i)/...
(x - 0.62996052494743658238360530363911 + 1.0911236359717214035600726141898i)
```

Replace `2` in the same expression with a symbolic parameter `a` and find partial fraction decomposition in the complex and full factorization modes. In the complex mode, `partfrac` factors only those expressions in the denominator whose coefficients can be converted to floating-point numbers. Thus, it returns this expression unchanged.

```
syms a
partfrac(1/(x^3 + a), x, 'FactorMode', 'complex')
```

```
ans =
1/(x^3 + a)
```

When you use the full factorization mode, `partfrac` factors expressions in the denominator symbolically. Thus, the partial fraction decomposition of the same expression in the full factorization mode is the following expression.

```
partfrac(1/(x^3 + a), x, 'FactorMode', 'full')
```

```
ans =
1/(3*(-a)^(2/3)*(x - (-a)^(1/3))) -...
((3^(1/2)*1i)/2 + 1/2)/(3*(-a)^(2/3)*(x + (-a)^(1/3)*((3^(1/2)*1i)/2 + 1/2))) +...
((3^(1/2)*1i)/2 - 1/2)/(3*(-a)^(2/3)*(x - (-a)^(1/3)*((3^(1/2)*1i)/2 - 1/2)))
```

Full Factorization Mode

In the full factorization mode,`partfrac` can also return partial fraction decomposition as a symbolic sum of polynomial roots expressed as `RootOf`.

Find the partial fraction decomposition of this expression.

```
syms x
s = partfrac(1/(x^3 + x - 3), x, 'FactorMode','full')
```

```
s =
symsum(-((6*root(z^3 + z - 3, z, k)^2)/247 +...
         (27*root(z^3 + z - 3, z, k))/247 +...
          4/247)/(root(z^3 + z - 3, z, k) - x), k, 1, 3)
```

Approximate the result with floating-point numbers by using `vpa`.

```
vpa(s)
```

```
ans =
0.1846004942289254798185772017286/(x - 1.2134116627622296341321313773815) +...
(- 0.092300247114462739909288600864302 + 0.11581130283490645120989658654914i)/...
(x + 0.60670583138111481706606568869074 - 1.4506122491884415265154422203395i) +...
(- 0.092300247114462739909288600864302 - 0.11581130283490645120989658654914i)/...
(x + 0.60670583138111481706606568869074 + 1.4506122491884415265154422203395i)
```

Numerators and Denominators of Partial Fraction Decomposition

Find a vector of numerators and a vector of denominators of the partial fraction decomposition.

Find the partial fraction decomposition of this expression.

```
syms x
P = partfrac(x^2/(x^3 - 3*x + 2), x)
```

```
P =
5/(9*(x - 1)) + 1/(3*(x - 1)^2) + 4/(9*(x + 2))
```

Partial fraction decomposition is a sum of fractions. Use the `children` function to return a vector containing the terms of that sum. then use `numden` to extract numerators and denominators of the terms.

```
[N,D] = numden(children(P))
```

```
N =
[ 5, 1, 4]

D =
[ 9*x - 9, 3*(x - 1)^2, 9*x + 18]
```

Reconstruct the partial fraction decomposition from the vectors of numerators and denominators.

```
P1 = sum(N./D)
```

```
P1 =
1/(3*(x - 1)^2) + 5/(9*x - 9) + 4/(9*x + 18)
```

Verify that the reconstructed expression, `P1`, is equivalent to the original partial fraction decomposition, `P`.

```
isAlways(P1 == P)
```

```
ans =
  logical
     1
```

Input Arguments

`expr` — Rational expression
symbolic expression | symbolic function

Rational expression, specified as a symbolic expression or function.

`var` — Variable of interest
symbolic variable

Variable of interest, specified as a symbolic variable.

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`.

Example: `partfrac(1/(x^3 - 2),x,'FactorMode','real')`

`FactorMode` — Factorization mode
`'rational'` (default) | `'real'` | `'complex'` | `'full'`

Factorization mode, specified as the comma-separated pair consisting of `'FactorMode'` and one of these character vectors.

`'rational'`	Factorization over rational numbers.
`'real'`	Factorization over real numbers. A real numeric factorization is a factorization into linear and quadratic irreducible polynomials with real coefficients. This factorization mode requires the coefficients of the input to be convertible to real floating-point numbers. All other inputs (for example, those inputs containing symbolic or complex coefficients) are treated as irreducible.

`'complex'`	Factorization over complex numbers. A complex numeric factorization is a factorization into linear factors whose coefficients are floating-point numbers. Such factorization is only available if the coefficients of the input are convertible to floating-point numbers, that is, if the roots can be determined numerically. Symbolic inputs are treated as irreducible.
`'full'`	Full factorization. A full factorization is a symbolic factorization into linear factors. The result shows these factors using radicals or as a `symsum` ranging over a `RootOf`.

Definitions

Partial Fraction Decomposition

Partial fraction decomposition of a rational expression

$$f(x) = g(x) + \frac{p(x)}{q(x)},$$

where the denominator can be written as $q(x) = q_1(x)q_2(x)\ldots$, is an expression of the form

$$f(x) = g(x) + \sum_j \frac{p_j(x)}{q_j(x)}$$

Here, the denominators $q_j(x)$ are irreducible polynomials or powers of irreducible polynomials. Also, the numerators $p_j(x)$ are polynomials of smaller degrees than the corresponding denominators $q_j(x)$.

Partial fraction decomposition can simplify integration by integrating each term of the returned expression separately.

See Also

`children` | `coeffs` | `collect` | `combine` | `compose` | `divisors` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

Introduced in R2015a

piecewise

Conditionally defined expression or function

Syntax

```
pw = piecewise(cond1,val1,cond2,val2,...)
pw = piecewise(cond1,val1,cond2,val2,...,otherwiseVal)
```

Description

`pw = piecewise(cond1,val1,cond2,val2,...)` returns the piecewise expression or function `pw` whose value is `val1` when condition `cond1` is true, is `val2` when `cond2` is true, and so on. If no condition is true, the value of `pw` is `NaN`.

`pw = piecewise(cond1,val1,cond2,val2,...,otherwiseVal)` returns the piecewise expression or function `pw` that has the value `otherwiseVal` if no condition is true.

Examples

Define and Evaluate Piecewise Expression

Define the following piecewise expression by using `piecewise`.

$$y = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}$$

```
syms x
y = piecewise(x<0, -1, x>0, 1)
```

```
y =
piecewise(x < 0, -1, 0 < x, 1)
```

Evaluate `y` at `-2`, `0`, and `2` by using `subs` to substitute for `x`. Because `y` is undefined at `x = 0`, the value is `NaN`.

```
subs(y, x, [-2 0 2])
```

```
ans =
[ -1, NaN, 1]
```

Define Piecewise Function

Define the following function symbolically.

$$y(x) = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}$$

```
syms y(x)
y(x) = piecewise(x<0, -1, x>0, 1)
```

```
y(x) =
piecewise(x < 0, -1, 0 < x, 1)
```

Because `y(x)` is a symbolic function, you can directly evaluate it for values of `x`. Evaluate `y(x)` at `-2`, `0`, and `2`. Because `y(x)` is undefined at `x = 0`, the value is `NaN`. For details, see "Create Symbolic Functions" on page 1-7.

```
y([-2 0 2])
```

```
ans =
[ -1, NaN, 1]
```

Set Value When No Conditions Is True

Set the value of a piecewise function when no condition is true (called *otherwise value*) by specifying an additional input argument. If an additional argument is not specified, the default otherwise value of the function is `NaN`.

Define the piecewise function

$$y(x) = \begin{cases} -2 & x < -2 \\ 0 & -2 < x < 0. \\ 1 & \text{otherwise} \end{cases}$$

```
syms y(x)
y(x) = piecewise(x<-2, -2, -2<x<0, 0, 1)
```

```
y(x) =
piecewise(x < -2, -2, x in Dom::Interval(-2, 0), 0, 1)
```

Evaluate `y(x)` between `-3` and `1` by generating values of `x` using `linspace`. At `-2` and `0`, `y(x)` evaluates to `1` because the other conditions are not true.

```
xvalues = linspace(-3,1,5)
yvalues = y(xvalues)
```

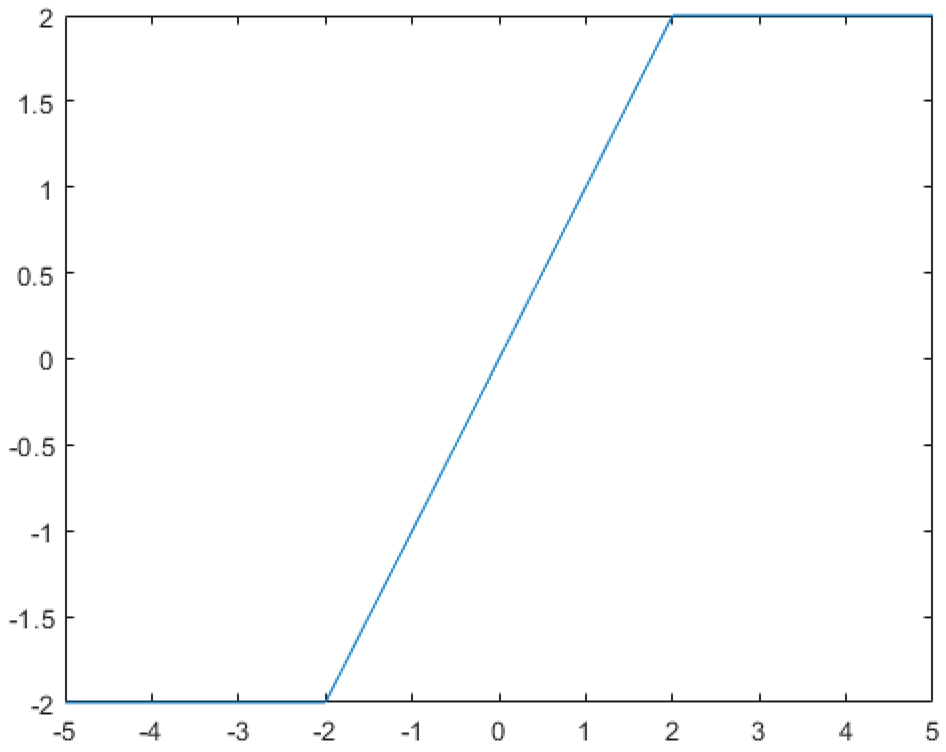
```
xvalues =
    -3    -2    -1     0     1
yvalues =
[ -2, 1, 0, 1, 1]
```

Plot Piecewise Expression

Plot the following piecewise expression by using `fplot`.

$$y = \begin{cases} -2 & x < -2 \\ x & -2 < x < 2. \\ 2 & x > 2 \end{cases}$$

```
syms x
y = piecewise(x<-2, -2, -2<x<2, x, x>2, 2);
fplot(y)
```



Assumptions and Piecewise Expressions

On creation, a piecewise expression applies existing assumptions. Apply assumptions set after creating the piecewise expression by using `simplify` on the expression.

Assume `x > 0`. Then define a piecewise expression with the same condition `x > 0`. `piecewise` automatically applies the assumption to simplify the condition.

```
syms x
assume(x > 0)
pw = piecewise(x<0, -1, x>0, 1)
pw =
1
```

Clear the assumption on x for further computations.

```
assume(x,'clear')
```

Create a piecewise expression pw with the condition x > 0. Then set the assumption that x > 0. Apply the assumption to pw by using simplify.

```
pw = piecewise(x<0, -1, x>0, 1);
assume(x > 0)
pw = simplify(pw)
```

```
pw =
1
```

Clear the assumption on x for further computations.

```
assume(x, 'clear')
```

Differentiate, Integrate, and Find Limits of Piecewise Expression

Differentiate, integrate, and find limits of a piecewise expression by using diff, int, and limit respectively.

Differentiate the following piecewise expression by using diff.

$$y = \begin{cases} 1/x & x < -1 \\ \sin(x)/x & x \geq -1 \end{cases}$$

```
syms x
y = piecewise(x<-1, 1/x, x>=-1, sin(x)/x);
diffy = diff(y, x)
```

```
diffy =
piecewise(x < -1, -1/x^2, -1 < x, cos(x)/x - sin(x)/x^2)
```

Integrate y by using int.

```
inty = int(y, x)
```

```
inty =
piecewise(x < -1, log(x), -1 <= x, sinint(x))
```

Find the limits of `y` at `0` and `-1` by using `limit`. Because `limit` finds the double-sided limit, the piecewise expression must be defined from both sides. Alternatively, you can find the right- or left-sided limit. For details, see `limit`.

```
limit(y, x, 0)
limit(y, x, -1)
```

```
ans =
1
ans =
limit(piecewise(x < -1, 1/x, -1 < x, sin(x)/x), x, -1)
```

Because the two conditions meet at `-1`, the limits from both sides differ and `limit` cannot find a double-sided limit.

Elementary Operations on Piecewise Expressions

Add, subtract, divide, and multiply two piecewise expressions. The resulting piecewise expression is only defined where the initial piecewise expressions are defined.

```
syms x
pw1 = piecewise(x<-1, -1, x>=-1, 1);
pw2 = piecewise(x<0, -2, x>=0, 2);
add = pw1 + pw2
sub = pw1 - pw2
mul = pw1 * pw2
div = pw1 / pw2
```

```
add =
piecewise(x < -1, -3, x in Dom::Interval([-1], 0), -1, 0 <= x, 3)
sub =
piecewise(x < -1, 1, x in Dom::Interval([-1], 0), 3, 0 <= x, -1)
mul =
piecewise(x < -1, 2, x in Dom::Interval([-1], 0), -2, 0 <= x, 2)
div =
piecewise(x < -1, 1/2, x in Dom::Interval([-1], 0), -1/2, 0 <= x, 1/2)
```

Modify or Extend Piecewise Expression

Modify a piecewise expression by replacing part of the expression using `subs`. Extend a piecewise expression by specifying the expression as the otherwise value of a new piecewise expression. This action combines the two piecewise expressions. `piecewise` does not check for overlapping or conflicting conditions. Instead, like an if-else ladder, `piecewise` returns the value for the first true condition.

Change the condition `x<2` in a piecewise expression to `x<0` by using `subs`.

```
syms x
pw = piecewise(x<2, -1, x>0, 1);
pw = subs(pw, x<2, x<0)

pw =
piecewise(x < 0, -1, 0 < x, 1)
```

Add the condition `x>5` with the value `1/x` to `pw` by creating a new piecewise expression with `pw` as the otherwise value.

```
pw = piecewise(x>5, 1/x, pw)

pw =
piecewise(5 < x, 1/x, x < 0, -1, 0 < x, 1)
```

Input Arguments

`cond` — Condition

symbolic condition | symbolic variable

Condition, specified as a symbolic condition or variable. A symbolic variable represents an unknown condition.

Example: x > 2

`val` — Value when condition is satisfied

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Value when condition is satisfied, specified as a number, vector, matrix, or multidimensional array, or as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`otherwiseVal` — **Value if no conditions are true**
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Value if no conditions are true, specified as a number, vector, matrix, or multidimensional array, or as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression. If `otherwiseVal` is not specified, its value is `NaN`.

Output Arguments

`pw` — **Piecewise expression or function**
symbolic expression | symbolic function

Piecewise expression or function, returned as a symbolic expression or function. The value of `pw` is the value `val` of the first condition `cond` that is true. To find the value of `pw`, use `subs` to substitute for variables in `pw`.

Tips

- `piecewise` does not check for overlapping or conflicting conditions. A piecewise expression returns the value of the first true condition and disregards any following true expressions. Thus, `piecewise` mimics an if-else ladder.

See Also

`and` | `assume` | `assumeAlso` | `assumptions` | `if` | `in` | `isAlways` | `not` | `or`

Introduced in R2016b

pinv

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.

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`.

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 - 0.0937i
```

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 + 1i/32, 1/24 - 1i/32]
[ - 7/32 - 5i/96, 5/16 + 7i/96]
[   7/24 + 1i/16, 1/96 - 3i/32]
```

Check that `A*X*A = A` and `X*A*X = X`.

```
isAlways(A*X*A == A)
```

```
ans =
  2×3 logical array
     1     1     1
     1     1     1
```

```
isAlways(X*A*X == X)
```

```
ans =
  3×2 logical array
     1     1
     1     1
     1     1
```

Now, verify that `A*X` and `X*A` are Hermitian matrices.

```
isAlways(A*X == (A*X)')
```

```
ans =
  2×2 logical array
     1     1
     1     1
```

```
isAlways(X*A == (X*A)')
```

```
ans =
  3×3 logical array
     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
```

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.

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`.

See Also

`inv` | `linalg::pseudoInverse` | `pinv` | `rank` | `svd`

Introduced in R2013a

plus+

Symbolic addition

Syntax

```
A + B
plus(A,B)
```

Description

`A + B` adds `A` and `B`.

`plus(A,B)` is equivalent to `A + B`.

Examples

Add Scalar to Array

`plus` adds `x` to each element of the array.

```
syms x
A = [x sin(x) 3];
A + x
```

```
ans =

[ 2*x, x + sin(x), x + 3]
```

Add Two Matrices

Add the identity matrix to matrix `M`.

```
syms x
M = [x x^2;Inf 0];
M + eye(2)
```

```
ans =
[ x + 1, x^2]
[   Inf,   1]
```

Alternatively, use `plus(M,eye(2))`.

```
plus(M,eye(2))

ans =
[ x + 1, x^2]
[   Inf,   1]
```

Add Symbolic Functions

```
syms f(x) g(x)
f(x) = x^2 + 5*x + 6;
g(x) = 3*x - 2;
h = f + g


h(x) =
x^2 + 8*x + 4
```

Add Expression to Symbolic Function

Add expression `expr` to function `f`.

```
syms f(x)
f(x) = x^2 + 3*x + 2;
expr = x^2 - 2;
f(x) = f(x) + expr

f(x) =
2*x^2 + 3*x
```

Input Arguments

`A` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression.

`B` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression.

Tips

- All nonscalar arguments must be the same size. If one input argument is nonscalar, then `plus` expands the scalar into an array of the same size as the nonscalar argument, with all elements equal to the scalar.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `power` | `rdivide` | `times` | `transpose`

Introduced before R2006a

pochhammer

Pochhammer symbol

Syntax

```
pochhammer(x,n)
```

Description

`pochhammer(x,n)` returns the “Pochhammer Symbol” on page 4-1300 $(x)_n$.

Examples

Find Pochhammer Symbol for Numeric and Symbolic Inputs

Find the Pochhammer symbol for the numeric inputs `x = 3` at `n = 2`.

```
pochhammer(3,2)
```

```
ans =
    12
```

Find the Pochhammer symbol for the symbolic input `x` at `n = 3` . The `pochhammer` function does not automatically return the expanded form of the expression. Use `expand` to force `pochhammer` to return the form of the expanded expression.

```
syms x
P = pochhammer(x, 3)
P = expand(P)
```

```
P =
pochhammer(x, 3)
P =
x^3 + 3*x^2 + 2*x
```

Rewrite and Factor Outputs of Pochhammer

If conditions are satisfied, `expand` rewrites the solution using `gamma`.

```
syms n x
assume(x>0)
assume(n>0)
P = pochhammer(x, n);
P = expand(P)

P =
gamma(n + x)/gamma(x)
```

Clear assumptions on `n` and `x` to use them in further computations.

```
syms n x clear
```

To convert expanded output of `pochhammer` into its factors, use `factor`.

```
P = expand(pochhammer(x, 4));
P = factor(P)

P =
[ x, x + 3, x + 2, x + 1]
```

Differentiate Pochhammer Symbol

Differentiate `pochhammer` once with respect to `x`.

```
syms n x
diff(pochhammer(x,n),x)

ans =
pochhammer(x, n)*(psi(n + x) - psi(x))
```

Differentiate `pochhammer` twice with respect to `n`.

```
diff(pochhammer(x,n),n,2)

ans =
pochhammer(x, n)*psi(n + x)^2 + pochhammer(x, n)*psi(1, n + x)
```

Taylor Series Expansion of Pochhammer Symbol

Use `taylor` to find the Taylor series expansion of `pochhammer` with `n = 3` around the expansion point `x = 2`.

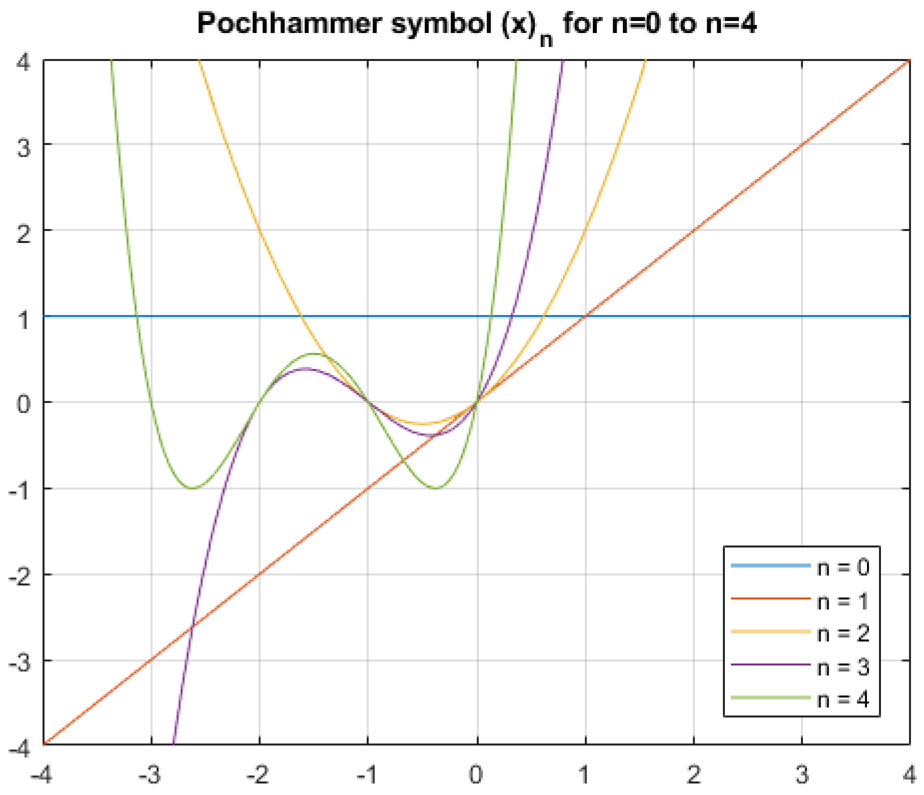
```
syms x
taylor(pochhammer(x,3),x,2)
```

```
ans =
26*x + 9*(x - 2)^2 + (x - 2)^3 - 28
```

Plot Pochhammer Symbol

Plot the Pochhammer symbol from `n = 0` to `n = 4` for `x`. Use `axis` to display the region of interest. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(pochhammer(x,0:4))
axis([-4 4 -4 4])

grid on
legend('n = 0','n = 1','n = 2','n = 3','n = 4','Location','Best')
title('Pochhammer symbol (x)_n for n=0 to n=4')
```



Input Arguments

x — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

n — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix, or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Definitions

Pochhammer Symbol

Pochhammer's symbol is defined as

$$(x)_n = \frac{\Gamma(x+n)}{\Gamma(x)},$$

where Γ is the Gamma function.

If n is a positive integer, Pochhammer's symbol is

$$(x)_n = x(x+1)\ldots(x+n-1)$$

Algorithms

- If `x` and `n` are numerical values, then an explicit numerical result is returned. Otherwise, a symbolic function call is returned.
- If both `x` and `x + n` are nonpositive integers, then

  $$(x)_n = (-1)^n \frac{\Gamma(1-x)}{\Gamma(1-x-n)}.$$

- The following special cases are implemented.

$$(x)_0 = 1$$
$$(x)_1 = x$$
$$(x)_{-1} = \frac{1}{x-1}$$
$$(1)_n = \Gamma(n+1)$$
$$(2)_n = \Gamma(n+2)$$

- If `n` is a positive integer, then `expand(pochhammer(x,n))` returns the expanded polynomial $x(x+1)...(x+n-1)$.
- If `n` is not an integer, then `expand(pochhammer(x,n))` returns a representation in terms of `gamma`.

See Also

`factorial` | `gamma`

Introduced in R2014b

poles

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`.

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 =
1i

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: Unable to determine poles.
ans =
Empty sym: 0-by-1
```

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: 0-by-1
```

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
```

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.

See Also

`limit` | `solve` | `symvar` | `vpasolve`

Introduced in R2012b

poly2sym

Create symbolic polynomial from vector of coefficients

Syntax

```
p = poly2sym(c)
p = poly2sym(c,var)
```

Description

`p = poly2sym(c)` creates the symbolic polynomial expression `p` from the vector of coefficients `c`. The polynomial variable is `x`. If `c = [c1,c2,...,cn]`, then `p = poly2sym(c)` returns $c_1x^{n-1} + c_2x^{n-2} + ... + c_n$.

This syntax does not create the symbolic variable `x` in the MATLAB Workspace.

`p = poly2sym(c,var)` uses `var` as a polynomial variable when creating the symbolic polynomial expression `p` from the vector of coefficients `c`.

Examples

Create Polynomial Expression

Create a polynomial expression from a symbolic vector of coefficients. If you do not specify a polynomial variable, `poly2sym` uses `x`.

```
syms a b c d
p = poly2sym([a, b, c, d])
```

```
p =
a*x^3 + b*x^2 + c*x + d
```

Create a polynomial expression from a symbolic vector of rational coefficients.

```
p = poly2sym(sym([1/2, -1/3, 1/4]))

p =
x^2/2 - x/3 + 1/4
```

Create a polynomial expression from a numeric vector of floating-point coefficients. The toolbox converts floating-point coefficients to rational numbers before creating a polynomial expression.

```
p = poly2sym([0.75, -0.5, 0.25])

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

Specify Polynomial Variable

Create a polynomial expression from a symbolic vector of coefficients. Use `t` as a polynomial variable.

```
syms a b c d t
p = poly2sym([a, b, c, d], t)

p =
a*t^3 + b*t^2 + c*t + d
```

To use a symbolic expression, such as `t^2 + 1` or `exp(t)`, instead of a polynomial variable, substitute the variable using `subs`.

```
p1 = subs(p, t, t^2 + 1)
p2 = subs(p, t, exp(t))

p1 =
d + a*(t^2 + 1)^3 + b*(t^2 + 1)^2 + c*(t^2 + 1)

p2 =
d + c*exp(t) + a*exp(3*t) + b*exp(2*t)
```

Input Arguments

c — Polynomial coefficients
numeric vector | symbolic vector

Polynomial coefficients, specified as a numeric or symbolic vector. Argument `c` can be a column or row vector.

`var` — Polynomial variable
symbolic variable

Polynomial variable, specified as a symbolic variable.

Output Arguments

`p` — Polynomial
symbolic expression

Polynomial, returned as a symbolic expression.

Tips

- When you call `poly2sym` for a numeric vector `c`, the toolbox converts the numeric vector to a vector of symbolic numbers using the default (rational) conversion mode of `sym`.

See Also

`coeffs` | `sym` | `sym2poly`

Introduced before R2006a

polylog

Polylogarithm

Syntax

```
polylog(n,x)
```

Description

`polylog(n,x)` returns the polylogarithm of the order `n` and the argument `x`.

Examples

Polylogarithm for Numeric and Symbolic Arguments

Depending on its arguments, `polylog` returns floating-point or exact symbolic results.

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

```
A = [polylog(3,-1/2), polylog(4,1/3), polylog(5,3/4)]
```

```
A =
   -0.4726    0.3408    0.7697
```

Compute polylogarithms for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `polylog` returns unresolved symbolic calls.

```
symA = [polylog(3,sym(-1/2)), polylog(sym(4),1/3), polylog(5,sym(3/4))]
```

```
symA =
[ polylog(3, -1/2), polylog(4, 1/3), polylog(5, 3/4)]
```

Use `vpa` to approximate symbolic results with the required number of digits.

```
vpa(symA)
```

```
ans =
[ -0.47259784465889687461862319312655,...
0.34079113085625075247764094401 22,...
0.76973541059975738097269173152535]
```

Explicit Expressions for Polylogarithms

If the order of the polylogarithm is `0`, `1`, or a negative integer, then `polylog` returns an explicit expression.

The polylogarithm of `n = 1` is a logarithm function.

```
syms x
polylog(1,x)

ans =
-log(1 - x)
```

The polylogarithms of `n < 1` are rational expressions.

```
polylog(0,x)

ans =
-x/(x - 1)

polylog(-1,x)

ans =
x/(x - 1)^2

polylog(-2,x)

ans =
-(x^2 + x)/(x - 1)^3

polylog(-3,x)

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

polylog(-10,x)

ans =
-(x^10 + 1013*x^9 + 47840*x^8 + 455192*x^7 + ...
1310354*x^6 + 1310354*x^5 + 455192*x^4 +...
47840*x^3 + 1013*x^2 + x)/(x - 1)^11
```

More Special Values

The `polylog` function has special values for some parameters.

If the second argument is `0`, then the polylogarithm equals `0` for any integer value of the first argument. If the second argument is `1`, then the polylogarithm is the Riemann zeta function of the first argument.

```
syms n
[polylog(n,0), polylog(n,1)]
```

```
ans =
[ 0, zeta(n)]
```

If the second argument is `-1`, then the polylogarithm has a special value for any integer value of the first argument except `1`.

```
assume(n ~= 1)
polylog(n,-1)
```

```
ans =
zeta(n)*(2^(1 - n) - 1)
```

For further computations, clear the assumption.

```
syms n clear
```

Other special values of the polylogarithm include the following.

```
[polylog(4,sym(1)), polylog(sym(5),-1), polylog(2,sym(i))]
```

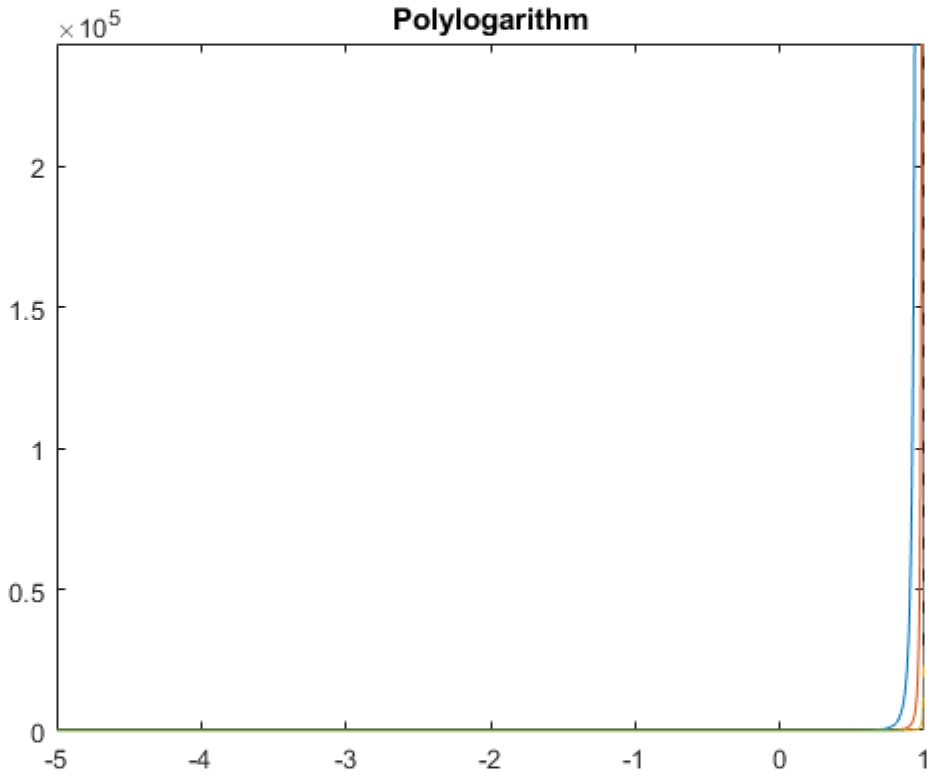
```
ans =
[ pi^4/90, -(15*zeta(5))/16, catalan*1i - pi^2/48]
```

Plot Polylogarithm

Plot the polylogarithms of the orders from -3 to 1. Prior to R2016a, use `ezplot` instead of `fplot`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
for n = -3:1
  fplot(polylog(n,x),[-5 1])
  hold on
end
```

```
title('Polylogarithm')
hold off
```



Handle Expressions Containing Polylogarithms

Many functions, such as `diff` and `int`, can handle expressions containing `polylog`.

Differentiate these expressions containing polylogarithms.

```
syms n x
diff(polylog(n, x), x)
diff(x*polylog(n, x), x)
```

```
ans =
polylog(n - 1, x)/x

ans =
polylog(n, x) + polylog(n - 1, x)
```

Compute integrals of these expressions containing polylogarithms.

```
int(polylog(n, x)/x, x)
int(polylog(n, x) + polylog(n - 1, x), x)
```

```
ans =
polylog(n + 1, x)

ans =
x*polylog(n, x)
```

Input Arguments

n — Index of polylogarithm

integer

Index of the polylogarithm, specified as an integer.

x — Argument of polylogarithm

number | symbolic variable | symbolic expression | symbolic function | vector | matrix

Argument of the polylogarithm, specified as a number, symbolic variable, expression, function, vector, or matrix.

Definitions

Polylogarithm

For a complex number `z` of modulus `|z| < 1`, the polylogarithm of order `n` is defined as follows.

$$\mathrm{Li}_n(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^n}$$

This function is extended to the whole complex plane by analytic continuation, with a branch cut along the real interval [`1`, ∞) for `n` ≥ 1.

Tips

- `polylog(2,x)` is equivalent to `dilog(1 - x)`.
- The logarithmic integral function (the integral logarithm) uses the same notation, `Li(x)`, but without an index. The toolbox provides the `logint` function for the integral logarithm.

See Also

`dilog` | `log` | `log10` | `log2` | `logint` | `zeta`

Introduced in R2014b

potential

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.

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.

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
```

Definitions

Scalar Potential of 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:

$$P(X) = \int_0^1 (X-Y) \cdot V\big(Y + \lambda (X-Y)\big)\, d\lambda$$

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`.

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `jacobian` | `laplacian` | `vectorPotential`

Introduced in R2012a

power.^

Symbolic array power

Syntax

```
A.^B
power(A,B)
```

Description

`A.^B` computes `A` to the `B` power and is an elementwise operation.

`power(A,B)` is equivalent to `A.^B`.

Examples

Square Each Matrix Element

Create a `2`-by-`3` matrix.

```
A = sym('a', [2 3])
```

```
A =
[ a1_1, a1_2, a1_3]
[ a2_1, a2_2, a2_3]
```

Square each element of the matrix.

```
A.^2
```

```
ans =
[ a1_1^2, a1_2^2, a1_3^2]
[ a2_1^2, a2_2^2, a2_3^2]
```

Use Matrices for Base and Exponent

Create a 3-by-3 symbolic Hilbert matrix and a 3-by-3 diagonal matrix.

```
H = sym(hilb(3))
d = diag(sym([1 2 3]))

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

d =
[ 1, 0, 0]
[ 0, 2, 0]
[ 0, 0, 3]
```

Raise the elements of the Hilbert matrix to the powers of the diagonal matrix. The base and the exponent must be matrices of the same size.

```
H.^d

ans =
[ 1,   1,     1]
[ 1, 1/9,     1]
[ 1,   1, 1/125]
```

Input Arguments

A — Input

number | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression. Inputs A and B must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

B — Input

number | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `rdivide` | `times` | `transpose`

Introduced before R2006a

pretty

Prettyprint symbolic expressions

Note `pretty` is not recommended. Use Live Scripts instead. Live Scripts provide full math rendering while `pretty` uses plain-text formatting. See “What Is a Live Script?” (MATLAB)

Syntax

```
pretty(X)
```

Description

`pretty(X)` prints `X` in a plain-text format that resembles typeset mathematics. For true typeset rendering, use Live Scripts instead. See “What Is a Live Script?” (MATLAB)

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
```

```
/ 3   sqrt(5) \
| - - ------- |
| 2      2    |
|             |
| sqrt(5)   3 |
| ------- + - |
\    2      2 /
```

Solve this equation, and then use `pretty` to represent the solutions in the format similar to typeset mathematics:

```
syms x
s = solve(x^4 + 2*x + 1, x,'MaxDegree',3);
pretty(s)
```

For better readability, `pretty` uses abbreviations when representing long expressions:

```
/         -1         \
|                    |
|           2    1   |
|    #2 - ---- + -   |
|         9 #2   3   |
|                    |
|   1         #2   1 |
| ---- - #1 - -- + - |
| 9 #2         2   3 |
|                    |
|        1    #2   1 |
| #1 + ---- - -- + - |
\      9 #2    2   3 /

where

                /   2       \
         sqrt(3) | ---- + #2 | 1i
                \ 9 #2      /
   #1 == ------------------------
                    2

         / sqrt(11) sqrt(27)   17 \1/3
   #2 == | ----------------- - -- |
         \         27          27 /
```

Introduced before R2006a

prevprime

Previous prime number

Syntax

```
prevprime(n)
```

Description

`prevprime(n)` returns the largest prime number smaller than or equal to `n`. If `n` is a vector or matrix, then `prevprime` acts element-wise on `n`.

Examples

Find Previous Prime Number

Find the largest prime number smaller than `100`. Because `prevprime` only accepts symbolic input, wrap `100` with `sym`.

```
prevprime(sym(100))
```

```
ans =
97
```

Find the largest prime numbers smaller than `1000`, `10000`, and `100000` by specifying the input as a vector.

```
prevprime(sym([1000 10000 100000]))
```

```
ans =
[ 997, 9973, 99991]
```

Find Large Prime Number

When finding large prime numbers, if your input has 15 or more digits, then use quotation marks to represent the number accurately. The best way to find an arbitrary large prime is to use powers of 10, which are accurately represented without requiring quotation marks. For more information, see “Numeric to Symbolic Conversion” on page 2-125.

Find a large prime number by using `10^sym(18)`.

```
prevprime(10^sym(18))
```

```
ans =
999999999999999989
```

Find the prime number previous to `823572345728582545` by using quotation marks.

```
prevprime(sym('823572345728582545'))
```

```
ans =
823572345728582543
```

Input Arguments

`n` — Input

symbolic number | symbolic vector | symbolic matrix

Input, specified as a symbolic number, or as a symbolic vector or matrix of symbolic numbers. `n` is rounded down.

Example: `sym(100)`

See Also

`isprime` | `nextprime` | `primes`

Introduced in R2016b

psi

Digamma function

Syntax

```
psi(x)
psi(k,x)
```

Description

`psi(x)` computes the digamma function on page 4-1329 of `x`.

`psi(k,x)` computes the polygamma function on page 4-1329 of `x`, which is the `k`th derivative of the digamma function at `x`.

Input Arguments

`x`

Symbolic number, variable, expression, or a vector, matrix, or multidimensional array of these.

`k`

Nonnegative integer or vector, matrix or multidimensional array of nonnegative integers. If `x` is nonscalar and `k` is scalar, then `k` is expanded into a nonscalar of the same dimensions as `x` with each element being equal to `k`. If both `x` and `k` are nonscalars, they must have the same dimensions.

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))]
```

```
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 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]
```

Compute the polygamma function for elements of matrix `M` and vector `V`. The `psi` function acts elementwise on nonscalar inputs.

```
M = sym([0 inf; 1/3 1/2]);
polyGammaM = [1 3; 2 2];
V = sym([1, inf]);
polyGammaV = [6 6];
psi(polyGammaM,M)
psi(polyGammaV,V)

ans =
[                                  Inf,            0]
[ - 26*zeta(3) - (4*3^(1/2)*pi^3)/9, -14*zeta(3)]

ans =
[ -720*zeta(7), 0]
```

Because all elements of `polyGammaV` have the same value, you can replace `polyGammaV` by a scalar of that value. `psi` expands the scalar into a nonscalar of the same size as `V` and computes the result.

```
V = sym([1, inf]);
psi(6,V)
```

```
ans =
[ -720*zeta(7), 0]
```

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:

$$\psi^{(k)}(x) = \frac{d^{k+1}}{dx^{k+1}} \ln \Gamma(x) = \frac{d^k}{dx^k} \psi(x)$$

Tips

- Calling `psi` for a number that is not a symbolic object invokes the MATLAB `psi` function. This function accepts real nonnegative arguments `x`. 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)`.

See Also

`beta` | `factorial` | `gamma` | `nchoosek`

Introduced in R2011b

qr

QR factorization

Syntax

```
R = qr(A)
[Q,R] = qr(A)
[Q,R,P] = qr(A)

[C,R] = qr(A,B)
[C,R,P] = qr(A,B)

[Q,R,p] = qr(A,'vector')
[C,R,p] = qr(A,B,'vector')

___ = qr(___,'econ')
___ = qr(___,'real')
```

Description

`R = qr(A)` returns the R part of the QR decomposition on page 4-1341 `A = Q*R`. Here, `A` is an *m*-by-*n* matrix, `R` is an *m*-by-*n* upper triangular matrix, and `Q` is an *m*-by-*m* unitary matrix.

`[Q,R] = qr(A)` returns an upper triangular matrix `R` and a unitary matrix `Q`, such that `A = Q*R`.

`[Q,R,P] = qr(A)` returns an upper triangular matrix `R`, a unitary matrix `Q`, and a permutation matrix `P`, such that `A*P = Q*R`. If all elements of `A` can be approximated by the floating-point numbers, then this syntax chooses the column permutation `P` so that `abs(diag(R))` is decreasing. Otherwise, it returns `P = eye(n)`.

`[C,R] = qr(A,B)` returns an upper triangular matrix `R` and a matrix `C`, such that `C = Q'*B` and `A = Q*R`. Here, `A` and `B` must have the same number of rows.

`C` and `R` represent the solution of the matrix equation `A*X = B` as `X = R\C`.

`[C,R,P] = qr(A,B)` returns an upper triangular matrix `R`, a matrix `C`, such that `C = Q'*B`, and a permutation matrix `P`, such that `A*P = Q*R`. If all elements of `A` can be approximated by the floating-point numbers, then this syntax chooses the permutation matrix `P` so that `abs(diag(R))` is decreasing. Otherwise, it returns `P = eye(n)`. Here, `A` and `B` must have the same number of rows.

`C`, `R`, and `P` represent the solution of the matrix equation `A*X = B` as `X = P*(R\C)`.

`[Q,R,p] = qr(A,'vector')` returns the permutation information as a vector `p`, such that `A(:,p) = Q*R`.

`[C,R,p] = qr(A,B,'vector')` returns the permutation information as a vector `p`.

`C`, `R`, and `p` represent the solution of the matrix equation `A*X = B` as `X(p,:) = R\C`.

`___ = qr(___,'econ')` returns the "economy size" decomposition. If `A` is an `m`-by-`n` matrix with `m > n`, then `qr` computes only the first `n` columns of `Q` and the first `n` rows of `R`. For `m <= n`, the syntaxes with `'econ'` are equivalent to the corresponding syntaxes without `'econ'`.

When you use `'econ'`, `qr` always returns the permutation information as a vector `p`.

You can use `0` instead of `'econ'`. For example, `[Q,R] = qr(A,0)` is equivalent to `[Q,R] = qr(A,'econ')`.

`___ = qr(___,'real')` assumes that input arguments and intermediate results are real, and therefore, suppresses calls to `abs` and `conj`. When you use this flag, `qr` assumes that all symbolic variables represent real numbers. When using this flag, ensure that all numeric arguments are real numbers.

Use `'real'` to avoid complex conjugates in the result.

Examples

R part of QR Factorization

Compute the R part of the QR decomposition of the `4`-by-`4` Wilkinson's eigenvalue test matrix.

Create the 4-by-4 Wilkinson's eigenvalue test matrix:

```
A = sym(wilkinson(4))

A =
[ 3/2,   1,   0,   0]
[   1, 1/2,   1,   0]
[   0,   1, 1/2,   1]
[   0,   0,   1, 3/2]
```

Use the syntax with one output argument to return the R part of the QR decomposition without returning the Q part:

```
R = qr(A)

R =
[ 13^(1/2)/2,          (4*13^(1/2))/13,             (2*13^(1/2))/13,                                 0]
[          0, (13^(1/2)*53^(1/2))/26, (10*13^(1/2)*53^(1/2))/689,        (2*13^(1/2)*53^(1/2))/53]
[          0,                      0,   (53^(1/2)*381^(1/2))/106, (172*53^(1/2)*381^(1/2))/20193]
[          0,                      0,                          0,              (35*381^(1/2))/762]
```

QR Factorization of Pascal Matrix

Compute the QR decomposition of the 3-by-3 Pascal matrix.

Create the 3-by-3 Pascal matrix:

```
A = sym(pascal(3))

A =
[ 1, 1, 1]
[ 1, 2, 3]
[ 1, 3, 6]
```

Find the Q and R matrices representing the QR decomposition of A:

```
[Q,R] = qr(A)

Q =
[ 3^(1/2)/3, -2^(1/2)/2,  6^(1/2)/6]
[ 3^(1/2)/3,          0, -6^(1/2)/3]
[ 3^(1/2)/3,  2^(1/2)/2,  6^(1/2)/6]

R =
[ 3^(1/2), 2*3^(1/2), (10*3^(1/2))/3]
[       0,   2^(1/2),  (5*2^(1/2))/2]
[       0,         0,      6^(1/2)/6]
```

Verify that `A = Q*R` using `isAlways`:

```
isAlways(A == Q*R)

ans =
  3×3 logical array
     1     1     1
     1     1     1
     1     1     1
```

Permutation Information

Using permutations helps increase numerical stability of the QR factorization for floating-point matrices. The `qr` function returns permutation information either as a matrix or as a vector.

Set the number of significant decimal digits, used for variable-precision arithmetic, to 10. Approximate the `3`-by-`3` symbolic Hilbert matrix by floating-point numbers:

```
previoussetting = digits(10);
A = vpa(hilb(3))

A =
[          1.0,          0.5, 0.3333333333]
[          0.5, 0.3333333333,         0.25]
[ 0.3333333333,         0.25,          0.2]
```

First, compute the QR decomposition of `A` without permutations:

```
[Q,R] = qr(A)

Q =
[ 0.8571428571, -0.5016049166,  0.1170411472]
[ 0.4285714286,  0.5684855721, -0.7022468832]
[ 0.2857142857,  0.6520863915,  0.7022468832]

R =
[ 1.166666667, 0.6428571429,           0.45]
[           0, 0.1017143303,   0.1053370325]
[           0,            0, 0.003901371573]
```

Compute the difference between A and Q*R. The computed Q and R matrices do not strictly satisfy the equality A*P = Q*R because of the round-off errors.

```
A - Q*R
```

```
ans =
[ -1.387778781e-16, -3.989863995e-16, -2.064320936e-16]
[ -3.469446952e-18, -8.847089727e-17, -1.084202172e-16]
[ -2.602085214e-18, -6.591949209e-17, -6.678685383e-17]
```

To increase numerical stability of the QR decomposition, use permutations by specifying the syntax with three output arguments. For matrices that do not contain symbolic variables, expressions, or functions, this syntax triggers pivoting, so that abs(diag(R)) in the returned matrix R is decreasing.

```
[Q,R,P] = qr(A)
```

```
Q =
[ 0.8571428571, -0.4969293466, -0.1355261854]
[ 0.4285714286,  0.5421047417,  0.7228063223]
[ 0.2857142857,  0.6776309272, -0.6776309272]
R =
[ 1.166666667,          0.45,   0.6428571429]
[           0, 0.1054092553,   0.1016446391]
[           0,            0, 0.003764616262]
P =
     1     0     0
     0     0     1
     0     1     0
```

Check the equality A*P = Q*R again. QR factorization with permutations results in smaller round-off errors.

```
A*P - Q*R
```

```
ans =
[ -3.469446952e-18, -4.33680869e-18, -6.938893904e-18]
[                0, -8.67361738e-19, -1.734723476e-18]
[                0, -4.33680869e-19, -1.734723476e-18]
```

Now, return the permutation information as a vector by using the 'vector' argument:

```
[Q,R,p] = qr(A,'vector')
```

```
Q =
[ 0.8571428571, -0.4969293466, -0.1355261854]
[ 0.4285714286,  0.5421047417,  0.7228063223]
[ 0.2857142857,  0.6776309272, -0.6776309272]
R =
[ 1.166666667,         0.45,   0.6428571429]
[           0, 0.1054092553,   0.1016446391]
[           0,            0, 0.003764616262]
p =
     1     3     2
```

Verify that `A(:,p) = Q*R`:

```
A(:,p) - Q*R
```

```
ans =
[ -3.469446952e-18, -4.33680869e-18, -6.938893904e-18]
[                0, -8.67361738e-19, -1.734723476e-18]
[                0, -4.33680869e-19, -1.734723476e-18]
```

Exact symbolic computations let you avoid roundoff errors:

```
A = sym(hilb(3));
[Q,R] = qr(A);
A - Q*R
```

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

Restore the number of significant decimal digits to its default setting:

```
digits(previoussetting)
```

Use QR Decomposition to Solve Matrix Equation

You can use `qr` to solve systems of equations in a matrix form.

Suppose you need to solve the system of equations `A*X = b`, where `A` and `b` are the following matrix and vector:

```
A = sym(invhilb(5))
b = sym([1:5]')
```

```
A =
[    25,   -300,    1050,   -1400,    630]
[  -300,   4800,  -18900,   26880, -12600]
[  1050, -18900,   79380, -117600,  56700]
[ -1400,  26880, -117600,  179200, -88200]
[   630, -12600,   56700,  -88200,  44100]
b =

 1
 2
 3
 4
 5
```

Use `qr` to find matrices `C` and `R`, such that `C = Q'*B` and `A = Q*R`:

```
[C,R] = qr(A,b);
```

Compute the solution `X`:

```
X = R\C
```

```
X =
        5
    71/20
   197/70
  657/280
 1271/630
```

Verify that `X` is the solution of the system `A*X = b` using `isAlways`:

```
isAlways(A*X == b)
```

```
ans =
  5×1 logical array
     1
     1
     1
     1
     1
```

Use QR Decomposition with Permutation Information to Solve Matrix Equation

When solving systems of equations that contain floating-point numbers, the QR decomposition with the permutation matrix or vector.

Suppose you need to solve the system of equations `A*X = b`, where `A` and `b` are the following matrix and vector:

```
previoussetting = digits(10);
A = vpa([2 -3 -1; 1 1 -1; 0 1 -1]);
b = vpa([2; 0; -1]);
```

Use `qr` to find matrices `C` and `R`, such that `C = Q'*B` and `A = Q*R`:

```
[C,R,P] = qr(A,b)

C =
  -2.110579412
 -0.2132007164
  0.7071067812
R =
[ 3.31662479, 0.3015113446, -1.507556723]
[          0,  1.705605731, -1.492405014]
[          0,            0, 0.7071067812]
P =
     0     0     1
     1     0     0
     0     1     0
```

Compute the solution `X`:

```
X = P*(R\C)

X =
   1.0
 -0.25
  0.75
```

Alternatively, return the permutation information as a vector:

```
[C,R,p] = qr(A,b,'vector')

C =
  -2.110579412
 -0.2132007164
  0.7071067812
R =
[ 3.31662479, 0.3015113446, -1.507556723]
[          0,  1.705605731, -1.492405014]
[          0,            0, 0.7071067812]
```

```
p =
     2     3     1
```

In this case, compute the solution `X` as follows:

```
X(p,:) = R\C

X =
   1.0
 -0.25
  0.75
```

Restore the number of significant decimal digits to its default setting:

```
digits(previoussetting)
```

"Economy Size" Decomposition

Use `'econ'` to compute the “economy size” QR decomposition.

Create a matrix that consists of the first two columns of the `4`-by-`4` Pascal matrix:

```
A = sym(pascal(4));
A = A(:,1:2)

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

Compute the QR decomposition for this matrix:

```
[Q,R] = qr(A)

Q =
[ 1/2, -(3*5^(1/2))/10,    (3^(1/2)*10^(1/2))/10,          0]
[ 1/2,     -5^(1/2)/10, -(2*3^(1/2)*10^(1/2))/15,  6^(1/2)/6]
[ 1/2,      5^(1/2)/10,   -(3^(1/2)*10^(1/2))/30, -6^(1/2)/3]
[ 1/2,  (3*5^(1/2))/10,    (3^(1/2)*10^(1/2))/15,  6^(1/2)/6]

R =
[ 2,       5]
[ 0, 5^(1/2)]
[ 0,       0]
[ 0,       0]
```

Now, compute the “economy size” QR decomposition for this matrix. Because the number of rows exceeds the number of columns, `qr` computes only the first `2` columns of `Q` and the first `2` rows of `R`.

```
[Q,R] = qr(A,'econ')

Q =
[ 1/2, -(3*5^(1/2))/10]
[ 1/2,     -5^(1/2)/10]
[ 1/2,      5^(1/2)/10]
[ 1/2,  (3*5^(1/2))/10]

R =
[ 2,       5]
[ 0, 5^(1/2)]
```

Avoid Complex Conjugates

Use the `'real'` flag to avoid complex conjugates in the result.

Create a matrix, one of the elements of which is a variable:

```
syms x
A = [1 2; 3 x]

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

Compute the QR factorization of this matrix. By default, `qr` assumes that `x` represents a complex number, and therefore, the result contains expressions with the `abs` function.

```
[Q,R] = qr(A)

Q =
[     10^(1/2)/10, -((3*x)/10 - 9/5)/(abs(x/10 - 3/5)^2...
                          + abs((3*x)/10 - 9/5)^2)^(1/2)]
[ (3*10^(1/2))/10,      (x/10 - 3/5)/(abs(x/10 - 3/5)^2...
                          + abs((3*x)/10 - 9/5)^2)^(1/2)]

R =
[ 10^(1/2),                            (10^(1/2)*(3*x + 2))/10]
[        0, (abs(x/10 - 3/5)^2 + abs((3*x)/10 - 9/5)^2)^(1/2)]
```

When you use `'real'`, `qr` assumes that all symbolic variables represent real numbers, and can return shorter results:

```
[Q,R] = qr(A,'real')

Q =
[     10^(1/2)/10, -((3*x)/10 - 9/5)/(x^2/10 - (6*x)/5...
                                         + 18/5)^(1/2)]
[ (3*10^(1/2))/10,      (x/10 - 3/5)/(x^2/10 - (6*x)/5...
                                         + 18/5)^(1/2)]


R =
[ 10^(1/2),         (10^(1/2)*(3*x + 2))/10]
[        0, (x^2/10 - (6*x)/5 + 18/5)^(1/2)]
```

Input Arguments

`A` — Input matrix

m-by-*n* symbolic matrix

Input matrix, specified as an *m*-by-*n* symbolic matrix.

`B` — Input

symbolic vector | symbolic matrix

Input, specified as a symbolic vector or matrix. The number of rows in `B` must be the same as the number of rows in `A`.

Output Arguments

`R` — R part of the QR decomposition

m-by-*n* upper triangular symbolic matrix

R part of the QR decomposition, returned as an *m*-by-*n* upper triangular symbolic matrix.

`Q` — Q part of the QR decomposition

m-by-*m* unitary symbolic matrix

Q part of the QR decomposition, returned as an *m*-by-*m* unitary symbolic matrix.

`P` — Permutation information

matrix of double-precision values

Permutation information, returned as a matrix of double-precision values, such that `A*P = Q*R`.

`p` — Permutation information

vector of double-precision values

Permutation information, returned as a vector of double-precision values, such that `A(:,p) = Q*R`.

`C` — Matrix representing solution of matrix equation `A*X = B`

symbolic matrix

Matrix representing solution of matrix equation `A*X = B`, returned as a symbolic matrix, such that `C = Q'*B`.

Definitions

QR Factorization of Matrix

The QR factorization expresses an `m`-by-`n` matrix `A` as `A = Q*R`. Here, `Q` is an `m`-by-`m` unitary matrix, and `R` is an `m`-by-`n` upper triangular matrix. If the components of `A` are real numbers, then `Q` is an orthogonal matrix.

Tips

- The upper triangular matrix `A` satisfies the following condition: `R = chol(A'*A)`.
- The arguments `'econ'` and `0` only affect the shape of the returned matrices.
- Calling `qr` for numeric matrices that are not symbolic objects (not created by `sym`, `syms`, or `vpa`) invokes the MATLAB `qr` function.

- If you use `'matrix'` instead of `'vector'`, then `qr` returns permutation matrices, as it does by default. If you use `'matrix'` and `'econ'`, then `qr` throws an error.

See Also

`chol` | `eig` | `lu` | `svd`

Introduced in R2014a

quorem

Quotient and remainder

Syntax

```
[Q,R] = quorem(A,B,var)
[Q,R] = quorem(A,B)
```

Description

`[Q,R] = quorem(A,B,var)` divides `A` by `B` and returns the quotient `Q` and remainder `R` of the division, such that `A = Q*B + R`. This syntax regards `A` and `B` as polynomials in the variable `var`.

If `A` and `B` are matrices, `quorem` performs elements-wise division, using `var` are a variable. It returns the quotient `Q` and remainder `R` of the division, such that `A = Q.*B + R`.

`[Q,R] = quorem(A,B)` uses the variable determined by `symvar(A,1)`. If `symvar(A, 1)` returns an empty symbolic object `sym([])`, then `quorem` uses the variable determined by `symvar(B,1)`.

If both `symvar(A,1)` and `symvar(B,1)` are empty, then `A` and `B` must both be integers or matrices with integer elements. In this case, `quorem(A,B)` returns symbolic integers `Q` and `R`, such that `A = Q*B + R`. If `A` and `B` are matrices, then `Q` and `R` are symbolic matrices with integer elements, such that `A = Q.*B + R`, and each element of `R` is smaller in absolute value than the corresponding element of `B`.

Examples

Divide Multivariate Polynomials

Compute the quotient and remainder of the division of these multivariate polynomials with respect to the variable y:

```
syms x y
p1 = x^3*y^4 - 2*x*y + 5*x + 1;
p2 = x*y;
[q, r] = quorem(p1, p2, y)

q =
x^2*y^3 - 2

r =
5*x + 1
```

Divide Univariate Polynomials

Compute the quotient and remainder of the division of these univariate polynomials:

```
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
```

Divide Integers

Compute the quotient and remainder of the division of these integers:

```
[q, r] = quorem(sym(10)^5, sym(985))

q =
101
```

```
r =
515
```

Input Arguments

`A` — Dividend (numerator)
symbolic integer | polynomial | symbolic vector | symbolic matrix

Dividend (numerator), specified as a symbolic integer, polynomial, or a vector or matrix of symbolic integers or polynomials.

`B` — Divisor (denominator)
symbolic integer | polynomial | symbolic vector | symbolic matrix

Divisor (denominator), specified as a symbolic integer, polynomial, or a vector or matrix of symbolic integers or polynomials.

`var` — Polynomial variable
symbolic variable

Polynomial variable, specified as a symbolic variable.

Output Arguments

`Q` — Quotient of the division
symbolic integer | symbolic expression | symbolic vector | symbolic matrix

Quotient of the division, returned as a symbolic integer, expression, or a vector or matrix of symbolic integers or expressions.

`R` — Remainder of the division
symbolic integer | symbolic expression | symbolic vector | symbolic matrix

Remainder of the division, returned as a symbolic integer, expression, or a vector or matrix of symbolic integers or expressions.

See Also

`deconv` | `mod`

Introduced before R2006a

rank

Find rank of symbolic matrix

Syntax

```
rank(A)
```

Description

`rank(A)` returns the rank of symbolic matrix `A`.

Examples

Find Rank of Matrix

```
syms a b c d
A = [a b; c d];
rank(A)
```

```
ans =
     2
```

Rank of Symbolic Matrices Is Exact

Symbolic calculations return the exact rank of a matrix while numeric calculations can suffer from round-off errors. This exact calculation is useful for ill-conditioned matrices, such as the Hilbert matrix. The rank of a Hilbert matrix of order n is n.

Find the rank of the Hilbert matrix of order `15` numerically. Then convert the numeric matrix to a symbolic matrix using `sym` and find the rank symbolically.

```
H = hilb(15);
rank(H)
rank(sym(H))
```

```
ans =
    12
ans =
    15
```

The symbolic calculation returns the correct rank of 15. The numeric calculation returns an incorrect rank of 12 due to round-off errors.

Rank Function Does Not Simplify Symbolic Calculations

Consider this matrix

$$A = \begin{bmatrix} 1-\sin^2(x) & \cos^2(x) \\ 1 & 1 \end{bmatrix}.$$

After simplification of `1-sin(x)^2` to `cos(x)^2`, the matrix has a rank of `1`. However, `rank` returns an incorrect rank of `2` because it does not take into account identities satisfied by special functions occurring in the matrix elements. Demonstrate the incorrect result.

```
syms x
A = [1-sin(x) cos(x); cos(x) 1+sin(x)];
rank(A)
```

```
ans =
     2
```

`rank` returns an incorrect result because the outputs of intermediate steps are not simplified. While there is no fail-safe workaround, you can simplify symbolic expressions by using numeric substitution and evaluating the substitution using `vpa`.

Find the correct rank by substituting `x` with a number and evaluating the result using `vpa`.

```
rank(vpa(subs(A,x,1)))
```

```
ans =
     1
```

However, even after numeric substitution, `rank` can return incorrect results due to round-off errors.

Input Arguments

`A` — Input

number | vector | matrix | symbolic number | symbolic vector | symbolic matrix

Input, specified as a number, vector, or matrix or a symbolic number, vector, or matrix.

See Also

`eig` | `null` | `rref` | `size`

Introduced before R2006a

rdivide./

Symbolic array right division

Syntax

```
A./B
rdivide(A,B)
```

Description

`A./B` divides `A` by `B`.

`rdivide(A,B)` is equivalent to `A./B`.

Examples

Divide Scalar by Matrix

Create a `2`-by-`3` matrix.

```
B = sym('b', [2 3])
```

```
B =
[ b1_1, b1_2, b1_3]
[ b2_1, b2_2, b2_3]
```

Divide the symbolic expression `sin(a)` by each element of the matrix `B`.

```
syms a
sin(a)./B
```

```
ans =
[ sin(a)/b1_1, sin(a)/b1_2, sin(a)/b1_3]
[ sin(a)/b2_1, sin(a)/b2_2, sin(a)/b2_3]
```

Divide Matrix by Matrix

Create a 3-by-3 symbolic Hilbert matrix and a 3-by-3 diagonal matrix.

```
H = sym(hilb(3))
d = diag(sym([1 2 3]))
```

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

d =
[ 1, 0, 0]
[ 0, 2, 0]
[ 0, 0, 3]
```

Divide d by H by using the elementwise right division operator .\. This operator divides each element of the first matrix by the corresponding element of the second matrix. The dimensions of the matrices must be the same.

```
d./H
```

```
ans =
[ 1, 0,  0]
[ 0, 6,  0]
[ 0, 0, 15]
```

Divide Expression by Symbolic Function

Divide a symbolic expression by a symbolic function. The result is a symbolic function.

```
syms f(x)
f(x) = x^2;
f1 = (x^2 + 5*x + 6)./f
```

```
f1(x) =
(x^2 + 5*x + 6)/x^2
```

Input Arguments

`A` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

`B` — Input

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a symbolic variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `times` | `transpose`

Introduced before R2006a

read

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.

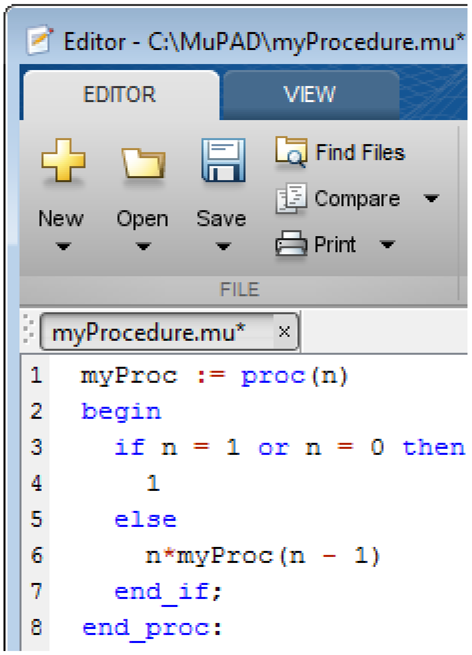
Input Arguments

`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
```

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 4-1355.

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:

```
feval(symengine, 'read',' "myProcedure.mu" ', 'Plain')
```

If your program file contains aliases or uses the aliases predefined by MATLAB, do not use `Plain`:

```
feval(symengine, 'read',' "myProcedure.mu" ')
```

See Also

`evalin` | `feval` | `symengine`

Topics

“Use Your Own MuPAD Procedures” on page 3-62

Introduced in R2011b

real

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`.

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(2*exp(1 + sym(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:

```
syms x
A = [-1 + sym(i), sinh(x); exp(10 + sym(7)*i), exp(sym(pi)*i)];
real(A)
```

```
ans =
[             -1, real(sinh(x))]
[ cos(7)*exp(10),            -1]
```

Tips

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

Alternatives

You can compute the real part of `z` via the conjugate: `real(z)= (z + conj(z))/2`.

See Also

`conj` | `imag` | `in` | `sign` | `signIm`

Introduced before R2006a

rectangularPulse

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)`.

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.

Examples

Find Rectangular Pulse Function

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
```

Edge Values of Rectangular Pulse

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
```

Fixed Rectangular Pulse of Width 1

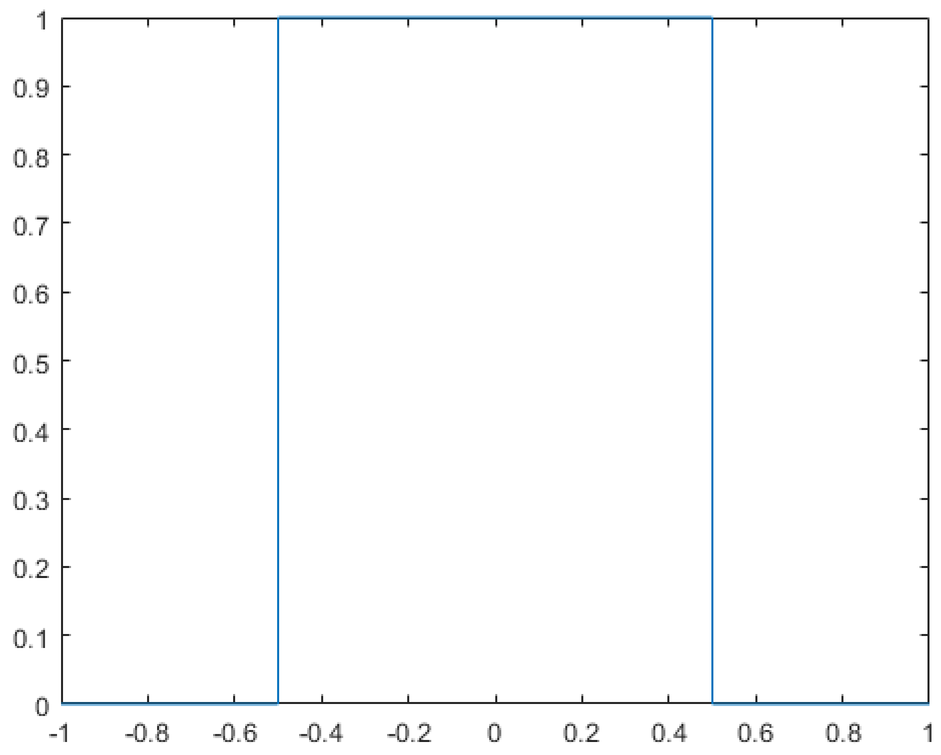
Use `rectangularPulse` with one input argument as a shortcut for computing `rectangularPulse(-1/2, 1/2, x)`:

```
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 Rectangular Pulse Function

Plot the rectangular pulse function using `fplot`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(rectangularPulse(x), [-1 1])
```



Relation Between Heaviside and Rectangular Pulse

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
```

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.

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.

See Also

`dirac` | `heaviside` | `triangularPulse`

Introduced in R2012b

reduceDAEIndex

Convert system of first-order differential algebraic equations to equivalent system of differential index 1

Syntax

```
[newEqs,newVars] = reduceDAEIndex(eqs,vars)
[newEqs,newVars,R] = reduceDAEIndex(eqs,vars)
[newEqs,newVars,R,oldIndex] = reduceDAEIndex(eqs,vars)
```

Description

`[newEqs,newVars] = reduceDAEIndex(eqs,vars)` converts a high-index system of first-order differential algebraic equations `eqs` to an equivalent system `newEqs` of differential index 1.

`reduceDAEIndex` keeps the original equations and variables and introduces new variables and equations. After conversion, `reduceDAEIndex` checks the differential index of the new system by calling `isLowIndexDAE`. If the index of `newEqs` is 2 or higher, then `reduceDAEIndex` issues a warning.

`[newEqs,newVars,R] = reduceDAEIndex(eqs,vars)` returns matrix `R` that expresses the new variables in `newVars` as derivatives of the original variables `vars`.

`[newEqs,newVars,R,oldIndex] = reduceDAEIndex(eqs,vars)` returns the differential index, `oldIndex`, of the original system of DAEs, `eqs`.

Examples

Reduce Differential Index of DAE System

Check if the following DAE system has a low (`0` or `1`) or high (`>1`) differential index. If the index is higher than `1`, then use `reduceDAEIndex` to reduce it.

Create the following system of two differential algebraic equations. Here, the symbolic functions `x(t)`, `y(t)`, and `z(t)` represent the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms  x(t) y(t) z(t) f(t)
eqs = [diff(x) == x + z, diff(y) == f(t), x == y];
vars = [x(t), y(t), z(t)];
```

Use `isLowIndexDAE` to check the differential index of the system. For this system, `isLowIndexDAE` returns `0` (`false`). This means that the differential index of the system is `2` or higher.

```
isLowIndexDAE(eqs, vars)

ans =
  logical
    0
```

Use `reduceDAEIndex` to rewrite the system so that the differential index is `1`. The new system has one additional state variable, `Dyt(t)`.

```
[newEqs, newVars] = reduceDAEIndex(eqs, vars)

newEqs =
 diff(x(t), t) - z(t) - x(t)
               Dyt(t) - f(t)
                 x(t) - y(t)
      diff(x(t), t) - Dyt(t)

newVars =
   x(t)
   y(t)
   z(t)
 Dyt(t)
```

Check if the differential order of the new system is lower than `2`.

```
isLowIndexDAE(newEqs, newVars)

ans =
  logical
     1
```

Reduce the Index and Return More Details

Reduce the differential index of a system that contains two second-order differential algebraic equation. Because the equations are second-order equations, first use `reduceDifferentialOrder` to rewrite the system to a system of first-order DAEs.

Create the following system of two second-order DAEs. Here, `x(t)`, `y(t)`, and `F(t)` are the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms t x(t) y(t) F(t) r g
eqs = [diff(x(t), t, t) == -F(t)*x(t),...
       diff(y(t), t, t) == -F(t)*y(t) - g,...
       x(t)^2 + y(t)^2 == r^2 ];
vars = [x(t), y(t), F(t)];
```

Rewrite this system so that all equations become first-order differential equations. The `reduceDifferentialOrder` function replaces the second-order DAE by two first-order expressions by introducing the new variables `Dxt(t)` and `Dyt(t)`. It also replaces the first-order equations by symbolic expressions.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
     diff(Dxt(t), t) + F(t)*x(t)
 diff(Dyt(t), t) + g + F(t)*y(t)
           x(t)^2 + y(t)^2 - r^2
          Dxt(t) - diff(x(t), t)
          Dyt(t) - diff(y(t), t)
 
vars =
   x(t)
   y(t)
   F(t)
 Dxt(t)
 Dyt(t)
```

Use `reduceDAEIndex` to rewrite the system so that the differential index is `1`.

```
[eqs, vars, R, originalIndex] = reduceDAEIndex(eqs, vars)
```

```
eqs =
                                                   Dxtt(t) + F(t)*x(t)
                                                   g + Dytt(t) + F(t)*y(t)
                                                     x(t)^2 + y(t)^2 - r^2
                                                          Dxt(t) - Dxt1(t)
                                                          Dyt(t) - Dyt1(t)
                                             2*Dxt1(t)*x(t) + 2*Dyt1(t)*y(t)
 2*Dxt1t(t)*x(t) + 2*Dxt1(t)^2 + 2*Dyt1(t)^2 + 2*y(t)*diff(Dyt1(t), t)
                                                        Dxtt(t) - Dxt1t(t)
                                                Dytt(t) - diff(Dyt1(t), t)
                                                   Dyt1(t) - diff(y(t), t)

vars =
     x(t)
     y(t)
     F(t)
   Dxt(t)
   Dyt(t)
  Dytt(t)
  Dxtt(t)
  Dxt1(t)
  Dyt1(t)
 Dxt1t(t)

R =
[  Dytt(t),  diff(Dyt(t), t)]
[  Dxtt(t),  diff(Dxt(t), t)]
[  Dxt1(t),    diff(x(t), t)]
[  Dyt1(t),    diff(y(t), t)]
[ Dxt1t(t), diff(x(t), t, t)]

originalIndex =
     3
```

Use `reduceRedundancies` to shorten the system.

```
[eqs, vars] = reduceRedundancies(eqs, vars)
```

```
eqs =
                                           Dxtt(t) + F(t)*x(t)
                                       g + Dytt(t) + F(t)*y(t)
                                         x(t)^2 + y(t)^2 - r^2
                                 2*Dxt(t)*x(t) + 2*Dyt(t)*y(t)
    2*Dxtt(t)*x(t) + 2*Dytt(t)*y(t) + 2*Dxt(t)^2 + 2*Dyt(t)^2
                                     Dytt(t) - diff(Dyt(t), t)
                                        Dyt(t) - diff(y(t), t)
```

```
vars =
    x(t)
    y(t)
    F(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Input Arguments

`eqs` — System of first-order DAEs
vector of symbolic equations | vector of symbolic expressions

System of first-order DAEs, specified as a vector of symbolic equations or expressions.

`vars` — State variables
vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]`

Output Arguments

`newEqs` — System of first-order DAEs of differential index 1
column vector of symbolic expressions

System of first-order DAEs of differential index 1, returned as a column vector of symbolic expressions.

`newVars` — Extended set of variables
column vector of symbolic function calls

Extended set of variables, returned as a column vector of symbolic function calls. This vector includes the original state variables `vars` followed by the generated variables that replace the second- and higher-order derivatives in `eqs`.

`R` — Relations between new and original variables
symbolic matrix

Relations between new and original variables, returned as a symbolic matrix with two columns. The first column contains the new variables. The second column contains their definitions as derivatives of the original variables `vars`.

`oldIndex` — Differential index of original DAE system
integer

Differential index of original DAE system, returned as an integer or `NaN`.

Algorithms

The implementation of `reduceDAEIndex` uses the Pantelides algorithm. This algorithm reduces higher-index systems to lower-index systems by selectively adding differentiated forms of the original equations. The Pantelides algorithm can underestimate the differential index of a new system, and therefore, can fail to reduce the differential index to `1`. In this case, `reduceDAEIndex` issues a warning and, for the syntax with four output arguments, returns the value of `oldIndex` as `NaN`. The `reduceDAEToODE` function uses more reliable, but slower Gaussian elimination. Note that `reduceDAEToODE` requires the DAE system to be semilinear.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

“Solve Differential Algebraic Equations (DAEs)” on page 2-193

Introduced in R2014b

reduceDAEToODE

Convert system of first-order semilinear differential algebraic equations to equivalent system of differential index 0

Syntax

```
newEqs = reduceDAEToODE(eqs,vars)
[newEqs,constraintEqs] = reduceDAEToODE(eqs,vars)
[newEqs,constraintEqs,oldIndex] = reduceDAEToODE(eqs,vars)
```

Description

`newEqs = reduceDAEToODE(eqs,vars)` converts a high-index system of first-order semilinear algebraic equations `eqs` to an equivalent system of ordinary differential equations, `newEqs`. The differential index of the new system is `0`, that is, the Jacobian of `newEqs` with respect to the derivatives of the variables in `vars` is invertible.

`[newEqs,constraintEqs] = reduceDAEToODE(eqs,vars)` returns a vector of constraint equations.

`[newEqs,constraintEqs,oldIndex] = reduceDAEToODE(eqs,vars)` returns the differential index `oldIndex` of the original system of semilinear DAEs, `eqs`.

Examples

Convert DAE System to Implicit ODE System

Convert a system of differential algebraic equations (DAEs) to a system of implicit ordinary differential equations (ODEs).

Create the following system of two differential algebraic equations. Here, the symbolic functions `x(t)`, `y(t)`, and `z(t)` represent the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x(t) y(t) z(t)
eqs = [diff(x,t)+x*diff(y,t) == y,...
       x*diff(x, t)+x^2*diff(y) == sin(x),...
       x^2 + y^2 == t*z];
vars = [x(t), y(t), z(t)];
```

Use `reduceDAEToODE` to rewrite the system so that the differential index is `0`.

```
newEqs = reduceDAEToODE(eqs, vars)
```

```
newEqs =
                             x(t)*diff(y(t), t) - y(t) + diff(x(t), t)
                diff(x(t), t)*(cos(x(t)) - y(t)) - x(t)*diff(y(t), t)
 z(t) - 2*x(t)*diff(x(t), t) - 2*y(t)*diff(y(t), t) + t*diff(z(t), t)
```

Reduce System and Return More Details

Check if the following DAE system has a low (`0` or `1`) or high (`>1`) differential index. If the index is higher than `1`, first try to reduce the index by using `reduceDAEIndex` and then by using `reduceDAEToODE`.

Create the system of differential algebraic equations. Here, the functions `x1(t)`, `x2(t)`, and `x3(t)` represent the state variables of the system. The system also contains the functions `q1(t)`, `q2(t)`, and `q3(t)`. These functions do not represent state variables. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x1(t) x2(t) x3(t) q1(t) q2(t) q3(t)
eqs = [diff(x2) == q1 - x1,
       diff(x3) == q2 - 2*x2 - t*(q1-x1),
       q3 - t*x2 - x3];
vars = [x1(t), x2(t), x3(t)];
```

Use `isLowIndexDAE` to check the differential index of the system. For this system, `isLowIndexDAE` returns `0` (`false`). This means that the differential index of the system is `2` or higher.

```
isLowIndexDAE(eqs, vars)
```

```
ans =
  logical
     0
```

Use `reduceDAEIndex` as your first attempt to rewrite the system so that the differential index is `1`. For this system, `reduceDAEIndex` issues a warning because it cannot reduce the differential index of the system to `0` or `1`.

```
[newEqs, newVars] = reduceDAEIndex(eqs, vars)
```

```
Warning: Index of reduced DAEs is larger than 1.

newEqs =
                      x1(t) - q1(t) + diff(x2(t), t)
       Dx3t(t) - q2(t) + 2*x2(t) + t*(q1(t) - x1(t))
                             q3(t) - x3(t) - t*x2(t)
 diff(q3(t), t) - x2(t) - t*diff(x2(t), t) - Dx3t(t)

newVars =
   x1(t)
   x2(t)
   x3(t)
 Dx3t(t)
```

If `reduceDAEIndex` cannot reduce the semilinear system so that the index is `0` or `1`, try using `reduceDAEToODE`. This function can be much slower, therefore it is not recommended as a first choice. Use the syntax with two output arguments to also return the constraint equations.

```
[newEqs, constraintEqs] = reduceDAEToODE(eqs, vars)
```

```
newEqs =
                                             x1(t) - q1(t) + diff(x2(t), t)
                     2*x2(t) - q2(t) + t*q1(t) - t*x1(t) + diff(x3(t), t)
 diff(x1(t), t) - diff(q1(t), t) + diff(q2(t), t, t) - diff(q3(t), t, t, t)

constraintEqs =
 x1(t) - q1(t) + diff(q2(t), t) - diff(q3(t), t, t)
                            x3(t) - q3(t) + t*x2(t)
                    x2(t) - q2(t) + diff(q3(t), t)
```

Use the syntax with three output arguments to return the new equations, constraint equations, and the differential index of the original system, `eqs`.

```
[newEqs, constraintEqs, oldIndex] = reduceDAEToODE(eqs, vars)
```

```
newEqs =
                                             x1(t) - q1(t) + diff(x2(t), t)
```

```
                      2*x2(t) - q2(t) + t*q1(t) - t*x1(t) + diff(x3(t), t)
 diff(x1(t), t) - diff(q1(t), t) + diff(q2(t), t, t) - diff(q3(t), t, t, t)

constraintEqs =
 x1(t) - q1(t) + diff(q2(t), t) - diff(q3(t), t, t)
                            x3(t) - q3(t) + t*x2(t)
                     x2(t) - q2(t) + diff(q3(t), t)

oldIndex =
     3
```

Input Arguments

`eqs` **— System of first-order semilinear DAEs**
vector of symbolic equations | vector of symbolic expressions

System of first-order semilinear DAEs, specified as a vector of symbolic equations or expressions.

`vars` **— State variables**
vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]` or `[x(t);y(t)]`

Output Arguments

`newEqs` **— System of implicit ordinary differential equations**
column vector of symbolic expressions

System of implicit ordinary differential equations, returned as a column vector of symbolic expressions. The differential index of this system is `0`.

`constraintEqs` **— Constraint equations encountered during system reduction**
column vector of symbolic expressions

Constraint equations encountered during system reduction, returned as a column vector of symbolic expressions. These expressions depend on the variables `vars`, but not on their derivatives. The constraints are conserved quantities of the differential equations in `newEqs`, meaning that the time derivative of each constraint vanishes modulo the equations in `newEqs`.

You can use these equations to determine consistent initial conditions for the DAE system.

`oldIndex` — Differential index of original DAE system `eqs`
integer

Differential index of original DAE system `eqs`, returned as an integer.

Algorithms

The implementation of `reduceDAEToODE` is based on Gaussian elimination. This algorithm is more reliable than the Pantelides algorithm used by `reduceDAEIndex`, but it can be much slower.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDifferentialOrder` | `reduceRedundancies`

Topics

"Solve Semilinear DAE System" on page 2-205

Introduced in R2014b

reduceDifferentialOrder

Reduce system of higher-order differential equations to equivalent system of first-order differential equations

Syntax

```
[newEqs,newVars] = reduceDifferentialOrder(eqs,vars)
[newEqs,newVars,R] = reduceDifferentialOrder(eqs,vars)
```

Description

`[newEqs,newVars] = reduceDifferentialOrder(eqs,vars)` rewrites a system of higher-order differential equations `eqs` as a system of first-order differential equations `newEqs` by substituting derivatives in `eqs` with new variables. Here, `newVars` consists of the original variables `vars` augmented with these new variables.

`[newEqs,newVars,R] = reduceDifferentialOrder(eqs,vars)` returns the matrix `R` that expresses the new variables in `newVars` as derivatives of the original variables `vars`.

Examples

Reduce Differential Order of DAE System

Reduce a system containing higher-order DAEs to a system containing only first-order DAEs.

Create the system of differential equations, which includes a second-order expression. Here, `x(t)` and `y(t)` are the state variables of the system, and `c1` and `c2` are parameters. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x(t) y(t) c1 c2
eqs = [diff(x(t), t, t) + sin(x(t)) + y(t) == c1*cos(t),...
```

```
                                diff(y(t), t) == c2*x(t)];
vars = [x(t), y(t)];

[newEqs, newVars] = reduceDifferentialOrder(eqs, vars)
```

Rewrite this system so that all equations become first-order differential equations. The `reduceDifferentialOrder` function replaces the higher-order DAE by first-order expressions by introducing the new variable `Dxt(t)`. It also represents all equations as symbolic expressions.

```
[newEqs, newVars] = reduceDifferentialOrder(eqs, vars)

newEqs =
 sin(x(t)) + y(t) + diff(Dxt(t), t) - c1*cos(t)
                        diff(y(t), t) - c2*x(t)
                         Dxt(t) - diff(x(t), t)


newVars =
   x(t)
   y(t)
 Dxt(t)
```

Show Relations Between Generated and Original Variables

Reduce a system containing a second- and a third-order expression to a system containing only first-order DAEs. In addition, return a matrix that expresses the variables generated by `reduceDifferentialOrder` via the original variables of this system.

Create a system of differential equations, which includes a second- and a third-order expression. Here, `x(t)` and `y(t)` are the state variables of the system. Specify the equations and variables as two symbolic vectors: equations as a vector of symbolic equations, and variables as a vector of symbolic function calls.

```
syms x(t) y(t) f(t)
eqs = [diff(x(t),t,t) == diff(f(t),t,t,t), diff(y(t),t,t,t) == diff(f(t),t,t)];
vars = [x(t), y(t)];
```

Call `reduceDifferentialOrder` with three output arguments. This syntax returns matrix `R` with two columns: the first column contains the new variables, and the second column expresses the new variables as derivatives of the original variables, `x(t)` and `y(t)`.

```
[newEqs, newVars, R] = reduceDifferentialOrder(eqs, vars)

newEqs =
 diff(Dxt(t), t) - diff(f(t), t, t, t)
   diff(Dytt(t), t) - diff(f(t), t, t)
                Dxt(t) - diff(x(t), t)
                Dyt(t) - diff(y(t), t)
             Dytt(t) - diff(Dyt(t), t)

newVars =
    x(t)
    y(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)

R =
[  Dxt(t),    diff(x(t), t)]
[  Dyt(t),    diff(y(t), t)]
[ Dytt(t), diff(y(t), t, t)]
```

Input Arguments

eqs — System containing higher-order differential equations
vector of symbolic equations | vector of symbolic expressions

System containing higher-order differential equations, specified as a vector of symbolic equations or expressions.

vars — Variables of original differential equations
vector of symbolic functions | vector of symbolic function calls

Variables of original differential equations, specified as a vector of symbolic functions, or function calls, such as `x(t)`.

Example: `[x(t),y(t)]`

Output Arguments

newEqs — System of first-order differential equations
column vector of symbolic expressions

System of first-order differential equations, returned as a column vector of symbolic expressions.

`newVars` — Extended set of variables
column vector of symbolic function calls

Extended set of variables, returned as a column vector of symbolic function calls. This vector includes the original state variables `vars` followed by the generated variables that replace the higher-order derivatives in `eqs`.

`R` — Relations between new and original variables
symbolic matrix

Relations between new and original variables, returned as a symbolic matrix with two columns. The first column contains the new variables `newVars`. The second column contains their definition as derivatives of the original variables `vars`.

See Also
`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceRedundancies`

Topics
“Solve Differential Algebraic Equations (DAEs)” on page 2-193

Introduced in R2014b

reduceRedundancies

Simplify system of first-order differential algebraic equations by eliminating redundant equations and variables

Syntax

```
[newEqs,newVars] = reduceRedundancies(eqs,vars)
[newEqs,newVars,R] = reduceRedundancies(eqs,vars)
```

Description

`[newEqs,newVars] = reduceRedundancies(eqs,vars)` eliminates simple equations from the system of first-order differential algebraic equations `eqs`. It returns a column vector `newEqs` of symbolic expressions and a column vector `newVars` of those variables that remain in the new DAE system `newEqs`. The expressions in `newEqs` represent equations with a zero right side.

`[newEqs,newVars,R] = reduceRedundancies(eqs,vars)` returns a structure array `R` containing information on the eliminated equations and variables.

Examples

Shorten DAE System by Removing Redundant Equations

Use `reduceRedundancies` to simplify a system of five differential algebraic equations in four variables to a system of two equations in two variables.

Create the following system of five differential algebraic equations in four state variables `x1(t)`, `x2(t)`, `x3(t)`, and `x4(t)`. The system also contains symbolic parameters `a1`, `a2`, `a3`, `a4`, `b`, `c`, and the function `f(t)` that is not a state variable.

```
syms x1(t) x2(t) x3(t) x4(t) a1 a2 a3 a4 b c f(t)
eqs = [a1*diff(x1(t),t)+a2*diff(x2(t),t) == b*x4(t),...
```

```
       a3*diff(x2(t),t)+a4*diff(x3(t),t) == c*x4(t),...
       x1(t) == 2*x2(t),...
       x4(t) == f(t), ...
       f(t) == sin(t)];
vars = [x1(t), x2(t), x3(t), x4(t)];
```

Use `reduceRedundancies` to eliminate redundant equations and corresponding state variables.

```
[newEqs, newVars] = reduceRedundancies(eqs, vars)
```

```
newEqs =
 a1*diff(x1(t), t) + (a2*diff(x1(t), t))/2 - b*f(t)
 (a3*diff(x1(t), t))/2 + a4*diff(x3(t), t) - c*f(t)

newVars =
 x1(t)
 x3(t)
```

Obtain Information About Eliminated Equations

Call `reduceRedundancies` with three output arguments to simplify a system and return information about eliminated equations.

Create the following system of five differential algebraic equations in four state variables `x1(t)`, `x2(t)`, `x3(t)`, and `x4(t)`. The system also contains symbolic parameters `a1`, `a2`, `a3`, `a4`, `b`, `c`, and the function `f(t)` that is not a state variable.

```
syms x1(t) x2(t) x3(t) x4(t) a1 a2 a3 a4 b c f(t)
eqs = [a1*diff(x1(t),t)+a2*diff(x2(t),t) == b*x4(t),...
       a3*diff(x2(t),t)+a4*diff(x3(t),t) == c*x4(t),...
       x1(t) == 2*x2(t),...
       x4(t) == f(t), ...
       f(t) == sin(t)];
vars = [x1(t), x2(t), x3(t), x4(t)];
```

Call `reduceRedundancies` with three output variables.

```
[newEqs, newVars, R] = reduceRedundancies(eqs, vars)
```

```
newEqs =
 a1*diff(x1(t), t) + (a2*diff(x1(t), t))/2 - b*f(t)
 (a3*diff(x1(t), t))/2 + a4*diff(x3(t), t) - c*f(t)
```

```
newVars =
 x1(t)
 x3(t)

R =
  struct with fields:

      solvedEquations: [2×1 sym]
    constantVariables: [1×2 sym]
    replacedVariables: [1×2 sym]
       otherEquations: [1×1 sym]
```

Here, `R` is a structure array with four fields. The `solvedEquations` field contains equations that `reduceRedundancies` used to replace those state variables from `vars` that do not appear in `newEqs`.

```
R.solvedEquations
```

```
ans =
 x1(t) - 2*x2(t)
    x4(t) - f(t)
```

The `constantVariables` field contains a matrix with the following two columns. The first column contains those state variables from `vars` that `reduceRedundancies` replaced by constant values. The second column contains the corresponding constant values.

```
R.constantVariables
```

```
ans =
[ x4(t), f(t)]
```

The `replacedVariables` field contains a matrix with the following two columns. The first column contains those state variables from `vars` that `reduceRedundancies` replaced by expressions in terms of other variables. The second column contains the corresponding values of the eliminated variables.

```
R.replacedVariables
```

```
ans =
[ x2(t), x1(t)/2]
```

The `otherEquations` field contains those equations from `eqs` that do not contain any of the state variables `vars`.

```
R.otherEquations

ans =
f(t) - sin(t)
```

Input Arguments

`eqs` — System of first-order DAEs
vector of symbolic equations | vector of symbolic expressions

System of first-order DAEs, specified as a vector of symbolic equations or expressions.

`vars` — State variables
vector of symbolic functions | vector of symbolic function calls

State variables, specified as a vector of symbolic functions or function calls, such as `x(t)`.

Example: `[x(t),y(t)]`

Output Arguments

`newEqs` — System of first-order DAEs
column vector of symbolic expressions

System of first-order DAEs, returned as a column vector of symbolic expressions

`newVars` — Reduced set of variables
column vector of symbolic function calls

Reduced set of variables, returned as a column vector of symbolic function calls.

`R` — Information about eliminated variables
structure array

Information about eliminated variables, returned as a structure array. To access this information, use:

- `R.solvedEquations` to return a symbolic column vector of all equations that `reduceRedundancies` used to replace those state variables that do not appear in `newEqs`.

- `R.constantVariables` to return a matrix with the following two columns. The first column contains those original state variables of the vector `vars` that were eliminated and replaced by constant values. The second column contains the corresponding constant values.
- `R.replacedVariables` to return a matrix with the following two columns. The first column contains those original state variables of the vector `vars` that were eliminated and replaced in terms of other variables. The second column contains the corresponding values of the eliminated variables.
- `R.otherEquations` to return a column vector containing all original equations `eqs` that do not contain any of the input variables `vars`.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `massMatrixForm` | `odeFunction` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder`

Topics

"Solve Differential Algebraic Equations (DAEs)" on page 2-193

Introduced in R2014b

rem

Remainder after division

Syntax

```
rem(a,b)
```

Description

`rem(a,b)` finds the remainder after division. If `b <> 0`, then `rem(a,b) = a - fix(a/b)*b`. If `b = 0` or `b = Inf` or `b = -Inf`, then `rem` returns `NaN`.

The `rem` function does not support complex numbers: all values must be real numbers.

To find the remainder after division of polynomials, use `quorem`.

Examples

Divide Integers by Integers

Find the remainder after division in case both the dividend and divisor are integers.

Find the modulus after division for these numbers.

```
[rem(sym(27), 4), rem(sym(27), -4), rem(sym(-27), 4), rem(sym(-27), -4)]
```

```
ans =
[ 3, 3, -3, -3]
```

Divide Rationals by Integers

Find the remainder after division in case the dividend is a rational number, and the divisor is an integer.

Find the remainder after division for these numbers.

```
[rem(sym(22/3), 5), rem(sym(1/2), -7), rem(sym(27/6), -11)]

ans =
[ 7/3, 1/2, 9/2]
```

Divide Elements of Matrices

For vectors and matrices, `rem` finds the remainder after division element-wise. Nonscalar arguments must be the same size.

Find the remainder after division for the elements of these two matrices.

```
A = sym([27, 28; 29, 30]);
B = sym([2, 3; 4, 5]);
rem(A,B)

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

Find the remainder after division for the elements of matrix `A` and the value `9`. Here, `rem` expands `9` into the `2`-by-`2` matrix with all elements equal to `9`.

```
rem(A,9)

ans =
[ 0, 1]
[ 2, 3]
```

Input Arguments

`a` — Dividend (numerator)

number | symbolic number | vector | matrix

Dividend (numerator), specified as a number, symbolic number, or a vector or matrix of numbers or symbolic numbers.

`b` — Divisor (denominator)

number | symbolic number | vector | matrix

Divisor (denominator), specified as a number, symbolic number, or a vector or matrix of numbers or symbolic numbers.

Tips

- Calling `rem` for numbers that are not symbolic objects invokes the MATLAB `rem` function.
- All nonscalar arguments must be the same size. If one input arguments is nonscalar, then `mod` expands the scalar into a vector or matrix of the same size as the nonscalar argument, with all elements equal to the corresponding scalar.

See Also

`mod` | `quorem`

Introduced before R2006a

removeUnit

Remove unit

Syntax

```
removeUnit(unit)
```

Description

`removeUnit(unit)` removes the symbolic unit `unit`. You can remove only user-defined units created with `newUnit`. You cannot remove predefined units. If `unit` is a vector, `removeUnit` removes all units in `unit`.

Examples

Remove Unit

Remove units you define by using `removeUnit`. Create the unit `warp3`, use the unit in calculations, and then remove the unit.

Define the unit `warp3` as 3 times the speed of light.

```
u = symunit;
warp3 = newUnit('warp3',3*u.c_0)
```

```
warp3 =
[warp3]
```

Convert `1e10` meter per second to `u.warp3`.

```
speed = rewrite(1e10*u.m/u.s,u.warp3)
```

```
speed =
(5000000000/449688687)*[warp3]
```

After calculations, remove the unit `u.warp3` by using `removeUnit`.

```
removeUnit(u.warp3)
```

Conversion to `u.warp3` now throws an error.

- "Units of Measurement Tutorial" on page 2-5
- "Unit Conversions and Unit Systems" on page 2-30
- "Units List" on page 2-13

Input Arguments

`unit` — Unit name
symbolic unit | vector of symbolic units

Unit name, specified as a symbolic unit or a vector of symbolic units.

See Also

`checkUnits` | `isUnit` | `newUnit` | `newUnitSystem` | `symunit`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

reset

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`

Introduced in R2008b

reshape

Reshape symbolic array

Syntax

```
reshape(A,n1,n2)
reshape(A,n1,...,nM)
reshape(A,...,[],...)
reshape(A,sz)
```

Description

`reshape(A,n1,n2)` returns the `n1`-by-`n2` matrix, which has the same elements as `A`. The elements are taken column-wise from `A` to fill in the elements of the `n1`-by-`n2` matrix.

`reshape(A,n1,...,nM)` returns the `n1`-by-`...`-by-`nM` array, which has the same elements as `A`. The elements are taken column-wise from `A` to fill in the elements of the `n1`-by-`...`-by-`nM` array.

`reshape(A,...,[],...)` lets you represent a size value with the placeholder `[]` while calculating the magnitude of that size value automatically. For example, if `A` has size 2-by-6, then `reshape(A,4,[])` returns a 4-by-3 array.

`reshape(A,sz)` reshapes `A` into an array with size specified by `sz`, where `sz` is a vector.

Examples

Reshape Symbolic Row Vector into Column Vector

Reshape `V`, which is a 1-by-4 row vector, into the 4-by-1 column vector `Y` . Here, `V` and `Y` must have the same number of elements.

Create the vector `V`.

```
syms f(x) y
V = [3 f(x) -4 y]
```

```
V =
[ 3, f(x), -4, y]
```

Reshape `V` into `Y`.

```
Y = reshape(V,4,1)
```

```
Y =
   3
f(x)
  -4
   y
```

Alternatively, use `Y = V.'` where `.'` is the nonconjugate transpose.

Reshape Symbolic Matrix

Reshape the 2-by-6 symbolic matrix `M` into a 4-by-3 matrix.

```
M = sym([1 9 4 3 0 1; 3 9 5 1 9 2])
N = reshape(M,4,3)
```

```
M =
[ 1, 9, 4, 3, 0, 1]
[ 3, 9, 5, 1, 9, 2]

N =
[ 1, 4, 0]
[ 3, 5, 9]
[ 9, 3, 1]
[ 9, 1, 2]
```

`M` and `N` must have the same number of elements. `reshape` reads `M` column-wise to fill in the elements of `N` column-wise.

Alternatively, use a size vector to specify the dimensions of the reshaped matrix.

```
sz = [4 3];
N = reshape(M,sz)
```

```
N =
[ 1, 4, 0]
[ 3, 5, 9]
[ 9, 3, 1]
[ 9, 1, 2]
```

Automatically Set Dimension of Reshaped Matrix

When you replace a dimension with the placeholder `[]`, `reshape` calculates the required magnitude of that dimension to reshape the matrix.

Create the matrix `M`.

```
M = sym([1 9 4 3 0 1; 3 9 5 1 9 2])
```

```
M =
[ 1, 9, 4, 3, 0, 1]
[ 3, 9, 5, 1, 9, 2]
```

Reshape `M` into a matrix with three columns.

```
reshape(M,[],3)
```

```
ans =
[ 1, 4, 0]
[ 3, 5, 9]
[ 9, 3, 1]
[ 9, 1, 2]
```

`reshape` calculates that a reshaped matrix of three columns needs four rows.

Reshape Matrix Row-wise

Reshape a matrix row-wise by transposing the result.

Create matrix `M`.

```
syms x
M = sym([1 9 0 sin(x) 2 2; NaN x 5 1 4 7])
```

```
M =
[   1, 9, 0, sin(x), 2, 2]
[ NaN, x, 5,      1, 4, 7]
```

Reshape `M` row-wise by transposing the result.

```
reshape(M,4,3).'
```

```
ans =
[ 1, NaN,      9, x]
[ 0,   5, sin(x), 1]
[ 2,   4,      2, 7]
```

Note that `.'` returns the non-conjugate transpose while `'` returns the conjugate transpose.

Reshape 3-D Array into 2-D Matrix

Reshape the 3-by-3-by-2 array `M` into a 9-by-2 matrix.

`M` has 18 elements. Because a 9-by-2 matrix also has 18 elements, `M` can be reshaped into it. Construct `M`.

```
syms x
M = [sin(x) x 4; 3 2 9; 8 x x];
M(:,:,2) = M'
```

```
M(:,:,1) =
[ sin(x), x, 4]
[      3, 2, 9]
[      8, x, x]
M(:,:,2) =
[ sin(conj(x)), 3,       8]
[      conj(x), 2, conj(x)]
[            4, 9, conj(x)]
```

Reshape `M` into a 9-by-2 matrix.

```
N = reshape(M,9,2)
```

```
N =
[ sin(x), sin(conj(x))]
[      3,      conj(x)]
[      8,            4]
[      x,            3]
[      2,            2]
[      x,            9]
[      4,            8]
[      9,      conj(x)]
[      x,      conj(x)]
```

Use reshape to Break Up Arrays

Use `reshape` instead of loops to break up arrays for further computation. Use `reshape` to break up the vector `V` to find the product of every three elements.

Create vector `V`.

```
syms x
V = [exp(x) 1 3 9 x 2 7 7 1 8 x^2 3 4 sin(x) x]
```

```
V =
[ exp(x), 1, 3, 9, x, 2, 7, 7, 1, 8, x^2, 3, 4, sin(x), x]
```

Specify `3` for the number of rows. Use the placeholder `[]` for the number of columns. This lets `reshape` automatically calculate the number of columns required for three rows.

```
M = prod( reshape(V,3,[]) )
```

```
M =
[ 3*exp(x), 18*x, 49, 24*x^2, 4*x*sin(x)]
```

`reshape` calculates that five columns are required for a matrix of three rows. `prod` then multiples the elements of each column to return the result.

Input Arguments

`A` — Input array

symbolic vector | symbolic matrix | symbolic multidimensional array

Input array, specified as a symbolic vector, matrix, or multidimensional array.

`n1,n2` — Dimensions of reshaped matrix

comma-separated scalars

Dimensions of reshaped matrix, specified as comma-separated scalars. For example, `reshape(A,3,2)` returns a 3-by-2 matrix. The number of elements in the output array specified by `n1,n2` must be equal to `numel(A)`.

`n1,...,nM` — Dimensions of reshaped array

comma-separated scalars

Dimensions of reshaped array, specified as comma-separated scalars. For example, `reshape(A,3,2,2)` returns a 3-by-2-by-2 matrix. The number of elements in the output array specified by `n1,...,nM` must be equal to `numel(A)`.

`sz` — Size of reshaped array
numeric vector

Size of reshaped array, specified as a numeric vector. For example, `reshape(A,[3 2])` returns a 3-by-2 matrix. The number of elements in the output array specified by `sz` must be equal to `numel(A)`.

See Also

`colon` | `numel` | `transpose`

Introduced before R2006a

removeUnitSystem

Remove unit system

Syntax

```
removeUnitSystem(unitSystem)
```

Description

`removeUnitSystem(unitSystem)` removes the unit system `unitSystem`. You can remove only user-defined unit systems created with `newUnitSystem`. You cannot remove the predefined unit systems `SI`, `CGS`, and `US`.

Examples

Remove Unit System

Define a unit system, use the unit system to rewrite units, and then remove the unit system by using `removeUnitSystem`.

Define the unit system `mySystem` with SI base units and the derived unit kilowatt hour.

```
u = symunit;
bunits = baseUnits('SI');
dunits = [u.kWh];
mySystem = newUnitSystem('mySystem',bunits,dunits)
```

```
mySystem = 
    "mySystem"
```

Convert 50,000 Joules to derived units of `mySystem` by using `rewrite` with the third argument `'Derived'`. As expected, the result is in kilowatt hour.

```
rewrite(50000*u.J,mySystem,'Derived')
```

```
ans =
(1/72)*[kWh]
```

Remove the unit system `mySystem` by using `removeUnitSystem`.

```
removeUnitSystem(mySystem)
```

Converting units to `mySystem` now throws an error.

- "Units of Measurement Tutorial" on page 2-5
- "Unit Conversions and Unit Systems" on page 2-30
- "Units List" on page 2-13

Input Arguments

`unitSystem` **— Name of unit system**
string | character vector

Name of the unit system, specified as a string or character vector.

See Also

`baseUnits` | `derivedUnits` | `newUnitSystem` | `removeUnit` | `rewrite` | `symunit` | `unitSystems`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

rewrite

Rewrite expression in terms of another function

Syntax

```
rewrite(expr,target)
rewrite(unit,unitSystem)
rewrite(unit,unitSystem,'Derived')
___ = rewrite(___,Name,Value)
```

Description

`rewrite(expr,target)` rewrites the symbolic expression `expr` in terms of the target function `target`. The rewritten expression is mathematically equivalent to the original expression. If `expr` is a vector or matrix, `rewrite` acts element-wise on `expr`.

`rewrite(unit,unitSystem)` converts the symbolic unit `unit` to the unit system `unitSystem`. By default, the SI, CGS, and US unit systems are available. You can also define custom unit systems.

`rewrite(unit,unitSystem,'Derived')` converts the symbolic unit `unit` to derived units of `unitSystem`.

`___ = rewrite(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Rewrite Between Trigonometric and Exponential Functions

Rewrite any trigonometric function in terms of the exponential function by specifying the target `'exp'`.

```
syms x
sin2exp = rewrite(sin(x), 'exp')
tan2exp = rewrite(tan(x), 'exp')

sin2exp =
(exp(-x*1i)*1i)/2 - (exp(x*1i)*1i)/2

tan2exp =
-(exp(x*2i)*1i - 1i)/(exp(x*2i) + 1)
```

Rewrite the exponential function in terms of any trigonometric function by specifying the trigonometric function as the target. For a full list of targets, see `target`.

```
syms x
exp2sin = rewrite(exp(x), 'sin')
exp2tan = rewrite(-(exp(x*2i)*1i - 1i)/(exp(x*2i) + 1), 'tan')

exp2sin =
1 - 2*sin((x*1i)/2)^2 - sin(x*1i)*1i
exp2tan =
-(((tan(x) - 1i)*1i)/(tan(x) + 1i) + 1i)/((tan(x) - 1i)/(tan(x) + 1i) - 1)
```

Simplify `exp2tan` into the expected form by using `simplify`.

```
exp2tan = simplify(exp2tan)

exp2tan =
tan(x)
```

Rewrite Between Trigonometric Functions

Rewrite any trigonometric function in terms of any other trigonometric function by specifying the target. For a full list of targets, see `target`.

Rewrite `tan(x)` in terms of the sine function by specifying the target `'sin'`.

```
syms x
tan2sin = rewrite(tan(x), 'sin')

tan2sin =
-sin(x)/(2*sin(x/2)^2 - 1)
```

Rewrite Between Symbolic Units

Rewrite a symbolic unit to another unit by using `rewrite`. You can also rewrite to SI units.

Rewrite `5` cm in terms of inches.

```
u = symunit;
length = 5*u.cm;
length = rewrite(length,u.in)
```

```
length =
(250/127)*[in]
```

Rewrite `length` in terms of SI units. The result is in meters.

```
length = rewrite(length,'SI')
```

```
length =
(1/20)*[m]
```

By default, temperatures are assumed to represent temperature differences. To rewrite between absolute temperatures, specify the `Temperature` input as `'absolute'`.

Rewrite `23` degrees Celsius to degrees Kelvin, treating it first as a temperature difference and then as an absolute temperature.

```
u = symunit;
T = 23*u.Celsius;
relK = rewrite(T,u.K,'Temperature','difference')
```

```
relK =
23*[K]
```

```
absK = rewrite(T,u.K,'Temperature','absolute')
```

```
absK =
(5923/20)*[K]
```

For defining unit systems and converting between unit systems, see “Unit Conversions and Unit Systems” on page 2-30.

Rewrite Between Hyperbolic Functions and Trigonometric Functions

Rewrite any hyperbolic function in terms of any trigonometric function by specifying the trigonometric function as the target. For a full list of targets, see `target`.

Rewrite `tanh(x)` in terms of the sine function by specifying the target `'sin'`.

```
syms x
tanh2sin = rewrite(tanh(x), 'sin')
```

```
tanh2sin =
(sin(x*1i)*1i)/(2*sin((x*1i)/2)^2 - 1)
```

Similarly, rewrite trigonometric functions in terms of hyperbolic functions by specifying the hyperbolic function as the target.

Rewrite Between Inverse Trigonometric Functions and Logarithm Function

Rewrite any inverse trigonometric function in terms of the logarithm function by specifying the target `'log'`. For a full list of targets, see `target`.

Rewrite `acos(x)` and `acot(x)` in terms of the `log` function.

```
syms x
acos2log = rewrite(acos(x), 'log')
acot2log = rewrite(acot(x), 'log')
```

```
acos2log =
-log(x + (1 - x^2)^(1/2)*1i)*1i

acot2log =
(log(1 - 1i/x)*1i)/2 - (log(1i/x + 1)*1i)/2
```

Similarly, rewrite the logarithm function in terms of an inverse trigonometric function by specifying the inverse trigonometric function as the target.

Rewrite Elements of Matrix

Rewrite each element of a matrix by calling `rewrite` on the matrix.

Rewrite all elements of a matrix in terms of the `exp` function.

```
syms x
matrix = [sin(x) cos(x); sinh(x) cosh(x)];
rewrite(matrix, 'exp')
```

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

Rewrite Between Sine and Cosine Functions

Rewrite the cosine function in terms of the sine function. Here, `rewrite` replaces the cosine function using the identity `cos(2*x) = 1 - 2*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 `sin(x)` with either $-\sqrt{1-\cos^2(x)}$ or $\sqrt{1-\cos^2(x)}$ because these expressions are not valid for all `x`. However, using the square of these expressions to replace `sin(x)^2` is valid for all `x`. Thus, `rewrite` replaces `sin(x)^2`.

```
syms x
rewrite(sin(x),'cos')
rewrite(sin(x)^2,'cos')
```

```
ans =
sin(x)
ans =
1 - cos(x)^2
```

Input Arguments

`expr` — Input to rewrite

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Input to rewrite, specified as a symbolic number, variable, expression, function, vector, matrix, or multidimensional array.

`target` — Target function
character vector

Target function, specified as a character vector. This table summarizes the rewriting rules for all allowed targets.

Target	Rewrites These Functions	In Terms of These Functions
`'exp'`	All trigonometric and hyperbolic functions including inverse functions	`exp`, `log`
`'log'`	All inverse trigonometric and hyperbolic functions	`log`
`'sincos'`	`tan`, `cot`, `exp`, `sinh`, `cosh`, `tanh`, `coth`	`sin`, `cos`
`'sin'`, `'cos'`, `'tan'`, or `'cot'`	`sin`, `cos`, `exp`, `tan`, `cot`, `sinh`, `cosh`, `tanh`, `coth` except the target	Target trigonometric function
`'sinhcosh'`	`tan`, `cot`, `exp`, `sin`, `cos`, `tanh`, `coth`	`sinh`, `cosh`
`'sinh'`, `'cosh'`, `'tanh'`, `'coth'`	`tan`, `cot`, `exp`, `sin`, `cos`, `sinh`, `cosh`, `tanh`, `coth` except the target	Target hyperbolic function
`'asin'`, `'acos'`, `'atan'`, `'acot'`	`log`, and all inverse trigonometric and inverse hyperbolic functions	Target inverse trigonometric function
`'asinh'`, `'acosh'`, `'atanh'`, `'acoth'`	`log`, and all inverse trigonometric and inverse hyperbolic functions	Target inverse hyperbolic function
`'sqrt'`	`abs(x + 1i*y)`	`sqrt(x^2 + y^2)`
`'heaviside'`	`sign`, `triangularPulse`, `rectangularPulse`	`heaviside`
`'piecewise'`	`abs`, `heaviside`, `sign`, `triangularPulse`, `rectangularPulse`	`piecewise`
Symbolic unit	Units	Target unit

`unit` — Unit of measurement

symbolic unit | array of symbolic units

Unit of measurement, specified as a symbolic unit, or an array of symbolic units.

`unitSystem` — Unit system

string | character vector

Unit system, specified as a string or character vector. By default, the SI, CGS, and US unit systems are available. You can also define custom unit systems. See "Unit Conversions and Unit Systems" on page 2-30.

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`.

Example: `rewrite(23*u.Celsius,u.K,'Temperature','absolute')`

`Temperature` — Assume that temperatures represent absolute temperatures or temperature differences

`'difference'` (default) | `'absolute'`

Assume that temperatures represent absolute temperatures or temperature differences, specified as `'difference'` or `'absolute'`. The `Temperature` argument only affects conversion between units of temperature.

Tips

- `rewrite` replaces symbolic function calls in `expr` with the target function only if the replacement is mathematically valid. Otherwise, it keeps the original function calls.

See Also

`collect` | `combine` | `expand` | `factor` | `horner` | `numden` | `simplify` | `simplifyFraction` | `symunit`

Topics

“Choose Function to Rearrange Expression” on page 2-94

Introduced in R2012a

rhs

Right side (RHS) of equation

Syntax

`rhs(eqn)`

Description

`rhs(eqn)` returns the right side of the symbolic equation `eqn`. The value of `eqn` also can be a symbolic condition, such as `x > 0`. If `eqn` is an array, then `rhs` returns an array of the right sides of the equations in `eqn`.

Examples

Find Right Side of Equation

Find the right side of the equation `2*y == x^2` by using `rhs`.

First, declare the equation.

```
syms x y
eqn = 2*y == x^2
```

```
eqn =
2*y == x^2
```

Find the right side of `eqn` by using `rhs`.

```
rhsEqn = rhs(eqn)
```

```
rhsEqn =
 x^2
```

Find Right Side of Condition

Find the right side of the condition `x < y + 1` by using `rhs`.

First, declare the condition.

```
syms x y
cond = x < y + 1
```

```
cond =
x < y + 1
```

Find the right side of `cond` by using `rhs`.

```
rhsCond = rhs(cond)
```

```
rhsCond =
y + 1
```

Note Conditions that use the > operator are internally rewritten using the < operator. Therefore, `rhs` returns the original left side. For example, `rhs(x > a)` returns `x`.

Find Right Side of Equations in Array

For an array that contains equations and conditions, `rhs` returns an array of the right sides of those equations or conditions. The output array is the same size as the input array.

Find the right side of the equations and conditions in the vector `V`.

```
syms x y
V = [y^2 == x^2, x ~= 0, x*y >= 1]
```

```
V =
[ y^2 == x^2, x ~= 0, 1 <= x*y]
```

```
rhsV = rhs(V)
```

```
rhsV =
[ x^2, 0, x*y]
```

Because any condition using the >= operator is internally rewritten using the <= operator, the sides of the last condition in `V` are exchanged.

Input Arguments

`eqn` — Equation or condition

symbolic equation | symbolic condition | vector of symbolic equations or conditions | matrix of symbolic equations or conditions | multidimensional array of symbolic equations or conditions

Equation or condition, specified as a symbolic equation or condition, or a vector, matrix, or multidimensional array of symbolic equations or conditions.

See Also

`assume` | `children` | `lhs` | `subs`

Introduced in R2017a

root

Represent roots of polynomial

Syntax

```
root(p,x)
root(p,x,k)
```

Description

`root(p,x)` returns a column vector of numbered roots of symbolic polynomial `p` with respect to `x`. Symbolically solving a high-degree polynomial for its roots can be complex or mathematically impossible. In this case, the Symbolic Math Toolbox uses the `root` function to represent the roots of the polynomial.

`root(p,x,k)` represents the `k`th root of symbolic polynomial `p` with respect to `x`.

Examples

Represent Roots of High-Degree Polynomial

Represent the roots of the polynomial x^3+1 using `root`. The `root` function returns a column vector. The elements of this vector represent the three roots of the polynomial.

```
syms x
p = x^3 + 1;
root(p,x)

ans =
 root(x^3 + 1, x, 1)
 root(x^3 + 1, x, 2)
 root(x^3 + 1, x, 3)
```

`root(x^3 + 1, x, 1)` represents the first root of `p`, while `root(x^3 + 1, x, 2)` represents the second root, and so on. Use this syntax to represent roots of high-degree polynomials.

Find Roots of High-Degree Polynomial

When solving a high-degree polynomial, `solve` represents the roots by using `root`. Alternatively, you can either return an explicit solution by using the `MaxDegree` option or return a numerical result by using `vpa`.

Find the roots of `x^3 + 3*x - 16`.

```
syms x
p = x^3 + 3*x - 16;
R = solve(p,x)
```

```
R =
 root(z^3 + 3*z - 16, z, 1)
 root(z^3 + 3*z - 16, z, 2)
 root(z^3 + 3*z - 16, z, 3)
```

Find the roots explicitly by setting the `MaxDegree` option to the degree of the polynomial. Polynomials with a degree greater than `4` do not have explicit solutions.

```
Rexplicit = solve(p,x,'MaxDegree',3)
```

```
Rexplicit =
                   (65^(1/2) + 8)^(1/3) - 1/(65^(1/2) + 8)^(1/3)
 1/(2*(65^(1/2) + 8)^(1/3)) - (65^(1/2) + 8)^(1/3)/2 -...
 (3^(1/2)*(1/(65^(1/2) + 8)^(1/3) + (65^(1/2) + 8)^(1/3))*1i)/2
 1/(2*(65^(1/2) + 8)^(1/3)) - (65^(1/2) + 8)^(1/3)/2 +...
 (3^(1/2)*(1/(65^(1/2) + 8)^(1/3) + (65^(1/2) + 8)^(1/3))*1i)/2
```

Calculate the roots numerically by using `vpa` to convert `R` to high-precision floating point.

```
Rnumeric = vpa(R)
```

```
RRnumeric =
                                        2.1267693318103912337456401562601
 - 1.0633846659051956168728200781301 - 2.5283118563671914055545884653776i
 - 1.0633846659051956168728200781301 + 2.5283118563671914055545884653776i
```

If the call to `root` contains parameters, substitute the parameters with numbers by using `subs` before calling `vpa`.

Use `root` in Symbolic Computations

You can use the `root` function as input to Symbolic Math Toolbox functions such as `simplify`, `subs`, and `diff`.

Simplify an expression containing `root` using the `simplify` function.

```
syms x
r = root(x^6 + x, x, 1);
simplify(sin(r)^2 + cos(r)^2)

ans =
1
```

Substitute for parameters in `root` with numbers using `subs`.

```
syms b
subs(root(x^2 + b*x, x, 1), b, 5)

ans =
root(x^2 + 5*x, x, 1)
```

Substituting for parameters using `subs` is necessary before converting `root` to numeric form using `vpa`.

Differentiate an expression containing `root` with respect to a parameter using `diff`.

```
diff(root(x^2 + b*x, x, 1), b)

ans =
root(b^2*x^2 + b^2*x, x, 1)
```

Find Inverse Laplace Transform of Ratio of Polynomials

Find the inverse Laplace transform of a ratio of two polynomials using `ilaplace`. The inverse Laplace transform is returned in terms of `root`.

```
syms s
G = (s^3 + 1)/(s^6 + s^5 + s^2);
H = ilaplace(G)

H =
t - symsum(exp(root(s3^4 + s3^3 + 1, s3, k)*t)/...
(4*root(s3^4 + s3^3 + 1, s3, k) + 3), k, 1, 4)
```

When you get the `root` function in output, you can use the `root` function as input in subsequent symbolic calculations. However, if a numerical result is required, convert the `root` function to a high-precision numeric result using `vpa`.

Convert the inverse Laplace transform to numeric form using `vpa`.

```
H_vpa = simplify(vpa(H))
```

```
H_vpa =
t +...
0.30881178580997278695808136329347*exp(-1.0189127943851558447865795886366*t)*...
                                   cos(0.60256541999859902604398442197193*t) -...
0.30881178580997278695808136329347*exp(0.5189127943851558447865795886366*t)*...
                                   cos(0.66660984493201857915375880073 3*t) -...
0.69196894793554437794633558135 96*exp(-1.0189127943851558447865795886366*t)*...
                                   sin(0.60256541999859902604398442197193*t) -...
0.16223098826244593894459034019473*exp(0.5189127943851558447865795886366*t)*...
                                   sin(0.66660984493201857915375880073 3*t)
```

Input Arguments

`p` — Symbolic polynomial
symbolic expression

Symbolic polynomial, specified as a symbolic expression.

`x` — Variable
symbolic variable

Variable, specified as a symbolic variable.

`k` — Number of polynomial root
number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic multidimensional array

Number of polynomial root, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, or multidimensional array. When `k` is a nonscalar, `root` acts element-wise on `k`.

Example: `root(f,x,3)` represents the third root of `f`.

See Also

`solve` | `vpa`

Introduced in R2015b

round

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

`ceil` | `fix` | `floor` | `frac`

Introduced before R2006a

rref

Reduced row echelon form of matrix (Gauss-Jordan elimination)

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.

To solve a system of linear equations, use `linsolve`.

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` | `linsolve` | `rank` | `size`

Introduced before R2006a

rsums

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 Middle 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 Middle Riemann sum by using the slider below the graph. The number of terms available ranges from 2 to 128. `f` can be a character vector or a symbolic expression. The height of each rectangle is determined by the value of the function in the middle of each interval.

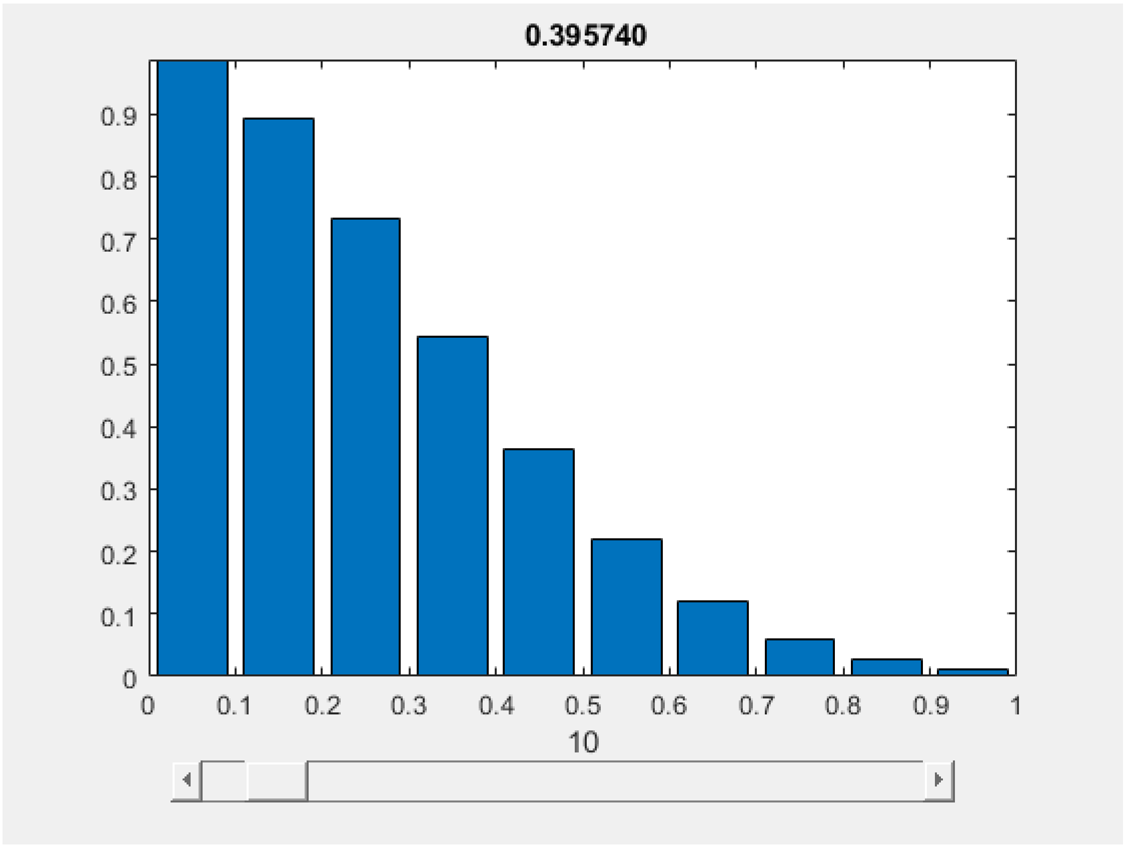
`rsums(f,a,b)` and `rsums(f,[a,b])` approximates the integral for *x* from `a` to `b`.

Examples

Visualize Riemann Sums

Use `rsums(exp(-5*x^2))` to create the following plot.

```
syms x
rsums(exp(-5*x^2))
```



Introduced before R2006a

sec

Symbolic secant function

Syntax

```
sec(X)
```

Description

`sec(X)` returns the secant function on page 4-1424 of `X`.

Examples

Secant Function for Numeric and Symbolic Arguments

Depending on its arguments, `sec` returns floating-point or exact symbolic results.

Compute the secant function for these numbers. Because these numbers are not symbolic objects, `sec` returns floating-point results.

```
A = sec([-2, -pi, pi/6, 5*pi/7, 11])
```

```
A =
   -2.4030   -1.0000    1.1547   -1.6039  225.9531
```

Compute the secant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sec` returns unresolved symbolic calls.

```
symA = sec(sym([-2, -pi, pi/6, 5*pi/7, 11]))
```

```
symA =
[ 1/cos(2), -1, (2*3^(1/2))/3, -1/cos((2*pi)/7), 1/cos(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

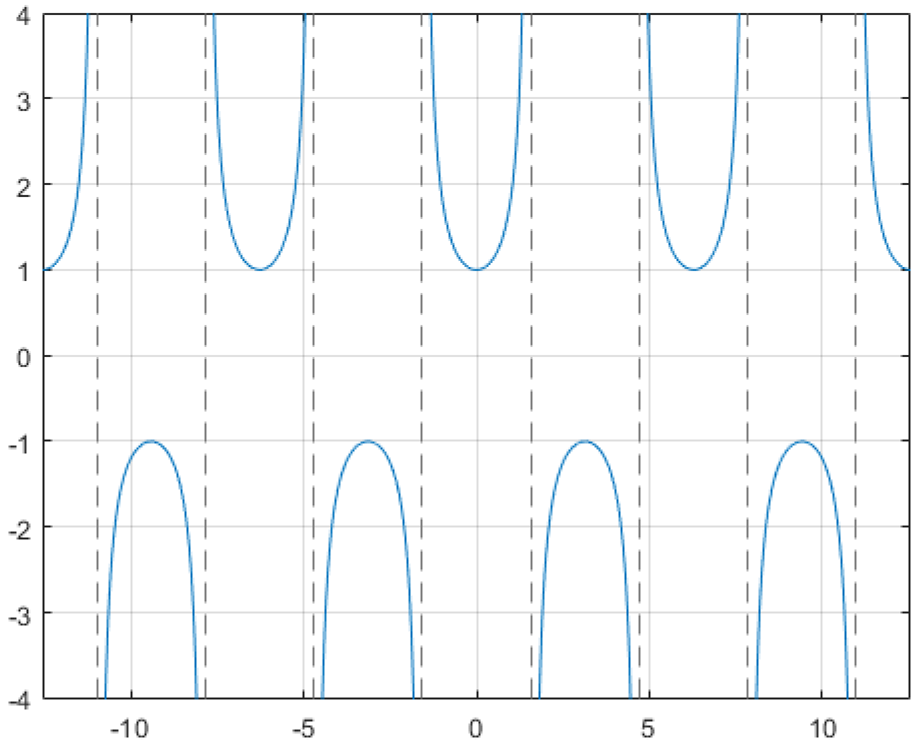
```
vpa(symA)
```

```
ans =
[ -2.4029979617223809897546004014201,...
-1.0,...
1.1547005383792515290182975610039,...
-1.6038754716096765049444092780298,...
225.95305931402493269037542703557]
```

Plot Secant Function

Plot the secant function on the interval from -4π to 4π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sec(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Secant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `sec`.

Find the first and second derivatives of the secant function:

```
syms x
diff(sec(x), x)
diff(sec(x), x, x)
```

```
ans =
sin(x)/cos(x)^2
```

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

Find the indefinite integral of the secant function:

```
int(sec(x), x)
```

```
ans =
log(1/cos(x)) + log(sin(x) + 1)
```

Find the Taylor series expansion of `sec(x)`:

```
taylor(sec(x), x)
```

```
ans =
(5*x^4)/24 + x^2/2 + 1
```

Rewrite the secant function in terms of the exponential function:

```
rewrite(sec(x), 'exp')
```

```
ans =
1/(exp(-x*1i)/2 + exp(x*1i)/2)
```

Evaluate Units with `sec` Function

`sec` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the secant of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
secf = sec(f)
```

```
secf =
[ 1/cos((pi*x)/180), 1/cos(2)]
```

You can calculate `secf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

`x` — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

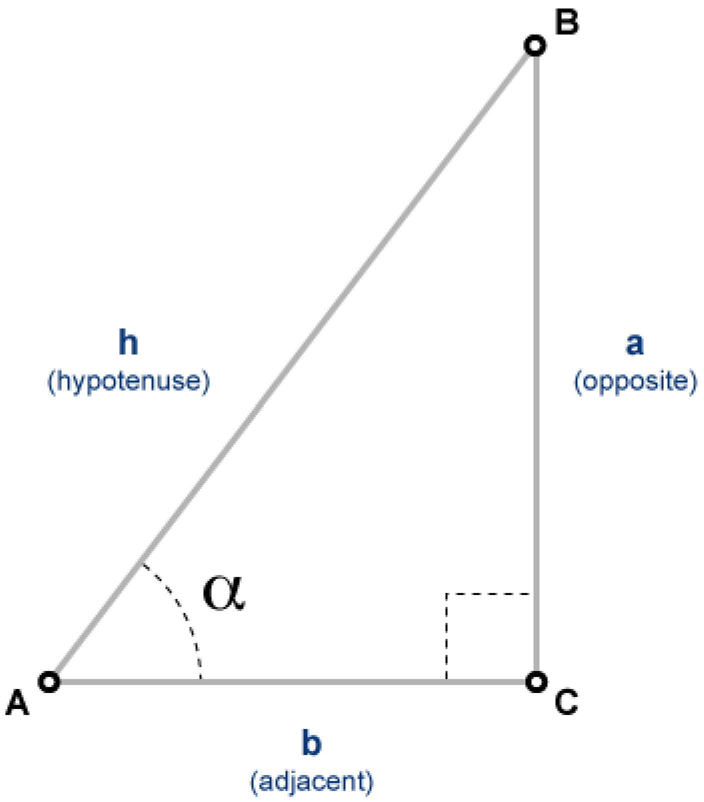
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Secant Function

The secant of an angle, α, defined with reference to a right angled triangle is

$$\sec(\alpha) = \frac{1}{\cos(\alpha)} = \frac{\text{hypotenuse}}{\text{adjacent side}} = \frac{h}{b}.$$



The secant of a complex angle, α, is

$$\sec(\alpha) = \frac{2}{e^{i\alpha} + e^{-i\alpha}}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sin` | `tan`

Introduced before R2006a

sech

Symbolic hyperbolic secant function

Syntax

```
sech(X)
```

Description

`sech(X)` returns the hyperbolic secant function of `X`.

Examples

Hyperbolic Secant Function for Numeric and Symbolic Arguments

Depending on its arguments, `sech` returns floating-point or exact symbolic results.

Compute the hyperbolic secant function for these numbers. Because these numbers are not symbolic objects, `sech` returns floating-point results.

```
A = sech([-2, -pi*i, pi*i/6, 0, pi*i/3, 5*pi*i/7, 1])
```

```
A =
    0.2658   -1.0000    1.1547    1.0000    2.0000   -1.6039    0.6481
```

Compute the hyperbolic secant function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sech` returns unresolved symbolic calls.

```
symA = sech(sym([-2, -pi*i, pi*i/6, 0, pi*i/3, 5*pi*i/7, 1]))
```

```
symA =
[ 1/cosh(2), -1, (2*3^(1/2))/3, 1, 2, -1/cosh((pi*2i)/7), 1/cosh(1)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

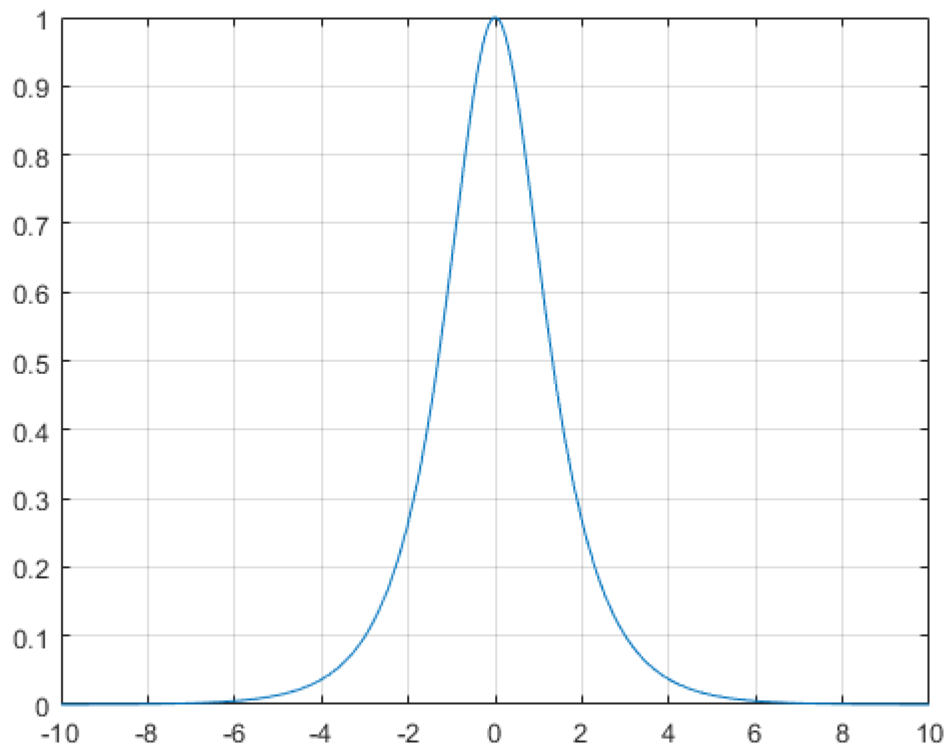
```
vpa(symA)
```

```
ans =
[ 0.26580222883407969212086273981989,...
-1.0,...
1.1547005383792515290182975610039,...
1.0,...
2.0,...
-1.6038754716096765049444092780298,...
0.64805427366388539957497735322615]
```

Plot Hyperbolic Secant Function

Plot the hyperbolic secant function on the interval from -10 to 10. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sech(x), [-10, 10])
grid on
```



Handle Expressions Containing Hyperbolic Secant Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `sech`.

Find the first and second derivatives of the hyperbolic secant function:

```
syms x
diff(sech(x), x)
diff(sech(x), x, x)
```

```
ans =
-sinh(x)/cosh(x)^2
```

```
ans =
(2*sinh(x)^2)/cosh(x)^3 - 1/cosh(x)
```

Find the indefinite integral of the hyperbolic secant function:

```
int(sech(x), x)
```

```
ans =
2*atan(exp(x))
```

Find the Taylor series expansion of `sech(x)`:

```
taylor(sech(x), x)
```

```
ans =
(5*x^4)/24 - x^2/2 + 1
```

Rewrite the hyperbolic secant function in terms of the exponential function:

```
rewrite(sech(x), 'exp')
```

```
ans =
1/(exp(-x)/2 + exp(x)/2)
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sinh` | `tanh`

Introduced before R2006a

separateUnits

Separate units from expression

Syntax

```
[Data,Units] = separateUnits(expr)
Data = separateUnits(expr)
```

Description

`[Data,Units] = separateUnits(expr)` returns the units of the symbolic expression `expr` in `Units` and the rest of `expr` in `Data`.

`Data = separateUnits(expr)` removes symbolic units from `expr` and then returns the rest.

Examples

Separate Units and Expression

Separate the units from the expression `10*t*u.m/u.s`, where `u = symunit`, by providing two output arguments for `separateUnits`.

```
u = symunit;
syms t
speed = 10*t*u.m/u.s;
[Data,Units] = separateUnits(speed)
```

```
Data =
10*t
Units =
1*([m]/[s])
```

Return only the expression with the units removed by providing one output argument.

```
Data = separateUnits(speed)

Data =
10*t
```

Separate Incompatible Units

When the expression has incompatible units, `separateUnits` errors. Units are incompatible when they do not have the same dimensions, such as length or time.

Separate the units from `2*u.m + 3*u.s`. where `u = symunit`. The `separateUnits` function throws an error. Instead, to list the units in the input, use `findUnits`.

```
u = symunit;
[Data,Units] = separateUnits(2*u.m + 3*u.s)

Error using separateUnits (line 51)
Input has incompatible units.
```

Separate Inconsistent Units

When the input has inconsistent units that can be converted to the same unit, then `separateUnits` performs the conversion and returns the separated result. Units are inconsistent when they cannot be converted to each other with a conversion factor of 1

Separate the units from `2*u.m + 30*u.cm`. Even though the units differ, `separateUnits` converts them to the same unit and returns the separated result.

```
u = symunit;
[Data,Units] = separateUnits(2*u.m + 30*u.cm)

Data =
230
Units =
[cm]
```

Input Arguments

`expr` — Input

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, returned as a number, vector, matrix or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

Output Arguments

`Data` — Expression after removing units

number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic equation | symbolic multidimensional array | symbolic function | symbolic expression

Expression after removing units, returned as a number, vector, matrix or multidimensional array, or a symbolic number, variable, vector, matrix, equation, multidimensional array, function, or expression.

`Units` — Units from input

symbolic units

Units from input, specified as symbolic units.

See Also

`checkUnits` | `findUnits` | `isUnit` | `newUnit` | `str2symunit` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

series

Puiseux series

Syntax

```
series(f,var)
series(f,var,a)
series(___,Name,Value)
```

Description

`series(f,var)` approximates `f` with the Puiseux series expansion of `f` up to the fifth order at the point `var = 0`. If you do not specify `var`, then `series` uses the default variable determined by `symvar(f,1)`.

`series(f,var,a)` approximates `f` with the Puiseux series expansion of `f` at the point `var = a`.

`series(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. You can specify `Name,Value` after the input arguments in any of the previous syntaxes.

Examples

Find Puiseux Series Expansion

Find the Puiseux series expansions of univariate and multivariate expressions.

Find the Puiseux series expansion of this expression at the point `x = 0`.

```
syms x
series(1/sin(x), x)
```

```
ans =
x/6 + 1/x + (7*x^3)/360
```

Find the Puiseux series expansion of this multivariate expression. If you do not specify the expansion variable, `series` uses the default variable determined by `symvar(f,1)`.

```
syms s t
f = sin(s)/sin(t);
symvar(f, 1)
series(f)
```

```
ans =
t

ans =
sin(s)/t + (7*t^3*sin(s))/360 + (t*sin(s))/6
```

To use another expansion variable, specify it explicitly.

```
syms s t
f = sin(s)/sin(t);
series(f, s)
```

```
ans =
s^5/(120*sin(t)) - s^3/(6*sin(t)) + s/sin(t)
```

Specify Expansion Point

Find the Puiseux series expansion of `psi(x)` around `x = Inf`. The default expansion point is 0. To specify a different expansion point, use the `ExpansionPoint` name-value pair.

```
series(psi(x), x, 'ExpansionPoint', Inf)
```

```
ans =
log(x) - 1/(2*x) - 1/(12*x^2) + 1/(120*x^4)
```

Alternatively, specify the expansion point as the third argument of `series`.

```
syms x
series(psi(x), x, Inf)
```

```
ans =
log(x) - 1/(2*x) - 1/(12*x^2) + 1/(120*x^4)
```

Specify Truncation Order

Find the Puiseux series expansion of `exp(x)/x` using different truncation orders.

Find the series expansion up to the default truncation order 6.

```
syms x
f = exp(x)/x;
s6 = series(f, x)
```

```
s6 =
x/2 + 1/x + x^2/6 + x^3/24 + x^4/120 + 1
```

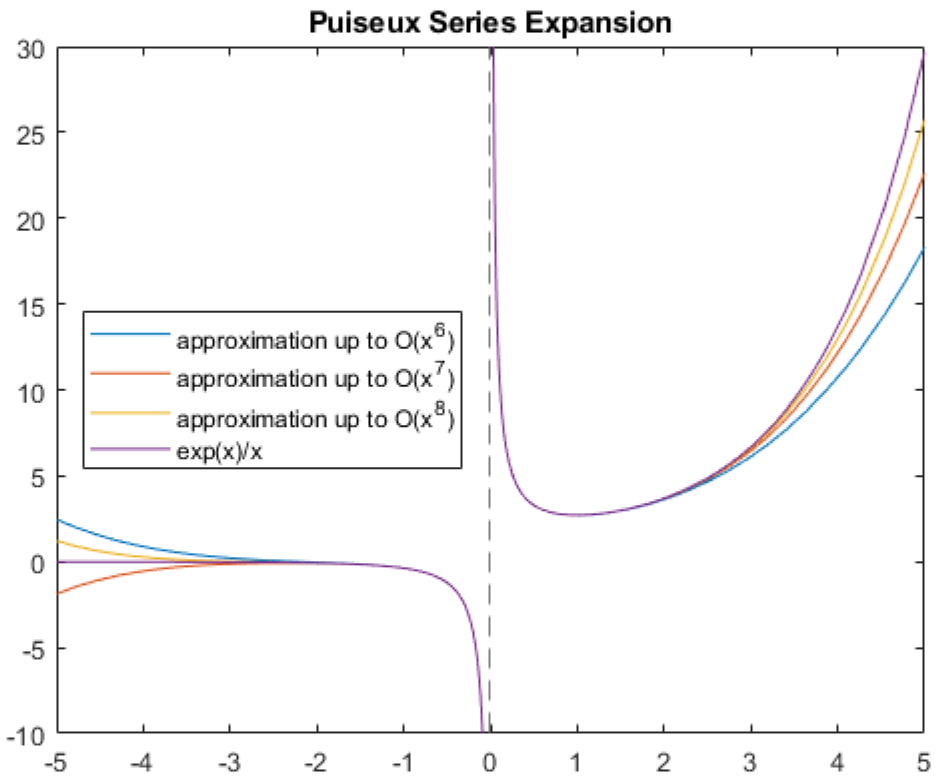
Use `Order` to control the truncation order. For example, approximate the same expression up to the orders 7 and 8.

```
s7 = series(f, x, 'Order', 7)
s8 = series(f, x, 'Order', 8)
```

```
s7 =
x/2 + 1/x + x^2/6 + x^3/24 + x^4/120 + x^5/720 + 1

s8 =
x/2 + 1/x + x^2/6 + x^3/24 + x^4/120 + x^5/720 + x^6/5040 + 1
```

Plot the original expression `f` and its approximations `s6`, `s7`, and `s8`. Note how the accuracy of the approximation depends on the truncation order. Prior to R2016a, use `ezplot` instead of `fplot`.

```
fplot([s6 s7 s8 f])
legend('approximation up to O(x^6)','approximation up to O(x^7)',...
            'approximation up to O(x^8)','exp(x)/x','Location', 'Best')
title('Puiseux Series Expansion')
```



Specify Direction of Expansion

Find the Puiseux series approximations using the `Direction` argument. This argument lets you change the convergence area, which is the area where `series` tries to find converging Puiseux series expansion approximating the original expression.

Find the Puiseux series approximation of this expression. By default, `series` finds the approximation that is valid in a small open circle in the complex plane around the expansion point.

```
syms x
series(sin(sqrt(-x)), x)
```

```
ans =
(-x)^(1/2) - (-x)^(3/2)/6 + (-x)^(5/2)/120
```

Find the Puiseux series approximation of the same expression that is valid in a small interval to the left of the expansion point. Then, find an approximation that is valid in a small interval to the right of the expansion point.

```
syms x
series(sin(sqrt(-x)), x)
series(sin(sqrt(-x)), x, 'Direction', 'left')
series(sin(sqrt(-x)), x, 'Direction', 'right')
```

```
ans =
(-x)^(1/2) - (-x)^(3/2)/6 + (-x)^(5/2)/120

ans =
- x^(1/2)*1i - (x^(3/2)*1i)/6 - (x^(5/2)*1i)/120

ans =
x^(1/2)*1i + (x^(3/2)*1i)/6 + (x^(5/2)*1i)/120
```

Try computing the Puiseux series approximation of this expression. By default, `series` tries to find an approximation that is valid in the complex plane around the expansion point. For this expression, such approximation does not exist.

```
series(real(sin(x)), x)
```

```
Error using sym/series>scalarSeries (line 90)
Cannot compute a series expansion of the input.
```

However, the approximation exists along the real axis, to both sides of `x = 0`.

```
series(real(sin(x)), x, 'Direction', 'realAxis')
```

```
ans =
x^5/120 - x^3/6 + x
```

Input Arguments

`f` — Input to approximate

symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Input to approximate, specified as a symbolic expression or function. It also can be a vector, matrix, or multidimensional array of symbolic expressions or functions.

`var` — Expansion variable
symbolic variable

Expansion variable, specified as a symbolic variable. If you do not specify `var`, then `series` uses the default variable determined by `symvar(f,1)`.

`a` — Expansion point
0 (default) | number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable.

You also can specify the expansion point as a `Name,Value` pair argument. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

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`.

Example: `series(psi(x),x,'ExpansionPoint',Inf,'Order',9)`

`ExpansionPoint` — Expansion point
0 (default) | number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable.

You can also specify the expansion point using the input argument `a`. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

`Order` — Truncation order of Puiseux series expansion
6 (default) | positive integer | symbolic positive integer

Truncation order of Puiseux series expansion, specified as a positive integer or a symbolic positive integer.

`series` computes the Puiseux series approximation with the order `n - 1`. The truncation order `n` is the exponent in the *O*-term: $O(var^n)$.

`Direction` — Direction for area of convergence of Puiseux series expansion
`'complexPlane'` (default) | `'left'` | `'right'` | `'realAxis'`

Direction for area of convergence of Puiseux series expansion, specified as:

`'left'`	Find a Puiseux series approximation that is valid in a small interval to the left of the expansion point.
`'right'`	Find a Puiseux series approximation that is valid in a small interval to the right of the expansion point.
`'realAxis'`	Find a Puiseux series approximation that is valid in a small interval on the both sides of the expansion point.
`'complexPlane'`	Find a Puiseux series approximation that is valid in a small open circle in the complex plane around the expansion point. This is the default value.

Tips

- If you use both the third argument `a` and the `ExpansionPoint` name-value pair to specify the expansion point, the value specified via `ExpansionPoint` prevails.

See Also

`pade` | `taylor`

Introduced in R2015b

setVar

Assign variable in MuPAD notebook

Syntax

```
setVar(nb,MATLABvar)
setVar(nb,'MuPADvar',MATLABexpr)
```

Description

`setVar(nb,MATLABvar)` copies the symbolic variable `MATLABvar` and its value in the MATLAB workspace to the variable `MATLABvar` in the MuPAD notebook `nb`.

`setVar(nb,'MuPADvar',MATLABexpr)` assigns the symbolic expression `MATLABexpr` in the MATLAB workspace to the variable `MuPADvar` in the MuPAD notebook `nb`.

Examples

Copy Variable and Its Value from MATLAB to MuPAD

Copy a variable `y` with a value `exp(-x)` assigned to it from the MATLAB workspace to a MuPAD notebook. Do all three steps in the MATLAB Command Window.

Create the symbolic variable `x` and assign the expression `exp(-x)` to `y`:

```
syms x
y = exp(-x);
```

Create a new MuPAD notebook and specify a handle `mpnb` to that notebook:

```
mpnb = mupad;
```

Copy the variable `y` and its value `exp(-x)` to the MuPAD notebook `mpnb`:

```
setVar(mpnb,'y',y)
```

After executing this statement, the MuPAD engine associated with the `mpnb` notebook contains the variable `y`, with its value `exp(-x)`.

Assign MATLAB Symbolic Expression to Variable in MuPAD

Working in the MATLAB Command Window, assign an expression `t^2 + 1` to a variable `g` in a MuPAD notebook. Do all three steps in the MATLAB Command Window.

Create the symbolic variable `t`:

```
syms t
```

Create a new MuPAD notebook and specify a handle `mpnb` to that notebook:

```
mpnb = mupad;
```

Assign the value `t^2 + 1` to the variable `g` in the MuPAD notebook `mpnb`:

```
setVar(mpnb,'g',t^2 + 1)
```

After executing this statement, the MuPAD engine associated with the `mpnb` notebook contains the variable `g`, with its value `t^2 + 1`.

- “Copy Variables and Expressions Between MATLAB and MuPAD” on page 3-52

Input Arguments

`nb` — Pointer to MuPAD notebook
handle to notebook | vector of handles to notebooks

Pointer to a MuPAD notebook, specified as a MuPAD notebook handle or a vector of handles. You create the notebook handle when opening a notebook with the `mupad` or `openmn` function.

`MuPADvar` — Variable in MuPAD notebook
variable

Variable in a MuPAD notebook, specified as a variable.

`MATLABvar` **— Variable in MATLAB workspace**
symbolic variable

Variable in the MATLAB workspace, specified as a symbolic variable.

`MATLABexpr` **— Expression in MATLAB workspace**
symbolic expression

Expression in the MATLAB workspace, specified as a symbolic expression.

See Also

`getVar` | `mupad` | `openmu`

Topics

"Copy Variables and Expressions Between MATLAB and MuPAD" on page 3-52

Introduced in R2008b

sign

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`.

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 + 2i/5),                              -1]
[ 2^(1/2)*(- 1/2 + 1i/2), 5^(1/2)*18^(1/2)*(1/30 - 1i/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
```

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.

Definitions

Sign Function

The sign function of any number z is defined via the absolute value of z:

$$\operatorname{sign}(z) = \frac{z}{|z|}$$

Thus, the sign function of a real number z can be defined as follows:

$$\operatorname{sign}(z) = \begin{cases} -1 \text{ if } x < 0 \\ \ \ 0 \text{ if } x = 0 \\ \ \ 1 \text{ if } x > 0 \end{cases}$$

Tips

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

See Also

`abs` | `angle` | `imag` | `real` | `signIm`

Introduced in R2013a

signIm

Sign of the imaginary part of complex number

Syntax

```
signIm(z)
```

Description

`signIm(z)` returns the sign of the imaginary part of a complex number `z`. For all complex numbers with a nonzero imaginary part, `singIm(z) = sign(imag(z))`. For real numbers, `signIm(z) = -sign(z)`.

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

Examples

Symbolic Results Including signIm

Results of symbolic computations, especially symbolic integration, can include the `signIm` function.

Integrate this expression. For complex values `a` and `x`, this integral includes `signIm`.

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

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

Signs of Imaginary Parts of Numbers

Find the signs of imaginary parts of complex numbers with nonzero imaginary parts and of real numbers.

Use `signIm` to find the signs of imaginary parts of these numbers. For complex numbers with nonzero imaginary parts, `signIm` returns the sign of the imaginary part of the number.

```
[signIm(-18 + 3*i), signIm(-18 - 3*i),...
signIm(10 + 3*i), signIm(10 - 3*i),...
signIm(Inf*i), signIm(-Inf*i)]
```

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

For real positive numbers, `signIm` returns `-1`.

```
[signIm(2/3), signIm(1), signIm(100), signIm(Inf)]
```

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

For real negative numbers, `signIm` returns `1`.

```
[signIm(-2/3), signIm(-1), signIm(-100), signIm(-Inf)]
```

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

`signIm(0)` is `0`.

```
[signIm(0), signIm(0 + 0*i), signIm(0 - 0*i)]
```

```
ans =
     0     0     0
```

Signs of Imaginary Parts of Symbolic Expressions

Find the signs of imaginary parts of symbolic expressions that represent complex numbers.

Call `signIm` for these symbolic expressions without additional assumptions. Because `signIm` cannot determine if the imaginary part of a symbolic expression is positive, negative, or zero, it returns unresolved symbolic calls.

```
syms x y z
[signIm(z), signIm(x + y*i), signIm(x - 3*i)]

ans =
[ signIm(z), signIm(x + y*1i), signIm(x - 3i)]
```

Assume that `x`, `y`, and `z` are positive values. Find the signs of imaginary parts of the same symbolic expressions.

```
syms x y z positive
[signIm(z), signIm(x + y*i), signIm(x - 3*i)]

ans =
[ -1, 1, -1]
```

For further computations, clear the assumptions.

```
syms x y z clear
```

Find the first derivative of the `signIm` function. `signIm` is a constant function, except for the jump discontinuities along the real axis. The `diff` function ignores these discontinuities.

```
syms z
diff(signIm(z), z)

ans =
0
```

Signs of Imaginary Parts of Matrix Elements

`singIm` accepts vectors and matrices as its input argument. This lets you find the signs of imaginary parts of several numbers in one function call.

Find the signs of imaginary parts of the real and complex elements of matrix `A`.

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

```
ans =
[ 1,  1]
[ 1, -1]
```

Input Arguments

z — Input representing complex number

number | symbolic number | symbolic variable | symbolic expression | vector | matrix

Input representing complex number, specified as a number, symbolic number, symbolic variable, expression, vector, or matrix.

Tips

- `signIm(NaN)` returns `NaN`.

See Also

`conj` | `imag` | `real` | `sign`

Introduced in R2014b

simplify

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.

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 Matrix Elements

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
M = [(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)];
simplify(M)

ans =
[  x + 3, cos(x)]
[ sin(x),      4]
```

Get Simpler Results For Logarithms and Powers

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 More Simplification 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*2i)*1i - 1i)/(exp(x*2i) + 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 =
2i/(exp(x*2i) + 1) - 1i

ans =
((cos(x) - sin(x)*1i)*1i)/cos(x) - 1i

ans =
tan(x)
```

If you are unable to return the desired result, try alternate simplification functions. See “Choose Function to Rearrange Expression” on page 2-94.

Separate Real and Imaginary Parts

Attempt to separate real and imaginary parts of an expression by setting 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 + exp(x*i)/2)*i)/2;
simplify(f, 'Criterion','preferReal', 'Steps', 100)
```

```
ans =
sin(sin(x))*cosh(x) + cos(sin(x))*sinh(x)*1i
```

If `Criterion` is not set to `preferReal`, then `simplify` returns a shorter result but the real and imaginary parts are not separated.

```
simplify(f,'Steps',100)
```

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

When you set `Criterion` to `preferReal`, the simplifier disfavors expression forms where complex values appear inside subexpressions. In nested subexpressions, the deeper the complex value appears inside an expression, the least preference this form of an expression gets.

Avoid Imaginary Terms in Exponents

Attempt to avoid imaginary terms in exponents by setting `Criterion` to `preferReal`.

Show this behavior by simplifying a complex symbolic expression with and without setting `Criterion` to `preferReal`. When `Criterion` is set to `preferReal`, then `simplify` places the imaginary term outside the exponent.

```
expr = sym(i)^(i+1);
withoutPreferReal = simplify(expr,'Steps',100)
```

```
withoutPreferReal =
(-1)^(1/2 + 1i/2)
```

```
withPreferReal = simplify(expr,'Criterion','preferReal','Steps',100)
```

```
withPreferReal =
exp(-pi/2)*1i
```

Simplify Units

Simplify expressions containing symbolic units of the same dimension by using `simplify`.

```
u = symunit;
expr = 300*u.cm + 40*u.inch + 2*u.m;
expr = simplify(expr)
```

```
expr =
(3008/5)*[cm]
```

`simplify` automatically chooses the unit to rewrite into. To choose a specific unit, use `rewrite`.

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`.

Example: `'Seconds',60` limits the simplification process to 60 seconds.

`Criterion` — Simplification criterion

`'default'` (default) | `'preferReal'`

Simplification criterion, specified as the comma-separated pair consisting of `'Criterion'` and one of these character vectors.

`'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.

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.

Algorithms

When you use `IgnoreAnalyticConstraints`, then `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:
  - $\log(e^x) = x$

- asin(sin(x)) = x, acos(cos(x)) = x, atan(tan(x)) = x
- asinh(sinh(x)) = x, acosh(cosh(x)) = x, atanh(tanh(x)) = x
- $\mathrm{W}_k(x \cdot e^x) = x$ for all values of k

See Also

`collect` | `combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplifyFraction`

Topics

"Simplify Symbolic Expressions" on page 2-86
"Choose Function to Rearrange Expression" on page 2-94

Introduced before R2006a

simplifyFraction

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.

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)
```

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.

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` | `combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify`

Topics

"Simplify Symbolic Expressions" on page 2-86
"Choose Function to Rearrange Expression" on page 2-94

Introduced in R2011b

simscapeEquation

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` converts expressions with the second and higher-order derivatives to a system of first-order equations, introducing new variables, such as `x1`, `x2`, and so on.

`simscapeEquation(LHS,RHS)` returns a Simscape equation `LHS == RHS`.

Examples

Convert the following expressions to Simscape language equations.

```
syms t x(t) y(t)
phi = diff(x) + 5*y + sin(t);
simscapeEquation(phi)
simscapeEquation(diff(y),phi)

ans =
    'phi == sin(time)+y*5.0+x.der;'

ans =
    'y.der == sin(time)+y*5.0+x.der;'
```

Convert this expression containing the second derivative.

```
syms x(t)
eqn1 = diff(x,2) - diff(x) + sin(t);
simscapeEquation(eqn1)

ans =
    'x.der == x1;
       eqn1 == sin(time)-x1+x1.der;'
```

Convert this expression containing the fourth and second derivatives.

```
eqn2 = diff(x,4) + diff(x,2) - diff(x) + sin(t);
simscapeEquation(eqn2)

ans =
    'x.der == x1;
       x1.der == x2;
       x2.der == x3;
       eqn2 == sin(time)-x1+x2+x3.der;'
```

Tips

The equation section of a Simscape component file supports a limited number of functions. For details and the list of supported functions, see Simscape `equations`. 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 and want to convert the results to Simscape equations, use the `generate::Simscape` function in MuPAD.

See Also

`ccode` | `fortran` | `matlabFunction` | `matlabFunctionBlock` | `symWriteSSC`

Topics

"Generate Simscape Equations from Symbolic Expressions" on page 2-261

Introduced in R2010a

sin

Symbolic sine function

Syntax

```
sin(X)
```

Description

`sin(X)` returns the sine function on page 4-1468 of `X`.

Examples

Sine Function for Numeric and Symbolic Arguments

Depending on its arguments, `sin` returns floating-point or exact symbolic results.

Compute the sine function for these numbers. Because these numbers are not symbolic objects, `sin` returns floating-point results.

```
A = sin([-2, -pi, pi/6, 5*pi/7, 11])
```

```
A =
   -0.9093   -0.0000    0.5000    0.7818   -1.0000
```

Compute the sine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sin` returns unresolved symbolic calls.

```
symA = sin(sym([-2, -pi, pi/6, 5*pi/7, 11]))
```

```
symA =
[ -sin(2), 0, 1/2, sin((2*pi)/7), sin(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

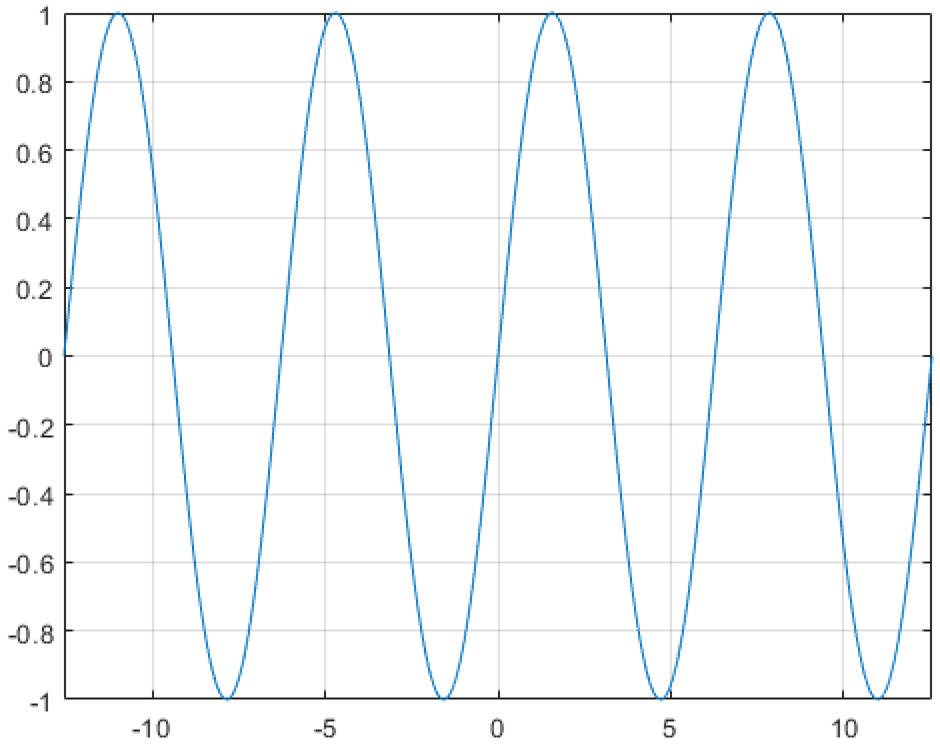
```
vpa(symA)
```

```
ans =
[ -0.90929742682568169539601986591174,...
0,...
0.5,...
0.78183148246802980870844452667406,...
-0.99999020655070345705156489902552]
```

Plot Sine Function

Plot the sine function on the interval from -4π to 4π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sin(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Sine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `sin`.

Find the first and second derivatives of the sine function:

```
syms x
diff(sin(x), x)
diff(sin(x), x, x)
```

```
ans =
cos(x)
```

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

Find the indefinite integral of the sine function:

```
int(sin(x), x)
```

```
ans =
-cos(x)
```

Find the Taylor series expansion of `sin(x)`:

```
taylor(sin(x), x)
```

```
ans =
x^5/120 - x^3/6 + x
```

Rewrite the sine function in terms of the exponential function:

```
rewrite(sin(x), 'exp')
```

```
ans =
(exp(-x*1i)*1i)/2 - (exp(x*1i)*1i)/2
```

Evaluate Units with `sin` Function

`sin` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the sine of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
sinf = sin(f)
```

```
sinf =
[ sin((pi*x)/180), sin(2)]
```

You can calculate `sinf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

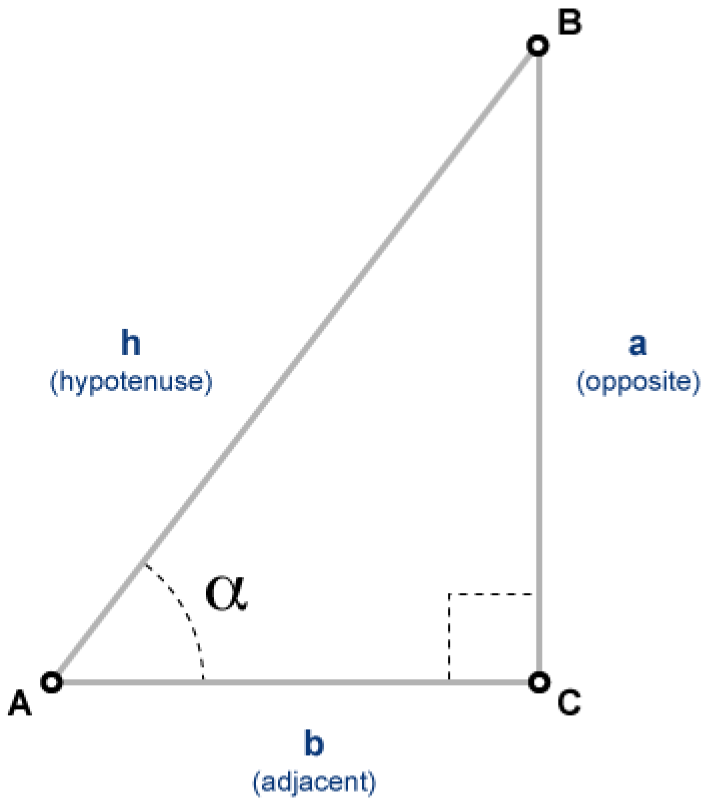
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Sine Function

The sine of an angle, α, defined with reference to a right angled triangle is

$$\sin(\alpha) = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{a}{h}.$$



The sine of a complex angle, α, is

$$\sin(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{2i}.$$

See Also

acos | acot | acsc | asec | asin | atan | cos | cot | csc | sec | tan

Introduced before R2006a

single

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

`double` | `sym` | `vpa`

Introduced before R2006a

sinh

Symbolic hyperbolic sine function

Syntax

```
sinh(X)
```

Description

`sinh(X)` returns the hyperbolic sine function of `X`.

Examples

Hyperbolic Sine Function for Numeric and Symbolic Arguments

Depending on its arguments, `sinh` returns floating-point or exact symbolic results.

Compute the hyperbolic sine function for these numbers. Because these numbers are not symbolic objects, `sinh` returns floating-point results.

```
A = sinh([-2, -pi*i, pi*i/6, 5*pi*i/7, 3*pi*i/2])
```

```
A =
  -3.6269 + 0.0000i   0.0000 - 0.0000i   0.0000 + 0.5000i...
   0.0000 + 0.7818i   0.0000 - 1.0000i
```

Compute the hyperbolic sine function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sinh` returns unresolved symbolic calls.

```
symA = sinh(sym([-2, -pi*i, pi*i/6, 5*pi*i/7, 3*pi*i/2]))
```

```
symA =
[ -sinh(2), 0, 1i/2, sinh((pi*2i)/7), -1i]
```

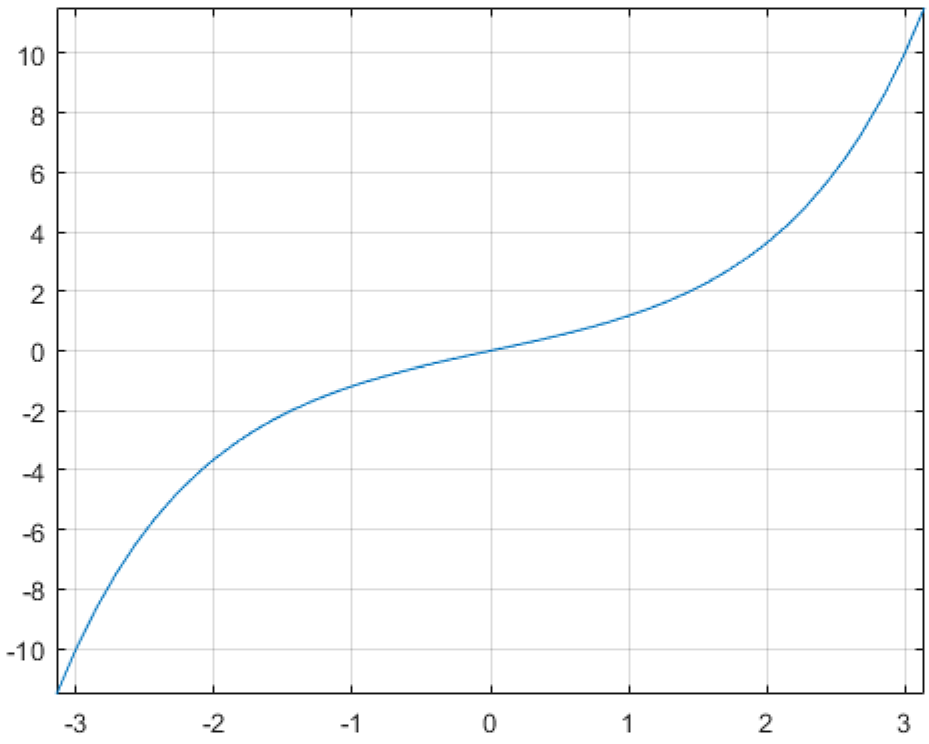
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -3.6268604078470187676682139828013,...
0,...
0.5i,...
0.78183148246802980870844452667406i,...
-1.0i]
```

Plot Hyperbolic Sine Function

Plot the hyperbolic sine function on the interval from $-\pi$ to π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sinh(x), [-pi, pi])
grid on
```



Handle Expressions Containing Hyperbolic Sine Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `sinh`.

Find the first and second derivatives of the hyperbolic sine function:

```
syms x
diff(sinh(x), x)
diff(sinh(x), x, x)
```

```
ans =
cosh(x)
```

```
ans =
sinh(x)
```

Find the indefinite integral of the hyperbolic sine function:

```
int(sinh(x), x)
```

```
ans =
cosh(x)
```

Find the Taylor series expansion of `sinh(x)`:

```
taylor(sinh(x), x)
```

```
ans =
x^5/120 + x^3/6 + x
```

Rewrite the hyperbolic sine function in terms of the exponential function:

```
rewrite(sinh(x), 'exp')
```

```
ans =
exp(x)/2 - exp(-x)/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `tanh`

Introduced before R2006a

sinhint

Hyperbolic sine integral function

Syntax

```
sinhint(X)
```

Description

`sinhint(X)` returns the hyperbolic sine integral function on page 4-1478 of `X`.

Examples

Hyperbolic Sine Integral Function for Numeric and Symbolic Arguments

Depending on its arguments, `sinhint` returns floating-point or exact symbolic results.

Compute the hyperbolic sine integral function for these numbers. Because these numbers are not symbolic objects, `sinhint` returns floating-point results.

```
A = sinhint([-pi, -1, 0, pi/2, 2*pi])
```

```
A =
   -5.4696   -1.0573         0    1.8027   53.7368
```

Compute the hyperbolic sine integral function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sinhint` returns unresolved symbolic calls.

```
symA = sinhint(sym([-pi, -1, 0, pi/2, 2*pi]))
```

```
symA =
[ -sinhint(pi), -sinhint(1), 0, sinhint(pi/2), sinhint(2*pi)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

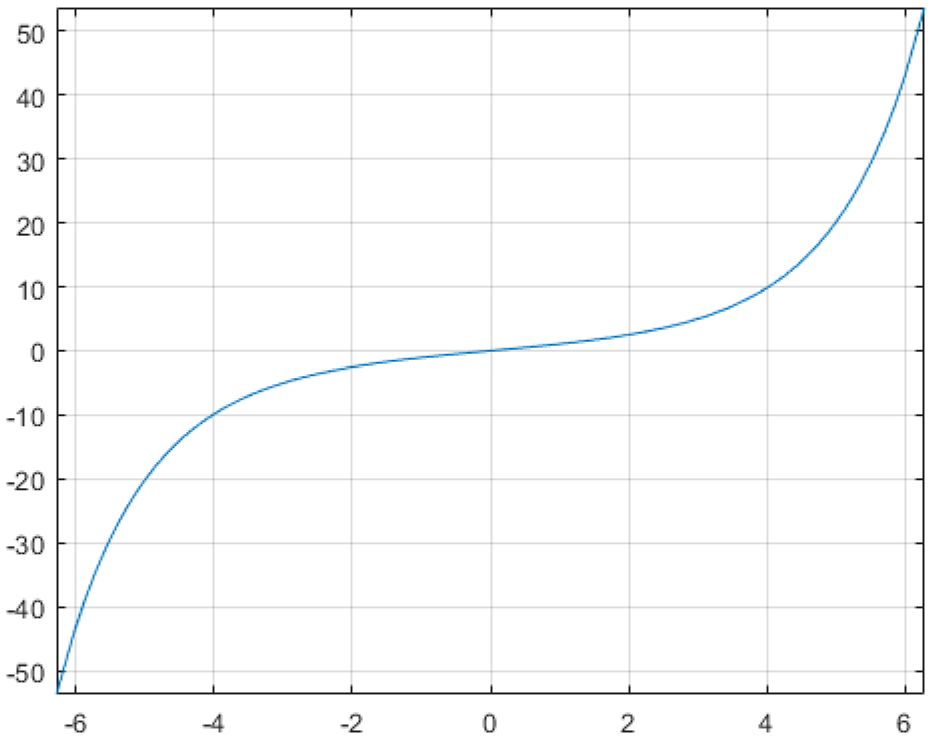
```
vpa(symA)
```

```
ans =
[ -5.4696403451153421506369580091277,...
-1.0572508753757285145718423548959,...
0,...
1.802743198288293882089794577617,...
53.736750620859153990408011863262]
```

Plot Hyperbolic Sine Integral Function

Plot the hyperbolic sine integral function on the interval from `-2*pi` to `2*pi`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sinhint(x), [-2*pi, 2*pi])
grid on
```



Handle Expressions Containing Hyperbolic Sine Integral Function

Many functions, such as `diff`, `int`, and `taylor`, can handle expressions containing `sinhint`.

Find the first and second derivatives of the hyperbolic sine integral function:

```
syms x
diff(sinhint(x), x)
diff(sinhint(x), x, x)
```

```
ans =
sinh(x)/x
```

```
ans =
cosh(x)/x - sinh(x)/x^2
```

Find the indefinite integral of the hyperbolic sine integral function:

```
int(sinhint(x), x)

ans =
x*sinhint(x) - cosh(x)
```

Find the Taylor series expansion of `sinhint(x)`:

```
taylor(sinhint(x), x)

ans =
x^5/600 + x^3/18 + x
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Hyperbolic Sine Integral Function

The hyperbolic sine integral function is defined as follows:

$$\mathrm{Shi}(x) = \int_0^x \frac{\sinh(t)}{t}\,dt$$

References

[1] Gautschi, W. and W. F. Cahill. “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

`coshint` | `cosint` | `eulergamma` | `int` | `sin` | `sinint` | `ssinint`

Introduced in R2014a

sinint

Sine integral function

Syntax

```
sinint(X)
```

Description

`sinint(X)` returns the sine integral function of `X`.

Examples

Sine Integral Function for Numeric and Symbolic Arguments

Depending on its arguments, `sinint` returns floating-point or exact symbolic results.

Compute the sine integral function for these numbers. Because these numbers are not symbolic objects, `sinint` returns floating-point results.

```
A = sinint([- pi, 0, pi/2, pi, 1])
```

```
A =
   -1.8519         0    1.3708    1.8519    0.9461
```

Compute the sine integral function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `sinint` returns unresolved symbolic calls.

```
symA = sinint(sym([- pi, 0, pi/2, pi, 1]))
```

```
symA =
[ -sinint(pi), 0, sinint(pi/2), sinint(pi), sinint(1)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

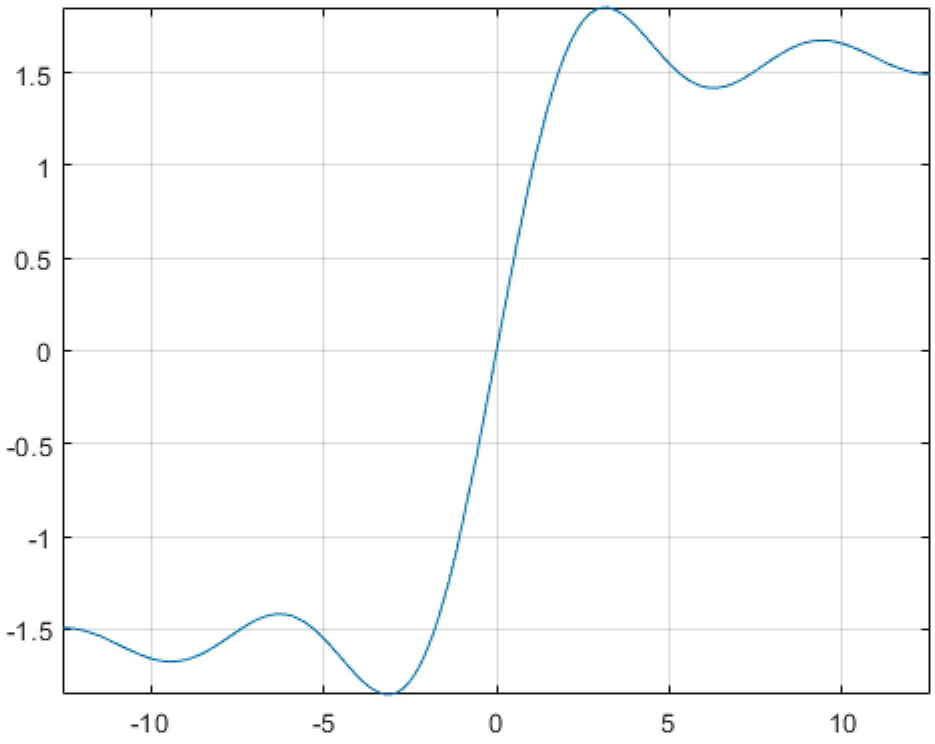
```
vpa(symA)
```

```
ans =
[ -1.8519370519824661703610533701581,...
0,...
1.3707621681544884800696782883816,...
1.8519370519824661703610533701581,...
0.94608307036718301494135331382318]
```

Plot Sine Integral Function

Plot the sine integral function on the interval from `-4*pi` to `4*pi`. Before R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(sinint(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Sine Integral Function

Many functions, such as `diff`, `int`, and `taylor`, can handle expressions containing `sinint`.

Find the first and second derivatives of the sine integral function:

```
syms x
diff(sinint(x), x)
diff(sinint(x), x, x)
```

```
ans =
sin(x)/x
```

```
ans =
cos(x)/x - sin(x)/x^2
```

Find the indefinite integral of the sine integral function:

```
int(sinint(x), x)
```

```
ans =
cos(x) + x*sinint(x)
```

Find the Taylor series expansion of `sinint(x)`:

```
taylor(sinint(x), x)
```

```
ans =
x^5/600 - x^3/18 + x
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Sine Integral Function

The sine integral function is defined as follows:

$$\mathrm{Si}(x) = \int_0^x \frac{\sin(t)}{t}\,dt$$

References

[1] Gautschi, W. and W. F. Cahill. "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

`coshint` | `cosint` | `eulergamma` | `int` | `sin` | `sinhint` | `ssinint`

Introduced before R2006a

size

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`

Introduced before R2006a

smithForm

Smith form of matrix

Syntax

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

___ = smithForm(A,var)
```

Description

`S = smithForm(A)` returns the Smith normal form on page 4-1492 of a square invertible matrix `A`. The elements of `A` must be integers or polynomials in a variable determined by `symvar(A,1)`. The Smith form `S` is a diagonal matrix.

`[U,V,S] = smithForm(A)` returns the Smith normal form of `A` and unimodular transformation matrices `U` and `V`, such that `S = U*A*V`.

`___ = smithForm(A,var)` assumes that the elements of `A` are univariate polynomials in the specified variable `var`. If `A` contains other variables, `smithForm` treats those variables as symbolic parameters.

You can use the input argument `var` in any of the previous syntaxes.

If `A` does not contain `var`, then `smithForm(A)` and `smithForm(A,var)` return different results.

Examples

Smith Form for Matrix of Integers

Find the Smith form of an inverse Hilbert matrix.

```
A = sym(invhilb(5))
S = smithForm(A)

A =
[    25,   -300,    1050,   -1400,    630]
[  -300,   4800,  -18900,   26880, -12600]
[  1050, -18900,   79380, -117600,  56700]
[ -1400,  26880, -117600,  179200, -88200]
[   630, -12600,   56700,  -88200,  44100]

S =
[ 5,  0,   0,   0,    0]
[ 0, 60,   0,   0,    0]
[ 0,  0, 420,   0,    0]
[ 0,  0,   0, 840,    0]
[ 0,  0,   0,   0, 2520]
```

Smith Form for Matrix of Univariate Polynomials

Create a 2-by-2 matrix, the elements of which are polynomials in the variable x.

```
syms x
A = [x^2 + 3, (2*x - 1)^2; (x + 2)^2, 3*x^2 + 5]

A =
[   x^2 + 3, (2*x - 1)^2]
[ (x + 2)^2,   3*x^2 + 5]
```

Find the Smith form of this matrix.

```
S = smithForm(A)

S =
[ 1,                               0]
[ 0, x^4 + 12*x^3 - 13*x^2 - 12*x - 11]
```

Smith Form for Matrix of Multivariate Polynomials

Create a 2-by-2 matrix containing two variables: x and y.

```
syms x y
A = [2/x + y, x^2 - y^2; 3*sin(x) + y, x]
```

```
A =
[      y + 2/x, x^2 - y^2]
[ y + 3*sin(x),         x]
```

Find the Smith form of this matrix. If you do not specify the polynomial variable, `smithForm` uses `symvar(A,1)` and thus determines that the polynomial variable is `x`. Because `3*sin(x) + y` is not a polynomial in `x`, `smithForm` throws an error.

```
S = smithForm(A)
```

```
Error using mupadengine/feval (line 163)
Cannot convert the matrix entries to integers or univariate polynomials.
```

Find the Smith form of `A` specifying that all elements of `A` are polynomials in the variable `y`.

```
S = smithForm(A,y)
```

```
S =
[ 1,                                                 0]
[ 0, 3*y^2*sin(x) - 3*x^2*sin(x) + y^3 + y*(- x^2 + x) + 2]
```

Smith Form and Transformation Matrices

Find the Smith form and transformation matrices for an inverse Hilbert matrix.

```
A = sym(invhilb(3));
[U,V,S] = smithForm(A)
```

```
U =
[  1,  1, 1]
[ -4, -1, 0]
[ 10,  5, 3]

V =
[ 1, -2, 0]
[ 0,  1, 5]
[ 0,  1, 4]

S =
[ 3,  0,  0]
[ 0, 12,  0]
[ 0,  0, 60]
```

Verify that `S = U*A*V`.

```
isAlways(S == U*A*V)

ans =
  3×3 logical array
     1     1     1
     1     1     1
     1     1     1
```

Find the Smith form and transformation matrices for a matrix of polynomials.

```
syms x y
A = [2*(x - y), 3*(x^2 - y^2);
     4*(x^3 - y^3), 5*(x^4 - y^4)];
[U,V,S] = smithForm(A,x)

U =
[ 0,                        1]
[ 1, - x/(10*y^3) - 3/(5*y^2)]

V =
[ -x/(4*y^3), - (5*x*y^2)/2 - (5*x^2*y)/2 - (5*x^3)/2 - (5*y^3)/2]
[  1/(5*y^3),                               2*x^2 + 2*x*y + 2*y^2]

S =
[ x - y,                             0]
[     0, x^4 + 6*x^3*y - 6*x*y^3 - y^4]
```

Verify that `S = U*A*V`.

```
isAlways(S == U*A*V)

ans =
  2×2 logical array
     1     1
     1     1
```

If You Specify Variable for Integer Matrix

If a matrix does not contain a particular variable, and you call `smithForm` specifying that variable as the second argument, then the result differs from what you get without specifying that variable. For example, create a matrix that does not contain any variables.

```
A = [9 -36 30; -36 192 -180; 30 -180 180]
```

```
A =
     9   -36    30
   -36   192  -180
    30  -180   180
```

Call `smithForm` specifying variable `x` as the second argument. In this case, `smithForm` assumes that the elements of `A` are univariate polynomials in `x`.

```
syms x
smithForm(A,x)
```

```
ans =
     1     0     0
     0     1     0
     0     0     1
```

Call `smithForm` without specifying variables. In this case, `smithForm` treats `A` as a matrix of integers.

```
smithForm(A)
```

```
ans =
     3     0     0
     0    12     0
     0     0    60
```

Input Arguments

`A` — Input matrix

square invertible symbolic matrix

Input matrix, specified as a square invertible symbolic matrix, the elements of which are integers or univariate polynomials. If the elements of `A` contain more than one variable, use the `var` argument to specify a polynomial variable, and treat all other variables as symbolic parameters. If `A` is multivariate, and you do not specify `var`, then `smithForm` uses `symvar(A,1)` to determine a polynomial variable.

`var` — Polynomial variable

symbolic variable

Polynomial variable, specified as a symbolic variable.

Output Arguments

`S` — Smith normal form of input matrix

symbolic diagonal matrix

Smith normal form of input matrix, returned as a symbolic diagonal matrix. The first diagonal element divides the second, the second divides the third, and so on.

`U` — Transformation matrix

unimodular symbolic matrix

Transformation matrix, returned as a unimodular symbolic matrix. If elements of `A` are integers, then elements of `U` are also integers, and `det(U) = 1` or `det(U) = -1`. If elements of `A` are polynomials, then elements of `U` are univariate polynomials, and `det(U)` is a constant.

`V` — Transformation matrix

unimodular symbolic matrix

Transformation matrix, returned as a unimodular symbolic matrix. If elements of `A` are integers, then elements of `V` are also integers, and `det(V) = 1` or `det(V) = -1`. If elements of `A` are polynomials, then elements of `V` are univariate polynomials, and `det(V)` is a constant.

Definitions

Smith Normal Form

Smith normal form of a an n-by-n matrix `A` is an n-by-n diagonal matrix S, such that $S_{i,i}$ divides $S_{i+1,i+1}$ for all $i < n$.

See Also

`hermiteForm` | `jordan`

Introduced in R2015b

solve

Equations and systems solver

Note Character vector inputs will be removed in a future release. Instead, use `syms` to declare variables and replace inputs such as `solve('2*x == 1','x')` with `solve(2*x == 1,x)`.

Syntax

```
S = solve(eqn,var)
S = solve(eqn,var,Name,Value)

Y = solve(eqns,vars)
Y = solve(eqns,vars,Name,Value)

[y1,...,yN] = solve(eqns,vars)
[y1,...,yN] = solve(eqns,vars,Name,Value)
[y1,...,yN,parameters,conditions] = solve(eqns,vars,'
ReturnConditions',true)
```

Description

`S = solve(eqn,var)` solves the equation `eqn` for the variable `var`. If you do not specify `var`, the `symvar` function determines the variable to solve for. For example, `solve(x + 1 == 2, x)` solves the equation $x + 1 = 2$ for x.

`S = solve(eqn,var,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`Y = solve(eqns,vars)` solves the system of equations `eqns` for the variables `vars` and returns a structure that contains the solutions. If you do not specify `vars`, `solve` uses `symvar` to find the variables to solve for. In this case, the number of variables that `symvar` finds is equal to the number of equations `eqns`.

`Y = solve(eqns,vars,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[y1,...,yN] = solve(eqns,vars)` solves the system of equations `eqns` for the variables `vars`. The solutions are assigned to the variables `y1,...,yN`. If you do not specify the variables, `solve` uses `symvar` to find the variables to solve for. In this case, the number of variables that `symvar` finds is equal to the number of output arguments `N`.

`[y1,...,yN] = solve(eqns,vars,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[y1,...,yN,parameters,conditions] = solve(eqns,vars,' ReturnConditions',true)` returns the additional arguments `parameters` and `conditions` that specify the parameters in the solution and the conditions on the solution.

Examples

Solve an Equation

Use the == operator to specify the equation `sin(x) == 1` and solve it.

```
syms x
eqn = sin(x) == 1;
solx = solve(eqn,x)

solx =
pi/2
```

Find the complete solution of the same equation by specifying the `ReturnConditions` option as `true`. Specify output variables for the solution, the parameters in the solution, and the conditions on the solution.

```
[solx, params, conds] = solve(eqn, x, 'ReturnConditions', true)

solx =
pi/2 + 2*pi*k

params =
k
```

```
conds =
in(k, 'integer')
```

The solution `pi/2 + 2*pi*k` contains the parameter `k` which is valid under the condition `in(k, 'integer')`. This condition means the parameter `k` must be an integer.

If `solve` returns an empty object, then no solutions exist. If `solve` returns an empty object with a warning, solutions might exist but `solve` did not find any solutions.

```
eqns = [3*x+2, 3*x+1];
solve(eqns, x)
```

```
ans =
Empty sym: 0-by-1
```

Use Parameters and Conditions Returned by solve to Refine Solution

Return the complete solution of an equation with parameters and conditions of the solution by specifying `ReturnConditions` as `true`.

Solve the equation $\sin(x) = 0$. Provide two additional output variables for output arguments `parameters` and `conditions`.

```
syms x
eqn = sin(x) == 0;
[solx, param, cond] = solve(eqn, x, 'ReturnConditions', true)
```

```
solx =
pi*k
param =
k
cond =
in(k, 'integer')
```

The solution `pi*k` contains the parameter `k` and is valid under the condition `in(k,'integer')`. This condition means the parameter `k` must be an integer. `k` does not exist in the MATLAB workspace and must be accessed using `param`.

Find a valid value of `k` for `0 < x < 2*pi` by assuming the condition, `cond`, and using `solve` to solve these conditions for `k`. Substitute the value of `k` found into the solution for `x`.

```
assume(cond)
interval = [solx > 0, solx < 2*pi];
solk = solve(interval, param)
valx = subs(solx, param, solk)

solk =
1
valx =
pi
```

A valid value of `k` for `0 < x < 2*pi` is `1`. This produces the value `x = pi`.

Alternatively, find a solution for `x` by choosing a value of `k`. Check if the value chosen satisfies the condition on `k` using `isAlways`.

Check if `k = 4` satisfies the condition on `k`.

```
condk4 = subs(cond, param, 4);
isAlways(condk4)

ans =
  logical
   1
```

`isAlways` returns logical `1` (`true`), meaning `4` is a valid value for `k`. Substitute `k` with `4` to obtain a solution for `x`. Use `vpa` to obtain a numeric approximation.

```
valx = subs(solx, param, 4)
vpa(valx)

valx =
4*pi
ans =
12.566370614359172953850573533118
```

Solve Multivariate Equations and Assign Outputs to Variables

Avoid ambiguities when solving equations with symbolic parameters by specifying the variable for which you want to solve an equation. If you do not specify the variable, `solve` chooses a variable using `symvar`. First, solve the quadratic equation without specifying a variable. `solve` chooses *x* to return the familiar solution. Then solve the quadratic equation for `a` to return the solution for `a`.

```
syms a b c x
eqn = a*x^2 + b*x + c == 0;
sol = solve(eqn)
sola = solve(eqn, a)
```

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

When solving for more than one variable, the order in which you specify the variables defines the order in which the solver returns the solutions.

Solve this system of equations and assign the solutions to variables `solv` and `solu` by specifying the variables explicitly. The solver returns an array of solutions for each variable.

```
syms u v
eqns = [2*u^2 + v^2 == 0, u - v == 1];
vars = [v u];
[solv, solu] = solve(eqns, vars)
```

```
solv =
 - (2^(1/2)*1i)/3 - 2/3
   (2^(1/2)*1i)/3 - 2/3
solu =
 1/3 - (2^(1/2)*1i)/3
 (2^(1/2)*1i)/3 + 1/3
```

Entries with the same index form the solutions of a system.

```
solutions = [solv solu]
```

```
solutions =
[ - (2^(1/2)*1i)/3 - 2/3, 1/3 - (2^(1/2)*1i)/3]
[   (2^(1/2)*1i)/3 - 2/3, (2^(1/2)*1i)/3 + 1/3]
```

A solution of the system is `v = - (2^(1/2)*1i)/3 - 2/3`, and `u = 1/3 - (2^(1/2)*1i)/3`.

Solve Multivariate Equations and Assign Outputs to Structure

When solving for multiple variables, it can be more convenient to store the outputs in a structure array than in separate variables. The `solve` function returns a structure when you specify a single output argument and multiple outputs exist.

Solve a system of equations to return the solutions in a structure array.

```
syms u v
eqns = [2*u + v == 0, u - v == 1];
S = solve(eqns, [u v])

S =
  struct with fields:

    u: [1×1 sym]
    v: [1×1 sym]
```

Access the solutions by addressing the elements of the structure.

```
S.u
S.v

ans =
1/3
ans =
-2/3
```

Using a structure array allows you to conveniently substitute solutions into expressions. The `subs` function substitutes the correct values irrespective of which variables you substitute.

Substitute solutions into expressions using the structure `S`.

```
expr1 = u^2;
subs(expr1, S)
expr2 = 3*v+u;
subs(expr2, S)

ans =
1/9
ans =
-5/3
```

Return Complete Solution of System of Equations Using Structure

Return the complete solution of a system of equations with parameters and conditions of the solution by specifying `ReturnConditions` as `true`.

```
syms x y
eqns = [sin(x)^2 == cos(y), 2*x == y];
S = solve(eqns, [x y], 'ReturnConditions', true);
S.x
S.y
S.conditions
S.parameters
```

```
ans =
 pi*k - asin(3^(1/2)/3)
 asin(3^(1/2)/3) + pi*k
ans =
 2*pi*k - 2*asin(3^(1/2)/3)
 2*asin(3^(1/2)/3) + 2*pi*k
ans =
 in(k, 'integer')
 in(k, 'integer')
ans =
k
```

A solution is formed by the elements of the same index in `S.x`, `S.y`, and `S.conditions`. Any element of `S.parameters` can appear in any solution. For example, a solution is `x = pi*k - asin(3^(1/2)/3)`, and `y = 2*pi*k - 2*asin(3^(1/2)/3)`, with the parameter `k` under the condition `in(k, 'integer')`. This condition means `k` must be an integer for the solution to be valid. `k` does not exist in the MATLAB workspace and must be accessed with `S.parameters`.

For the first solution, find a valid value of `k` for `0 < x < pi` by assuming the condition `S.conditions(1)` and using `solve` to solve these conditions for `k`. Substitute the value of `k` found into the solution for `x`.

```
assume(S.conditions(1))
interval = [S.x(1)>0, S.x(1)<pi];
solk = solve(interval, S.parameters)
solx = subs(S.x(1), S.parameters, solk)
```

```
solk =
1
```

```
solx =
pi - asin(3^(1/2)/3)
```

A valid value of `k` for `0 < x < pi` is `1`. This produces the value `x = pi - asin(3^(1/2)/3)`.

Alternatively, find a solution for `x` by choosing a value of `k`. Check if the value chosen satisfies the condition on `k` using `isAlways`.

Check if `k = 4` satisfies the condition on `k`.

```
condk4 = subs(S.conditions(1), S.parameters, 4);
isAlways(condk4)
```

```
ans =
  logical
   1
```

`isAlways` returns logical `1` (`true`) meaning `4` is a valid value for `k`. Substitute `k` with `4` to obtain a solution for `x`. Use `vpa` to obtain a numeric approximation.

```
valx = subs(S.x(1), S.parameters, 4)
vpa(valx)
```

```
valx =
4*pi - asin(3^(1/2)/3)
ans =
11.950890905688785612783108943994
```

Return Numeric Solutions

Try solving the following equation. The symbolic solver cannot find an exact symbolic solution for this equation, and therefore issues a warning before calling 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 it finds.

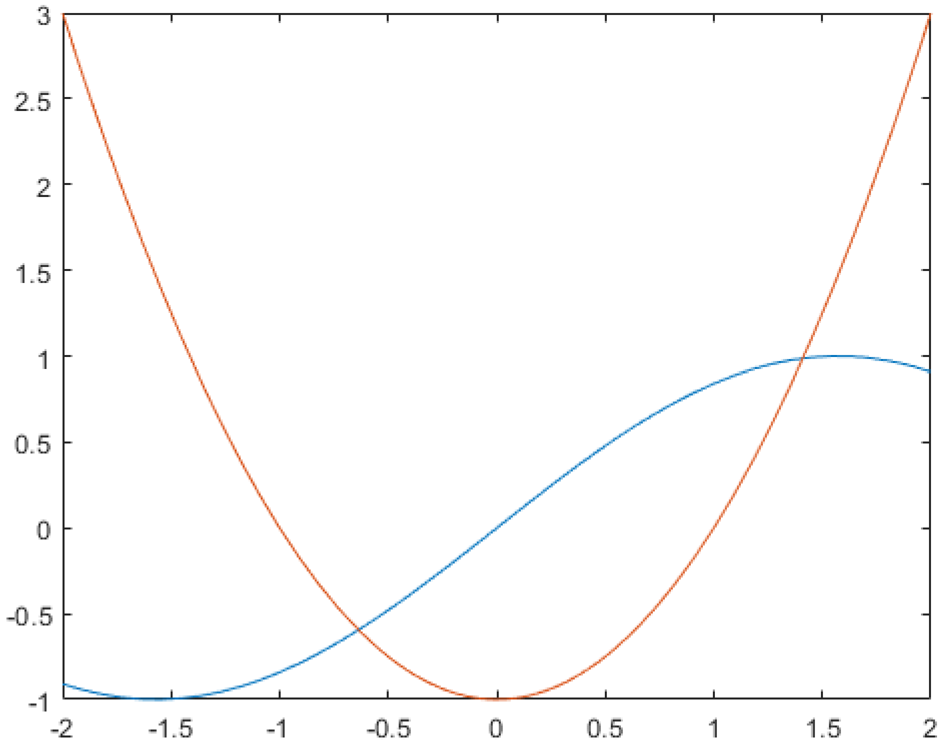
```
syms x
eqn = sin(x) == x^2 - 1;
solve(eqn, x)
```

```
Warning: Unable to solve symbolically. Returning a
numeric approximation instead.
```

```
> In solve (line 304)
ans =
-0.63673265080528201088799090383828
```

Plot the left and the right sides of the equation in one graph. The graph shows that the equation also has a positive solution.

```
fplot(sin(x), [-2 2])
hold on
fplot(x^2 - 1, [-2 2])
hold off
```



Find this solution by calling the numeric solver `vpasolve` directly and specifying the interval where this solution can be found.

```
eqn = sin(x) == x^2 - 1;
vpasolve(eqn, x, [0 2])

ans =
1.4096240040025962492355939705895
```

Solve Inequalities

`solve` can solve inequalities to find a solution that satisfies the inequalities.

Solve the following inequalities. Set `ReturnConditions` to `true` to return any parameters in the solution and conditions on the solution.

$$x > 0$$
$$y > 0$$
$$x^2 + y^2 + xy < 1$$

```
syms x y
cond1 = x^2 + y^2 + x*y < 1;
cond2 = x > 0;
cond3 = y > 0;
conds = [cond1 cond2 cond3];

sol = solve(conds, [x y], 'ReturnConditions', true);

sol.x
sol.y
sol.parameters
sol.conditions

ans =
(- 3*v^2 + u)^(1/2)/2 - v/2
ans =
v
ans =
[ u, v]
ans =
4*v^2 < u & u < 4 & 0 < v
```

The parameters `u` and `v` do not exist in the MATLAB workspace and must be accessed using `sol.parameters`.

Check if the values `u = 7/2` and `v = 1/2` satisfy the condition using `subs` and `isAlways`.

```
condWithValues = subs(sol.conditions, sol.parameters, [7/2,1/2]);
isAlways(condWithValues)
```

```
ans =
  logical
   1
```

`isAlways` returns logical `1` (`true`) indicating that these values satisfy the condition. Substitute these parameter values into `sol.x` and `sol.y` to find a solution for `x` and `y`.

```
xSol = subs(sol.x, sol.parameters, [7/2,1/2])
ySol = subs(sol.y, sol.parameters, [7/2,1/2])
```

```
xSol =
11^(1/2)/4 - 1/4

ySol =
1/2
```

Convert the solution into numeric form by using `vpa`.

```
vpa(xSol)
vpa(ySol)
```

```
ans =
0.57915619758884996227873318416767

ans =
0.5
```

Return Real Solutions

Solve this equation. It has five solutions.

```
syms x
eqn = x^5 == 3125;
solve(eqn, x)
```

```
ans =
                                                   5
 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*5i)/4 - (5*5^(1/2))/4 - 5/4
   (2^(1/2)*(5 - 5^(1/2))^(1/2)*5i)/4 - (5*5^(1/2))/4 - 5/4
   (5*5^(1/2))/4 - (2^(1/2)*(5^(1/2) + 5)^(1/2)*5i)/4 - 5/4
   (5*5^(1/2))/4 + (2^(1/2)*(5^(1/2) + 5)^(1/2)*5i)/4 - 5/4
```

Return only real solutions by setting argument `Real` to `true`. The only real solution of this equation is `5`.

```
solve(eqn, x, 'Real', true)
```

```
ans =
5
```

Return One Solution

Solve this equation. Instead of returning an infinite set of periodic solutions, the solver picks these three solutions that it considers to be most practical.

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

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

Pick only one solution using `PrincipalValue`.

```
eqn = sin(x) + cos(2*x) == 1;
solve(eqn, x, 'PrincipalValue', true)
```

```
ans =
0
```

Shorten Result with Simplification Rules

Try to solve this equation. By default, `solve` does not apply simplifications that are not always mathematically correct. As a result, `solve` cannot solve this equation symbolically.

```
syms x
eqn = exp(log(x)*log(3*x)) == 4;
solve(eqn, x)
```

```
Warning: Unable to solve symbolically.
Returning a numeric approximation instead.
```

```
ans =
- 14.009379055223370038369334703094 - 2.9255310052111119036668717988769i
```

Set `IgnoreAnalyticConstraints` to `true` to apply simplifications that might allow `solve` to find a result. For details, see "Algorithms" on page 4-1514.

```
S = solve(eqn, x, 'IgnoreAnalyticConstraints', true)

S =
 (3^(1/2)*exp(-(log(256) + log(3)^2)^(1/2)/2))/3
  (3^(1/2)*exp((log(256) + log(3)^2)^(1/2)/2))/3
```

`solve` applies simplifications that allow it to find a solution. The simplifications applied do not always hold. Thus, the solutions in this mode might not be correct or complete, and need verification.

Ignore Assumptions on Variables

The `sym` and `syms` functions let you set assumptions for symbolic variables.

Assume that the variable *x* can have only positive values.

```
syms x positive
```

When you solve an equation or a system of equations for a variable under assumptions, the solver only returns solutions consistent with the assumptions. Solve this equation for `x`.

```
eqn = x^2 + 5*x - 6 == 0;
solve(eqn, x)

ans =
1
```

Allow solutions that do not satisfy the assumptions by setting `IgnoreProperties` to `true`.

```
solve(eqn, x, 'IgnoreProperties', true)

ans =
 -6
  1
```

For further computations, clear the assumption that you set on the variable *x*.

```
syms x clear
```

Numerically Approximating Symbolic Solutions That Contain `root`

When solving polynomials, `solve` might return solutions containing `root`. To numerically approximate these solutions, use `vpa`. Consider the following equation and solution.

```
syms x
eqn = x^4 + x^3 + 1 == 0;
s = solve(eqn, x)

s =
 root(z^4 + z^3 + 1, z, 1)
 root(z^4 + z^3 + 1, z, 2)
 root(z^4 + z^3 + 1, z, 3)
 root(z^4 + z^3 + 1, z, 4)
```

Because there are no parameters in this solution, use `vpa` to approximate it numerically.

```
vpa(s)

ans =
 - 1.0189127943851558447865795886366 - 0.60256541999859902604398442197193i
 - 1.0189127943851558447865795886366 + 0.60256541999859902604398442197193i
     0.5189127943851558447865795886366 - 0.666609844932018579153758800733i
     0.5189127943851558447865795886366 + 0.666609844932018579153758800733i
```

Solve Polynomial Equations of High Degree

When you solve a higher order polynomial equation, the solver might use `root` to return the results. Solve an equation of order 3.

```
syms x a
eqn = x^3 + x^2 + a == 0;
solve(eqn, x)

ans =
 root(z^3 + z^2 + a, z, 1)
 root(z^3 + z^2 + a, z, 2)
 root(z^3 + z^2 + a, z, 3)
```

Try to get an explicit solution for such equations by 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`. Increasing this value, you can get explicit solutions for higher order polynomials.

Solve the same equations for explicit solutions by increasing the value of `MaxDegree` to `3`.

```
S = solve(eqn, x, 'MaxDegree', 3);
pretty(S)
```

```
/                  1         1                \
|                ---- + #1 - -                |
|                9 #1        3                |
|                                             |
|           /   1       \                     |
|   sqrt(3) | ---- - #1 | 1i                  |
|           \ 9 #1      /        1     #1   1 |
| - ------------------------ - ----- - -- - - |
|               2              18 #1    2   3 |
|                                             |
|          /   1       \                      |
|  sqrt(3) | ---- - #1 | 1i                   |
|          \ 9 #1      /        1     #1   1  |
|  ------------------------ - ----- - -- - -  |
\              2              18 #1    2   3  /

where

         /     / / a    1 \2    1  \   a    1 \1/3
   #1 == | sqrt| | - + -- |  - --- | - - - -- |
         \     \ \ 2   27 /    729 /   2   27 /
```

Input Arguments

`eqn` — Equation to solve

symbolic expression | symbolic equation

Equation to solve, specified as a symbolic expression or symbolic equation. The relation operator == defines symbolic equations . 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 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 elements of `eqns` are symbolic expressions (without the right side), `solve` equates the element to `0`.

`vars` — Variables for which you solve an equation or 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

Example: `'Real',true` specifies that the solver returns real solutions.

`ReturnConditions` — Flag for returning parameters conditions

`false` (default) | `true`

Flag for returning parameters in solution and conditions under which the solution is true, specified as the comma-separated pair consisting of `'ReturnConditions'` and one of these values.

`false`	Do not return parameterized solutions. Do not return the conditions under which the solution holds. The `solve` function replaces parameters with appropriate values.
`true`	Return the parameters in the solution and the conditions under which the solution holds. For a call with a single output variable, `solve` returns a structure with the fields `parameters` and `conditions`. For multiple output variables, `solve` assigns the parameters and conditions to the last two output variables. This behavior means that the number of output variables must be equal to the number of variables to solve for plus two.

Example: `[v1, v2, params, conditions] = solve(sin(x) +y == 0,y^2 == 3,'ReturnConditions',true)` returns the parameters in `params` and conditions in `conditions`.

`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. Setting `IgnoreAnalyticConstraints` to `true` can lead to wrong or incomplete results.

`IgnoreProperties` — Flag for returning solutions inconsistent with 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` — Maximum degree of polynomial equations for which solver uses explicit formulas

`2` (default) | positive integer smaller than 5

Maximum degree of polynomial equations for which 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 one solution

`false` (default) | `true`

Flag for returning 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 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 system of equations

structure

Solutions of a system of equations, returned as a structure. The number of fields in the structure correspond to the number of independent variables in a system. If `ReturnConditions` is set to `true`, the `solve` function returns two additional fields that contain the parameters in the solution, and the conditions under which the solution is true.

`y1,...,yN` — Solutions of 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.

`parameters` — Parameters in solution
vector of generated parameters

Parameters in a solution, returned as a vector of generated parameters. This output argument is only returned if `ReturnConditions` is `true`. If a single output argument is provided, `parameters` is returned as a field of a structure. If multiple output arguments are provided, `parameters` is returned as the second-to-last output argument. The generated parameters do not appear in the MATLAB workspace. They must be accessed using `parameters`.

Example: `[solx, params, conditions] = solve(sin(x) == 0, 'ReturnConditions', true)` returns the parameter `k` in the argument `params`.

`conditions` — Conditions under which solutions are valid
vector of symbolic expressions

Conditions under which solutions are valid, returned as a vector of symbolic expressions. This output argument is only returned if `ReturnConditions` is `true`. If a single output argument is provided, `conditions` is returned as a field of a structure. If multiple output arguments are provided, `conditions` is returned as the last output argument.

Example: `[solx, params, conditions] = solve(sin(x) == 0, 'ReturnConditions', true)` returns the condition `in(k, 'integer')` in `conditions`. The solution in `solx` is valid only under this condition.

Tips

- If `solve` cannot find a solution and `ReturnConditions` is `false`, the `solve` function internally calls the numeric solver `vpasolve` that tries to find a numeric solution. If `solve` cannot find a solution and `ReturnConditions` is `true`, `solve` returns an empty solution with a warning. If no solutions exist, `solve` returns an empty solution without a warning. For polynomial equations and systems without symbolic parameters, the numeric solver returns all solutions. For nonpolynomial equations and systems without symbolic parameters, the numeric solver returns only one solution (if a solution exists).

- If the solution contains parameters and `ReturnConditions` is `true`, `solve` returns the parameters in the solution and the conditions under which the solutions are true. If `ReturnConditions` is `false`, the `solve` function either chooses values of the parameters and returns the corresponding results, or returns parameterized solutions without choosing particular values. In the latter case, `solve` also issues a warning indicating the values of parameters in the returned solutions.
- If a parameter does not appear in any condition, it means the parameter can take any complex value.
- The output of `solve` can contain parameters from the input equations in addition to parameters introduced by `solve`.
- Parameters introduced by `solve` do not appear in the MATLAB workspace. They must be accessed using the output argument that contains them. Alternatively, to use the parameters in the MATLAB workspace use `syms` to initialize the parameter. For example, if the parameter is `k`, use `syms k`.
- The variable names `parameters` and `conditions` are not allowed as inputs to `solve`.
- The syntax `S = solve(eqn,var,'ReturnConditions',true)` returns `S` as a structure instead of a symbolic array.
- 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, when you run `[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` to `a` and the solutions for `b` to `b`.

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:
  - $\log(e^x) = x$
  - $\operatorname{asin}(\sin(x)) = x$, $\operatorname{acos}(\cos(x)) = x$, $\operatorname{atan}(\tan(x)) = x$
  - $\operatorname{asinh}(\sinh(x)) = x$, $\operatorname{acosh}(\cosh(x)) = x$, $\operatorname{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k
- 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` | `isolate` | `linsolve` | `root` | `subs` | `symvar` | `vpasolve`

Topics

“Solve Algebraic Equation” on page 2-145
“Solve System of Algebraic Equations” on page 2-153
“Solve System of Linear Equations” on page 2-169
“Select Numeric or Symbolic Solver” on page 2-151
“Troubleshoot Equation Solutions from solve Function” on page 2-164

Introduced before R2006a

sort

Sort elements of symbolic vectors or matrices

Syntax

```
Y = sort(X)
[Y,I] = sort(___)
___= sort(X,dim)
___= sort(___,'descend')
```

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 lexicographic order. If `X` is a matrix, `sort(X)` sorts each column of `X`.

`[Y,I] = sort(___)` shows the indices that each element of `Y` had in the original vector or matrix `X`.

If `X` is an `m`-by-`n` matrix and you sort elements of each column (`dim = 2`), 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.

`___= sort(X,dim)` sorts the elements of `X` along the dimension `dim`. Thus, if `X` is a two-dimensional matrix, then `sort(X,1)` sorts elements of each column of `X`, and `sort(X,2)` sorts elements of each row.

`___= sort(___,'descend')` sorts `X` in descending order. By default, `sort` uses ascending order.

Examples

Sort Vector Elements

By default, `sort` sorts the element of a vector or a matrix in ascending order.

Sort the elements of the following symbolic vector:

```
syms a b c d e
sort([7 e 1 c 5 d a b])
```

```
ans =
[ 1, 5, 7, a, b, c, d, e]
```

Find Indices That Elements of Sorted Matrix Had in Original Matrix

To find the indices that each element of a new vector or matrix `Y` had in the original vector or matrix `X`, call `sort` with two output arguments.

Sort the matrix `X` returning the matrix of indices that each element of the sorted matrix had in `X`:

```
X = sym(magic(3));
[Y, I] = sort(X)
```

```
Y =
[ 3, 1, 2]
[ 4, 5, 6]
[ 8, 9, 7]

I =
     2     1     3
     3     2     1
     1     3     2
```

Sort Matrix Along Its Columns and Rows

When sorting elements of a matrix, `sort` can work along the columns or rows of that matrix.

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 in Descending Order

`sort` can sort the elements of a vector or a matrix in descending order.

Sort the elements of this vector in descending order:

```
syms a b c d e
sort([7 e 1 c 5 d a b], 'descend')

ans =
[ e, d, c, b, a, 7, 5, 1]
```

Sort the elements of each column of this matrix `X` in descending order:

```
X = sym(magic(3))
sort(X,'descend')

X =
[ 8, 1, 6]
[ 3, 5, 7]
[ 4, 9, 2]

ans =
[ 8, 9, 7]
[ 4, 5, 6]
[ 3, 1, 2]
```

Now, 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]
```

Input Arguments

`X` — Input that needs to be sorted

symbolic vector | symbolic matrix

Input that needs to be sorted, specified as a symbolic vector or matrix.

`dim` — Dimension to operate along

positive integer

Dimension to operate along, specified as a positive integer. The default value is `1`. If `dim` exceeds the number of dimensions of `X`, then `sort(X,dim)` returns `X`, and `[Y,I] = sort(X,dim)` returns `Y = X` and `I = ones(size(X))`.

Output Arguments

`Y` — Sorted output

symbolic vector | symbolic matrix

Sorted output, returned as a symbolic vector or matrix.

`I` — Indices that elements of `Y` had in `X`

symbolic vector | symbolic matrix

Indices that elements of `Y` had in `X`, returned as a symbolic vector or matrix. `[Y,I] = sort(X,dim)` also returns matrix `I = ones(size(X))` if the value `dim` exceeds the number of dimensions of `X`.

Tips

- Calling `sort` for vectors or matrices of numbers that are not symbolic objects invokes the MATLAB `sort` function.
- For complex input `X`, `sort` compares elements by their magnitudes (complex moduli), computed with `abs(X)`. If complex numbers have the same complex modulus, `sort` compares their phase angles, `angle(X)`.
- If you use `'ascend'` instead of `'descend'`, then `sort` returns elements in ascending order, as it does by default.
- `sort` uses the following rules:

  - It sorts symbolic numbers and floating-point numbers numerically.
  - It sorts symbolic variables alphabetically.
  - In all other cases, including symbolic expressions and functions, `sort` uses internal sorting rules.

See Also

`max` | `min`

Introduced before R2006a

sqrtm

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')`.

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')`.

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 =
  3×3 logical array
   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, 1.290994448735805628393088466594]

resnorm =
2.9387358770557187699218413430556e-40
```

Tips

- Calling `sqrtm` for a matrix that is not a symbolic object invokes the MATLAB `sqrtm` function.

See Also

`cond` | `eig` | `expm` | `funm` | `jordan` | `logm` | `norm`

Introduced in R2013a

ssinint

Shifted sine integral function

Syntax

```
ssinint(X)
```

Description

`ssinint(X)` returns the shifted sine integral function on page 4-1526 `ssinint(X) = sinint(X) — pi/2`.

Examples

Shifted Sine Integral Function for Numeric and Symbolic Arguments

Depending on its arguments, `ssinint` returns floating-point or exact symbolic results.

Compute the shifted sine integral function for these numbers. Because these numbers are not symbolic objects, `ssinint` returns floating-point results.

```
A = ssinint([- pi, 0, pi/2, pi, 1])
```

```
A =
   -3.4227   -1.5708   -0.2000    0.2811   -0.6247
```

Compute the shifted sine integral function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `ssinint` returns unresolved symbolic calls.

```
symA = ssinint(sym([- pi, 0, pi/2, pi, 1]))
```

```
symA =
[ - pi - ssinint(pi), -pi/2, ssinint(pi/2), ssinint(pi), ssinint(1)]
```

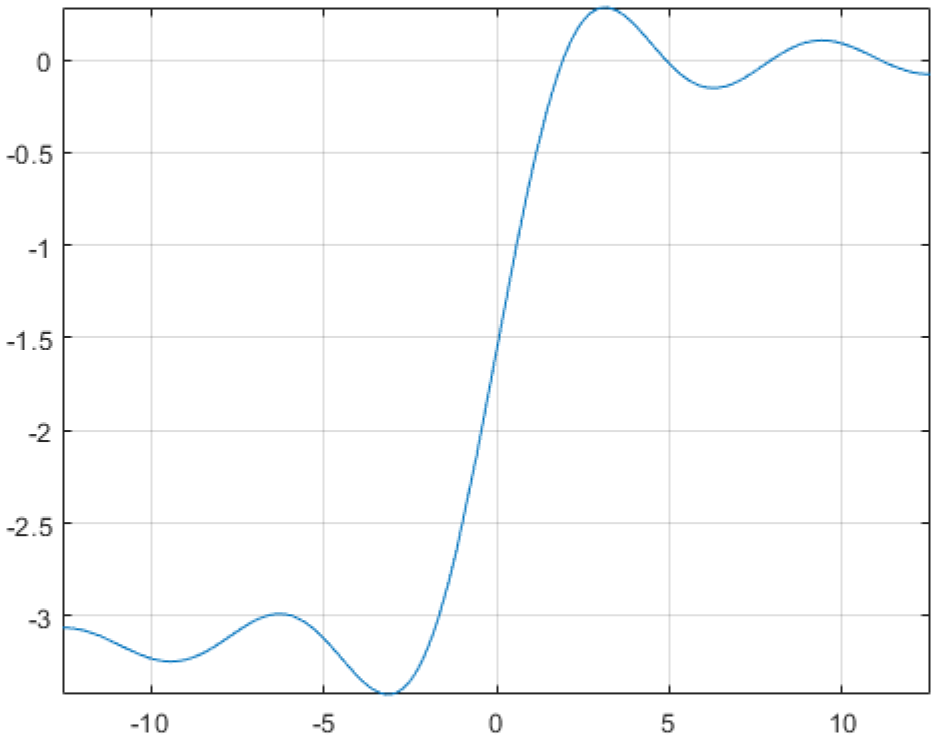
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -3.4227333787773627895923750617977,...
-1.5707963267948966192313216916398,...
-0.20003415864040813916164340325818,...
0.28114072518756955112973167851824,...
-0.62471325642771360428996837781657]
```

Plot Shifted Sine Integral Function

Plot the shifted sine integral function on the interval from `-4*pi` to `4*pi`. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(ssinint(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing Shifted Sine Integral Function

Many functions, such as `diff`, `int`, and `taylor`, can handle expressions containing `ssinint`.

Find the first and second derivatives of the shifted sine integral function:

```
syms x
diff(ssinint(x), x)
diff(ssinint(x), x, x)
```

```
ans =
sin(x)/x
```

```
ans =
cos(x)/x - sin(x)/x^2
```

Find the indefinite integral of the shifted sine integral function:

```
int(ssinint(x), x)
```

```
ans =
cos(x) + x*ssinint(x)
```

Find the Taylor series expansion of `ssinint(x)`:

```
taylor(ssinint(x), x)
```

```
ans =
x^5/600 - x^3/18 + x - pi/2
```

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

Definitions

Sine Integral Function

The sine integral function is defined as follows:

$$\mathrm{Si}(x) = \int_0^x \frac{\sin(t)}{t}\,dt$$

Shifted Sine Integral Function

The sine integral function is defined as $\mathrm{Ssi}(x) = \mathrm{Si}(x) - \pi/2$.

References

[1] Gautschi, W. and W. F. Cahill. “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

`coshint` | `cosint` | `eulergamma` | `int` | `sin` | `sinhint` | `sinhint` | `sinint`

Introduced in R2014a

str2symunit

Convert character vector or string to unit

Syntax

```
str2symunit(unitStr)
str2symunit(unitStr,toolbox)
```

Description

`str2symunit(unitStr)` converts the character vector or string `unitStr` to symbolic units.

`str2symunit(unitStr,toolbox)` converts the character vector `unitStr` assuming it represents units in the toolbox `toolbox`. The allowed values of `toolbox` are `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`.

Examples

Convert Character Vector to Unit

Convert the character vector `'km/hour'` to symbolic units.

```
unit = str2symunit('km/hour')
```

```
unit =
1*([km]/[h])
```

Use this unit to define a speed of `50` km/hour.

```
speed = 50*unit
```

```
speed =
50*([km]/[h])
```

Convert Units of Specified Toolbox

Convert units from other toolboxes to symbolic units by specifying the toolbox name as the second argument to `str2symunit`. The allowed names are `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`.

Convert `'km/h-s'` from Aerospace Toolbox to symbolic units.

```
unit = str2symunit('km/h-s','Aerospace')
```

```
unit =
1*([km]/([h]*[s]))
```

Convert `'molecules/s'` from SimBiology® to symbolic units.

```
unit = str2symunit('molecule/s','SimBiology')
```

```
unit =
1*([molecule]/[s])
```

Convert `'gee/km'` from Simscape to symbolic units.

```
unit = str2symunit('gee/km','Simscape')
```

```
unit =
1*([g_n]/[km])
```

Convert `'rad/second'` from Simulink to symbolic units.

```
unit = str2symunit('rad/second','Simulink')
```

```
unit =
1*([rad]/[s])
```

Input Arguments

`unitStr` — Input units
character vector | string

Input, specified as a character vector or string.

Example: `str2symunit('km/hour')`

`toolbox` — Toolbox to which units belong
`'Aerospace'` | `'SimBiology'` | `'Simscape'` | `'Simulink'`

Toolbox to which input belongs, specified as `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`.

Example: `str2symunit('km/h-s', 'Aerospace')`

See Also

`checkUnits` | `findUnits` | `isUnit` | `newUnit` | `separateUnits` | `symunit` | `symunit2str` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

str2sym

Evaluate string representing symbolic expression

Syntax

```
str2sym(symstr)
```

Description

`str2sym(symstr)` evaluates `symstr` where `symstr` is a string representing a symbolic expression. Enter symbolic expressions as strings only when reading expressions from text files or when specifying numbers exactly. Otherwise, do not use strings for symbolic input.

Examples

Evaluate String as Symbolic Expression

Evaluate the string `'sin(pi)'`. `str2sym` returns the expected result.

```
str2sym('sin(pi)')
```

```
ans =
0
```

`str2sym` assumes the = operator represents an equation, not an assignment. Also, `str2sym` does not add the variables contained in the string to the workspace.

Show this behavior by evaluating `'x^2 = 4'`. The `str2sym` function returns the equation `x^2 == 4` but `x` does not appear in the workspace.

```
eqn = str2sym('x^2 = 4')
```

```
eqn =
x^2 == 4
```

Find the variable in eqn by using symvar. The variable var now refers to x.

```
var = symvar(eqn)

var =
x
```

Assign values from eqn by solving eqn for var and assigning the result.

```
varVal = solve(eqn,var)

varVal =
 -2
  2
```

Substitute Workspace Values into String Input

str2sym does not substitute values from the workspace for variables in the input. Therefore, str2sym has reproducible output. Instead, substitute workspace values by using subs on the output of str2sym.

Set y to 2. Then, evaluate 'y^2' with and without subs to show how subs substitutes y with its value.

```
y = 2;
withoutSubs = str2sym('y^2')

withoutSubs =
y^2

withSubs = subs(str2sym('y^2'))

withSubs =
4
```

Evaluate Strings from File as Symbolic Expressions

When symbolic expressions are stored as strings in a file, evaluate the strings by reading the file and using str2sym.

Assume the file mySym.txt contains this text.

```
a = 2.431
y = a*exp(t)
diff(z(t),t) = b*y*z
```

Evaluate expressions in `mySym.txt` using `str2sym`.

```
filename = 'mySym.txt';
filetext = fileread(filename);
filetext = splitlines(filetext);
str2sym(filetext)
```

```
ans =
           a == 2.431
        y == a*exp(t)
 diff(z(t), t) == b*y*z
```

The output of `str2sym` is independent of workspace values, which means the output is reproducible. Show this reproducibility by assigning a value to `b` and re-evaluating the stored expressions.

```
b = 5;
str2sym(filetext)
```

```
ans =
           a == 2.431
        y == a*exp(t)
 diff(z(t), t) == b*y*z
```

To use workspace values or a value from input equations, use `subs` (solve the equation first using `solve`), as described in "Evaluate String as Symbolic Expression" on page 4-1531 and "Substitute Workspace Values into String Input" on page 4-1532.

Execute Functions in String Input

`str2sym` executes functions in input when the functions are on the path. Otherwise, `str2sym` returns the symbolic object as expected. This behavior means that the output is reproducible.

Show this behavior by reading a differential equation and initial condition from a file. Solve the equation for the condition. Because `str2sym` does not evaluate `y(t)` in the equation, the output is reproducible.

```
filename = 'mySym.txt';
filetext = fileread(filename);
filetext = splitlines(filetext);
eqn = str2sym(filetext(1))

eqn =
diff(y(t), t) == -y(t)

cond = str2sym(filetext(2))

cond =
y(0) == 2

ySol = dsolve(eqn,cond)

ySol =
2*exp(-t)
```

Exactly Represent Large Numbers and High-Precision Numbers

Because the MATLAB parser automatically converts all numbers to double precision, maintain original precision by entering large numbers and high-precision numbers as strings. Instead of `str2sym`, enter integers using `sym` and floating-point numbers using `vpa` because `sym` and `vpa` are faster.

Show the error between entering a ratio of large integers directly versus the exact string representation.

```
num = sym(12230984290/38490293482)

num =
5724399718238385/18014398509481984

numExact = sym('12230984290/38490293482')

numExact =
6115492145/19245146741

error = num - numExact

error =
-7827162395/346689742765832461975814144
```

Show the error between entering a high-precision number directly versus the exact string representation.

```
num = vpa(8.023098429038490293482)
```

```
num =
8.0230984290384910195825796108693
```

```
numExact = vpa('8.023098429038490293482')
```

```
numExact =
8.023098429038490293482
```

```
error = num - numExact
```

```
error =
0.00000000000000072610057961086928844451883343504
```

For details, see “Numeric to Symbolic Conversion” on page 2-125. For full workflows, see “Numerical Computations With High Precision” and “Prime Factorizations”.

Input Arguments

`symstr` — String representing symbolic expression

character vector | string | cell array of character vectors

String representing a symbolic expression, specified as a character vector, string, or cell array of character vectors.

Tips

- `str2sym` assumes the = operator represents an equation, not an assignment.
- `str2sym` does not create variables contained in the input.
- `str2sym('inf')` returns infinity (`Inf`).
- `str2sym('i')` returns the imaginary number `1i`.

See Also

`subs` | `sym` | `syms` | `vpa`

Topics

“Numeric to Symbolic Conversion” on page 2-125

Introduced in R2017b

subexpr

Rewrite symbolic expression in terms of common subexpressions

Syntax

```
[r,sigma] = subexpr(expr)
[r,var] = subexpr(expr,'var')
[r,var] = subexpr(expr,var)
```

Description

`[r,sigma] = subexpr(expr)` rewrites the symbolic expression `expr` in terms of a common subexpression, substituting this common subexpression with the symbolic variable `sigma`. The input expression `expr` cannot contain the variable `sigma`.

`[r,var] = subexpr(expr,'var')` substitutes the common subexpression by `var`. The input expression `expr` cannot contain the symbolic variable `var`.

`[r,var] = subexpr(expr,var)` is equivalent to `[r,var] = subexpr(expr,'var')`, except that the symbolic variable `var` must already exist in the MATLAB workspace.

This syntax overwrites the value of the variable `var` with the common subexpression found in `expr`. To avoid overwriting the value of `var`, use another variable name as the second output argument. For example, use `[r,var1] = subexpr(expr,var)`.

Examples

Rewrite Expression Using Abbreviations

Solve the following equation. The solutions are very long expressions. To see them, remove the semicolon at the end of the `solve` command.

```
syms a b c d x
solutions = solve(a*x^3 + b*x^2 + c*x + d == 0, x, 'MaxDegree', 3);
```

These long expressions have common subexpressions. To shorten the expressions, abbreviate the common subexpression by using `subexpr`. If you do not specify the variable to use for abbreviations as the second input argument of `subexpr`, then `subexpr` uses the variable `sigma`.

```
[r, sigma] = subexpr(solutions)
```

```
r =
sigma^(1/3) - b/(3*a) - (- b^2/(9*a^2) + c/(3*a))/sigma^(1/3)
(- b^2/(9*a^2) + c/(3*a))/(2*sigma^(1/3)) -...
    sigma^(1/3)/2 - (3^(1/2)*(sigma^(1/3) +...
(- b^2/(9*a^2) + c/(3*a))/sigma^(1/3))*1i)/2 - b/(3*a)
(- b^2/(9*a^2) + c/(3*a))/(2*sigma^(1/3)) -...
    sigma^(1/3)/2 + (3^(1/2)*(sigma^(1/3) +...
(- b^2/(9*a^2) + c/(3*a))/sigma^(1/3))*1i)/2 - b/(3*a)

sigma =
((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)
```

Customize Abbreviation Variables

Solve a quadratic equation.

```
syms a b c x
solutions = solve(a*x^2 + b*x + c == 0, x)
```

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

Use `syms` to create the symbolic variable `s`, and then replace common subexpressions in the result with this variable.

```
syms s
[abbrSolutions,s] = subexpr(solutions,s)
```

```
abbrSolutions =
 -(b + s)/(2*a)
 -(b - s)/(2*a)

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

Alternatively, use `'s'` to specify the abbreviation variable.

```
[abbrSolutions,s] = subexpr(solutions,'s')
```

```
abbrSolutions =
 -(b + s)/(2*a)
 -(b - s)/(2*a)

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

Both syntaxes overwrite the value of the variable `s` with the common subexpression. Therefore, you cannot, for example, substitute `s` with some value.

```
subs(abbrSolutions,s,0)
```

```
ans =
 -(b + s)/(2*a)
 -(b - s)/(2*a)
```

To avoid overwriting the value of the variable `s`, use another variable name for the second output argument.

```
syms s
[abbrSolutions,t] = subexpr(solutions,'s')
```

```
abbrSolutions =
 -(b + s)/(2*a)
 -(b - s)/(2*a)

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

```
subs(abbrSolutions,s,0)
```

```
ans =
 -b/(2*a)
 -b/(2*a)
```

Input Arguments

`expr` — Long expression containing common subexpressions

symbolic expression | symbolic function

Long expression containing common subexpressions, specified as a symbolic expression or function.

`var` — Variable to use for substituting common subexpressions

character vector | symbolic variable

Variable to use for substituting common subexpressions, specified as a character vector or symbolic variable.

`subexpr` throws an error if the input expression `expr` already contains `var`.

Output Arguments

`r` — Expression with common subexpressions replaced by abbreviations

symbolic expression | symbolic function

Expression with common subexpressions replaced by abbreviations, returned as a symbolic expression or function.

`var` — Variable used for abbreviations

symbolic variable

Variable used for abbreviations, returned as a symbolic variable.

See Also

`children` | `pretty` | `simplify` | `subs`

Topics

"Abbreviate Common Terms in Long Expressions" on page 2-92

Introduced before R2006a

subs

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 evaluates `s`.

`subs(s,new)` returns a copy of `s`, replacing all occurrences of the default variable in `s` with `new`, and then evaluates `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 evaluates `s`. Variables with no assigned values remain as variables.

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
```

Default Substitution Variable

Substitute the default variable in this expression with `a`. If you do not specify the variable or expression 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
```

Update Expression with New Values

Solve this ordinary differential equation.

```
syms a y(t)
y = dsolve(diff(y) == -a*y)

y =
C3*exp(-a*t)
```

Specify the values of the symbolic parameters `a` and `C2`.

```
a = 980;
C2 = 3;
```

Check that `y` is not updated with these values, although the values appear in the MATLAB Workspace.

```
y
```

```
y =
C3*exp(-a*t)
```

To evaluate y with the new values of a and C2, use subs.

```
subs(y)

ans =
C3*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)
```

Alternatively, for multiple substitutions, use cell arrays.

```
subs(cos(a) + sin(b), {a, b}, {sym('alpha'), 2})

ans =
sin(2) + cos(alpha)
```

Substitute Scalars with Arrays

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]
```

You can also replace an element of a vector, matrix, or array with a nonscalar value. For example, create these 2-by-2 matrices.

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

A =
[ A1_1, A1_2]
[ A2_1, A2_2]

B =
[ B1_1, B1_2]
[ B2_1, B2_2]
```

Replace the first element of the matrix `A` with the matrix `B`. While making this substitution, `subs` expands the 2-by-2 matrix `A` into this 4-by-4 matrix.

```
A44 = subs(A, A(1,1), B)

A44 =
[ B1_1, B1_2, A1_2, A1_2]
[ B2_1, B2_2, A1_2, A1_2]
[ A2_1, A2_1, A2_2, A2_2]
[ A2_1, A2_1, A2_2, A2_2]
```

`subs` does not let you replace a nonscalar with a scalar.

Substitute Multiple Scalars with Arrays

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 element-wise.

```
[0 1; -1 0].*[1 -1; -2 1]

ans =
     0    -1
     2     0
```

Substitutions in Equations

Eliminate variables from an equation by using the variable's value from another equation. In the second equation, isolate the variable on the left side using `isolate`, and then substitute the right side with the variable in the first equation.

First, declare the equations `eqn1` and `eqn2`.

```
syms x y
eqn1 = sin(x)+y == x^2 + y^2;
eqn2 = y*x == cos(x);
```

Isolate `y` in `eqn2` by using `isolate`.

```
eqn2 = isolate(eqn2,y)
```

```
eqn2 =
y == cos(x)/x
```

Eliminate `y` from `eqn1` by substituting the right side of `eqn2` with the left side of `eqn2` in `eqn1`.

```
eqn1 = subs(eqn1,lhs(eqn2),rhs(eqn2))
```

```
eqn1 =
sin(x) + cos(x)/x == cos(x)^2/x^2 + x^2
```

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]
```

Replace the arguments of a symbolic function explicitly.

```
syms x y
f(x, y) = x + y;
f(a, y) = subs(f, x, a);
f

f(a, y) =
a + y
```

Substitute Variables with Corresponding Values from Structure

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
```

Verify the solutions by substituting the solutions into the original system.

```
isAlways(subs(eqs, S))

ans =
  2×2 logical array
   1   1
   1   1
```

Input Arguments

`s` — Input
symbolic variable | symbolic expression | symbolic equation | symbolic function | symbolic array | symbolic matrix

Input, specified as a symbolic variable, expression, equation, function, array, or matrix.

`old` — Element to substitute
symbolic variable | symbolic expression | symbolic array

Element to substitute, specified as a symbolic variable, expression, or array.

`new` — New element
number | symbolic number | symbolic variable | symbolic expression | symbolic array | structure

New element to substitute with, specified as a number, symbolic number, variable, expression, array, or a structure.

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` with 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 element-wise. All constant terms in `s` are replaced with the constant multiplied by 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` as a matrix. All constant terms are replaced with the constant multiplied by an identity matrix.

See Also

`double` | `lhs` | `rhs` | `simplify` | `subexpr` | `vpa`

Topics

"Substitutions in Symbolic Expressions" on page 1-18
"Substitute Variables in Symbolic Expressions" on page 2-107
"Substitute Elements in Symbolic Matrices" on page 2-109
"Substitute Scalars with Matrices" on page 2-111
"Evaluate Symbolic Expressions Using subs" on page 2-113

Introduced before R2006a

svd

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 convertible to floating-point numbers. For example, it can be a matrix of symbolic numbers.

`[U,S,V] = svd(X,0)` returns the thin, or economy, SVD. 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 returns the thin, or economy, SVD. 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.

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
 0.0000000000000000000042127245515076439434819165724023i
```

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]
```

Thin or Economy SVD

Use the second input argument `0` to compute the thin, or economy, SVD 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 thin, or economy, 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]
```

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 convertible to floating-point numbers.

Output Arguments

`sigma` — Singular values

symbolic vector | vector of symbolic numbers

Singular values of a matrix, returned as a vector. If `sigma` is a vector of numbers, then its elements are sorted in descending order.

`U` — Singular vectors

matrix of symbolic numbers

Singular vectors, returned as a 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 unitary matrix. Each column of this matrix is a singular vector.

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.

See Also

`chol` | `digits` | `eig` | `inv` | `lu` | `qr` | `svd` | `vpa`

Topics

"Singular Value Decomposition" on page 2-143

Introduced before R2006a

sym

Create symbolic variables, expressions, functions, matrices

Note Support of character vectors that are not valid variable names and do not define a number will be removed in a future release. To create symbolic expressions, first create symbolic variables, and then use operations on them. For example, use `syms x; x + 1` instead of `sym('x + 1')`, `exp(sym(pi))` instead of `sym('exp(pi)')`, and `syms f(var1,...varN)` instead of `f(var1,...varN) = sym('f(var1,...varN)')`.

Syntax

```
x = sym('x')
A = sym('a', [n1 ... nM])
A = sym('a', n)

sym(___, set)
sym(___, 'clear')

sym(num)
sym(num, flag)

symexpr = sym(h)
```

Description

`x = sym('x')` creates symbolic variable `x`.

`A = sym('a', [n1 ... nM])` creates an `n1`-by-`...`-by-`nM` symbolic array filled with automatically generated elements. For example, `A = sym('a',[1 3])` creates the row vector `A = [a1 a2 a3]`. The auto-generated elements do not appear in the MATLAB workspace. For arrays, these elements have the prefix `a` followed by the element's index using _ as a delimiter, such as `a1_3_2`.

`A = sym('a', n)` creates an `n`-by-`n` symbolic matrix filled with automatically generated elements.

`sym(___, set)` creates a symbolic variable or array and sets the assumption that the variable or all array elements belong to a `set`. Here, `set` can be `'real'`, `'positive'`, `'integer'`, or `'rational'`.

`sym(___, 'clear')` clears assumptions set on a symbolic variable or array. You can specify `'clear'` after the input arguments in any of the previous syntaxes, except combining `'clear'` and `set`. You cannot set and clear an assumption in the same function call to `sym`.

`sym(num)` converts a number or numeric matrix to a symbolic number or symbolic matrix.

`sym(num, flag)` uses the technique specified by `flag` for converting floating-point numbers to symbolic numbers.

`symexpr = sym(h)` creates a symbolic expression or matrix `symexpr` from an anonymous MATLAB function associated with the function handle `h`.

Examples

Create Symbolic Variables

Create the symbolic variables `x` and `y`.

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

Create Symbolic Vector

Create a 1-by-4 symbolic vector `a` with auto-generated elements `a1`, ..., `a4`.

```
a = sym('a', [1 4])
```

```
a =
[ a1, a2, a3, a4]
```

Format the names of elements of `a` by using a format character vector as the first argument. `sym` replaces `%d` in the format character vector with the index of the element to generate the element names.

```
a = sym('x_%d', [1 4])
```

```
a =
[ x_1, x_2, x_3, x_4]
```

This syntax does not create symbolic variables `x_1`, ..., `x_4` in the MATLAB workspace. Access elements of `a` using standard indexing methods.

```
a(1)
a(2:3)
```

```
ans =
x_1
ans =
[ x_2, x_3]
```

Create Symbolic Matrices

Create a 3-by-4 symbolic matrix with automatically generated elements. The elements are of the form `ai_j`, which generates the 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]
```

Create a 4-by-4 matrix with the element names `x_1_1`, ..., `x_4_4` by using a format character vector as the first argument. `sym` replaces `%d` in the format character vector with the index of the element to generate the element names.

```
B = sym('x_%d_%d',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 create symbolic variables `A1_1`, ..., `A3_4`, `x_1_1`, ..., `x_4_4` in the MATLAB workspace. To access an element of a matrix, use parentheses.

```
A(2,3)
B(4,2)
```

```
ans =
A2_3

ans =
x_4_2
```

Create Symbolic Multidimensional Arrays

Create a 2-by-2-by-2 symbolic array with automatically generated elements `A1_1_1`, ..., `A2_2_2`.

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

```
A(:,:,1) =
[ a1_1_1, a1_2_1]
[ a2_1_1, a2_2_1]
A(:,:,2) =
[ a1_1_2, a1_2_2]
[ a2_1_2, a2_2_2]
```

Create Symbolic Numbers

Convert numeric values to symbolic numbers or expressions. Use `sym` on subexpressions instead of the entire expression for better accuracy. Using `sym` on entire expressions is inaccurate because MATLAB first converts the expression to a floating-point number, which loses accuracy. `sym` cannot always recover this lost accuracy.

```
inaccurate1 = sym(1/1234567)
accurate1 = 1/sym(1234567)

inaccurate2 = sym(sqrt(1234567))
accurate2 = sqrt(sym(1234567))

inaccurate3 = sym(exp(pi))
accurate3 = exp(sym(pi))
```

```
inaccurate1 =
7650239286923505/9444732965739290427392
accurate1 =
1/1234567

inaccurate2 =
4886716562018589/4398046511104
```

```
accurate2 =
1234567^(1/2)

inaccurate3 =
6513525919879993/281474976710656
accurate3 =
exp(pi)
```

Create Large Symbolic Numbers

When creating symbolic numbers with 15 or more digits, use quotation marks to accurately represent the numbers.

```
inaccurateNum = sym(11111111111111111111)
accurateNum = sym('11111111111111111111')
```

```
inaccurateNum =
11111111111111110656
accurateNum =
11111111111111111111
```

When you use quotation marks to create symbolic complex numbers, specify the imaginary part of a number as `1i`, `2i`, and so on.

```
sym('1234567 + 1i')
```

```
ans =
1234567 + 1i
```

Create Symbolic Expressions from Function Handles

Create a symbolic expression and a symbolic matrix from anonymous functions associated with MATLAB handles.

```
h_expr = @(x)(sin(x) + cos(x));
sym_expr = sym(h_expr)
```

```
sym_expr =
cos(x) + sin(x)
```

```
h_matrix = @(x)(x*pascal(3));
sym_matrix = sym(h_matrix)
```

```
sym_matrix =
[ x,   x,   x]
[ x, 2*x, 3*x]
[ x, 3*x, 6*x]
```

Set Assumptions While Creating Variables

Create the symbolic variables `x`, `y`, `z`, and `t` simultaneously assuming that `x` is real, `y` is positive, `z` integer, and `t` is rational.

```
x = sym('x','real');
y = sym('y','positive');
z = sym('z','integer');
t = sym('t','rational');
```

Check the assumptions on `x`, `y`, and `z` using `assumptions`.

```
assumptions

ans =
[ in(z, 'integer'), in(t, 'rational'), in(x, 'real'), 0 < y]
```

For further computations, clear the assumptions using `assume`.

```
assume([x y z t],'clear')
assumptions

ans =
Empty sym: 1-by-0
```

Set Assumptions on Matrix Elements

Create a symbolic matrix and set assumptions on each element of that matrix.

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

A =
[ A11, A12]
[ A21, A22]
```

Solve an equation involving the first element of `A`. MATLAB assumes that this element is positive.

```
solve(A(1, 1)^2 - 1, A(1, 1))
```

```
ans =
1
```

Check the assumptions set on the elements of `A` by using `assumptions`.

```
assumptions(A)
```

```
ans =
[ 0 < A11, 0 < A12, 0 < A21, 0 < A22]
```

Clear all previously set assumptions on elements of a symbolic matrix by using `assume`.

```
assume(A,'clear');
assumptions(A)
```

```
ans =
Empty sym: 1-by-0
```

Solve the same equation again.

```
solve(A(1, 1)^2 - 1, A(1, 1))
```

```
ans =
 -1
  1
```

Choose Conversion Technique for Floating-Point Values

Convert `pi` to a symbolic value.

Choose the conversion technique by specifying the optional second argument, which can be `'r'`, `'f'`, `'d'`, or `'e'`. The default is `'r'`. See the Input Arguments section for the details about conversion techniques.

```
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
```

Input Arguments

`x` — Variable name

character vector

Variable name, specified as a character vector. Argument `x` must a valid variable name. That is, `x` must begin with a letter and can contain only alphanumeric characters and underscores. To verify that the name is a valid variable name, use `isvarname`.

Example: `x`, `y123`, `z_1`

`h` — Anonymous function

MATLAB function handle

Anonymous function, specified as a MATLAB function handle

Example: `h = @(x)sin(x); symexpr = sym(h)`

`a` — Prefix for automatically generated matrix elements

character vector

Prefix for automatically generated matrix elements, specified as a character vector. Argument `a` must a valid variable name. That is, `a` must begin with a letter and can contain only alphanumeric characters and underscores. To verify that the name is a valid variable name, use `isvarname`.

Example: `a`, `b`, `a_bc`

`[n1 ... nM]` — Vector, matrix, or array dimensions

vector of integers

Vector, matrix, or array dimensions, specified as a vector of integers. As a shortcut, you also can use one integer to create a square matrix. For example, `A = sym('A',3)` creates a square `3`-by-`3` matrix.

Example: `[2 3]`, `[2,3]`, `[2;3]`

`set` — Assumptions on symbolic variable or matrix
`'real'` | `'positive'` | `'integer'` | `'rational'`

Assumptions on a symbolic variable or matrix, specified as one of these character vectors: `'real'`, `'positive'`, `'integer'`, or `'rational'`.

`num` — Numeric value to be converted to symbolic number or matrix
number | matrix of numbers

Numeric value to be converted to symbolic number or matrix, specified as a number or a matrix of numbers.

Example: `10`, `pi`,`hilb(3)`

`flag` — Conversion technique
`'r'` (default) | `'d'` | `'e'` | `'f'`

Conversion technique, specified as one of the characters listed in this table.

`'r'`	When `sym` uses the *rational* mode, it converts 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` 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 `sym` cannot find simple rational approximation, then it uses the same technique as it would use with the flag `'f'`.
`'d'`	When `sym` uses the *decimal* mode, it takes the number of digits from the current setting of `digits`. Conversions with fewer than 16 digits lose some accuracy, while more than 16 digits might not be warranted. For example, `sym(4/3,'d')` with the 10-digit accuracy returns `1.333333333`, while with the 20-digit accuracy it returns `1.3333333333333332593`. The latter does not end in `3`s, but it is an accurate decimal representation of the floating-point number nearest to `4/3`.
`'e'`	When `sym` uses the *estimate error* mode, it supplements a result obtained in the rational mode by a term involving the variable `eps`. This term estimates the difference between the theoretical rational expression and its actual floating-point value. For example, `sym(3*pi/4,'e')` returns `(3*pi)/4 - (103*eps)/249`.

'f'	When sym uses the *floating-point* mode, it represents all values in the form N*2^e or -N*2^e, where N >= 0 and e are integers. For example, sym(1/10,'f') returns 3602879701896397/36028797018963968 . The returned rational value is the exact value of the floating-point number that you convert to a symbolic number.

Output Arguments

x — Variable
symbolic variable

Variable, returned as a symbolic variable.

symexpr — Expression or matrix generated from anonymous MATLAB function
symbolic expression | symbolic matrix

Expression or matrix generated from an anonymous MATLAB function, returned as a symbolic expression or matrix.

A — Vector or matrix with automatically generated elements
symbolic vector | symbolic matrix

Vector or matrix with automatically generated elements, returned as a symbolic vector or matrix. The elements of this vector or matrix do not appear in the MATLAB workspace.

Tips

- 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.
- `sym` always treats `i` in character vector input as an identifier. To input the imaginary number `i`, use `1i` instead.
- `clear x` does not clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`, `sym`, or `syms`. To remove assumptions, use one of these options:

  - `assume(x,'clear')` removes all assumptions affecting `x`.
  - `clear all` clears all objects in the MATLAB workspace and resets the symbolic engine.
  - `assume` and `assumeAlso` provide more flexibility for setting assumptions on variable.
- When you replace one or more elements of a numeric vector or matrix with a symbolic number, MATLAB converts that number to a double-precision number.

  ```
  A = eye(3);
  A(1,1) = sym('pi')

  A =
      3.1416         0         0
           0    1.0000         0
           0         0    1.0000
  ```

  You cannot replace elements of a numeric vector or matrix with a symbolic variable, expression, or function because these elements cannot be converted to double-precision numbers. For example, `A(1,1) = sym('a')` throws an error.

Alternative Functionality

Alternative Approaches for Creating Symbolic Variables

To create several symbolic variables in one function call, use `syms`.

See Also

`assume` | `clear all` | `double` | `reset` | `str2sym` | `symfun` | `syms` | `symvar`

Topics

"Create Symbolic Numbers, Variables, and Expressions" on page 1-3
"Create Symbolic Functions" on page 1-7
"Create Symbolic Matrices" on page 1-9
"Use Assumptions on Symbolic Variables" on page 1-28

Introduced before R2006a

sym2cell

Convert symbolic array to cell array

Syntax

```
C = sym2cell(S)
```

Description

`C = sym2cell(S)` converts a symbolic array `S` to a cell array `C`. The resulting cell array has the same size and dimensions as the input symbolic array.

Examples

Convert Symbolic Array to Cell Array

Convert a matrix of symbolic variables and numbers to a cell array.

Create the following symbolic matrix.

```
syms x y
S =  [x 2 3 4; y 6 7 8; 9 10 11 12]
```

```
S =
[ x,  2,  3,  4]
[ y,  6,  7,  8]
[ 9, 10, 11, 12]
```

Convert this matrix to a cell array by using `sym2cell`. The size of the resulting cell array corresponds to the size of the input matrix. Each cell contains an element of the symbolic matrix `S`.

```
C = sym2cell(S)
```

```
C =
  3×4 cell array
```

```
    {1×1 sym}    {1×1 sym}    {1×1 sym}    {1×1 sym}
    {1×1 sym}    {1×1 sym}    {1×1 sym}    {1×1 sym}
    {1×1 sym}    {1×1 sym}    {1×1 sym}    {1×1 sym}
```

To access an element in each cell, use curly braces.

```
[C{1,1:4}]

ans =
[ x, 2, 3, 4]
```

```
[C{1:3,1}]

ans =
[ x, y, 9]
```

Input Arguments

S — Input symbolic array

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Input symbolic array, specified as a symbolic vector, matrix, or multidimensional array. `S` also can be a scalar, that is, a symbolic number, variable, expression, or function.

Output Arguments

C — Resulting cell array

cell array

Resulting cell array, returned as a cell array such that `size(C) = size(S)`. Each element of the input symbolic array `S` is enclosed in a separate cell.

See Also

`cell2mat` | `cell2sym` | `mat2cell` | `num2cell`

Introduced in R2016a

sym2poly

Extract vector of all numeric coefficients, including zeros, from symbolic polynomial

Syntax

```
c = sym2poly(p)
```

Description

`c = sym2poly(p)` returns the numeric vector of coefficients `c` of the symbolic polynomial `p`. The returned vector `c` includes all coefficients, including those equal `0`.

`sym2poly` returns coefficients in order of descending powers of the polynomial variable.

If $c_1x^{n-1}+c_2x^{n-2}+...+c_n$, then `c = sym2poly(p)` returns `c = [c1 c2 ... cn]`.

Examples

Extract Numeric Coefficients of Polynomial

Create row vectors of coefficients of symbolic polynomials.

Extract integer coefficients of a symbolic polynomial into a numeric row vector.

```
syms x
c = sym2poly(x^3 - 2*x - 5)
```

```
c =
     1     0    -2    -5
```

Extract rational and integer coefficients of a symbolic polynomial into a vector. Because `sym2poly` returns numeric double-precision results, it approximates exact rational coefficients with double-precision numbers.

```
c = sym2poly(1/2*x^3 - 2/3*x - 5)
```

```
c =
    0.5000         0   -0.6667   -5.0000
```

Input Arguments

p — Polynomial
symbolic expression

Polynomial, specified as a symbolic expression.

Output Arguments

c — Polynomial coefficients
numeric row vector

Polynomial coefficients, returned as a numeric row vector.

Tips

- To extract symbolic coefficients of a polynomial, use `coeffs`. This function returns a symbolic vector of coefficients and omits all zeros. For example, `syms a b x; c = coeffs(a*x^3 - 5*b,x)` returns `c = [ -5*b, a]`.

See Also

`coeffs` | `poly2sym`

Introduced before R2006a

symengine

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

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)

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

See Also

`evalin` | `feval` | `read`

Introduced in R2008b

symfun

Create symbolic functions

Syntax

```
f = symfun(formula,inputs)
```

Description

`f = symfun(formula,inputs)` creates the symbolic function `f`. The symbolic variables `inputs` represent its input arguments. The symbolic expression `formula` defines the body of the function `f`.

Examples

Create Symbolic Functions

Use `syms` to create symbolic variables. Then use `symfun` to create a symbolic function with these variables as its input arguments.

```
syms x y
f = symfun(x + y, [x y])
```

```
f(x, y) =
x + y
```

Call the function for `x = 1` and `y = 2`.

```
f(1,2)
```

```
ans =
3
```

Input Arguments

`formula` — Function body

symbolic expression | vector of symbolic expressions | matrix of symbolic expressions

Function body, specified as a symbolic expression, vector of symbolic expressions, or matrix of symbolic expressions.

Example: `x + y`

`inputs` — Input argument or arguments of function

symbolic variable | array of symbolic variables

Input argument or arguments of a function, specified as a symbolic variable or an array of symbolic variables, respectively.

Example: `[x,y]`

Output Arguments

`f` — Function

symbolic function (`symfun` data type)

Function, returned as a symbolic function (`symfun` data type).

Tips

- When you replace one or more elements of a numeric vector or matrix with a symbolic number, MATLAB converts that number to a double-precision number.

  ```
  A = eye(3);
  A(1,1) = sym('pi')

  A =
      3.1416         0         0
           0    1.0000         0
           0         0    1.0000
  ```

You cannot replace elements of a numeric vector or matrix with a symbolic variable, expression, or function because these elements cannot be converted to double-precision numbers. For example, `syms f(t); A(1,1) = f` throws an error.

- Symbolic functions are always scalars, therefore, you cannot index into a function. To access `x^2` and `x^4` in this example, use `formula` to get the expression that defines `f`, and then index into that expression.

  ```
  syms x
  f = symfun([x^2, x^4], x);

  expr = formula(f);

  expr(1)
  expr(2)

  ans =
  x^2

  ans =
  x^4
  ```

Alternative Functionality

Alternative Approaches for Creating Symbolic Functions

- 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, and the body of the function must be a symbolic number, variable, or expression. Assigning a number, such as `f(x,y) = 1`, causes an error.

  ```
  syms x y
  f(x,y) = x + y
  ```
- Use `syms` to create an abstract symbolic function `f(x,y)` and its arguments. The following command creates the symbolic function `f` and the symbolic variables `x` and `y`. Using `syms`, you also can create multiple symbolic functions in one function call.

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

See Also

`argnames` | `dsolve` | `formula` | `matlabFunction` | `odeToVectorField` | `sym` | `syms` | `symvar`

Topics

"Create Symbolic Functions" on page 1-7

Introduced in R2012a

sympref

Set symbolic preferences

Syntax

```
sympref(pref,value)
sympref(pref,'default')
sympref(pref)

sympref()
sympref('default')
sympref(allPref)
```

Description

`sympref(pref,value)` sets the symbolic preference `pref` to `value` and returns the previous value of `pref`. Symbolic preferences can affect the functions `fourier`, `ifourier`, and `heaviside`. These preferences persist between successive MATLAB sessions.

`sympref(pref,'default')` sets `pref` to its default value and returns the previous value of `pref`.

`sympref(pref)` returns the value of symbolic preference `pref`.

`sympref()` returns the values of all symbolic preferences in a structure.

`sympref('default')` sets all symbolic preferences to their default values and returns the previous values in a structure.

`sympref(allPref)` restores all symbolic preferences to the values in structure `allPref` and returns the previous values in a structure. `allPref` is the structure returned by a previous call to `sympref`.

Note Symbolic preferences persist between successive MATLAB sessions. MATLAB does not restore them for a new session.

Examples

Change Parameter Values of Fourier Transform

The Fourier transform $F(w)$ of $f = f(t)$ is

$$F(w) = c \int_{-\infty}^{\infty} f(t) e^{iswt} dt,$$

where c and s are parameters with default values 1 and -1. Other common values for c are 1/2π and $1/\sqrt{2\pi}$, and for `s` are 1, -2π, and 2π.

Find the Fourier transform of `sin(t)` with default values of `c` and `s`.

```
syms t w
fourier(sin(t),t,w)

ans =
-pi*(dirac(w - 1) - dirac(w + 1))*1i
```

Find the same Fourier transform for `c = 1/(2π)` and `s = 1`. Set these parameter values by using the `FourierParameter` preference of `sympref`. Represent `π` exactly using `sym`. The values of `c` and `s` are specified as the vector `[1/(2*sym(pi)) 1]`. Store the previous values returned by `sympref` to restore them later.

```
oldparam = sympref('FourierParameters',[1/(2*sym(pi)) 1])
fourier(sin(t),t,w)

oldparam =
[ 1, -1]

ans =
(dirac(w - 1)*1i)/2 - (dirac(w + 1)*1i)/2
```

The preferences set by `sympref` persist through your current and future MATLAB sessions. Restore the old values of `c` and `s` using the previous parameter values stored in `oldparam`.

```
sympref('FourierParameters',oldparam);
```

Alternatively, you can restore the default values of `c` and `s` by specifying the `'default'` option.

```
sympref('FourierParameters','default');
```

Change Value of Heaviside at Origin

The default value of the Heaviside function at the origin is 1/2 in the Symbolic Math Toolbox. Return the value of `heaviside(0)`. Find the Z-Transform of `heaviside(x)` for this default value of `heaviside(0)`.

```
syms x
heaviside(sym(0))
ztrans(heaviside(x))
```

```
ans =
1/2

ans =
1/(z - 1) + 1/2
```

Other common values for the Heaviside function at the origin are 0 and 1. Set `heaviside(0)` to `1` using the `'HeavisideAtOrigin'` preference of `sympref`. Store the old parameter value returned by `sympref` to restore it later.

```
oldparam = sympref('HeavisideAtOrigin',1)
```

```
oldparam =
1/2
```

Check the new value of `heaviside(0)`. Find the Z-Transform of `heaviside(x)` for this value.

```
heaviside(sym(0))
ztrans(heaviside(x))
```

```
ans =
1
```

```
ans =
1/(z - 1) + 1
```

The new output of `heaviside(0)` modifies the output of `ztrans`.

The preferences set by `sympref` persist throughout your current and future MATLAB sessions. Restore the previous value of `heaviside(0)` by loading the old parameter stored in `oldparam`.

```
sympref('HeavisideAtOrigin',oldparam);
```

Alternatively, you can restore the default value of `'HeavisideAtOrigin'` by specifying the `'default'` option.

```
sympref('HeavisideAtOrigin','default');
```

Modify Display of Symbolic Expressions in Live Scripts

By default, symbolic expressions in Live Scripts are typeset and, long expressions are abbreviated. You can turn off the use of typesetting and abbreviation using symbolic preferences.

Turn off abbreviations of long outputs by setting the preference `'AbbreviateOutput'` to `false`. First, show the abbreviated output.

```
syms a b c d x
f = a*x^3 + b*x^2 + c*x + d;
outputAbbrev = sin(f) + cos(f) + tan(f) + log(f) + 1/f
```

```
outputAbbrev =
```

$$\cos(\sigma_1) + \log(\sigma_1) + \sin(\sigma_1) + \tan(\sigma_1) + \frac{1}{\sigma_1}$$

where

$$\sigma_1 = a\,x^3 + b\,x^2 + c\,x + d$$

Turn off abbreviated output and display the same expression again.

```
sympref('AbbreviateOutput',false);
outputLong = sin(f) + cos(f) + tan(f) + log(f) + 1/f

outputLong =
```

$$\cos(a\,x^3+b\,x^2+c\,x+d)+\log(a\,x^3+b\,x^2+c\,x+d)+\sin(a\,x^3+b\,x^2+c\,x+d)+\tan(a\,x^3+b\,x^2+c\,x+d)+\frac{1}{a\,x^3+b\,x^2+c\,x+d}$$

Turn off rendered output and use ASCII output instead by setting the preference `'TypesetOutput'` to `false`. First, show the typeset output.

```
syms a b c d x
f = exp(a^b)+pi

f =
```

$$\pi+e^{a^b}$$

Turn off typeset output and display the same expression again.

```
sympref('TypesetOutput',false);
f = exp(a^b)+pi


f =

pi + exp(a^b)
```

The preferences set by `sympref` persist throughout your current and future MATLAB sessions. Restore the default values of `'AbbreviateOutput'` and `'TypesetOutput'` by using the option `'default'`.

```
sympref('AbbreviateOutput','default');
sympref('TypesetOutput','default');
```

Saving and Restoring All Symbolic Preferences

`sympref` can save and restore all symbolic preferences simultaneously in place of working with individual preferences.

Return the values of all symbolic preferences using `sympref`. The `sympref` function returns a structure of values of preferences. Access individual preferences by addressing the fields of the structure.

```
S = sympref;
S.FourierParameters
S.HeavisideAtOrigin
```

```
ans =
[ 1, -1]

ans =
1/2
```

`S` stores the values of all symbolic preferences.

Assume that you have changed the preferences. Since the preferences persist through your current and future MATLAB sessions, you want to restore your previous preferences in `S`. Restore the saved preferences using `sympref(S)`.

```
sympref(S);
```

Alternatively, you can set all symbolic preferences to their defaults by specifying the option `'default'`.

```
sympref('default');
```

Input Arguments

`pref` — Symbolic preference

character vector

Symbolic preference, specified as a character vector.

Example: `'HeavisideAtOrigin'`

`value` — Value of symbolic preference

numeric number | symbolic number

Value of the symbolic preference, specified as a numeric or symbolic number.

`allPref` — Values of all symbolic preferences

structure

Values of all symbolic preferences, specified as a structure. Typically, `allPref` is generated by a previous call to `sympref`.

Tips

- The commands `clear(all)` and `reset(symengine)` do *not* reset or affect symbolic preferences. Use `sympref` to manipulate symbolic preferences.

See Also

`fourier` | `heaviside` | `ifourier`

Introduced in R2015a

symprod

Product of series

Syntax

```
F = symprod(f,k,a,b)
F = symprod(f,k)
```

Description

`F = symprod(f,k,a,b)` returns the product of the series with terms that expression `f` specifies, which depend on symbolic variable `k`. The value of `k` ranges from `a` to `b`. If you do not specify `k`, `symprod` uses the variable that `symvar` determines. If `f` is a constant, then the default variable is `x`.

`F = symprod(f,k)` returns the product of the series that expression `f` specifies, which depend on symbolic variable `k`. The value of `k` starts at `1` with an unspecified upper bound. The product `F` is returned in terms of `k` where `k` represents the upper bound. This product `F` differs from the indefinite product. If you do not specify `k`, `symprod` uses the variable that `symvar` determines. If `f` is a constant, then the default variable is `x`.

Examples

Find Product of Series Specifying Bounds

Find the following products of series

$$P1 = \prod_{k=2}^{\infty} 1 - \frac{1}{k^2},$$

$$P2 = \prod_{k=2}^{\infty} \frac{k^2}{k^2 - 1}.$$

```
syms k
P1 = symprod(1 - 1/k^2, k, 2, Inf)
P2 = symprod(k^2/(k^2 - 1), k, 2, Inf)
```

```
P1 =
1/2
P2 =
2
```

Alternatively, specify bounds as a row or column vector.

```
syms k
P1 = symprod(1 - 1/k^2, k, [2 Inf])
P2 = symprod(k^2/(k^2 - 1), k, [2; Inf])
```

```
P1 =
1/2
P2 =
2
```

Find Product of Series Specifying Product Index and Bounds

Find the product of series

$$P = \prod_{k=1}^{10000} \frac{e^{kx}}{x}.$$

```
syms k x
P = symprod(exp(k*x)/x, k, 1, 10000)
```

```
P =
exp(50005000*x)/x^10000
```

Find Product of Series with Unspecified Bounds

When you do not specify the bounds of a series are unspecified, the variable `k` starts at `1`. In the returned expression, `k` itself represents the upper bound.

Find the products of series with an unspecified upper bound

$$P1 = \prod_{k} k,$$

$$P2 = \prod_{k} \frac{2k-1}{k^2}.$$

```
syms k
P1 = symprod(k, k)
P2 = symprod((2*k - 1)/k^2, k)

P1 =
factorial(k)
P2 =
(1/2^(2*k)*2^(k + 1)*factorial(2*k))/(2*factorial(k)^3)
```

Input Arguments

`f` — Expression defining terms of series

symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic number

Expression defining terms of series, specified as a symbolic expression, function, constant, or a vector or matrix of symbolic expressions, functions, or constants.

`k` — Product index

symbolic variable

Product index, specified as a symbolic variable. If you do not specify this variable, `symprod` uses the default variable that `symvar(expr,1)` determines. If `f` is a constant, then the default variable is `x`.

`a` — Lower bound of product index

number | symbolic number | symbolic variable | symbolic expression | symbolic function

Lower bound of product index, specified as a number, symbolic number, variable, expression, or function (including expressions and functions with infinities).

`b` — Upper bound of product index

number | symbolic number | symbolic variable | symbolic expression | symbolic function

Upper bound of product index, specified as a number, symbolic number, variable, expression, or function (including expressions and functions with infinities).

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

The indefinite product of x_i over i is

$$f(i) = \prod_{i} x_i$$

This definition holds under the assumption that the following identity is true for all values of i.

$$\frac{f(i+1)}{f(i)} = x_i$$

Note `symprod` does not compute indefinite products.

See Also

`cumprod` | `cumsum` | `int` | `syms` | `symsum` | `symvar`

Introduced in R2011b

symReadSSCParameters

Load parameters from Simscape component

Syntax

```
[names,values,units] = symReadSSCParameters(componentName)
```

Description

`[names,values,units] = symReadSSCParameters(componentName)` returns cell arrays containing the names, values, and units of all parameters from the Simscape component called `componentName`.

Examples

Parameters of Simscape Component

Load the names, values, and units of the parameters of a Simscape component.

Suppose you have the Simscape component `friction.ssc` in your current folder.

```
type('friction.ssc');
```

```
component friction < foundation.mechanical.rotational.branch

parameters
    brkwy_trq = { 25, 'N*m' };          % Breakaway friction torque
    Col_trq = { 20, 'N*m' };            % Coulomb friction torque
    visc_coef = { 0.001, 'N*m*s/rad' }; % Viscous friction coefficient
    trans_coef = { 10, 's/rad' };       % Transition approximation coefficient
    vel_thr = { 1e-4, 'rad/s' };        % Linear region velocity threshold
end

parameters (Access=private)
    brkwy_trq_th = { 24.995, 'N*m' };   % Breakaway torque at threshold velocity
```

```
end

function setup
    % Parameter range checking
    if brkwy_trq <= 0
        pm_error('simscape:GreaterThanZero','Breakaway friction torque' )
    end
    if Col_trq <= 0
        pm_error('simscape:GreaterThanZero','Coulomb friction torque' )
    end
    if Col_trq > brkwy_trq
        pm_error('simscape:LessThanOrEqual','Coulomb friction torque',...
                  'Breakaway friction torque')
    end
    if visc_coef < 0
        pm_error('simscape:GreaterThanOrEqualToZero','Viscous friction coefficient')
    end
    if trans_coef <= 0
        pm_error('simscape:GreaterThanZero','Transition approximation coefficient')
    end
    if vel_thr <= 0
        pm_error('simscape:GreaterThanZero','Linear region velocity threshold')
    end

    % Computing breakaway torque at threshold velocity
    brkwy_trq_th = visc_coef * vel_thr + Col_trq + (brkwy_trq - Col_trq) * ...
        exp(-trans_coef * vel_thr);
end

equations
    if (abs(w) <= vel_thr)
        % Linear region
        t == brkwy_trq_th * w / vel_thr;
    elseif w > 0
        t == visc_coef * w + Col_trq + ...
            (brkwy_trq - Col_trq) * exp(-trans_coef * w);
    else
        t == visc_coef * w - Col_trq - ...
            (brkwy_trq - Col_trq) * exp(-trans_coef * abs(w));
    end
end

end
```

Load the names, values, and units of the parameters of the component `friction.ssc`.

```
[names,values,units] = symReadSSCParameters('friction.ssc');
```

In this example, all elements of the resulting cell arrays are scalars. You can convert the cell arrays to symbolic vectors.

```
names_sym = cell2sym(names)
```

```
names_sym =
[ Col_trq, brkwy_trq, brkwy_trq_th, trans_coef, vel_thr, visc_coef]
```

```
values_sym = cell2sym(values)
```

```
values_sym =
[ 20, 25, 4999/200, 10, 1/10000, 1/1000]
```

Create individual symbolic variables from the elements of the cell array `names` in the MATLAB workspace. This command creates the symbolic variables `Col_trq`, `brkwy_trq`, `brkwy_trq_th`, `trans_coef`, `vel_thr`, and `visc_coef` as `sym` objects in the workspace.

```
syms(names)
```

Input Arguments

`componentName` — Simscape component name
file name enclosed in single quotes

Simscape component name, specified as a file name enclosed in single quotes. The file must have the extension `.ssc`. If you do not provide the file extension, `symReadSSCParameters` assumes it to be `.ssc`. The component must be on the MATLAB path or in the current folder.

Example: `'MyComponent.ssc'`

Output Arguments

`names` — Names of all parameters of Simscape component
cell array

Names of all parameters of a Simscape component, returned as a cell array.

Data Types: `cell`

`values` — Values of all parameters of Simscape component
cell array

Values of all parameters of a Simscape component, returned as a cell array.

Data Types: `cell`

`units` — Units of all parameters of Simscape component
cell array

Units of all parameters of a Simscape component, returned as a cell array.

Data Types: `cell`

See Also

`symReadSSCVariables` | `symWriteSSC`

Introduced in R2016a

symReadSSCVariables

Load variables from Simscape component

Syntax

```
[names,values,units] = symReadSSCVariables(componentName)
[names,values,units] = symReadSSCVariables(componentName,Name,Value)
```

Description

`[names,values,units] = symReadSSCVariables(componentName)` returns cell arrays containing the names, values, and units of all variables from the Simscape component called `componentName`.

`[names,values,units] = symReadSSCVariables(componentName,Name,Value)` uses additional options specified by `Name,Value` pair arguments.

Examples

Variables of Simscape Component

Load the names, values, and units of the variables of a Simscape component.

Suppose you have the Simscape component `friction.ssc` in your current folder.

```
type('friction.ssc');
```

```
component friction < foundation.mechanical.rotational.branch

parameters
    brkwy_trq = { 25, 'N*m' };          % Breakaway friction torque
    Col_trq = { 20, 'N*m' };            % Coulomb friction torque
    visc_coef = { 0.001, 'N*m*s/rad' }; % Viscous friction coefficient
    trans_coef = { 10, 's/rad' };       % Transition approximation coefficient
```

```
        vel_thr = { 1e-4, 'rad/s' };        % Linear region velocity threshold
    end

    parameters (Access=private)
        brkwy_trq_th = { 24.995, 'N*m' };   % Breakaway torque at threshold velocity
    end

    function setup
        % Parameter range checking
        if brkwy_trq <= 0
            pm_error('simscape:GreaterThanZero','Breakaway friction torque' )
        end
        if Col_trq <= 0
            pm_error('simscape:GreaterThanZero','Coulomb friction torque' )
        end
        if Col_trq > brkwy_trq
            pm_error('simscape:LessThanOrEqual','Coulomb friction torque',...
                     'Breakaway friction torque')
        end
        if visc_coef < 0
            pm_error('simscape:GreaterThanOrEqualToZero','Viscous friction coefficient')
        end
        if trans_coef <= 0
            pm_error('simscape:GreaterThanZero','Transition approximation coefficient')
        end
        if vel_thr <= 0
            pm_error('simscape:GreaterThanZero','Linear region velocity threshold')
        end

        % Computing breakaway torque at threshold velocity
        brkwy_trq_th = visc_coef * vel_thr + Col_trq + (brkwy_trq - Col_trq) * ...
            exp(-trans_coef * vel_thr);
    end

    equations
        if (abs(w) <= vel_thr)
            % Linear region
            t == brkwy_trq_th * w / vel_thr;
        elseif w > 0
            t == visc_coef * w + Col_trq + ...
                (brkwy_trq - Col_trq) * exp(-trans_coef * w);
        else
            t == visc_coef * w - Col_trq - ...
                (brkwy_trq - Col_trq) * exp(-trans_coef * abs(w));
    end
end

end
```

Load the names, values, and units of the variables of the component `friction.ssc`.

```
[names,values,units] = symReadSSCVariables('friction.ssc');
```

In this example, all elements of the resulting cell arrays are scalars. You can convert the cell arrays to symbolic vectors.

```
names_sym = cell2sym(names)
```

```
names_sym =
[ t, w]
```

```
values_sym = cell2sym(values)
```

```
values_sym =
[ 0, 0]
```

Create individual symbolic variables from the elements of the cell array `names` in the MATLAB workspace. This command creates the symbolic variables `t` and `w` as `sym` objects in the workspace.

```
syms(names)
```

Variables of Simscape Component Returned as Functions

Load the names of the variables of a Simscape component while converting them to symbolic functions of the variable `t`.

Suppose you have the Simscape component `source.ssc` in your current folder.

```
type('source.ssc');
```

```
component source
% Electrical Source
% Defines an electrical source with positive and negative external nodes.
% Also defines associated through and across variables.

nodes
```

```
    p = foundation.electrical.electrical; % :top
    n = foundation.electrical.electrical; % :bottom
end

variables(Access=protected)
    i = { 0, 'A' }; % Current
    v = { 0, 'V' }; % Voltage
end

branches
    i : p.i -> n.i;
end

equations
     v == p.v - n.v;
end

end
```

Load the names the variables of the component `source.ssc` setting `ReturnFunction` to `true`.

```
[names,~,~] = symReadSSCVariables('source.ssc','ReturnFunction',true);
```

In this example, all elements of the resulting cell arrays are scalars. You can convert the cell arrays to symbolic vectors.

```
names_symfun = cell2sym(names)
```

```
names_symfun =
[ i(t), v(t)]
```

Create individual symbolic functions from the elements of the cell array `names` in the MATLAB workspace. This command creates the symbolic functions `i` and `v` as `symfun` objects, and their variable `t` as a `sym` object in the workspace.

```
syms(names)
```

Input Arguments

`componentName` — Simscape component name

file name enclosed in single quotes

Simscape component name, specified as a file name enclosed in single quotes. The file must have the extension `.ssc`. If you do not provide the file extension, `symReadSSCVariables` assumes it to be `.ssc`. The component must be on the MATLAB path or in the current folder.

Example: `'MyComponent.ssc'`

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`.

Example: `'ReturnFunctions',true`

`ReturnFunctions` — Flag returning names of Simscape component variables as symbolic functions of variable **`t`**

`false` (default) | `true`

Flag returning names of Simscape component variables as symbolic functions of variable `t`, specified as `true` or `false`.

Example: `'ReturnFunctions',true`

Output Arguments

`names` — Names of all variables of Simscape component

cell array

Names of all variables of a Simscape component, returned as a cell array.

Data Types: `cell`

`values` — Values of all variables of Simscape component

cell array

Values of all variables of a Simscape component, returned as a cell array.

Data Types: `cell`

`units` — Units of all variables of Simscape component
cell array

Units of all variables of a Simscape component, returned as a cell array.

Data Types: `cell`

See Also

`symReadSSCParameters` | `symWriteSSC`

Introduced in R2016a

syms

Create symbolic variables and functions

Syntax

```
syms var1 ... varN
syms var1 ... varN set
syms var1 ... varN clear
syms f(var1,...,varN)

syms(symArray)

syms
S = syms
```

Description

`syms var1 ... varN` creates symbolic variables `var1 ... varN`. Separate variables by spaces.

`syms var1 ... varN set` sets an assumption that the created symbolic variables belong to a `set`.

`syms var1 ... varN clear` clears assumptions set on a symbolic variables `var1 ... varN`.

`syms f(var1,...,varN)` creates the symbolic function `f` and symbolic variables `var1,...,varN` representing the input arguments of `f`. You can create multiple symbolic functions in one call. For example, `syms f(x) g(t)` creates two symbolic functions (`f` and `g`) and two symbolic variables (`x` and `t`).

`syms(symArray)` creates the symbolic variables and functions contained in `symArray`, where `symArray` is either a vector of symbolic variables or a cell array of symbolic variables and functions. Use this syntax only when such an array is returned by another function, such as `solve` or `symReadSSCVariables`.

`syms` lists the names of all symbolic variables, functions, and arrays in the MATLAB workspace.

`S = syms` returns a cell array of the names of all symbolic variables, functions, and arrays.

Examples

Create Symbolic Variables

Create symbolic variables `x` and `y`.

```
syms x y
```

Set Assumptions While Creating Variables

Create symbolic variables `x` and `y`, and assume that they are integers.

```
syms x y integer
```

Check assumptions.

```
assumptions
```

```
ans =
[ in(x, 'integer'), in(y, 'integer')]
```

Alternatively, check assumptions on each variable. For example, check assumptions set on the variable `x`.

```
assumptions(x)
```

```
ans =
in(x, 'integer')
```

Clear assumptions on `x` and `y`.

```
assume([x y],'clear')
assumptions
```

```
ans =
Empty sym: 1-by-0
```

Create Symbolic Functions

Create symbolic functions with one and two arguments.

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

Both `s` and `f` are abstract symbolic functions. They do not have symbolic expressions assigned to them, so the bodies of these functions are `s(t)` and `f(x,y)`, respectively.

Specify the following formula for `f`.

```
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 Functions with Matrices as Formulas

Create a symbolic function and specify its formula by using a symbolic matrix.

```
syms x
f(x) = [x x^3; x^2 x^4]
```

```
f(x) =
[   x, x^3]
[ x^2, x^4]
```

Compute the function value at the point `x = 2`:

```
f(2)
```

```
ans =
[ 2,  8]
[ 4, 16]
```

Compute the value of this function for `x = [1 2 3; 4 5 6]`. The result is a cell array of symbolic matrices.

```
y = f([1 2 3; 4 5 6])

y =
  2×2 cell array
    {2×3 sym}    {2×3 sym}
    {2×3 sym}    {2×3 sym}
```

Access the contents of a cell in the cell array by using braces.

```
y{1}

ans =
[ 1, 2, 3]
[ 4, 5, 6]
```

Create Objects from Array of Symbolic Variables and Functions

Certain functions, such as `solve` and `symReadSSCVariables`, can return a vector of symbolic variables or a cell array of symbolic variables and functions. These variables or functions do not automatically appear in the MATLAB workspace. Create these variables or functions from the vector or cell array by using `syms`.

Solve the equation `sin(x) == 1` by using `solve`. The parameter `k` in the solution does not appear in the MATLAB workspace.

```
syms x
[sol, parameter, condition] = solve(sin(x) == 1, x, 'ReturnConditions', true);
parameter

parameter =
k
```

Create the parameter `k` by using `syms`. The parameter `k` now appears in the MATLAB workspace.

```
syms(parameter)
```

Similarly, use `syms` to create the symbolic objects contained in a vector or cell array. Examples of functions that return a cell array of symbolic objects are `symReadSSCVariables` and `symReadSSCParameters`.

List All Symbolic Variables, Functions, and Arrays

Create some symbolic variables, functions, and arrays.

```
syms a f(x)
A = sym('A',[2 3]);
```

Display a list of all symbolic objects that currently exist in the MATLAB workspace by using `syms`.

```
syms
```

```
Your symbolic variables are:

A  a  f  x
```

Instead of displaying a list, return a cell array of all symbolic objects by providing an output to `syms`.

```
S = syms
```

```
S =
  4×1 cell array
    {'A'}
    {'a'}
    {'f'}
    {'x'}
```

Check for a Symbolic Variable, Function, or Array

Create some symbolic variables, functions, and arrays.

```
syms a f(x)
A = sym('A',[2 3]);
```

Check if `x` exists in the output of `syms` by using `ismember` and `any`. The `any` function returns logical `1` (`true`), meaning `x` does exist in the output of `syms`.

```
checkVar = sym('x');
S = syms;
any(ismember(S,checkVar))
```

```
ans =
  logical
   1
```

Delete All Symbolic Variables, Functions, or Arrays

Create several symbolic objects.

```
syms a b c f(x)
```

Delete all symbolic objects by clearing the output of `syms`.

```
symObj = syms;
cellfun(@clear,symObj)
```

Check that you deleted all symbolic objects by calling `syms`. The output is empty meaning no symbolic objects exist in the MATLAB workspace.

```
syms
```

Input Arguments

`var1 ... varN` — Symbolic variables
valid variable names separated by spaces

Symbolic variables, specified as valid variable names separated by spaces. Each variable name must begin with a letter and can contain only alphanumeric characters and underscores. To verify that the name is a valid variable name, use `isvarname`.

Example: `x y123 z_1`

`set` — Assumptions on symbolic variables
`real` | `positive` | `integer` | `rational`

Assumptions on a symbolic variable or matrix, specified as `real`, `positive`, `integer`, or `rational`.

`f(var1,...,varN)` — Symbolic function with its input arguments
expression with parentheses

Symbolic function with its input arguments, specified as an expression with parentheses. The function name `f` and the variable names `var1...varN` must be valid variable names. That is, they must begin with a letter and can contain only alphanumeric characters and underscores. To verify that the name is a valid variable name, use `isvarname`.

Example: `s(t)`, `f(x,y)`

`symArray` — Symbolic variables and functions
vector of symbolic variables | cell array of symbolic variables and functions

Symbolic variables or functions, specified as a vector of symbolic variables or a cell array of symbolic variables and functions. Such a vector or array is typically the output of another function, such as `solve` or `symReadSSCVariables`.

Output Arguments

`S` — Names of all symbolic variables, functions, and arrays
cell array of character vectors

Names of all symbolic variables, functions, and arrays in the MATLAB workspace, returned as a cell array of character vectors.

Tips

- `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. You also can use `symfun` to create symbolic functions.
- 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 a MATLAB function inside a function or a script, use `sym`. For example, use `alpha = sym('alpha')`.
- The following variable names are invalid with `syms`: `integer`, `real`, `rational`, `positive`, and `clear`. To create variables with these names, use `sym`. For example, `real = sym('real')`.
- `clear x` does not clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`, `sym`, or `syms`. To remove assumptions, use one of these options:

  - `assume(x,'clear')` removes all assumptions affecting `x`.
  - `clear all` clears all objects in the MATLAB workspace and resets the symbolic engine.
  - `assume` and `assumeAlso` provide more flexibility for setting assumptions on variables.
- When you replace one or more elements of a numeric vector or matrix with a symbolic number, MATLAB converts that number to a double-precision number.

  ```
  A = eye(3);
  A(1,1) = sym('pi')

  A =
      3.1416         0         0
           0    1.0000         0
           0         0    1.0000
  ```

  You cannot replace elements of a numeric vector or matrix with a symbolic variable, expression, or function because these elements cannot be converted to double-precision numbers. For example, `syms a; A(1,1) = a` throws an error.

See Also

`assume` | `assumeAlso` | `assumptions` | `clear all` | `reset` | `sym` | `symfun` | `symvar`

Topics

"Create Symbolic Numbers, Variables, and Expressions" on page 1-3
"Create Symbolic Functions" on page 1-7
"Create Symbolic Matrices" on page 1-9
"Use Assumptions on Symbolic Variables" on page 1-28

Introduced before R2006a

symsum

Sum of series

Syntax

```
F = symsum(f,k,a,b)
F = symsum(f,k)
```

Description

`F = symsum(f,k,a,b)` returns the sum of the series with terms that expression `f` specifies, which depend on symbolic variable `k`. The value of `k` ranges from `a` to `b`. If you do not specify the variable, `symsum` uses the variable that `symvar` determines. If `f` is a constant, then the default variable is `x`.

`F = symsum(f,k)` returns the indefinite sum `F` of the series with terms that expression `f` specifies, which depend on variable `k`. The `f` argument defines the series such that the indefinite sum `F` is given by `F(k+1) - F(k) = f(k)`. If you do not specify the variable, `symsum` uses the variable that `symvar` determines. If `f` is a constant, then the default variable is `x`.

Examples

Find Sum of Series Specifying Bounds

Find the following sums of series.

$$S1 = \sum_{k=0}^{10} k^2$$

$$S2 = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

$$S3 = \sum_{k=1}^{\infty} \frac{x^k}{k!}$$

```
syms k x
S1 = symsum(k^2, k, 0, 10)
S2 = symsum(1/k^2, k, 1, Inf)
S3 = symsum(x^k/factorial(k), k, 0, Inf)
```

```
S1 =
385
S2 =
pi^2/6
S3 =
exp(x)
```

Alternatively, specify bounds as a row or column vector.

```
S1 = symsum(k^2, k, [0 10])
S2 = symsum(1/k^2, k, [1; Inf])
S3 = symsum(x^k/factorial(k), k, [0 Inf])
```

```
S1 =
385
S2 =
pi^2/6
S3 =
exp(x)
```

Find Indefinite Sum of Series

Find the indefinite sum of the series specified by the symbolic expressions `k` and `k^2`.

```
syms k
symsum(k, k)
symsum(1/k^2, k)
```

```
ans =
k^2/2 - k/2
```

```
ans =
-psi(1, k)
```

Difference between `symsum` and `sum`

The `sum` function finds the sum of elements of symbolic vectors and matrices.

Consider the definite sum

$$S = \sum_{k=1}^{10} \frac{1}{k^2}.$$

Contrast `symsum` and `sum` by summing this definite sum using both functions.

```
syms k
S_sum = sum(subs(1/k^2, k, 1:10))
S_symsum = symsum(1/k^2, k, 1, 10)
```

```
S_sum =
1968329/1270080
S_symsum =
1968329/1270080
```

For details on `sum`, see the information on the MATLAB `sum` page.

Input Arguments

`f` — Expression defining terms of series

symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic number

Expression defining terms of series, specified as a symbolic expression, function, or a vector or matrix of symbolic expressions, functions, or constants.

`k` — Summation index

symbolic variable

Summation index, specified as a symbolic variable. If you do not specify this variable, `symsum` uses the default variable determined by `symvar(expr,1)`. If `f` is a constant, then the default variable is `x`.

`a` — Lower bound of summation index

number | symbolic number | symbolic variable | symbolic expression | symbolic function

Lower bound of summation index, specified as a number, symbolic number, variable, expression, or function (including expressions and functions with infinities).

`b` — Upper bound of summation index

number | symbolic number | symbolic variable | symbolic expression | symbolic function

Upper bound of summation index, specified as a number, symbolic number, variable, expression, or function (including expressions and functions with infinities).

Definitions

Definite Sum

The definite sum of series is defined as

$$\sum_{k=a}^{b} x_k = x_a + x_{a+1} + \ldots + x_b.$$

Indefinite Sum

The indefinite sum of a series is defined as

$$F(x) = \sum_{x} f(x),$$

such that

$$F(x+1) - F(x) = f(x).$$

See Also

`cumsum` | `int` | `sum` | `symprod` | `syms` | `symvar`

Topics

“Symbolic Summation” on page 2-64

Introduced before R2006a

symunit

Units of measurement

Syntax

```
u = symunit
```

Description

`u = symunit` returns the units collection. Then, specify any unit by using `u.`*`unit`*. For example, specify `3` meters as `3*u.m`.

Examples

Specify Units of Measurement

Before specifying units, load units by using `symunit`. Then, specify a unit by using dot notation.

Specify a length of `3` meters. In displayed output, units are placed in square brackets `[]`.

```
u = symunit;
length = 3*u.m

length =
3*[m]
```

Tip Use tab expansion to find names of units. Type `u.`, press **Tab**, and continue typing.

Specify the acceleration due to gravity of `9.81` meters per second squared. Because units are symbolic expressions, numeric inputs are converted to exact symbolic values. Here, `9.81` is converted to `981/100`.

```
g = 9.81*u.m/u.s^2

g =
(981/100)*([m]/[s]^2)
```

If you are unfamiliar with the differences between symbolic and numeric arithmetic, see "Choose Symbolic or Numeric Arithmetic" on page 2-114.

Operations on Units and Conversion to Double

Units behave like symbolic expressions when you perform standard operations on them. For numeric operations, separate the value from the units, substitute for any symbolic parameters, and convert the result to double.

Find the speed required to travel `5` km in `2` hours.

```
u = symunit;
d = 5*u.km;
t = 2*u.hr;
s = d/t

s =
(5/2)*([km]/[h])
```

The value `5/2` is symbolic. You may prefer double output, or require double output for a MATLAB function that does not accept symbolic values. Convert to double by separating the numeric value using `seperateUnits` and then using `double`.

```
[sNum,sUnits] = separateUnits(s)

sNum =
5/2
sUnits =
1*([km]/[h])

sNum = double(sNum)

sNum =
    2.5000
```

For the complete units workflow, see "Units of Measurement Tutorial" on page 2-5.

Rewrite Between Units

Use your preferred unit by rewriting units using `rewrite`. Also, instead of specifying specific units, you can specify that the output should be in terms of SI units.

Calculate the force required to accelerate `2` kg by `5` m/s^2. The expression is not automatically rewritten in terms of Newtons.

```
u = symunit;
m = 2*u.kg;
a = 5*u.m/u.s^2;
F = m*a

F =
10*(([kg]*[m])/[s]^2)
```

Rewrite the expression in terms of Newtons by using `rewrite`.

```
F = rewrite(F,u.N)

F =
10*[N]
```

Rewrite `5` cm in terms of inches.

```
length = 5*u.cm;
length = rewrite(length,u.in)

length =
(250/127)*[in]
```

Rewrite `length` in terms of SI units. The result is in meters.

```
length = rewrite(length,'SI')

length =
(1/20)*[m]
```

Simplify Units of Same Dimension

Simplify expressions containing units of the same dimension by using `simplify`. Units are not automatically simplified or checked for consistency unless you call `simplify`.

```
u = symunit;
expr = 300*u.cm + 40*u.inch + 2*u.m

expr =
300*[cm] + 40*[in] + 2*[m]

expr = simplify(expr)

expr =
(3008/5)*[cm]
```

`simplify` automatically chooses the unit to rewrite in terms of. To choose a specific unit, see “Rewrite Between Units” on page 4-1612.

Temperature: Absolute and Difference Forms

By default, temperatures are assumed to represent temperature differences. For example, `5*u.Celsius` represents a temperature difference of 5 degrees Celsius. This assumption allows arithmetical operations on temperature values, and conversion between temperature scales.

For representing absolute temperatures, use degrees Kelvin, so that you do not have to distinguish an absolute temperature from a temperature difference.

Rewrite `23` degrees Celsius to Kelvin, treating it first as a temperature difference and then as an absolute temperature.

```
u = symunit;
T = 23*u.Celsius;
diffK = rewrite(T,u.K)

diffK =
23*[K]

absK = rewrite(T,u.K,'Temperature','absolute')

absK =
(5923/20)*[K]
```

Tips

- `1` represents a dimensionless unit. Hence, `isUnit(sym(1))` returns logical `1` (`true`).

- Certain non-linear units, such as decibels, are not implemented because arithmetic operations are not possible for these units.
- Instead of using dot notation to specify units, you can alternatively use string input as `symunit(unit)`. For example, `symunit('m')` specifies the unit meter.

See Also

`checkUnits` | `isUnit` | `newUnit` | `rewrite` | `separateUnits` | `symunit2str` | `unitConversionFactor` | `unitInfo`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

symunit2str

Convert unit to character vector

Syntax

```
symunit2str(unit)
symunit2str(unit,toolbox)
```

Description

`symunit2str(unit)` converts the symbolic unit `unit` to a character vector.

`symunit2str(unit,toolbox)` converts the symbolic unit `unit` to a character vector representing units in the toolbox `toolbox`. The allowed values of `toolbox` are `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`.

Examples

Convert Unit to Character Vector

Convert the symbolic unit `u.km` to a character vector, where `u = symunit`.

```
u = symunit;
unitStr = symunit2str(u.km)
```

```
unitStr =
    'km'
```

Convert Unit for Specified Toolbox

Convert symbolic units to character vectors representing units in other toolboxes by specifying the toolbox name as the second argument to `symunit2str`. The allowed toolboxes are `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`. The unit must exist in the target toolbox to be valid.

Where `u = symunit`, convert `u.km/(u.hour*u.s)` to a character vector representing units from Aerospace Toolbox.

```
u = symunit;
unit = symunit2str(u.km/(u.hour*u.s),'Aerospace')

unit =
    'km/h-s'
```

Convert `u.molecule/u.s` to a character vector representing units from SimBiology.

```
unit = symunit2str(u.molecule/u.s,'SimBiology')

unit =
    'molecule/second'
```

Convert `u.gn/u.km` to a character vector representing units from Simscape.

```
unit = symunit2str(u.gn/u.km,'Simscape')

unit =
    'gee/km'
```

Convert `u.rad/u.s` to a character vector representing units from Simulink.

```
unit = symunit2str(u.rad/u.s,'Simulink')

unit =
    'rad/s'
```

Input Arguments

`unit` — Symbolic unit to convert

symbolic unit

Symbolic unit to convert, specified as a symbolic unit.

`toolbox` — Toolbox to represent unit in

`'Aerospace'` | `'SimBiology'` | `'Simscape'` | `'Simulink'`

Toolbox to represent unit in, specified as `'Aerospace'`, `'SimBiology'`, `'Simscape'`, or `'Simulink'`.

Example: `symunit2str(u.km/u.h,'Aerospace')`

See Also

`checkUnits` | `findUnits` | `isUnit` | `newUnit` | `separateUnits` | `str2symunit` | `symunit` | `unitConversionFactor`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

symvar

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)` chooses the `n` symbolic variables in `s` that are alphabetically closest to `x` and sorts them alphabetically before returning them. If `s` is a symbolic function, `symvar(s,n)` returns the input arguments of `s` before other free variables in `s`.

Input Arguments

s

Symbolic expression, matrix, or function.

n

Integer or `Inf`. If `n` exceeds the number of variables in `s`, then `symvar(s,n)` is equivalent to `symvar(s,m)` where `m` is the number of variables in `s`.

Examples

Find all symbolic variables in this 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]
```

Find the first three symbolic variables in `f`. For a symbolic function, `symvar` with two arguments returns the function inputs before other variables.

```
symvar(f, 3)

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

Find the first three symbolic variables in an expression. For a symbolic expression or matrix, `symvar` chooses variables alphabetically closest to `x` and sorts them alphabetically.

```
symvar(a*x^2/(sin(3*y - b)), 3)

ans =
[ b, x, y]
```

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
```

Tips

- `symvar` treats the constants `pi`, `i`, and `j` as variables.
- If there are no symbolic variables in `s`, `symvar` returns the empty vector.
- When differentiating, integrating, substituting, 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`.

Algorithms

When `symvar` sorts the symbolic variables alphabetically, all uppercase letters have precedence over lowercase: 0 1 ... 9 A B ... Z a b ... z.

See Also

`argnames` | `sym` | `symfun` | `syms`

Topics

"Find Symbolic Variables in Expressions, Functions, Matrices" on page 2-3

Introduced in R2008b

symWriteSSC

Create new Simscape component

Syntax

```
symWriteSSC(newComponentName,templateComponentName,eqns)
symWriteSSC(newComponentName,templateComponentName,eqns,Name,Value)
```

Description

`symWriteSSC(newComponentName,templateComponentName,eqns)` creates a new Simscape component `newComponentName` using an existing component `templateComponentName` as a template and adding `eqns`. Thus, the new component has both the existing equations taken from the template component and the added equations.

`symWriteSSC(newComponentName,templateComponentName,eqns,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Component with Additional Equation

Create a new Simscape component by using an existing component as a template and adding an equation.

Suppose you have the Simscape component `spring.ssc` in your current folder.

```
type('spring.ssc');
```

```
component spring < foundation.mechanical.rotational.branch

parameters
    spr_rate = { 10, 'N*m/rad' };
```

```
end

variables
    phi = { value = { 0, 'rad'}, priority = priority.high };
end

function setup
    if spr_rate <= 0
        pm_error('simscape:GreaterThanZero','Spring rate' )
    end
end

equations
    w == phi.der;
    t == spr_rate*phi;
end

end
```

Create symbolic variables with names of the parameters and variables of the component you are going to use when creating new equations. Also create a symbolic variable, `u`, to denote the energy of the rotational spring.

```
syms spr_rate phi u
```

Create the equation defining the energy `u`.

```
eq = u == spr_rate*phi^2/2;
```

Create the new component, `myRotationalSpring.ssc`, that is a copy of the component `spring.ssc` with an additional equation defining the energy of the rotational spring.

```
symWriteSSC('myRotationalSpring.ssc','spring.ssc',eq)
```

```
Warning: Equations contain undeclared variables 'u'.
> In symWriteSSC (line 94)
```

`symWriteSSC` creates the component `myRotationalSpring.ssc`.

```
type('myRotationalSpring.ssc');
```

```
component myRotationalSpring

parameters
    spr_rate = { 10, 'N*m/rad' };
```

```
end

variables
    phi = { value = { 0, 'rad'}, priority = priority.high };
end

function setup
    if spr_rate <= 0
        pm_error('simscape:GreaterThanZero','Spring rate' )
    end
end

equations
    w == phi.der;
    t == spr_rate*phi;
    u == phi^2*spr_rate*(1.0/2.0);
end

end
```

Add Component Title and Description

Create a Simscape component with the title and descriptive text different from those of the template component.

Suppose you have the Simscape component `spring.ssc` in your current folder. This component does not have any title or descriptive text.

```
type('spring.ssc');
```

```
component spring < foundation.mechanical.rotational.branch

parameters
    spr_rate = { 10, 'N*m/rad' };
end

variables
    phi = { value = { 0, 'rad'}, priority = priority.high };
end

function setup
    if spr_rate <= 0
        pm_error('simscape:GreaterThanZero','Spring rate' )
    end
end

equations
    w == phi.der;
    t == spr_rate*phi;
end

end
```

Create symbolic variables with names of the parameters and variables of the component you are going to use when creating new equations. Also create a symbolic variable, `u`, to denote the energy of the rotational spring.

```
syms spr_rate phi u
```

Create the equation defining the energy `u`.

```
eq = u == spr_rate*phi^2/2;
```

Create the new component, `myRotationalSpring.ssc`, based on the `spring.ssc` component. Add the equation `eq`, the title “Rotational Spring”, and a few lines of descriptive text to the new component.

```
symWriteSSC('myRotationalSpring.ssc','spring.ssc',eq,...
'H1Header','% Rotational Spring',...
'HelpText',{'% The block represents an ideal mechanical rotational linear spring.',...
            '% Connections R and C are mechanical rotational conserving ports.'...
            '% The block positive direction is from port R to port C. This means'...
            '% that the torque is positive if it acts in the direction from R to C.'})
```

```
Warning: Equations contain undeclared variables 'u'.
> In symWriteSSC (line 94)
```

`symWriteSSC` creates the component `myRotationalSpring.ssc`.

```
type('myRotationalSpring.ssc');
```

```
component myRotationalSpring
% Rotational Spring
% The block represents an ideal mechanical rotational linear spring.
% Connections R and C are mechanical rotational conserving ports.
% The block positive direction is from port R to port C. This means
% that the torque is positive if it acts in the direction from R to C.
```

```
parameters
    spr_rate = { 10, 'N*m/rad' };
end

variables
    phi = { value = { 0, 'rad'}, priority = priority.high };
end

function setup
    if spr_rate <= 0
        pm_error('simscape:GreaterThanZero','Spring rate' )
    end
end

equations
    w == phi.der;
    t == spr_rate*phi;
    u == phi^2*spr_rate*(1.0/2.0);
end

end
```

Input Arguments

`newComponentName` — Name of Simscape component to create

file name enclosed in single quotes

Name of Simscape component to create, specified as a file name enclosed in single quotes. File must have the extension `.ssc`. If you do not provide file extension, `symWriteSSC` assumes it to be `.ssc`. If you do not specify the absolute path, `symWriteSSC` creates the new component in the current folder.

Example: `'MyNewComponent.ssc'`

`templateComponentName` — Name of template Simscape component

file name enclosed in single quotes

Name of template Simscape component, specified as a file name enclosed in single quotes. File must have the extension `.ssc`. If you do not provide the file extension, `symWriteSSC` assumes it to be `.ssc`. The component must be on the MATLAB path or in the current folder.

Example: `'TemplateComponent.ssc'`

`eqns` — Symbolic equations
row vector

Symbolic equations, specified as a row vector.

Example: `[ y(t) == diff(x(t), t), m*diff(y(t), t, t) + b*y(t) + k*x(t) == F]`

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`.

Example: `symWriteSSC('myComp.ssc','template.ssc',eq,'H1Header','% New title','HelpText',{'% Description of the','% new component'})`

`H1Header` — Title
row vector of characters

Title specified as a row vector of characters (type `char`) starting with %. If the first character is not %, then `symWriteSSC` adds %.

If the template component has a title and you use `H1Header`, the new component will have the title specified by `H1Header`. If the template component has a title and you call `symWriteSSC` without `H1Header`, the new component will have the same title as the template component.

Example: `'H1Header','% New title'`

`HelpText` — Descriptive text
cell array of row vectors of characters

Descriptive text, specified as a cell array of row vectors of characters. Each row vector must start with %. If the first character is not %, then `symWriteSSC` adds %.

If the template component has descriptive text and you use `HelpText`, the new component will have only the text specified by `HelpText`. In this case, `symWriteSSC` does not copy the descriptive text of the template component to the new component. If the template component has a title and you call `symWriteSSC` without `HelpText`, the new component will have the same descriptive text as the template component.

Example: `'HelpText',{'% Description of the','% new component'}`

See Also

`simscapeEquation` | `symReadSSCParameters` | `symReadSSCVariables`

Introduced in R2016a

tan

Symbolic tangent function

Syntax

`tan(X)`

Description

`tan(X)` returns the tangent function on page 4-1632 of `X`.

Examples

Tangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `tan` returns floating-point or exact symbolic results.

Compute the tangent function for these numbers. Because these numbers are not symbolic objects, `tan` returns floating-point results.

```
A = tan([-2, -pi, pi/6, 5*pi/7, 11])
```

```
A =
    2.1850    0.0000    0.5774   -1.2540 -225.9508
```

Compute the tangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `tan` returns unresolved symbolic calls.

```
symA = tan(sym([-2, -pi, pi/6, 5*pi/7, 11]))
```

```
symA =
[ -tan(2), 0, 3^(1/2)/3, -tan((2*pi)/7), tan(11)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

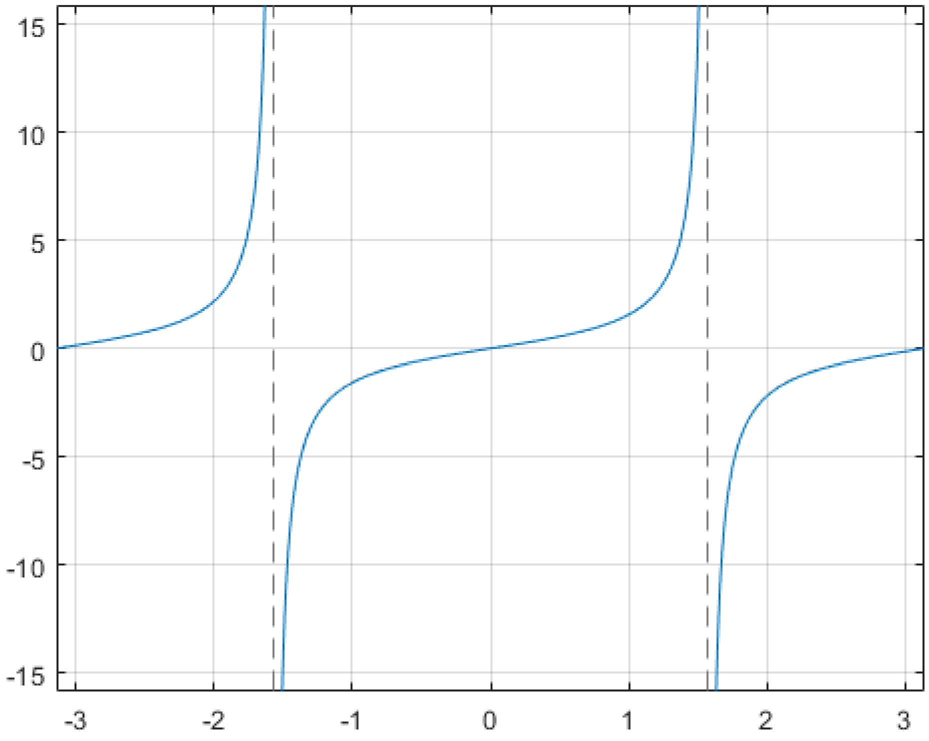
```
vpa(symA)
```

```
ans =
[ 2.1850398632615189916433061023137,...
0,...
0.57735026918962576450914878050196,...
-1.2539603376627038375709109783365,...
-225.95084645419514202579548320345]
```

Plot Tangent Function

Plot the tangent function on the interval from $-\pi$ to π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(tan(x), [-pi, pi])
grid on
```



Handle Expressions Containing Tangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `tan`.

Find the first and second derivatives of the tangent function:

```
syms x
diff(tan(x), x)
diff(tan(x), x, x)
```

```
ans =
tan(x)^2 + 1
```

```
ans =
2*tan(x)*(tan(x)^2 + 1)
```

Find the indefinite integral of the tangent function:

```
int(tan(x), x)

ans =
-log(cos(x))
```

Find the Taylor series expansion of `tan(x)`:

```
taylor(tan(x), x)

ans =
(2*x^5)/15 + x^3/3 + x
```

Rewrite the tangent function in terms of the sine and cosine functions:

```
rewrite(tan(x), 'sincos')

ans =
sin(x)/cos(x)
```

Rewrite the tangent function in terms of the exponential function:

```
rewrite(tan(x), 'exp')

ans =
-(exp(x*2i)*1i - 1i)/(exp(x*2i) + 1)
```

Evaluate Units with `tan` Function

`tan` numerically evaluates these units automatically: `radian`, `degree`, `arcmin`, `arcsec`, and `revolution`.

Show this behavior by finding the tangent of `x` degrees and `2` radians.

```
u = symunit;
syms x
f = [x*u.degree 2*u.radian];
tanf = tan(f)
tanf =
[ tan((pi*x)/180), tan(2)]
```

You can calculate `tanf` by substituting for `x` using `subs` and then using `double` or `vpa`.

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

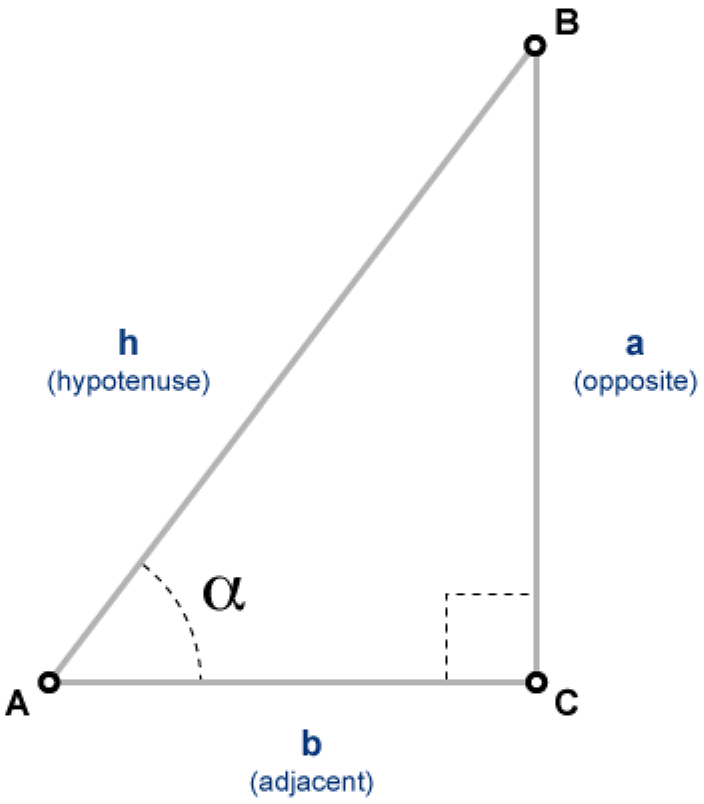
Definitions

Tangent Function

The tangent of an angle, α, defined with reference to a right angled triangle is

$$\tan(\alpha) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{a}{b}.$$

.



The tangent of a complex angle, α, is

$$\tan(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{i\left(e^{i\alpha} + e^{-i\alpha}\right)}.$$

.

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `sin`

Introduced before R2006a

tanh

Symbolic hyperbolic tangent function

Syntax

```
tanh(X)
```

Description

`tanh(X)` returns the hyperbolic tangent function of `X`.

Examples

Hyperbolic Tangent Function for Numeric and Symbolic Arguments

Depending on its arguments, `tanh` returns floating-point or exact symbolic results.

Compute the hyperbolic tangent function for these numbers. Because these numbers are not symbolic objects, `tanh` returns floating-point results.

```
A = tanh([-2, -pi*i, pi*i/6, pi*i/3, 5*pi*i/7])
```

```
A =
  -0.9640 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.5774i...
   0.0000 + 1.7321i   0.0000 - 1.2540i
```

Compute the hyperbolic tangent function for the numbers converted to symbolic objects. For many symbolic (exact) numbers, `tanh` returns unresolved symbolic calls.

```
symA = tanh(sym([-2, -pi*i, pi*i/6, pi*i/3, 5*pi*i/7]))
```

```
symA =
[ -tanh(2), 0, (3^(1/2)*1i)/3, 3^(1/2)*1i, -tanh((pi*2i)/7)]
```

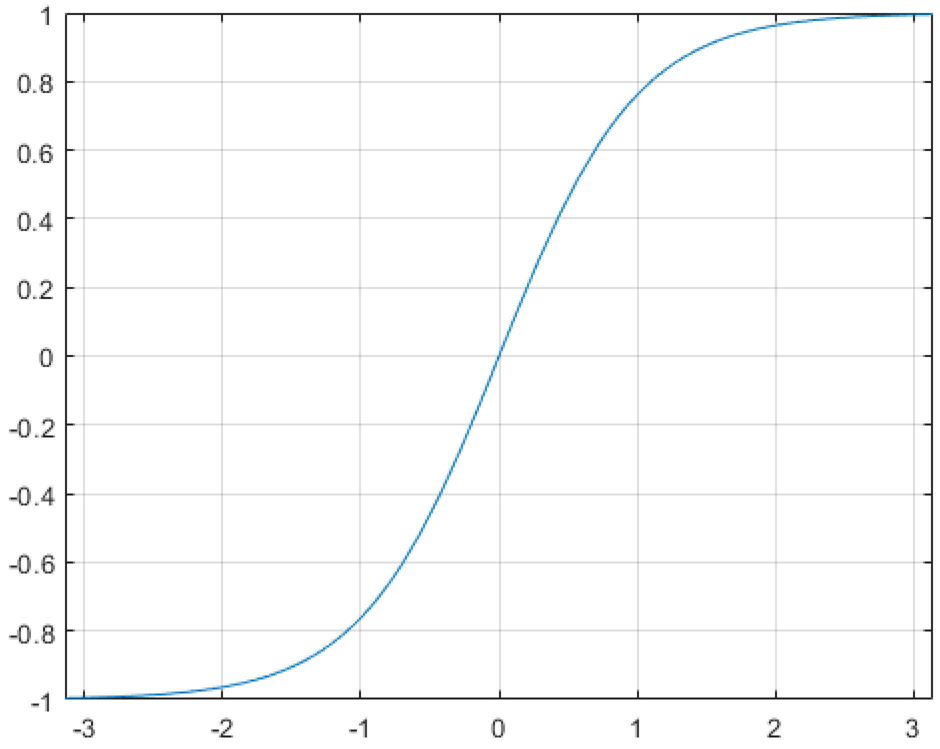
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.96402758007581688394641372410092,...
0,...
0.57735026918962576450914878050196i,...
1.7320508075688772935274463415059i,...
-1.2539603376627038375709109783365i]
```

Plot Hyperbolic Tangent Function

Plot the hyperbolic tangent function on the interval from $-\pi$ to π. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
fplot(tanh(x), [-pi, pi])
grid on
```



Handle Expressions Containing Hyperbolic Tangent Function

Many functions, such as `diff`, `int`, `taylor`, and `rewrite`, can handle expressions containing `tanh`.

Find the first and second derivatives of the hyperbolic tangent function:

```
syms x
diff(tanh(x), x)
diff(tanh(x), x, x)
```

```
ans =
1 - tanh(x)^2
```

```
ans =
2*tanh(x)*(tanh(x)^2 - 1)
```

Find the indefinite integral of the hyperbolic tangent function:

```
int(tanh(x), x)

ans =
log(cosh(x))
```

Find the Taylor series expansion of `tanh(x)`:

```
taylor(tanh(x), x)

ans =
(2*x^5)/15 - x^3/3 + x
```

Rewrite the hyperbolic tangent function in terms of the exponential function:

```
rewrite(tanh(x), 'exp')

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

Input Arguments

x — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

See Also

`acosh` | `acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh`

Introduced before R2006a

taylor

Taylor series

Syntax

```
taylor(f,var)
taylor(f,var,a)
taylor(___,Name,Value)
```

Description

`taylor(f,var)` approximates `f` with the Taylor series expansion on page 4-1645 of `f` up to the fifth order at the point `var = 0`. If you do not specify `var`, then `taylor` uses the default variable determined by `symvar(f,1)`.

`taylor(f,var,a)` approximates `f` with the Taylor series expansion of `f` at the point `var = a`.

`taylor(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. You can specify `Name,Value` after the input arguments in any of the previous syntaxes.

Examples

Find Maclaurin Series of Univariate Expressions

Find 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
```

Specify Expansion Point

Find the Taylor series expansions at $x = 1$ 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
```

Specify Truncation Order

Find the Maclaurin series expansion for `f = sin(x)/x`. The default truncation order is 6. Taylor series approximation of this expression does not have a fifth-degree term, so `taylor` approximates this expression with the fourth-degree polynomial:

```
syms x
f = sin(x)/x;
t6 = taylor(f, x)
```

```
t6 =
x^4/120 - x^2/6 + 1
```

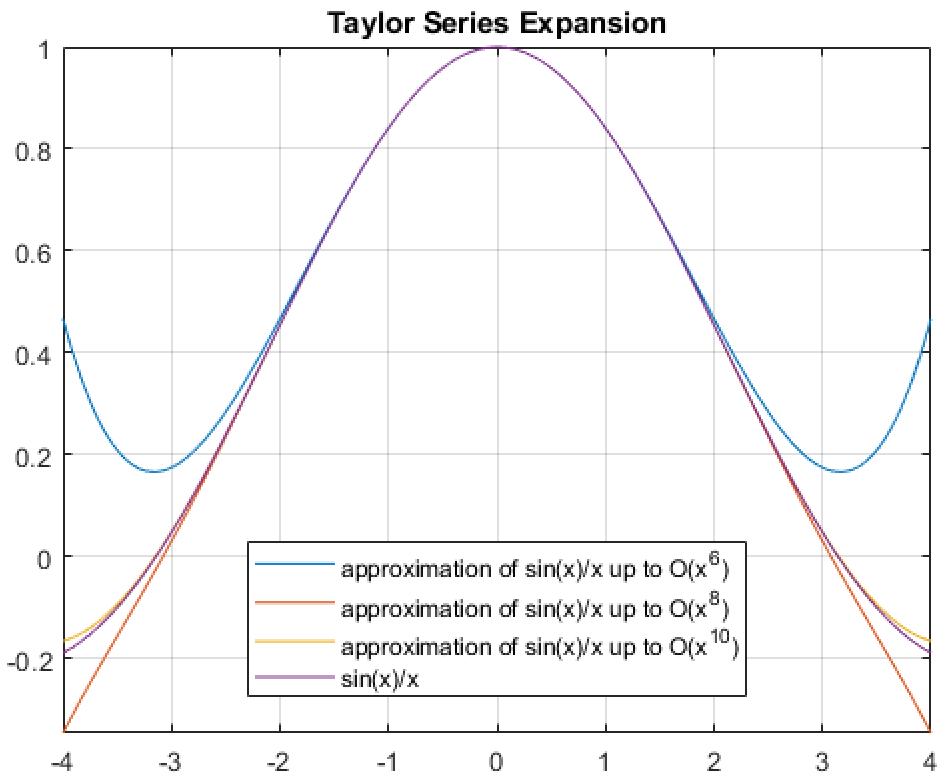
Use `Order` to control the truncation order. For example, approximate the same expression up to the orders 8 and 10:

```
t8 = taylor(f, x, 'Order', 8)
t10 = taylor(f, x, '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 expression `f` and its approximations `t6`, `t8`, and `t10`. Note how the accuracy of the approximation depends on the truncation order. Prior to R2016a, use `ezplot` instead of `fplot`.

```
fplot([t6 t8 t10 f])
xlim([-4 4])
grid on

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^{10})',...
       'sin(x)/x','Location','Best')
title('Taylor Series Expansion')
```



Specify Order Mode: Relative or Absolute

Find 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
```

Fnd the Taylor series expansion with a relative truncation order by using `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
```

Find Maclaurin Series of Multivariate Expressions

Find the Maclaurin series expansion of this multivariate expression. 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)
```

Find 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
```

Specify Expansion Point for Multivariate Expression

Find 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
```

Input Arguments

`f` — Input to approximate

symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic multidimensional array

Input to approximate, specified as a symbolic expression or function. It also can be a vector, matrix, or multidimensional array of symbolic expressions or functions.

`var` — Expansion variable

symbolic variable

Expansion variable, specified as a symbolic variable. If you do not specify `var`, then `taylor` uses the default variable determined by `symvar(f,1)`.

`a` — Expansion point

0 (default) | number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable. You also can specify the expansion point as a `Name,Value` pair argument. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

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`.

Example: `taylor(log(x),x,'ExpansionPoint',1,'Order',9)`

`ExpansionPoint` — Expansion point

0 (default) | number | symbolic number | symbolic variable | symbolic function | symbolic expression

Expansion point, specified as a number, or a symbolic number, variable, function, or expression. The expansion point cannot depend on the expansion variable. You can also specify the expansion point using the input argument `a`. If you specify the expansion point both ways, then the `Name,Value` pair argument takes precedence.

`Order` — Truncation order of Taylor series expansion
6 (default) | positive integer | symbolic positive integer

Truncation order of Taylor series expansion, specified as a positive integer or a symbolic positive integer. `taylor` computes the Taylor series approximation with the order `n - 1`. The truncation order `n` is the exponent in the *O*-term: $O(var^{n})$.

`OrderMode` — Order mode indicator
`'absolute'` (default) | `'relative'`

Order mode indicator, specified as `'absolute'` or `'relative'`. This indicator specifies whether you want to use absolute or relative order when computing the Taylor polynomial approximation.

Absolute order is the truncation order of the computed series. Relative order `n` means that the exponents of `var` in the computed series range from the leading order `m` to the highest exponent `m + n - 1`. Here `m + n` is the exponent of `var` in the *O*-term: $O(var^{m+n})$.

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:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots = \sum_{m=0}^{\infty} \frac{f^{(m)}(0)}{m!}x^m$$

Tips

- If you use both the third argument `a` and `ExpansionPoint` to specify the expansion point, the value specified via `ExpansionPoint` prevails.
- If `var` is a vector, then the expansion point `a` must be a scalar or a vector of the same length as `var`. If `var` is a vector and `a` is a scalar, then `a` is expanded into a vector of the same length as `var` with all elements equal to `a`.
- If the expansion point is infinity or negative infinity, then `taylor` computes the Laurent series expansion, which is a power series in `1/var`.

See Also

`pade` | `series` | `symvar`

Topics

"Taylor Series" on page 2-67

Introduced before R2006a

taylortool

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.

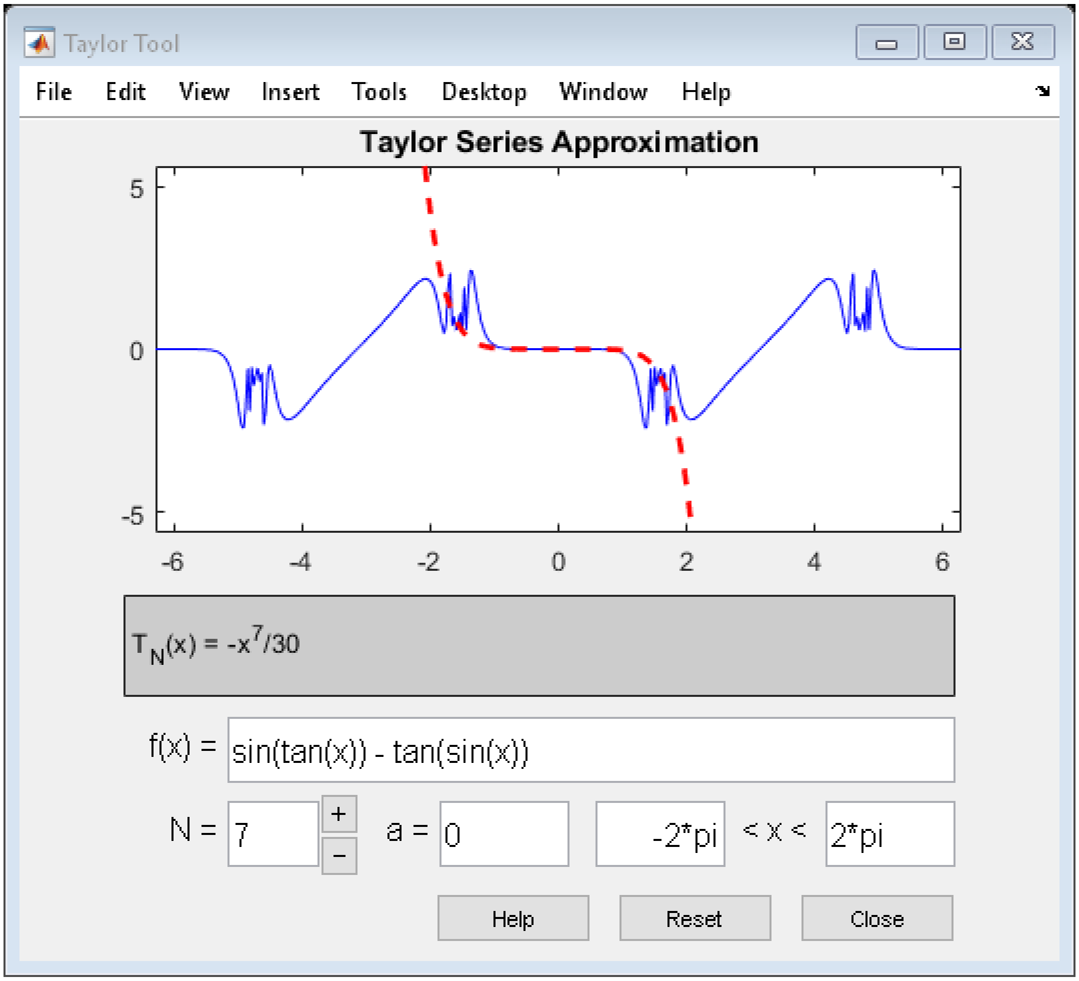
`taylortool('f')` initiates the GUI for the given expression `f`.

Examples

Open Taylor Series Calculator For Particular Expression

Open the Taylor series calculator for `sin(tan(x)) - tan(sin(x))`:

```
taylortool('sin(tan(x)) - tan(sin(x))')
```


Taylor Tool
File Edit View Insert Tools Desktop Window Help
Taylor Series Approximation
5
0
-5
-6 -4 -2 0 2 4 6
Help
Reset
Close
+
–

$T_N(x) = -x^7/30$

f(x) = sin(tan(x)) - tan(sin(x))

N = 7 a = 0 -2*pi < x < 2*pi

See Also

`funtool` | `rsums`

Topics

"Taylor Series" on page 2-67

Introduced before R2006a

texlabel

TeX representation of symbolic expression

Syntax

```
texlabel(expr)
texlabel(expr,'literal')
```

Description

`texlabel(expr)` converts the symbolic expression `expr` into the TeX equivalent for use in character vectors. `texlabel` converts Greek variable names, such as delta, into Greek letters. Annotation functions, such as `title`, `xlabel`, and `text` can use the TeX character vector as input. To obtain the LaTeX representation, use `latex`.

`texlabel(expr,'literal')` interprets Greek variable names literally.

Examples

Generate TeX Character Vector

Use `texlabel` to generate TeX character vectors for these symbolic expressions.

```
syms x y lambda12 delta
texlabel(sin(x) + x^3)
texlabel(3*(1-x)^2*exp(-(x^2) - (y+1)^2))
texlabel(lambda12^(3/2)/pi - pi*delta^(2/3))
```

```
ans =
    '{sin}({x}) + {x}^{3}'

ans =
    '{3} {exp}(- ({y} + {1})^{2} - {x}^{2}) ({x} - {1})^{2}'
```

```
ans =
    '{\lambda_{12}}^{{3}/{2}}/{\pi} - {\delta}^{{2}/{3}} {\pi}'
```

To make `texlabel` interpret Greek variable names literally, include the argument `'literal'`.

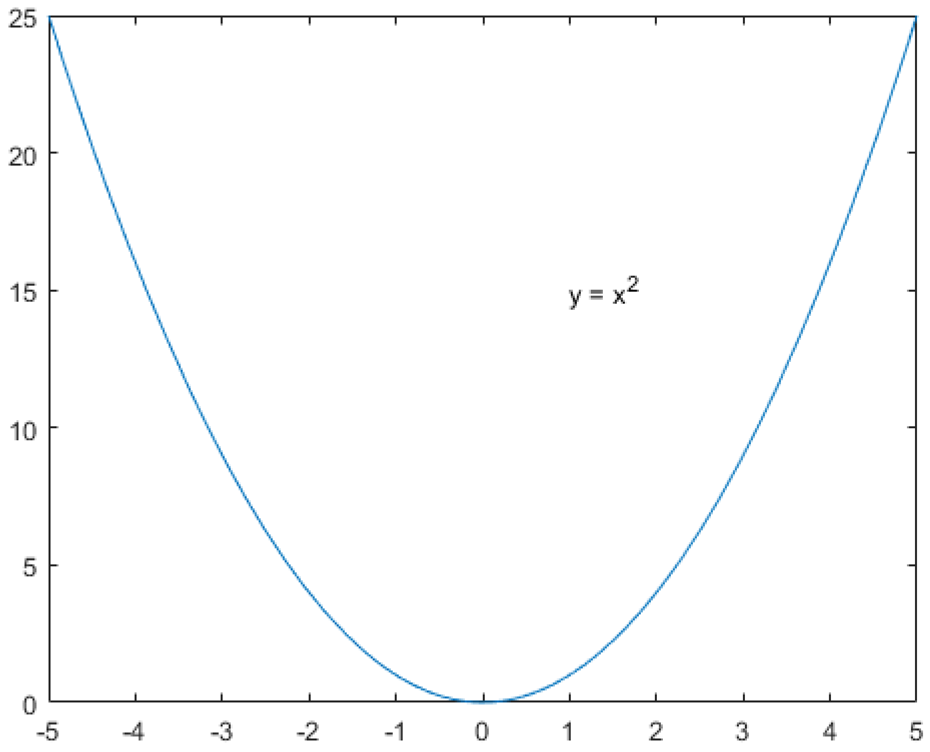
```
texlabel(lambda12,'literal')
```

```
ans =
    '{lambda12}'
```

Insert TeX in Figure

Plot `y = x^2` using `fplot`. Show the plotted expression `y` by using `texlabel` to generate a TeX character vector that `text` inserts into the figure. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms x
y = x^2;
fplot(y)
ylabel = texlabel(y);
text(1, 15, ['y = ' ylabel]);
```



Input Arguments

`expr` — Expression to be converted
symbolic expression

Expression to be converted, specified as a symbolic expression.

See Also

`latex` | `text` | `title` | `xlabel` | `ylabel` | `zlabel`

Introduced before R2006a

times.*

Symbolic array multiplication

Syntax

```
A.*B
times(A,B)
```

Description

`A.*B` performs elementwise multiplication of `A` and `B`.

`times(A,B)` is equivalent to `A.*B`.

Examples

Multiply Matrix by Scalar

Create a `2`-by-`3` matrix.

```
A = sym('a', [2 3])
```

```
A =
[ a1_1, a1_2, a1_3]
[ a2_1, a2_2, a2_3]
```

Multiply the matrix by the symbolic expression `sin(b)`. Multiplying a matrix by a scalar means multiplying each element of the matrix by that scalar.

```
syms b
A.*sin(b)
```

```
ans =
[ a1_1*sin(b), a1_2*sin(b), a1_3*sin(b)]
[ a2_1*sin(b), a2_2*sin(b), a2_3*sin(b)]
```

Multiply Two Matrices

Create a 3-by-3 symbolic Hilbert matrix and a 3-by-3 diagonal matrix.

```
H = sym(hilb(3))
d = diag(sym([1 2 3]))
```

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

d =
[ 1, 0, 0]
[ 0, 2, 0]
[ 0, 0, 3]
```

Multiply the matrices by using the elementwise multiplication operator `.*`. This operator multiplies each element of the first matrix by the corresponding element of the second matrix. The dimensions of the matrices must be the same.

```
H.*d
```

```
ans =
[ 1,   0,   0]
[ 0, 2/3,   0]
[ 0,   0, 3/5]
```

Multiply Expression by Symbolic Function

Multiply a symbolic expression by a symbolic function. The result is a symbolic function.

```
syms f(x)
f(x) = x^2;
f1 = (x^2 + 5*x + 6).*f
```

```
f1(x) =
x^2*(x^2 + 5*x + 6)
```

Input Arguments

`A` — Input

number | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

`B` — Input

number | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number or a symbolic number, variable, vector, matrix, multidimensional array, function, or expression. Inputs `A` and `B` must be the same size unless one is a scalar. A scalar value expands into an array of the same size as the other input.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `transpose`

Introduced before R2006a

toeplitz

Symbolic Toeplitz matrix

Syntax

```
toeplitz(c,r)
toeplitz(r)
```

Description

`toeplitz(c,r)` generates a nonsymmetric Toeplitz matrix on page 4-1660 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 $i \neq j$.

Input Arguments

`c`

Vector specifying the first column of a Toeplitz matrix.

`r`

Vector specifying the first row of a Toeplitz matrix.

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, 1i]
[   2, 1, 2]
[ -1i, 2, 1]
```

If the first element of the vector is real, then the resulting Toeplitz matrix is Hermitian:

```
isAlways(T == T')

ans =
  3×3 logical array
   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 =
[ 1i, 2, 1]
[ 2, 1i, 2]
[ 1, 2, 1i]

isAlways(T == T')

ans =
  3×3 logical array
     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 given column does not match first element of given row.
Column wins diagonal conflict.

ans =
[ a, b/2, a/2]
[ b,   a, b/2]
[ c,   b,   a]
```

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:

$$T = \begin{pmatrix} t_0 & t_1 & t_2 & & & & t_k \\ t_{-1} & t_0 & t_1 & \cdots & & & \\ t_{-2} & t_{-1} & t_0 & & & & \\ & \vdots & & \ddots & & \vdots & \\ & & & & t_0 & t_1 & t_2 \\ & & & \cdots & t_{-1} & t_0 & t_1 \\ t_{-k} & & & & t_{-2} & t_{-1} & t_0 \end{pmatrix}$$

Tips

- Calling `toeplitz` for numeric arguments that are not symbolic objects invokes the MATLAB `toeplitz` function.

See Also

`toeplitz`

Introduced in R2013a

transpose.'

Symbolic matrix transpose

Syntax

```
A.'
transpose(A)
```

Description

`A.'` computes the nonconjugate transpose on page 4-1664 of `A`.

`transpose(A)` is equivalent to `A.'`.

Examples

Transpose of Real Matrix

Create a `2`-by-`3` matrix, the elements of which represent real numbers.

```
syms x y real
A = [x x x; y y y]
```

```
A =
[ x, x, x]
[ y, y, y]
```

Find the nonconjugate transpose of this matrix.

```
A.'
```

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

If all elements of a matrix represent real numbers, then its complex conjugate transform equals its nonconjugate transform.

```
isAlways(A' == A.')

ans =
  3×2 logical array
     1     1
     1     1
     1     1
```

Transpose of Complex Matrix

Create a 2-by-2 matrix, the elements of which represent complex numbers.

```
syms x y real
A = [x + y*i x - y*i; y + x*i y - x*i]

A =
[ x + y*1i, x - y*1i]
[ y + x*1i, y - x*1i]
```

Find the nonconjugate transpose of this matrix. The nonconjugate transpose operator, `A.'`, performs a transpose without conjugation. That is, it does not change the sign of the imaginary parts of the elements.

```
A.'

ans =
[ x + y*1i, y + x*1i]
[ x - y*1i, y - x*1i]
```

For a matrix of complex numbers with nonzero imaginary parts, the nonconjugate transform is not equal to the complex conjugate transform.

```
isAlways(A.' == A','Unknown','false')

ans =
  2×2 logical array
     0     0
     0     0
```

Input Arguments

`A` **— Input**

number | symbolic number | symbolic variable | symbolic expression | symbolic vector | symbolic matrix | symbolic multidimensional array

Input, specified as a number or a symbolic number, variable, expression, vector, matrix, multidimensional array.

Definitions

Nonconjugate Transpose

The nonconjugate transpose of a matrix interchanges the row and column index for each element, reflecting the elements across the main diagonal. The diagonal elements themselves remain unchanged. This operation does not affect the sign of the imaginary parts of complex elements.

For example, if `B = A.'` and `A(3,2)` is `1+1i`, then the element `B(2,3)` is `1+1i`.

See Also

`ctranspose` | `ldivide` | `minus` | `mldivide` | `mpower` | `mrdivide` | `mtimes` | `plus` | `power` | `rdivide` | `times`

Introduced before R2006a

triangularPulse

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)`.

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.

Examples

Find Triangular Pulse Function

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
```

Special Cases of Triangular Pulse Function

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
```

Fixed-Form Triangular Pulse

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
```

Symmetrical Triangular Pulse

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)
```

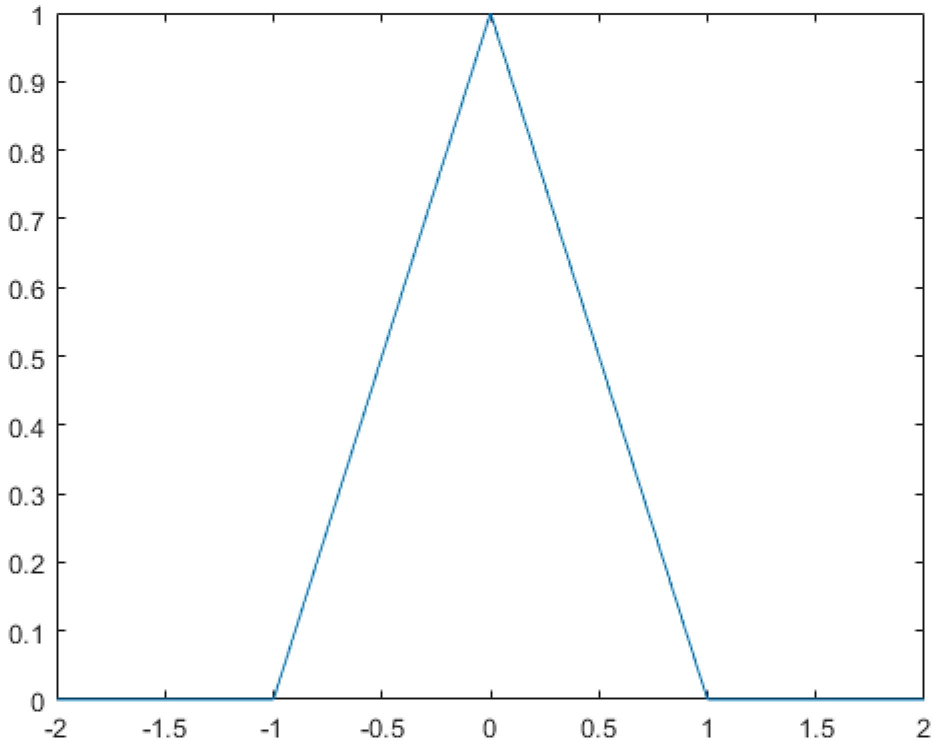
```
ans =
triangularPulse(a, a/2 + c/2, c, x)
```

```
[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 Triangular Pulse Function

```
syms x
fplot(triangularPulse(x), [-2 2])
```



Relation Between Heaviside, Rectangular and Triangular Pulse Functions

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
```

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.

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.

See Also

`dirac` | `heaviside` | `rectangularPulse`

Introduced in R2012b

tril

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)
```

```
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)
```

```
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)
```

```
ans =
[     0,     0, 0]
[     1,     0, 0]
[ a + 1, b + 2, 0]
```

See Also

`diag` | `triu`

Introduced before R2006a

triu

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)
```

```
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)
```

```
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)
```

```
ans =
[ a,     b,     c]
[ 1,     2,     3]
[ 0, b + 2, c + 3]
```

See Also

`diag` | `tril`

Introduced before R2006a

uint8uint16uint32uint64

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`

Function	Output Range	Output Type	Bytes per Element	Output Class
`uint64`	0 to 18,446,744,073,709, 551,615	Unsigned 64-bit integer	8	`uint64`

See Also

`double` | `int16` | `int32` | `int64` | `int8` | `single` | `sym` | `vpa`

Introduced before R2006a

unitConversionFactor

Conversion factor between units

Syntax

```
C = unitConversionFactor(unit1,unit2)
C = unitConversionFactor(unit1,unit2,'Force',true)
```

Description

`C = unitConversionFactor(unit1,unit2)` returns the conversion factor `C` between units `unit1` and `unit2` so that `unit1 = C*unit2`.

`C = unitConversionFactor(unit1,unit2,'Force',true)` forces `unitConversionFactor` to return a conversion factor even if the units are not dimensionally compatible.

Examples

Conversion Factor Between Units

Find the conversion factor between inches and centimeters.

```
u = symunit;
inch2cm = unitConversionFactor(u.inch,u.cm)
```

```
inch2cm =
127/50
```

Convert the conversion factor to double.

```
inch2cm = double(inch2cm)
```

```
inch2cm =
    2.5400
```

The same result is returned using `simplify` or `rewrite`.

```
inch2cm = simplify(u.inch/u.cm)
```

```
inch2cm =
127/50
```

```
inch2cm  = rewrite(u.inch,u.cm)
```

```
inch2cm =
(127/50)*[cm]
```

Find the conversion factor between Newtons and kg m/s^2. The conversion factor is `1`.

```
convFactor = unitConversionFactor(1*u.N, 1*u.kg*u.m/u.s^2)
```

```
convFactor =
1
```

Convert Between Dimensionally Incompatible Units

Convert between dimensionally incompatible units by specifying the argument `'Force'` as `true`.

Convert between Watts and Joules. `unitConversionFactor` returns the factor `1/[s]` since 1 W = 1 J/s.

```
u = symunit;
convFactor = unitConversionFactor(u.W, u.J, 'Force', true)
```

```
convFactor =
1/[s]
```

If you do not specify `'Force'` as `true` then `unitConversionFactor` errors.

```
unitConversionFactor(u.W, u.J)
```

```
Error using unitConversionFactor (line 73)
Incompatible units.
```

Input Arguments

`unit` — Units
symbolic units

Units, specified as symbolic units.

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`.

Example: `unitConversionFactor(u.W,u.J,'Force',true)`

`Force` — Force conversion of incompatible units
false (default) | true

Force conversion of incompatible units, specified as `true` or `false`.

See Also

`checkUnits` | `findUnits` | `isUnit` | `newUnit` | `separateUnits` | `str2symunit` | `symunit` | `symunit2str`

Topics

“Units of Measurement Tutorial” on page 2-5
“Unit Conversions and Unit Systems” on page 2-30
“Units List” on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017a

unitInfo

Information on units of measurement

Syntax

```
unitInfo(unit)
unitInfo(dim)
unitInfo
```

```
A = unitInfo( ___ )
```

Description

`unitInfo(unit)` returns information for the symbolic unit `unit`.

`unitInfo(dim)` returns available units for the dimension `dim`.

`unitInfo` returns a list of available physical dimensions.

`A = unitInfo( ___ )` returns the output in `A` using any of the input arguments in the previous syntaxes. Dimensions are returned as strings, and units are returned as symbolic units.

Examples

Find Information on Units and Dimensions

Find information on unit `u.Wb` where `u = symunit`. The `unitInfo` function specifies that `Wb` is the SI unit of magnetic flux.

```
u = symunit;
unitInfo(u.Wb)
```

```
weber - a physical unit of magnetic flux. ['SI']
```

```
Get all units for measuring 'MagneticFlux' by calling unitInfo('MagneticFlux').
SI units accept all SI prefixes. For example, m accepts nm, um, mm, cm, dm, km.
```

Find all available units for `'MagneticFlux'` as described.

```
unitInfo('MagneticFlux')
```

```
All units of dimension 'MagneticFlux':

Mx - maxwell
Wb - weber ['SI']
abWb - abweber
statWb - statweber

Get the base SI units of any unit above by calling rewrite(<unit>,'SI').
SI units accept all SI prefixes. For example, m accepts nm, um, mm, cm, dm, km.
```

Find All Physical Dimensions Available

Return all available dimensions using `unitInfo` without input arguments.

```
unitInfo
```

```
"AbsorbedDose"
"AbsorbedDoseOrDoseEquivalent"
"Acceleration"
...
...
"Length"
"Luminance"
"LuminousFlux"
...
...
"Time"
"Velocity"
"Volume"
```

Use Information on Units and Dimensions

Store information returned by `unitInfo` for use by providing an output.

Store the dimension of `u.C`.

```
u = symunit;
dimC = unitInfo(u.C)
```

```
dimC = 
    "ElectricCharge"
```

Find and store all units for the dimension `dimC`.

```
unitsEC = unitInfo(dimC)
```

```
unitsEC =
      [C]
     [Fr]
    [abC]
      [e]
  [statC]
```

Find information on the fourth unit of `unitsEC`.

```
unitInfo(unitsEC(4))
```

```
elementary charge - a physical unit of electric charge.

Get all units for measuring 'ElectricCharge' by calling unitInfo('ElectricCharge').
```

Store `[e]`. Then, approximate the electrons in a coulomb of electric charge.

```
electronCharge = unitsEC(4);
numElectrons = simplify(u.C/electronCharge)
```

```
numElectrons =
6241509125883257926.5158629382492
```

Show that approximately 6.24 x 10^{18} electrons are in a coulomb by converting the high-precision symbolic result to double.

```
numElectrons = double(numElectrons)
```

```
numElectrons =
   6.2415e+18
```

Input Arguments

`unit` — Unit name

symbolic unit | character vector | string

Unit name, specified as a symbolic unit, character vector, or string.

Example: `unitInfo(u.m)` where `u = symunit`

`dim` — Dimension
character vector | string

Dimension, specified as a character vector or string.

Example: `unitInfo('Length')`

See Also

`rewrite` | `simplify` | `symunit`

Introduced in R2017b

unitSystems

List available unit systems

Syntax

```
unitSystems
```

Description

`unitSystems` returns a row vector of available unit systems. To create custom unit systems, see `newUnitSystem`. To convert between units and unit systems, see “Unit Conversions and Unit Systems” on page 2-30.

Examples

Get Available Unit Systems

Get available unit systems by using `unitSystems`. Add a custom unit system and check that `unitSystems` lists it as available.

Check that the default unit systems are `SI`, `CGS`, and `US`.

```
unitSystems
```

```
ans = 
  1×3 string array
    "CGS"    "SI"    "US"
```

Add a custom unit system that modifies the SI base units. For details, see `newUnitSystem` and “Unit Conversions and Unit Systems” on page 2-30.

```
u = symunit;
SIUnits = baseUnits('SI');
newUnits = subs(SIUnits,[u.m u.s],[u.km u.hr]);
newUnitSystem('SI_km_hr',newUnits)
```

```
ans = 
    "SI_km_hr"
```

Check that the new unit system is available by using `unitSystems`.

```
unitSystems
```

```
ans = 
  1×4 string array
    "CGS"    "SI"    "SI_km_hr"    "US"
```

After calculations, remove the new unit system and check that it is unavailable.

```
removeUnitSystem('SI_km_hr');
unitSystems
```

```
ans = 
  1×3 string array
    "CGS"    "SI"    "US"
```

- "Units of Measurement Tutorial" on page 2-5
- "Unit Conversions and Unit Systems" on page 2-30
- "Units List" on page 2-13

See Also

`baseUnits` | `derivedUnits` | `newUnitSystem` | `removeUnitSystem` | `rewrite` | `symunit`

Topics

"Units of Measurement Tutorial" on page 2-5
"Unit Conversions and Unit Systems" on page 2-30
"Units List" on page 2-13

External Websites

The International System of Units (SI)

Introduced in R2017b

vectorPotential

Vector potential of vector field

Syntax

```
vectorPotential(V,X)
vectorPotential(V)
```

Description

`vectorPotential(V,X)` computes the vector potential of the vector field on page 4-1689 `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`.

Input Arguments

V

Three-dimensional vector of symbolic expressions or functions.

X

Three-dimensional vector with respect to which you compute the vector potential.

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) =
 z*(2*y^2 + 18) - (16*z^3)/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
```

Definitions

Vector Potential of a Vector Field

The vector potential of a vector field `V` is a vector field `A`, such that:

$$V = \nabla \times A = curl(A)$$

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`.

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `jacobian` | `laplacian` | `potential`

Introduced in R2012a

vertcat

Concatenate symbolic arrays vertically

Syntax

```
vertcat(A1,...,AN)
[A1;...;AN]
```

Description

`vertcat(A1,...,AN)` vertically concatenates the symbolic arrays `A1,...,AN`. For vectors and matrices, all inputs must have the same number of columns. For multidimensional arrays, `vertcat` concatenates inputs along the first dimension. The remaining dimensions must match.

`[A1;...;AN]` is a shortcut for `vertcat(A1,...,AN)`.

Examples

Concatenate Two Symbolic Vectors Vertically

Concatenate the two symbolic vectors `A` and `B` to form a symbolic matrix.

```
A = sym('a%d',[1 4]);
B = sym('b%d',[1 4]);
vertcat(A,B)
```

```
ans =
[ a1, a2, a3, a4]
[ b1, b2, b3, b4]
```

Alternatively, you can use the shorthand `[A;B]` to concatenate `A` and `B`.

```
[A;B]
```

```
ans =
[ a1, a2, a3, a4]
[ b1, b2, b3, b4]
```

Concatenate Multiple Symbolic Arrays Vertically

Concatenate multiple symbolic arrays into one symbolic matrix.

```
A = sym('a%d',[1 3]);
B = sym('b%d%d',[4 3]);
C = sym('c%d%d',[2 3]);
vertcat(C,A,B)
```

```
ans =
[ c11, c12, c13]
[ c21, c22, c23]
[  a1,  a2,  a3]
[ b11, b12, b13]
[ b21, b22, b23]
[ b31, b32, b33]
[ b41, b42, b43]
```

Concatenate Multidimensional Arrays Vertically

Create the 3-D symbolic arrays `A` and `B`.

```
A = [2 4; 1 7; 3 3];
A(:,:,2) = [8 9; 4 5; 6 2];
A = sym(A)
B = [8 3; 0 2];
B(:,:,2) = [6 2; 3 3];
B = sym(B)
```

```
A(:,:,1) =
[ 2, 4]
[ 1, 7]
[ 3, 3]
A(:,:,2) =
[ 8, 9]
[ 4, 5]
[ 6, 2]

B(:,:,1) =
[ 8, 3]
[ 0, 2]
B(:,:,2) =
[ 6, 2]
[ 3, 3]
```

Use `vertcat` to concatenate `A` and `B`.

```
vertcat(A,B)
```

```
ans(:,:,1) =
[ 2, 4]
[ 1, 7]
[ 3, 3]
[ 8, 3]
[ 0, 2]

ans(:,:,2) =
[ 8, 9]
[ 4, 5]
[ 6, 2]
[ 6, 2]
[ 3, 3]
```

Input Arguments

`A1,...,AN` — Input arrays

symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array

Input arrays, specified as symbolic variables, vectors, matrices, or multidimensional arrays.

See Also

`cat` | `horzcat`

Introduced before R2006a

vpa

Variable-precision arithmetic

Note Support of character vectors that are not valid variable names and do not define a number will be removed in a future release. Instead of character vectors, use symbolic expressions. To create symbolic expressions, first create symbolic numbers and variables, and then use operations on them. For example, use `vpa((1 + sqrt(sym(5)))/2)` instead of `vpa('(1 + sqrt(5))/2')`.

Syntax

```
vpa(x)
vpa(x,d)
```

Description

`vpa(x)` uses variable-precision floating-point arithmetic (VPA) to evaluate each element of the symbolic input `x` to at least `d` significant digits, where `d` is the value of the `digits` function. The default value of `digits` is 32.

`vpa(x,d)` uses at least `d` significant digits, instead of the value of `digits`.

Examples

Evaluate Symbolic Inputs with Variable-Precision Arithmetic

Evaluate symbolic inputs with variable-precision floating-point arithmetic. By default, `vpa` calculates values to 32 significant digits.

```
syms x
p = sym(pi);
piVpa = vpa(p)
piVpa =
3.1415926535897932384626433832795

a = sym(1/3);
f = a*sin(2*p*x);
fVpa = vpa(f)

fVpa =
0.33333333333333333333333333333333*sin(6.283185307179586476925286766559*x)
```

Evaluate elements of vectors or matrices with variable-precision arithmetic.

```
V = [x/p a^3];
M = [sin(p) cos(p/5); exp(p*x) x/log(p)];
vpa(V)
vpa(M)

ans =
[ 0.31830988618379067153776752674503*x, 0.037037037037037037037037037037037]
ans =
[                                     0,    0.80901699437494742410229341718282]
[ exp(3.1415926535897932384626433832795*x), 0.87356852683023186835397746476334*x]
```

Note You must wrap all inner inputs with `vpa`, such as `exp(vpa(200))`. Otherwise the inputs are automatically converted to double by MATLAB.

Change Precision Used by `vpa`

By default, `vpa` evaluates inputs to 32 significant digits. You can change the number of significant digits by using the `digits` function.

Approximate the expression `100001/10001` with seven significant digits using `digits`. Save the old value of `digits` returned by `digits(7)`. The `vpa` function returns only five significant digits, which can mean the remaining digits are zeros.

```
digitsOld = digits(7);
y = sym(100001)/10001;
vpa(y)

ans =
9.9991
```

Check if the remaining digits are zeros by using a higher precision value of `25`. The result shows that the remaining digits are in fact a repeating decimal.

```
digits(25)
vpa(y)

ans =
9.999100089991000899910009
```

Alternatively, to override `digits` for a single `vpa` call, change the precision by specifying the second argument.

Find π to 100 significant digits by specifying the second argument.

```
vpa(pi,100)

ans =
3.141592653589793238462643383279502884197169...
399375105820974944592307816406286208998628034...
4825342117068
```

Restore the original precision value in `digitsOld` for further calculations.

```
digits(digitsOld)
```

Numerically Approximate Symbolic Results

While symbolic results are exact, they might not be in a convenient form. You can use `vpa` to numerically approximate exact symbolic results.

Solve a high-degree polynomial for its roots using `solve`. The `solve` function cannot symbolically solve the high-degree polynomial and represents the roots using `root`.

```
syms x
y = solve(x^4 - x + 1, x)

y =
 root(z^4 - z + 1, z, 1)
 root(z^4 - z + 1, z, 2)
 root(z^4 - z + 1, z, 3)
 root(z^4 - z + 1, z, 4)
```

Use `vpa` to numerically approximate the roots.

```
yVpa = vpa(y)

yVpa =
 - 0.72713608449119683997667565867496 - 0.93409928946052943963903028710582i
 - 0.72713608449119683997667565867496 + 0.93409928946052943963903028710582i
   0.72713608449119683997667565867496 - 0.43001428832971577641651985839602i
   0.72713608449119683997667565867496 + 0.43001428832971577641651985839602i
```

`vpa` Uses Guard Digits to Maintain Precision

The value of the `digits` function specifies the minimum number of significant digits used. Internally, `vpa` can use more digits than `digits` specifies. These additional digits are called guard digits because they guard against round-off errors in subsequent calculations.

Numerically approximate `1/3` using four significant digits.

```
a = vpa(1/3, 4)

a =
0.3333
```

Approximate the result `a` using 20 digits. The result shows that the toolbox internally used more than four digits when computing `a`. The last digits in the result are incorrect because of the round-off error.

```
vpa(a, 20)

ans =
0.33333333333303016843
```

Avoid Hidden Round-off Errors

Hidden round-off errors can cause unexpected results.

Evaluate `1/10` with the default 32-digit precision, and then with the 10 digits precision.

```
a = vpa(1/10, 32)
b = vpa(1/10, 10)

a =
0.1
```

```
b =
0.1
```

Superficially, `a` and `b` look equal. Check their equality by finding `a - b`.

```
a - b
```

```
ans =
0.0000000000000000000086736173798840354720600815844403
```

The difference is not equal to zero because `b` was calculated with only `10` digits of precision and contains a larger round-off error than `a`. When you find `a - b`, `vpa` approximates `b` with 32 digits. Demonstrate this behavior.

```
a - vpa(b, 32)
```

```
ans =
0.0000000000000000000086736173798840354720600815844403
```

`vpa` Restores Precision of Common Double-Precision Inputs

Unlike exact symbolic values, double-precision values inherently contain round-off errors. When you call `vpa` on a double-precision input, `vpa` cannot restore the lost precision, even though it returns more digits than the double-precision value. However, `vpa` can recognize and restore the precision of expressions of the form p/q, $p\pi/q$, $(p/q)^{1/2}$, 2^q, and 10^q, where p and q are modest-sized integers.

First, demonstrate that `vpa` cannot restore precision for a double-precision input. Call `vpa` on a double-precision result and the same symbolic result.

```
dp = log(3);
s = log(sym(3));
dpVpa = vpa(dp)
sVpa = vpa(s)
d = sVpa - dpVpa
```

```
dpVpa =
1.0986122886681095600636126619065

sVpa =
1.0986122886681096913952452369225

d =
0.00000000000000013133163257501600766255995767652
```

As expected, the double-precision result differs from the exact result at the 16th decimal place.

Demonstrate that `vpa` restores precision for expressions of the form p/q, $p\pi/q$, $(p/q)^{1/2}$, 2^q, and 10^q, where p and q are modest sized integers, by finding the difference between the `vpa` call on the double-precision result and on the exact symbolic result. The differences are `0.0` showing that `vpa` restores lost precision in the double-precision input.

```
vpa(1/3) - vpa(1/sym(3))
vpa(pi) - vpa(sym(pi))
vpa(1/sqrt(2)) - vpa(1/sqrt(sym(2)))
vpa(2^66) - vpa(2^sym(66))
vpa(10^25) - vpa(10^sym(25))
```

```
ans =
0.0

ans =
0.0

ans =
0.0

ans =
0.0

ans =
0.0
```

Input Arguments

`x` — Input to evaluate

number | vector | matrix | multidimensional array | symbolic number | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic expression | symbolic function | symbolic character vector

Input to evaluate, specified as a number, vector, matrix, multidimensional array, or a symbolic number, vector, matrix, multidimensional array, expression, function, or character vector.

`d` — Number of significant digits

integer

Number of significant digits, specified as an integer. `d` must be greater than 1 and lesser than $2^{29}+1$.

Tips

- `vpa` does not convert fractions in the exponent to floating point. For example, `vpa(a^sym(2/5))` returns `a^(2/5)`.
- `vpa` uses more digits than the number of digits specified by `digits`. These extra digits guard against round-off errors in subsequent calculations and are called guard digits.
- When you call `vpa` on a numeric input, such as `1/3`, `2^(-5)`, or `sin(pi/4)`, the numeric expression is evaluated to a double-precision number that contains round-off errors. Then, `vpa` is called on that double-precision number. For accurate results, convert numeric expressions to symbolic expressions with `sym`. For example, to approximate `exp(1)`, use `vpa(exp(sym(1)))`.
- If the second argument `d` is not an integer, `vpa` rounds it to the nearest integer with `round`.
- `vpa` restores precision for numeric inputs that match the forms p/q, $p\pi/q$, $(p/q)^{1/2}$, 2^q, and 10^q, where p and q are modest-sized integers.
- Atomic operations using variable-precision arithmetic round to nearest.
- The differences between variable-precision arithmetic and IEEE Floating-Point Standard 754 are

  - Inside computations, division by zero throws an error.
  - The exponent range is larger than in any predefined IEEE mode. `vpa` underflows below approximately `10^(-323228496)`.
  - Denormalized numbers are not implemented.
  - Zeroes are not signed.
  - The number of *binary* digits in the mantissa of a result may differ between variable-precision arithmetic and IEEE predefined types.
  - There is only one `NaN` representation. No distinction is made between quiet and signaling `NaN`.
  - No floating-point number exceptions are available.

See Also

`digits` | `double` | `root` | `vpaintegral`

Topics

“Increase Precision of Numeric Calculations” on page 2-116
“Recognize and Avoid Round-Off Errors” on page 2-118
“Increase Speed by Reducing Precision” on page 2-123
“Choose Symbolic or Numeric Arithmetic” on page 2-114

Introduced before R2006a

vpaintegral

Numerical integration using variable precision

Syntax

```
vpaintegral(f,a,b)
vpaintegral(f,x,a,b)
vpaintegral(___,Name,Value)
```

Description

`vpaintegral(f,a,b)` numerically approximates $\int_a^b f(x)\mathrm{d}x$. The default variable `x` in `f` is found by `symvar`.

`vpaintegral(f,[a b])` is equal to `vpaintegral(f,a,b)`.

`vpaintegral(f,x,a,b)` performs numerical integration using the integration variable `x`.

`vpaintegral(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Numerically Integrate Symbolic Expression

Numerically integrate the symbolic expression `x^2` from `1` to `2`.

```
syms x
vpaintegral(x^2, 1, 2)
```

```
ans =
2.33333
```

Numerically Integrate Symbolic Function

Numerically integrate the symbolic function $y(x) = x^2$ from `1` to `2`.

```
syms y(x)
y(x) = x^2;
vpaintegral(y, 1, 2)
```

```
ans =
2.33333
```

High-Precision Numerical Integration

`vpaintegral` uses variable-precision arithmetic while the MATLAB `integral` function uses double-precision arithmetic. Using the default values of tolerance, `vpaintegral` can handle values that cause the MATLAB `integral` function to overflow or underflow.

Integrate `besseli(5,25*u).*exp(-u*25)` by using both `integral` and `vpaintegral`. The `integral` function returns `NaN` and issues a warning while `vpaintegral` returns the correct result.

```
syms u x
f = besseli(5,25*x).*exp(-x*25);
fun = @(u)besseli(5,25*u).*exp(-u*25);

usingIntegral = integral(fun, 0, 30)
usingVpaintegral = vpaintegral(f, 0, 30)
```

```
Warning: Infinite or Not-a-Number value encountered.
usingIntegral =
   NaN

usingVpaintegral =
0.688424
```

Increase Precision Using Tolerances

The `digits` function does not affect `vpaintegral`. Instead, increase the precision of `vpainteral` by decreasing the integration tolerances. Conversely, increase the speed of numerical integration by increasing the tolerances. Control the tolerance used by `vpaintegral` by changing the relative tolerance `RelTol` and absolute tolerance `AbsTol`, which affect the integration through the condition

$$|Q - I| \le \max(AbsTol, |Q| \bullet RelTol)$$
$$\text{where} \quad Q = \text{Calculated Integral}$$
$$I = \text{Exact Integral.}$$

Numerically integrate `besselj(0,x)` from `0` to `pi`, to 32 significant figures by setting `RelTol` to `10^(-32)`.

```
syms x
vpaintegral(besselj(0,x), [0 pi], 'RelTol', 1e-32)
```

```
ans =
1.3475263146739901712314731279612
```

Using lower tolerance values increases precision at the cost of speed.

Complex Path Integration Using Waypoints

Integrate `1/(2*z-1)` over the triangular path from `0` to `1+1i` to `1-1i` back to `0` by specifying waypoints.

```
syms z
vpaintegral(1/(2*z-1), [0 0], 'Waypoints', [1+1i 1-1i])
```

```
ans =
1.11022e-16 - 3.14159i
```

Reversing the direction of the integral, by changing the order of the waypoints and exchanging the limits, changes the sign of the result.

Multiple Integrals

Perform multiple integration by nesting calls to `vpaintegral`. Integrate

$$\int_{-1}^{2}\int_{1}^{3} xy \; dx \; dy.$$

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

```
ans =
6.0
```

The limits of integration can be symbolic expressions or functions. Integrate over the triangular region $0 \le x \le 1$ and `|y|` < x by specifying the limits of the integration over `y` in terms of `x`.

```
vpaintegral(vpaintegral(sin(x-y)/(x-y), y, [-x x]), x, [0 1])

ans =
0.89734
```

Input Arguments

`f` — Expression or function to integrate
symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Expression or function to integrate, specified as a symbolic number, variable, vector, matrix, multidimensional array, function, or expression.

`a,b` — Limits of integration
list of two numbers | list of two symbolic numbers | list of two symbolic variables | list of two symbolic functions | list of two symbolic expressions

Limits of integration, specified as a list of two numbers, symbolic numbers, symbolic variables, symbolic functions, or symbolic expressions.

`x` — Integration variable
symbolic variable

Integration variable, specified as a symbolic variable. If `x` is not specified, the integration variable is found by `symvar`.

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`.

Example: `'RelTol',1e-20`

`RelTol` — Relative error tolerance

`1e-6` (default) | positive real number

Relative error tolerance, specified as a positive real number. The default is `1e-6`. The `RelTol` argument determines the accuracy of the integration only if $RelTol \cdot |Q| > AbsTol$, where Q is the calculated integral. In this case, `vpaintegral` satisfies the condition $|Q - I| \leq RelTol \cdot |Q|$, where I is the exact integral value. To use only `RelTol` and turn off `AbsTol`, set `AbsTol` to `0`.

Example: `1e-8`

`AbsTol` — Absolute error tolerance

`1e-10` (default) | non-negative real number

Absolute error tolerance, specified as a non-negative real number. The default is `1e-10`. `AbsTol` determines the accuracy of the integration if $AbsTol > RelTol \cdot |Q|$, where Q is the calculated integral. In this case, `vpaintegral` satisfies the condition $|Q - I| \leq AbsTol$, where I is the exact integral value. To turn off `AbsTol` and use only `RelTol`, set `AbsTol` to `0`.

Example: `1e-12`

`Waypoints` — Integration path

vector of numbers | vector of symbolic numbers | vector of symbolic expressions | vector of symbolic functions

Integration path, specified as a vector of numbers, or as a vector of symbolic numbers, expressions, or functions. `vpaintegral` integrates along the sequence of straight-line paths (lower limit to the first waypoint, from the first to the second waypoint, and so on) and finally from the last waypoint to the upper limit. For contour integrals, set equal lower and upper limits and define the contour using waypoints.

`MaxFunctionCalls` — Maximum evaluations of input

`10^5` (default) | positive integer | positive symbolic integer

Maximum evaluations of input, specified as a positive integer or a positive symbolic integer. The default value is `10^5`. If the number of evaluations of `f` is greater than

`MaxFunctionCalls`, then `vpaintegral` throws an error. For unlimited evaluations, set `MaxFunctionCalls` to `Inf`.

Tips

- Ensure that the input is integrable. If the input is not integrable, the output of `vpaintegral` is unpredictable.
- The `digits` function does not affect `vpaintegral`. To increase precision, use the `RelTol` and `AbsTol` arguments instead.

See Also

`diff` | `int` | `integral` | `vpa`

Topics

"Integration" on page 2-56

Introduced in R2016b

vpasolve

Solve equations numerically

Syntax

```
S = vpasolve(eqn)
S = vpasolve(eqn,var)
S = vpasolve(eqn,var,init_guess)

Y = vpasolve(eqns)
Y = vpasolve(eqns,vars)
Y = vpasolve(eqns,vars,init_guess)

[y1,...,yN] = vpasolve(eqns)
[y1,...,yN] = vpasolve(eqns,vars)
[y1,...,yN] = vpasolve(eqns,vars,init_guess)

___ = vpasolve(___,Name,Value)
```

Description

`S = vpasolve(eqn)` numerically solves the equation `eqn` for the variable determined by `symvar`.

`S = vpasolve(eqn,var)` numerically solves the equation `eqn` for the variable specified by `var`.

`S = vpasolve(eqn,var,init_guess)` numerically solves the equation `eqn` for the variable specified by `var` using the starting point or search range specified in `init_guess`. If you do not specify `var`, `vpasolve` solves for variables determined by `symvar`.

`Y = vpasolve(eqns)` numerically solves the system of equations `eqns` for variables determined by `symvar`. This syntax returns `Y` as a structure array. You can access the solutions by indexing into the array.

`Y = vpasolve(eqns,vars)` numerically solves the system of equations `eqns` for variables specified by `vars`. This syntax returns a structure array that contains the solutions. The fields in the structure array correspond to the variables specified by `vars`.

`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`.

`[y1,...,yN] = vpasolve(eqns)` numerically solves the system of equations `eqns` for variables determined by `symvar`. This syntax assigns the solutions to variables `y1,...,yN`.

`[y1,...,yN] = vpasolve(eqns,vars)` numerically solves the system of equations `eqns` for the variables specified by `vars`.

`[y1,...,yN] = vpasolve(eqns,vars,init_guess)` numerically solves the system of equations `eqns` for the variables specified by `vars` using the starting values or the search range `init_guess`.

`___ = vpasolve(___,Name,Value)` numerically solves the equation or system of equations for the variable or variables using additional options specified by one or more `Name,Value` pair arguments.

Examples

Solve Polynomial Equation

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 + 0.76331583387715452512978468102263i
 - 0.88011377126068169817875190457835 - 0.76331583387715452512978468102263i
   0.50511377126068169817875190457835 + 0.81598965068946312853227067890656i
   0.50511377126068169817875190457835 - 0.81598965068946312853227067890656i
```

If `vpasolve` returns an empty object, then no solution was found.

```
eqns = [3*x+2, 3*x+1];
vpasolve(eqns, x)
```

```
ans =
Empty sym: 0-by-1
```

Solve Nonpolynomial Equation

For nonpolynomial equations, `vpasolve` returns the first solution that it finds:

```
syms x
vpasolve(sin(x^2) == 1/2, x)
```

```
ans =
-226.94447241941511682716953887638
```

Assign Solutions to 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 = vpasolve([x^3 + 2*x == y, y^2 == x], [x, y])
```

```
S = 
  struct with fields:

    x: [6×1 sym]
    y: [6×1 sym]
```

Display solutions by accessing the elements of the structure array `S`:

```
S.x
```

```
ans =
                                         0.236574294277334161761487152176 8
                                                                         0
 - 0.28124065338711968666197895499453 + 1.2348724236470142074859894531946i
 - 0.28124065338711968666197895499453 - 1.2348724236470142074859894531946i
   0.16295350624845260578123537890613 - 1.6151544650555366917886585417926i
   0.16295350624845260578123537890613 + 1.6151544650555366917886585417926i
```

```
S.y
```

```
ans =
                                         0.48638903593454300001655725369801
                                                                          0
   0.70187356885586188630668751791218 + 0.87969719792982402287026727381769i
   0.70187356885586188630668751791218 - 0.87969719792982402287026727381769i
```

```
 - 0.94506808682313338631496614476119 + 0.85451751443904587692179191887616i
 - 0.94506808682313338631496614476119 - 0.85451751443904587692179191887616i
```

Assign Solutions to Variables When Solving System of Equations

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])
```

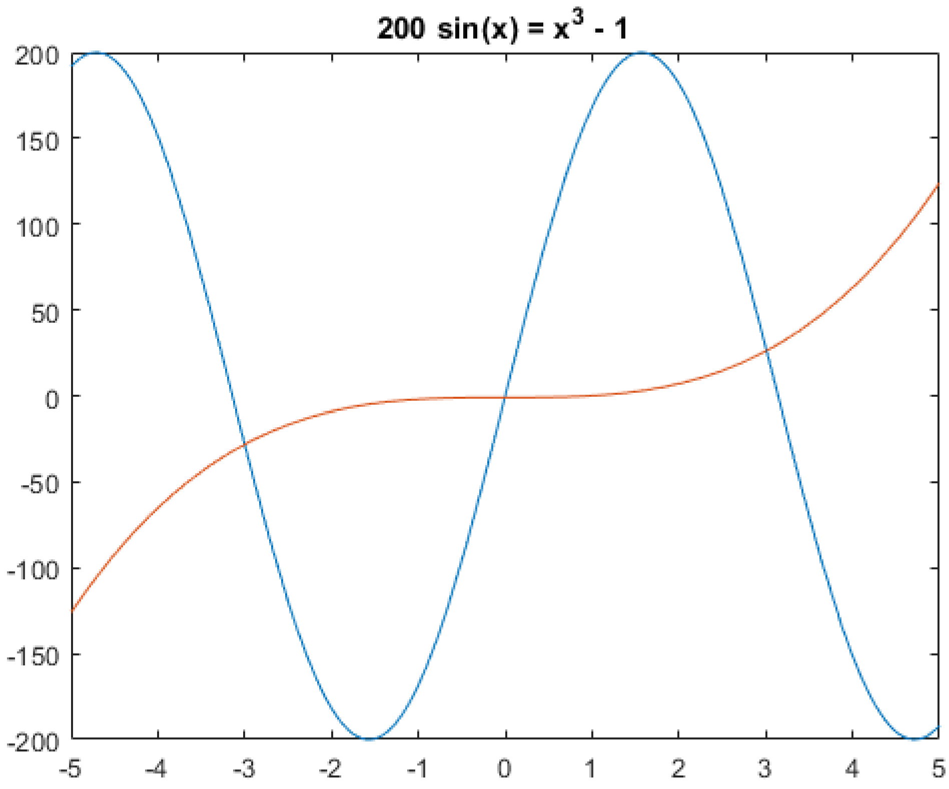
```
sol_x =
88.90707209659114864849280774681

sol_y =
0.00000000000013470479710676694388973703681918
```

Find Multiple Solutions by Specifying Starting Points

Plot the two sides of the equation, and then use the plot to specify initial guesses for the solutions.

Plot the left and right sides of the equation $200\sin(x) = x^3 - 1$.

```
syms x
eqnLeft = 200*sin(x);
eqnRight = x^3 - 1;
fplot([eqnLeft eqnRight])
title([texlabel(eqnLeft) ' = ' texlabel(eqnRight)])
```



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
```

Find one of the other solutions by specifying 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
```

Specify Ranges for Solutions

You can specify ranges for solutions of an equation. For example, if you want to restrict your search to only real solutions, you cannot use assumptions because `vpasolve` ignores assumptions. Instead, specify a search interval. For the following equation, if you do not specify ranges, the numeric solver returns all eight solutions of the equation:

```
syms x
vpasolve(x^8 - x^2 == 3, x)

ans =
                                        -1.2052497163799060695888397264341
                                         1.2052497163799060695888397264341
 - 0.77061431370803029127495426747428 + 0.85915207603993818859321142757163i
 - 0.77061431370803029127495426747428 - 0.85915207603993818859321142757164i
                                        -1.0789046020338265308047436284205i
                                         1.0789046020338265308047436284205i
   0.77061431370803029127495426747428 + 0.85915207603993818859321142757164i
   0.77061431370803029127495426747428 - 0.85915207603993818859321142757163i
```

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 + 0.85915207603993818859321142757163i
 - 0.77061431370803029127495426747428 - 0.85915207603993818859321142757164i
                                        -1.0789046020338265308047436284205i
                                         1.0789046020338265308047436284205i
   0.77061431370803029127495426747428 + 0.85915207603993818859321142757164i
   0.77061431370803029127495426747428 - 0.85915207603993818859321142757163i
```

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
```

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 + 0.85915207603993818859321142757164i
   0.77061431370803029127495426747428 + 0.85915207603993818859321142757164i
```

Find Multiple Solutions for Nonpolynomial Equation

By default, `vpasolve` returns the same solution on every call. To find more than one solution for nonpolynomial equations, set `random` to `true`. This makes `vpasolve` use a random starting value which can lead to different solutions on successive calls.

If `random` is not specified, `vpasolve` returns the same solution on every call.

```
syms x
f = x-tan(x);
for n = 1:3
  vpasolve(f,x)
end
```

```
ans =
0
ans =
0
ans =
0
```

When `random` is set to `true`, `vpasolve` returns a distinct solution on every call.

```
syms x
f = x-tan(x);
```

```
for n = 1:3
  vpasolve(f,x,'random',true)
end

ans =
 -227.76107684764829218924973598808
 ans =
 102.09196646490764333652956578441
 ans =
 61.244730260374400372753016364097
```

`random` can be used in conjunction with a search range:

```
vpasolve(f,x,[10 12],'random',true)

ans =
10.904121659428899827148702790189
```

Input Arguments

`eqn` — Equation to solve

symbolic equation | symbolic expression

Equation to solve, specified as a symbolic equation or symbolic expression. A symbolic equation is 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 to solve equation for

symbolic variable

Variable to solve equation for, specified as a symbolic variable. If `var` is not specified, `symvar` determines the variables.

`eqns` — System of equations or expressions to solve

symbolic vector | symbolic matrix | symbolic N-D array

System of equations or expressions to be solve, specified as a symbolic vector, matrix, or N-D array of equations or expressions. These equations or expressions can also be separated by commas. 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 to solve system of equations for
symbolic vector

Variables to solve system of equations for, specified as a symbolic vector. These variables are specified as a vector or comma-separated list. If `vars` is not specified, `symvar` determines the variables.

`init_guess` — Initial guess for solution
numeric value | vector | matrix with two columns

Initial guess for a solution, specified as a numeric value, vector, or matrix with two columns.

If `init_guess` is a number or, in the case of multivariate equations, a vector of numbers, then the numeric solver uses it as a starting point. If `init_guess` is specified as a scalar while the system of equations is multivariate, then the numeric solver uses the scalar value as a starting point for all variables.

If `init_guess` is a matrix with two columns, then the two entries of the rows specify the bounds of a search range for the corresponding variables. To specify a starting point in a matrix of search ranges, specify both columns as the starting point value.

To omit a search range for a variable, set the search range for that variable to `[NaN, NaN]` in `init_guess`. All other uses of `NaN` in `init_guess` will error.

By default, `vpasolve` uses its own internal choices for starting points and search ranges.

Name-Value Pair Arguments

Example: `vpasolve(x^2 - 4 == 0,x,'random',true)`

`random` — Use of random starting point for finding multiple solutions
false (default) | true

Use a random starting point for finding solutions, specified as a comma-separated pair consisting of `random` and a value, which is either `true` or `false`. This is useful when you solve nonpolynomial equations where there is no general method to find all the solutions. If the value is `false`, `vpasolve` uses the same starting value on every call. Hence, multiple calls to `vpasolve` with the same inputs always find the same solution, even if several solutions exist. If the value is `true`, however, starting values for the internal search are chosen randomly in the search range. Hence, multiple calls to `vpasolve` with the same inputs might lead to different solutions. Note that if you specify starting points for all variables, setting `random` to `true` has no effect.

Output Arguments

`S` — Solutions of univariate equation

symbolic value | symbolic array

Solutions of univariate equation, returned as symbolic value or symbolic array. The size of a symbolic array corresponds to the number of the solutions.

`Y` — Solutions of system of equations

structure array

Solutions of system of equations, returned as a structure array. The number of fields in the structure array corresponds to the number of variables to be solved for.

`y1,...,yN` — Variables that are assigned solutions of system of equations

array of numeric variables | array of symbolic variables

Variables that are assigned solutions of system of equations, returned as an array of numeric or symbolic variables. The number of output variables or symbolic arrays must equal the number of variables to be solved for. 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.

Tips

- If `vpasolve` returns an empty object, then no solution was found.
- `vpasolve` returns all solutions only for polynomial equations. For nonpolynomial equations, there is no general method of finding all solutions. When you look for numerical solutions of a nonpolynomial equation or system that has several solutions, then, by default, `vpasolve` returns only one solution, if any. To find more than just one solution, set `random` to true. Now, calling `vpasolve` repeatedly might return several different solutions.
- When you solve a system where there are not enough equations to determine all variables uniquely, the behavior of `vpasolve` behavior depends on whether the

system is polynomial or nonpolynomial. If polynomial, `vpasolve` returns all solutions by introducing an arbitrary parameter. If nonpolynomial, a single numerical solution is returned, if it exists.

- 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.
- 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`.
- Place equations and expressions to the left of the argument list, and the variables to the right. `vpasolve` checks for variables starting on the right, and on reaching the first equation or expression, assumes everything to the left is an equation or expression.
- 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.

Algorithms

- When you set `random` to `true` and specify a search range for a variable, random starting points within the search range are chosen using the internal random number generator. The distribution of starting points within finite search ranges is uniform.
- When you set `random` to `true` and do not specify a search range for a variable, random starting points are generated using a Cauchy distribution with a half-width of `100`. This means the starting points are real valued and have a large spread of values on repeated calls.

See Also

`dsolve` | `equationsToMatrix` | `fzero` | `linsolve` | `solve` | `symvar` | `vpa`

Introduced in R2012b

whittakerM

Whittaker M function

Syntax

`whittakerM(a,b,z)`

Description

`whittakerM(a,b,z)` returns the value of the Whittaker M function on page 4-1722.

Input Arguments

`a`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `a` is a vector or matrix, `whittakerM` returns the beta function for each element of `a`.

`b`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `b` is a vector or matrix, `whittakerM` returns the beta function for each element of `b`.

`z`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `x` is a vector or matrix, `whittakerM` returns the beta function for each element of `z`.

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
isAlways(diff(whittakerM(a,b,z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(a,b,z) == 0)

ans =
  logical
   1
```

Verify that `whittakerM(-a,-b,-z)` also is a valid solution of this differential equation:

```
syms a b z
isAlways(diff(whittakerM(-a,-b,-z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(-a,-b,-z) == 0)

ans =
  logical
   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 + 2i),
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)*1i, exp(x^2/2)*(x^2)^(1/2)]
[            0,        x^(1/2)*exp(x/2)]
```

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:

$$M_{a,b}(z)=e^{-z/2}\,z^{b+1/2}\,M\left(b-a+\frac{1}{2},1+2b,z\right)$$

Tips

- All non-scalar arguments must have the same size. If one or two input arguments are non-scalar, then `whittakerM` expands the scalars into vectors or matrices of the same size as the non-scalar arguments, with all elements equal to the corresponding scalar.

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` | `kummerU` | `whittakerW`

Introduced in R2012a

whittakerW

Whittaker W function

Syntax

`whittakerW(a,b,z)`

Description

`whittakerW(a,b,z)` returns the value of the Whittaker W function on page 4-1727.

Input Arguments

`a`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `a` is a vector or matrix, `whittakerW` returns the beta function for each element of `a`.

`b`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `b` is a vector or matrix, `whittakerW` returns the beta function for each element of `b`.

`z`

Symbolic number, variable, expression, function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `x` is a vector or matrix, `whittakerW` returns the beta function for each element of `z`.

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
isAlways(diff(whittakerW(a, b, z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(a, b, z) == 0)

ans =
  logical
   1
```

Verify that `whittakerW(-a, -b, -z)` also is a valid solution of this differential equation:

```
syms a b z
isAlways(diff(whittakerW(-a, -b, -z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(-a, -b, -z) == 0)

ans =
  logical
   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 + 2i),
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))/(x^b*pi^(1/2))

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)*(ei(1) + pi*1i)*1i,...
   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)]
```

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:

$$W_{a,b}(z)=e^{-z/2}z^{b+1/2}U\left(b-a+\frac{1}{2},1+2b,z\right)$$

Tips

- All non-scalar arguments must have the same size. If one or two input arguments are non-scalar, then `whittakerW` expands the scalars into vectors or matrices of the same size as the non-scalar arguments, with all elements equal to the corresponding scalar.

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` | `kummerU` | `whittakerM`

Introduced in R2012a

wrightOmega

Wright omega function

Syntax

```
wrightOmega(x)
wrightOmega(A)
```

Description

`wrightOmega(x)` computes the Wright omega function on page 4-1731 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.

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 =
  -1.0000 + 0.0000
```

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
```

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 ω(x) is a solution of the equation $Y + \log(Y) = X$.

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`

Introduced in R2011b

xor

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.

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 `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`.

```
isAlways(x1)
isAlways(x2)

ans =
  logical
     1

ans =
  logical
     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 `isAlways`:

```
isAlways(s1)
isAlways(s2)

ans =
  logical
     1
```

```
ans =
  logical
     0
```

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 `isAlways`.
- `assume` and `assumeAlso` do not accept assumptions that contain `xor`.

See Also

`all` | `and` | `any` | `isAlways` | `not` | `or`

Introduced in R2012a

zeta

Riemann zeta function

Syntax

```
zeta(z)
zeta(n,z)
```

Description

`zeta(z)` evaluates the Riemann zeta function at the elements of `z`, where `z` is a numeric or symbolic input.

`zeta(n,z)` returns the `n`th derivative of `zeta(z)`.

Examples

Find Riemann Zeta Function for Numeric and Symbolic Inputs

Find the Riemann zeta function for numeric inputs.

```
zeta([0.7 i 4 11/3])
```

```
ans =
  -2.7784 + 0.0000i   0.0033 - 0.4182i   1.0823 + 0.0000i   1.1094 + 0.0000i
```

Find the Riemann zeta function symbolically by converting the inputs to symbolic objects using `sym`. The `zeta` function returns exact results.

```
zeta(sym([0.7 i 4 11/3]))
```

```
ans =
[ zeta(7/10), zeta(i), pi^4/90, zeta(11/3)]
```

`zeta` returns unevaluated function calls for symbolic inputs that do not have results implemented. The implemented results are listed in “Algorithms” on page 4-1740.

Find the Riemann zeta function for a matrix of symbolic expressions.

```
syms x y
Z = zeta([x sin(x); 8*x/11 x + y])
```

```
Z =
[         zeta(x), zeta(sin(x))]
[ zeta((8*x)/11),  zeta(x + y)]
```

Find Riemann Zeta Function for Large Inputs

For values of `|z|>1000`, `zeta(z)` might return an unevaluated function call. Use `expand` to force `zeta` to evaluate the function call.

```
zeta(sym(1002))
expand(zeta(sym(1002)))
```

```
ans =
zeta(1002)
ans =
(1087503...312*pi^1002)/15156647...375
```

Differentiate Riemann Zeta Function

Find the third derivative of the Riemann zeta function at point `x`.

```
syms x
expr = zeta(3,x)
```

```
expr =
zeta(3, x)
```

Find the third derivative at `x = 4` by substituting `4` for `x` using `subs`.

```
expr = subs(expr,x,4)
```

```
expr =
zeta(3, 4)
```

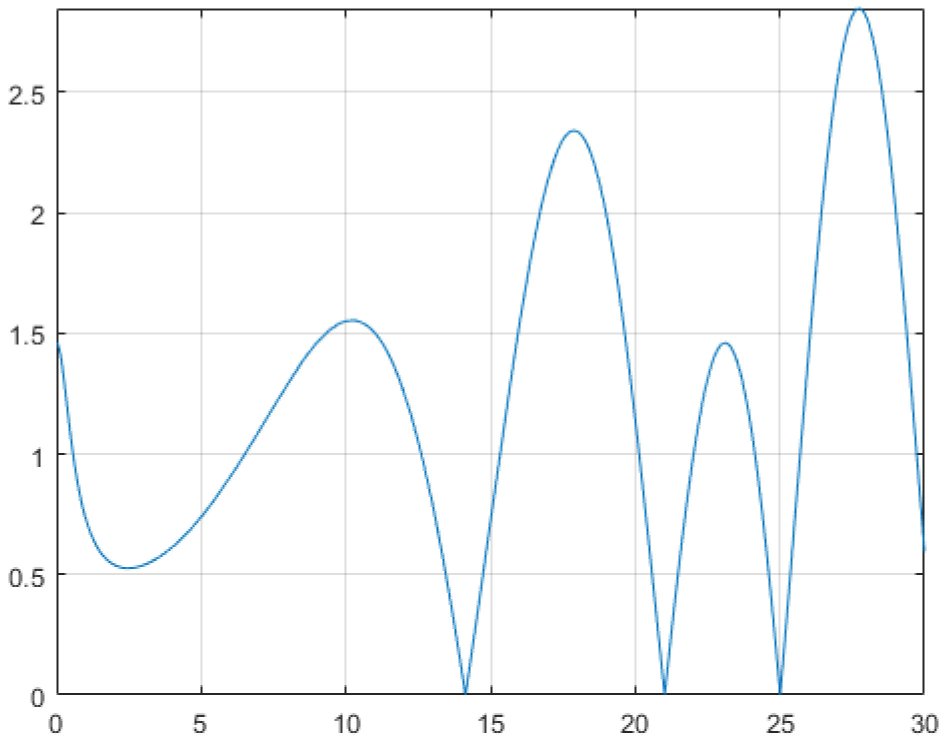
Evaluate `expr` using `vpa`.

```
expr = vpa(expr)

expr =
-0.07264084989132137196244616781177
```

Plot Zeros of Riemann Zeta Function

Zeros of the Riemann Zeta function `zeta(x+i*y)` are found along the line `x = 1/2`. Plot the absolute value of the function along this line for `0<y<30` to view the first three zeros. Prior to R2016a, use `ezplot` instead of `fplot`.

```
syms y
fplot(abs(zeta(1/2+1i*y)),[0 30])
grid on
```



Input Arguments

z — Input
number | vector | matrix | multidimensional array | symbolic number | symbolic variable | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic function | symbolic expression

Input, specified as a number, vector, matrix or multidimensional array, or a symbolic number, variable, vector, matrix, multidimensional array, function or expression.

n — Order of derivative
nonnegative integer

Order of derivative, specified as a nonnegative integer.

Definitions

Riemann Zeta Function

The Riemann zeta function is defined by

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$$

The series converges only if the real part of `z` is greater than 1. The definition of the function is extended to the entire complex plane, except for a simple pole `z = 1`, by analytic continuation.

Tips

- Floating point evaluation is slow for large values of `n`.

Algorithms

The following exact values are implemented.

- $$\zeta(0) = -\frac{1}{2}$$
- $$\zeta(0,1) = -\frac{\ln(\pi)}{2} - \frac{\ln(2)}{2}$$
- $$\zeta(\infty) = 1$$
- If $z < 0$ and `z` is an even integer, $\zeta(z) = 0$.
- If $z < 0$ and `z` is an odd integer

  $$\zeta(z) = -\frac{\text{bernoulli}(1-z)}{1-z}$$

  For $z < -1000$, `zeta(z)` returns an unevaluated function call. To force evaluation, use `expand(zeta(z))`.
- If $z > 0$ and `z` is an even integer

  $$\zeta(z) = \frac{(2\pi)^z \,|\,\text{bernoulli}(z)\,|}{2z!}$$

  For $z > 1000$, `zeta(z)` returns an unevaluated function call. To force evaluation, use `expand(zeta(z))`.
- If $n > 0$, $\zeta(n,\infty) = 0$.
- If the argument does not evaluate to a listed special value, `zeta` returns the symbolic function call.

See Also

`bernoulli`

Introduced before R2006a

ztrans

Z-transform

Syntax

```
ztrans(f)
ztrans(f,transVar)
ztrans(f,var,transVar)
```

Description

`ztrans(f)` finds the “Z-Transform” on page 4-1746 of `f`. By default, the independent variable is `n` and the transformation variable is `z`. If `f` does not contain `n`, `ztrans` uses `symvar`.

`ztrans(f,transVar)` uses the transformation variable `transVar` instead of `z`.

`ztrans(f,var,transVar)` uses the independent variable `var` and transformation variable `transVar` instead of `n` and `z`, respectively.

Examples

Z-Transform of Symbolic Expression

Compute the Z-transform of `sin(n)`. By default, the transform is in terms of `z`.

```
syms n
f = sin(n);
ztrans(f)
```

```
ans =
(z*sin(1))/(z^2 - 2*cos(1)*z + 1)
```

Specify Independent Variable and Transformation Variable

Compute the Z-transform of `exp(m+n)`. By default, the independent variable is `n` and the transformation variable is `z`.

```
syms m n
f = exp(m+n);
ztrans(f)

ans =
(z*exp(m))/(z - exp(1))
```

Specify the transformation variable as `y`. If you specify only one variable, that variable is the transformation variable. The independent variable is still `n`.

```
syms y
ztrans(f,y)

ans =
(y*exp(m))/(y - exp(1))
```

Specify both the independent and transformation variables as `m` and `y` in the second and third arguments, respectively.

```
ztrans(f,m,y)

ans =
(y*exp(n))/(y - exp(1))
```

Z-Transforms Involving Dirac and Heaviside Functions

Compute the Z-transform of 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))
```

```
ans =
z/(z - 1)^3
```

Z-Transform of Array Inputs

Find the Z-transform of the matrix `M`. Specify the independent and transformation variables for each matrix entry by using matrices of the same size. When the arguments are nonscalars, `ztrans` acts on them element-wise.

```
syms a b c d w x y z
M = [exp(x) 1; sin(y) i*z];
vars = [w x; y z];
transVars = [a b; c d];
ztrans(M,vars,transVars)

ans =
[                (a*exp(x))/(a - 1),       b/(b - 1)]
[ (c*sin(1))/(c^2 - 2*cos(1)*c + 1), (d*1i)/(d - 1)^2]
```

If `ztrans` is called with both scalar and nonscalar arguments, then it expands the scalars to match the nonscalars by using scalar expansion. Nonscalar arguments must be the same size.

```
syms w x y z a b c d
ztrans(x,vars,transVars)

ans =
[ (a*x)/(a - 1),   b/(b - 1)^2]
[ (c*x)/(c - 1), (d*x)/(d - 1)]
```

Z-Transform of Symbolic Function

Compute the Z-transform of symbolic functions. When the first argument contains symbolic functions, then the second argument must be a scalar.

```
syms f1(x) f2(x) a b
f1(x) = exp(x);
f2(x) = x;
ztrans([f1 f2],x,[a b])
```

```
ans =
[ a/(a - exp(1)), b/(b - 1)^2]
```

If Z-Transform Cannot Be Found

If `ztrans` cannot transform the input then it returns an unevaluated call.

```
syms f(n)
f(n) = 1/n;
F = ztrans(f,n,z)

F =
ztrans(1/n, n, z)
```

Return the original expression by using `iztrans`.

```
iztrans(F,z,n)

ans =
1/n
```

Input Arguments

`f` — Input

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input, specified as a symbolic expression, function, vector, or matrix.

`var` — Independent variable

`n` (default) | symbolic variable

Independent variable, specified as a symbolic variable. This variable is often called the "discrete time variable". If you do not specify the variable, then `ztrans` uses `n`. If `f` does not contain `n`, then `ztrans` uses the function `symvar`.

`transVar` — Transformation variable

`z` (default) | symbolic variable | symbolic expression | symbolic vector | symbolic matrix

Transformation variable, specified as a symbolic variable, expression, vector, or matrix. This variable is often called the "complex frequency variable." By default, `ztrans` uses `z`. If `z` is the independent variable of `f`, then `ztrans` uses `w`.

Definitions

Z-Transform

The Z-transform $F = F(z)$ of the expression $f = f(n)$ with respect to the variable `n` at the point `z` is

$$F(z) = \sum_{n=0}^{\infty} \frac{f(n)}{z^n}.$$

Tips

- If any argument is an array, then `ztrans` acts element-wise on all elements of the array.
- If the first argument contains a symbolic function, then the second argument must be a scalar.
- To compute the inverse Z-transform, use `iztrans`.

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `kroneckerDelta` | `laplace`

Topics

"Solve Difference Equations Using Z-Transform" on page 2-233

Introduced before R2006a