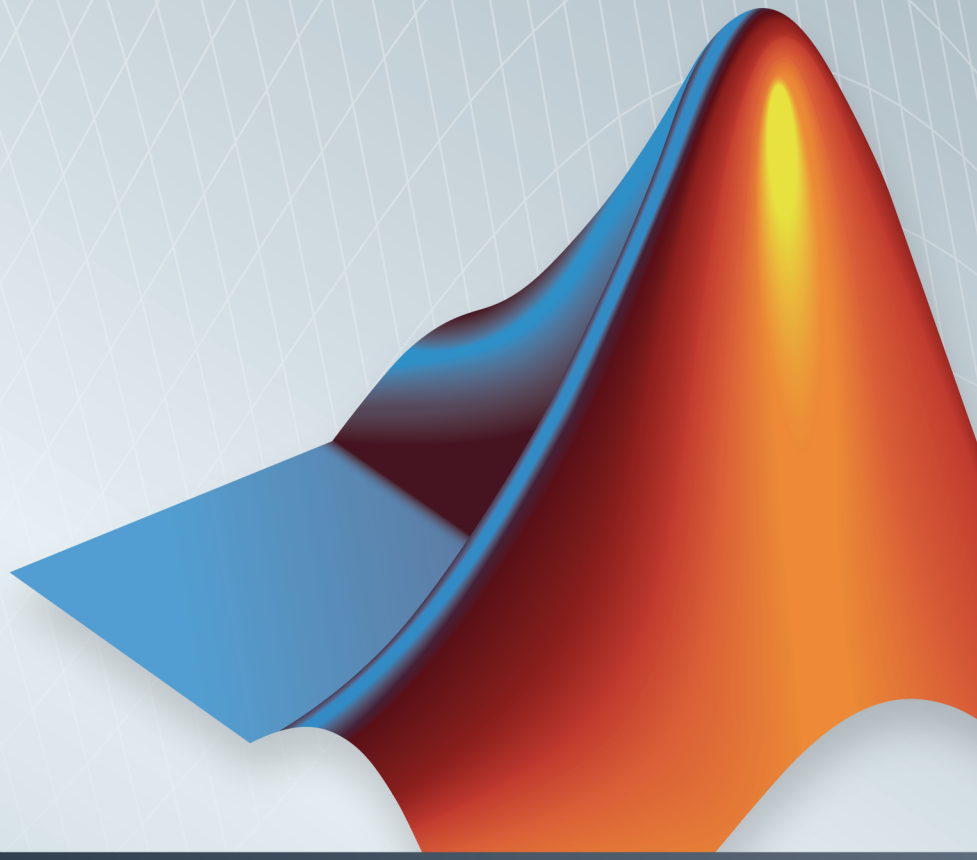
Symbolic Math Toolbox™

User's Guide

R2014**b**



MATLAB®

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)

Contents

2 Using Symbolic Math Toolbox Software

3 MuPAD in Symbolic Math Toolbox

Acknowledgments

1

Getting Started

Symbolic Math Toolbox Product Description

Perform symbolic math computations

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

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

Key Features

- Symbolic integration, differentiation, transforms, and linear algebra
- Algebraic and ordinary differential equation (ODE) solvers
- Simplification and manipulation of symbolic expressions
- Code generation from symbolic expressions for MATLAB, Simulink, Simscape, C, Fortran, MathML, and TeX
- Variable-precision arithmetic
- MuPAD Notebook for performing and documenting symbolic calculations
- MuPAD language and function libraries for combinatorics, number theory, and other mathematical areas

Access Symbolic Math Toolbox Functionality

In this section...

Work from MATLAB

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

Work from MuPAD

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

A debugger and other programming utilities provide tools for authoring custom symbolic functions and libraries in the MuPAD language. The MuPAD language supports multiple programming styles including imperative, functional, and object-oriented programming. The language treats variables as symbolic by default and is optimized for handling and operating on symbolic math expressions. You can call functions written in the MuPAD language from the MATLAB Command Window. For more information, see “Call Built-In MuPAD Functions from MATLAB”

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

Symbolic Objects

In this section...

Overview of Symbolic Objects

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

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

Symbolic objects are symbolic variables, symbolic numbers, symbolic expressions, symbolic matrices, and symbolic functions.

Symbolic Variables

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

```
syms x y
```

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

```
x + x + y
```

```
ans =
 2*x + y
```

You also can create formal symbolic mathematical expressions and symbolic matrices. See “Create Symbolic Variables and Expressions” on page 1-7 for more information.

Symbolic Numbers

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

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

If you create a symbolic number with 15 or fewer decimal digits, you can skip the quotes:

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

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

```
sqrt(2)
```

returns a double-precision floating-point number:

```
ans =
    1.4142
```

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

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

you get the precise symbolic result:

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

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

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

```
double(a)

ans =
```

```
    1.4142
```

You also can create a rational fraction involving symbolic numbers:

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

ans =
2/5
```

or more efficiently:

```
sym(2/5)

ans =
2/5
```

MATLAB performs arithmetic on symbolic fractions differently than it does on standard numeric fractions. By default, MATLAB stores all numeric values as double-precision floating-point data. For example:

```
2/5 + 1/3

ans =
    0.7333
```

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

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

ans =
 11/15
```

To learn more about symbolic representation of rational and decimal fractions, see “Estimate Precision of Numeric to Symbolic Conversions” on page 1-19.

Create Symbolic Variables and Expressions

In this section...

Create Symbolic Variables

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

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

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

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

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

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

The `sym` command:

- Requires parentheses and quotation marks: `x = sym('x')`. When creating a symbolic number with 15 or fewer decimal digits, you can skip the quotation marks: `f = sym(5)`.

- Creates one symbolic object with each call.
- Serves best for creating symbolic numbers and symbolic expressions.
- Serves best for creating symbolic objects in functions and scripts.

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

Create Symbolic Expressions

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

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

The command

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

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

```
f = phi^2 - phi - 1
```

returns

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

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

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

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

```
a = sym('a');
b = sym('b');
```

```
c = sym('c');
x = sym('x');
```

or simply

```
syms a b c x
```

Then, enter

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

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

Create Symbolic Functions

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

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

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

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

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

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

```
syms x y
```

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

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

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

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

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

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

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

```
f(y + 1, y)
```

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

Create Symbolic Objects with Identical Names

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

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

returns

```
f =
a + b
```

If later you enter

```
syms f
f
```

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

```
f =
```

```
f
```

You can use the `syms` command to clear variables of definitions that you previously assigned to them in your MATLAB session. However, `syms` does not clear the following assumptions of the variables: complex, real, and positive. These assumptions are stored separately from the symbolic object. See "Delete Symbolic Objects and Their Assumptions" on page 1-33 for more information.

Create a Matrix of Symbolic Variables

Use Existing Symbolic Objects as Elements

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

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

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

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

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

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

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

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

The sums are equal:

```
ans =
     1
```

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

Generate Elements While Creating a Matrix

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

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

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

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

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

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

Create a Matrix of Symbolic Numbers

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

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

generates the 3-by-3 Hilbert matrix:

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

By applying `sym` to `A`

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

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

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

For more information on numeric to symbolic conversions, see “Estimate Precision of Numeric to Symbolic Conversions” on page 1-19.

Find Symbolic Variables in Expressions, Functions, Matrices

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

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

ans =
[ n, x]
```

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

```
symvar(g)

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

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

```
symvar(g, 2)

ans =
[ t, b]
```

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

You also can find symbolic variables in a function:

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

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

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

```
symvar(f, 2)

ans =
[ w, z]
```

Find a Default Symbolic Variable

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

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

ans =
t

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

ans =
tx
```

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

Perform Symbolic Computations

In this section...

Simplify Symbolic Expressions

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

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

returns

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

You can simplify this answer by entering

```
simplify(f)
```

and get a very short answer:

```
ans =
0
```

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

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

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

ans =
x^10 - 1
```

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

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

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

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

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

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

For a list of Symbolic Math Toolbox simplification functions, see “Simplifications” on page 2-40.

Substitutions in Symbolic Expressions

Substitute Symbolic Variables with Numbers

You can substitute a symbolic variable with a numeric value by using the `subs` function. For example, evaluate the symbolic expression `f` at the point `x` = 1/3:

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

```
subs(f, 1/3)

ans =
2/9
```

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

```
f

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

Substitute in Multivariate Expressions

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

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

enter the command

```
subs(f, x, 3)

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

Substitute One Symbolic Variable for Another

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

```
subs(f, y, x)

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

Substitute a Matrix into a Polynomial

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

Element-by-Element Substitution

To substitute a matrix at each element, use the `subs` command:

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

ans =
[ 312, 250,  170]
[  78, -20, -118]
```

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

Substitution in a Matrix Sense

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

1. Create the polynomial:

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

2. Create the magic square matrix:

   ```
   A = magic(3)

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

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

   ```
   b = sym2poly(f)

   b =
        1   -15   -24   350
   ```

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

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

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

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

```
polyvalm(b,A)
```

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

Substitute the Elements of a Symbolic Matrix

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

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

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

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

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

The result is the matrix:

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

For more information, see “Substitution”.

Estimate Precision of Numeric to Symbolic Conversions

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

```
t = 0.1;
sym(t)

ans =
1/10
```

The technique for converting floating-point numbers is specified by the optional second argument, which can be `'f'`, `'r'`, `'e'` or `'d'`. The default option is `'r'`, which stands for rational approximation "Conversion to Rational Symbolic Form" on page 1-20.

Conversion to Floating-Point Symbolic Form

The `'f'` option to `sym` converts double-precision floating-point numbers to exact numeric values `N*2^e`, where `e` and `N` are integers, and `N` is nonnegative. For example,

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

returns the symbolic floating-point representation:

```
ans =
3602879701896397/36028797018963968
```

Conversion to Rational Symbolic Form

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

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

you get the results in the rational form:

```
ans =
1/10
```

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

```
sym(t)

ans =
1/10
```

Conversion to Rational Symbolic Form with Machine Precision

If you call the `sym` command with the option `'e'`, it returns the rational form of `t` plus the difference between the theoretical rational expression for `t` and its actual (machine) floating-point value in terms of `eps` (the floating-point relative precision):

```
sym(t, 'e')

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

Conversion to Decimal Symbolic Form

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

```
sym(t,'d')

ans =
0.10000000000000000555111512312578
```

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

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

ans =
0.1
```

Differentiate Symbolic Expressions

With the Symbolic Math Toolbox software, you can find

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

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

Expressions with One Variable

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

```
syms x
```

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

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

Partial Derivatives

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

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

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

For the complete set of rules MATLAB applies for choosing a default variable, see "Find a Default Symbolic Variable" on page 1-14.

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 2-12.

Indefinite Integrals of One-Variable Expressions

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

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

To find the indefinite integral, enter

```
int(f)

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

Indefinite Integrals of Multivariable Expressions

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

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

ans =
```

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

For the complete set of rules MATLAB applies for choosing a default variable, see “Find a Default Symbolic Variable” on page 1-14.

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

ans =
```

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

Solve Equations

You can solve different types of symbolic equations including:

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

For in-depth information on solving symbolic equations including differential equations, see “Equation Solving”.

Solve Algebraic Equations with One Symbolic Variable

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

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

```
ans =
 1
 2
 3
```

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

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

```
ans =
 1
 2
 3
```

Solve Algebraic Equations with Several Symbolic Variables

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

```
syms x y
```

```
solve(6*x^2 - 6*x^2*y + x*y^2 - x*y + y^3 - y^2 == 0, y)

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

If you do not specify any variable, you get the solution of an equation for the alphabetically closest to `x` variable. For the complete set of rules MATLAB applies for choosing a default variable see “Find a Default Symbolic Variable” on page 1-14.

Solve Systems of Algebraic Equations

You also can solve systems of equations. For example:

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

x =
 0
 2

y =
 0
 2

z =
 0
 8
```

Create Plots of Symbolic Functions

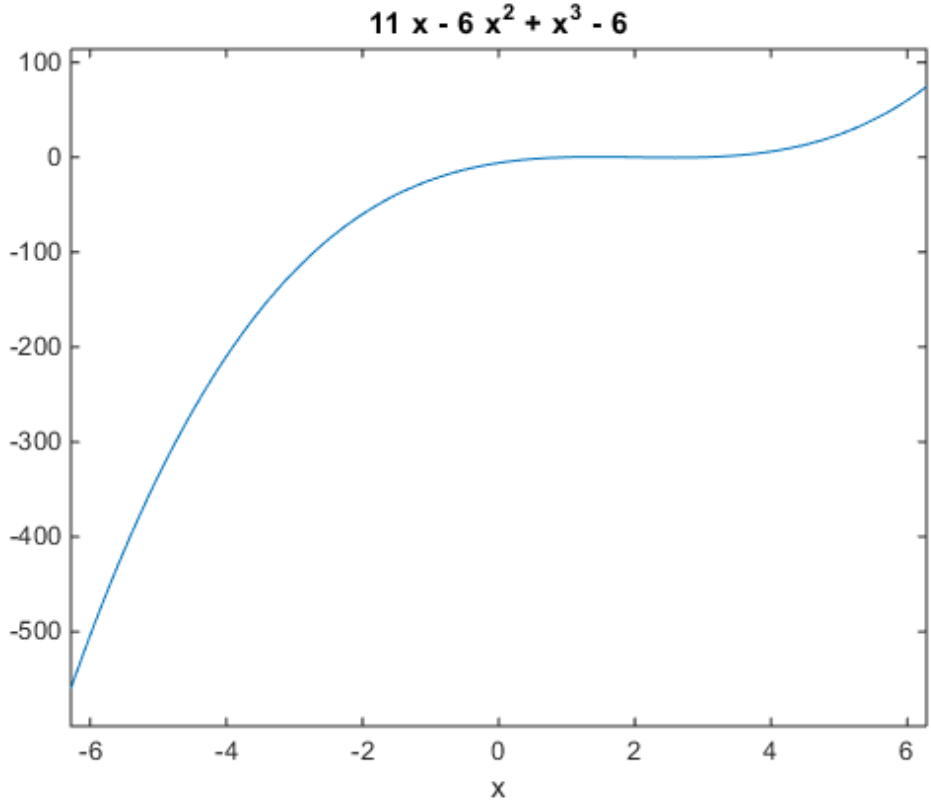
You can create different types of graphs including:

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

Explicit Function Plot

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

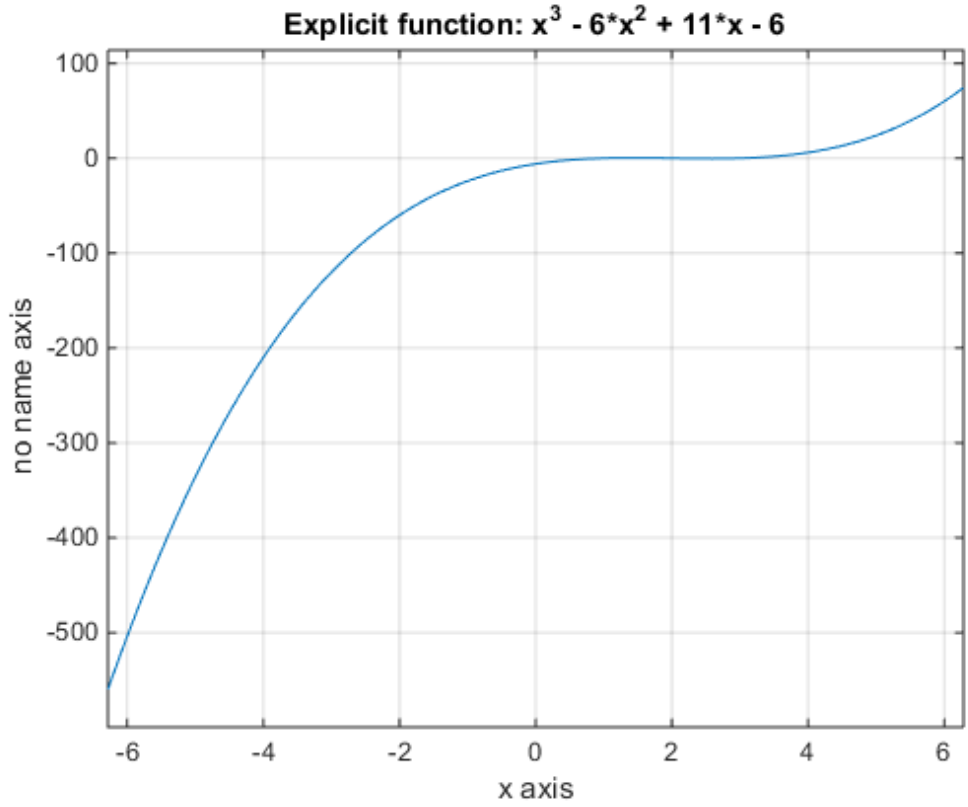
```
syms x
ezplot(x^3 - 6*x^2 + 11*x - 6)
hold on
```



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

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

```
grid on
hold off
```

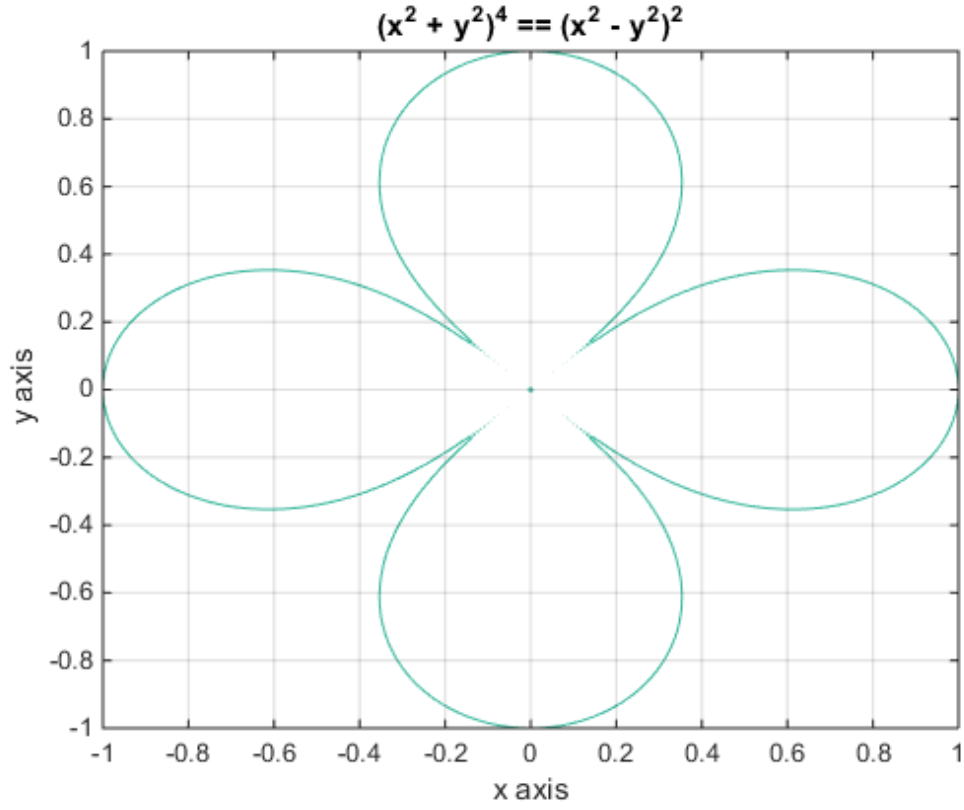


Implicit Function Plot

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

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

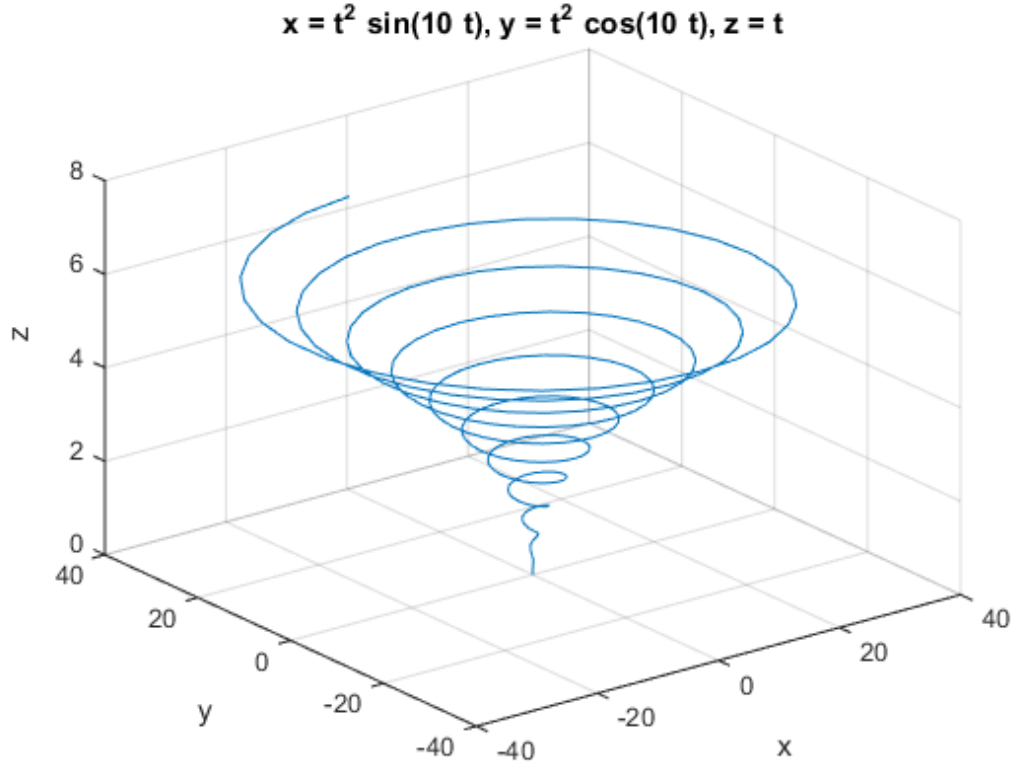
```
grid on
hold off
```



3-D Plot

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

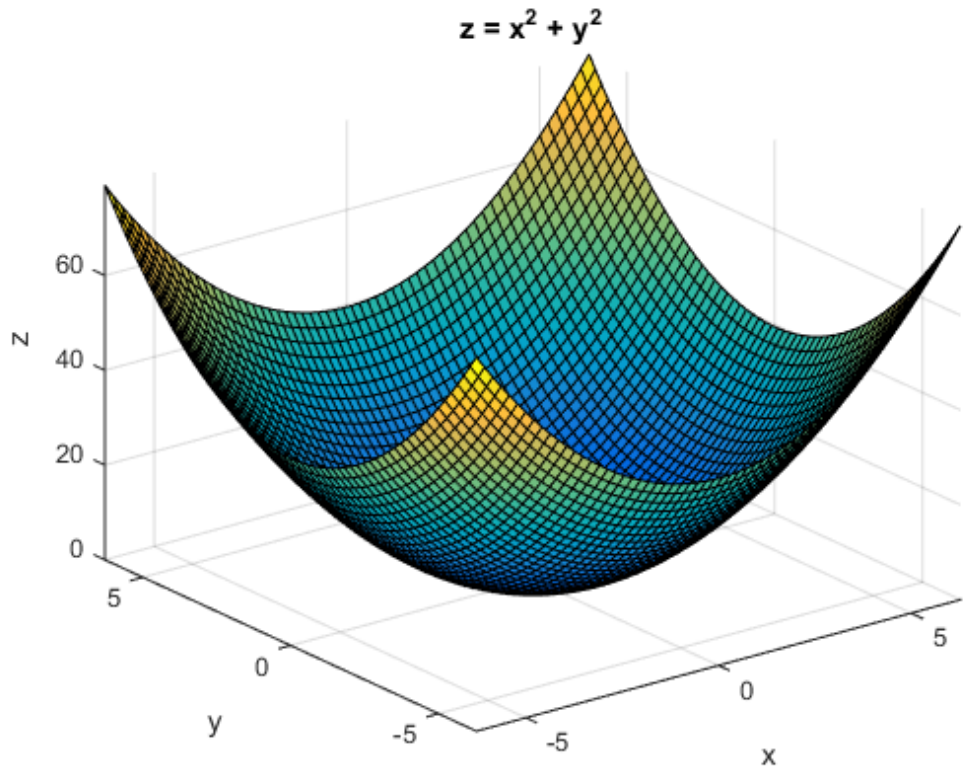
```
syms t
ezplot3(t^2*sin(10*t), t^2*cos(10*t), t)
```



Surface Plot

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

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


z = x^2 + y^2
z
60
40
20
0
5
0
-5
y
5
0
-5
x

Assumptions on Symbolic Objects

In this section...

Default Assumption

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

```
syms z
```

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

```
assumptions(z)
```

```
ans =
Empty sym: 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, rational numbers, and real numbers.

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

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

or more efficiently:

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

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

Check Existing Assumptions

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

```
assumptions(x)
```

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

```
assumptions
```

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

Delete Symbolic Objects and Their Assumptions

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

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

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

```
clear x
```

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

```
syms x real
clear x
syms x
solve(x^2 + 1 == 0, x)

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

```
syms x clear
```

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

1. To clear the assumption, enter

   ```
   syms x clear
   ```

2. To delete the symbolic object, enter

   ```
   clear x
   ```

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

Using Symbolic Math Toolbox Software

Differentiation

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

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

The command

```
diff(f)
```

differentiates `f` with respect to `x`:

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

As another example, let

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

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

```
diff(g)
```

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

To take the second derivative of `g`, enter

```
diff(g,2)
```

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

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

```
diff(diff(g))
```

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

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

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

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

returns

```
ans =
0
```

If you just enter

```
diff(5)
```

MATLAB returns

```
ans =
     []
```

because `5` is not a symbolic expression.

Derivatives of Expressions with Several Variables

To differentiate an expression that contains more than one symbolic variable, specify the variable that you want to differentiate with respect to. The `diff` command then calculates the partial derivative of the expression with respect to that variable. For example, given the symbolic expression

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

the command

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

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

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

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

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

which returns:

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

If you do not specify a variable to differentiate with respect to, MATLAB chooses a default variable. Basically, the default variable is the letter closest to x in the alphabet. See the complete set of rules in "Find a Default Symbolic Variable" on page 1-14. 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)`

f	diff(f)

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

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

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

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

which returns

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

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

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

which returns

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

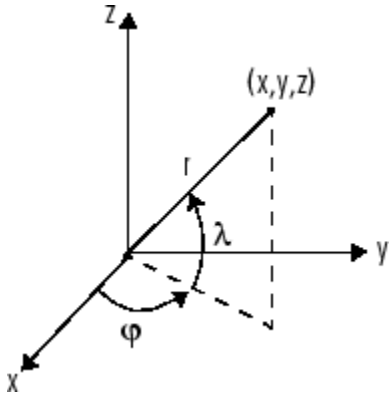
The command

```
diff(A)
```

returns

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

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



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

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

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

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

return the Jacobian

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

and the command

```
detJ = simplify(det(J))
```

returns

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

The arguments of the `jacobian` function can be column or row vectors. Moreover, since the determinant of the Jacobian is a rather complicated trigonometric expression, you can use `simplify` to make trigonometric substitutions and reductions (simplifications).

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

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

Limits

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

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

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

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

which return

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

and

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

which returns

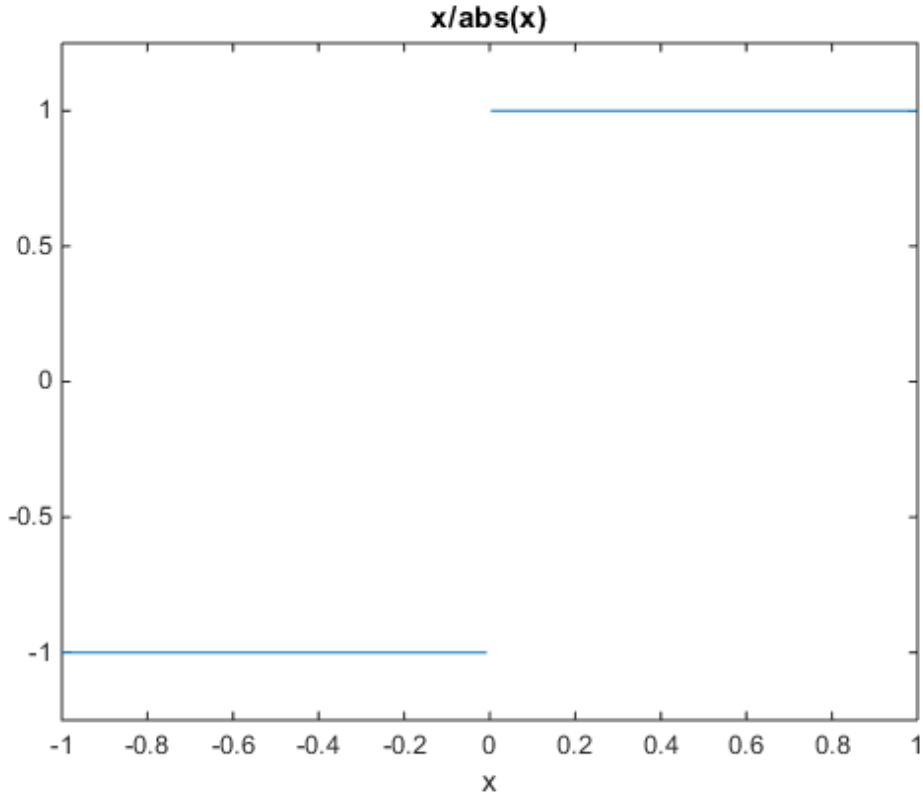
```
ans =
exp(x)
```

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

One-Sided Limits

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

```
syms x
ezplot(x/abs(x), -1, 1)
```



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
$\int x^n dx = \begin{cases} \log(x) & \text{if } n = -1 \\ \frac{x^{n+1}}{n+1} & \text{otherwise.} \end{cases}$	`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
```

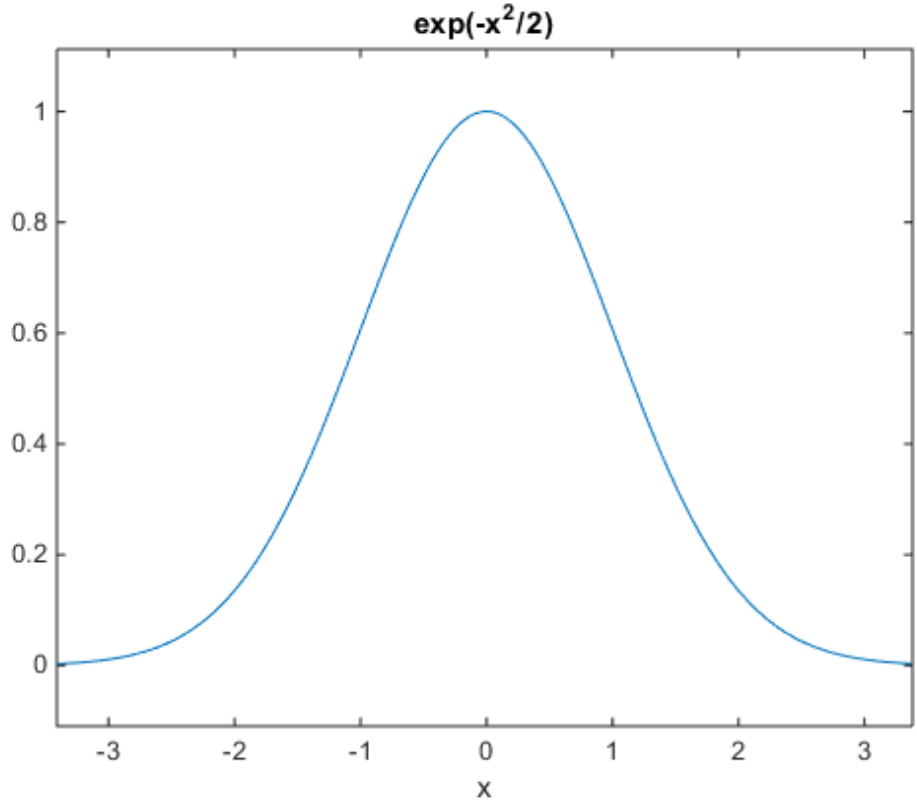
Integration with Real Parameters

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 $a = 1/2$, using the following commands:

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



However, if you try to calculate the integral

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

without assigning a value to a, MATLAB assumes that a represents a complex number, and therefore returns a piecewise answer that depends on the argument of a. If you are only interested in the case when a is a positive real number, use `assume` to set an assumption on `a`:

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

Now you can calculate the preceding integral using the commands

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

This returns

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

Integration with Complex Parameters

To calculate the integral

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

for complex values of `a`, enter

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

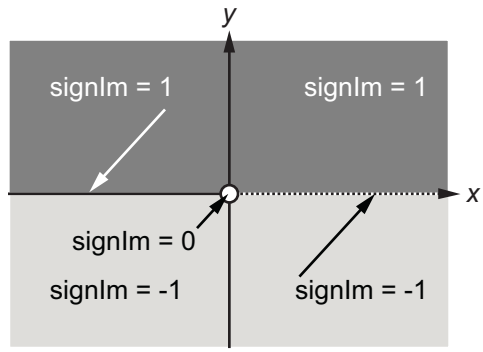
`syms` is used with the `clear` option to clear the all assumptions on `a`. For more information about symbolic variables and assumptions on them, see "Delete Symbolic Objects and Their Assumptions" on page 1-33.

The preceding commands produce the complex output

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

The function `signIm` is defined as:

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



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

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

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

```
double(g)
```

```
ans =
   1.5708 - 1.5708i
```

Symbolic Summation

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

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

sums to $\pi^2/6$, while the geometric series
$1 + x + x^2 + \ldots$

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

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

```
s1 =
pi^2/6

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

Taylor Series

The statements

```
syms x
f = 1/(5 + 4*cos(x));
T = taylor(f, 'Order', 8)
```

return

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

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

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

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

The command

```
pretty(T)
```

prints `T` in a format resembling typeset mathematics:

```
     6       4       2
 49 x     5 x     2 x     1
------ + ---- + ---- + -
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

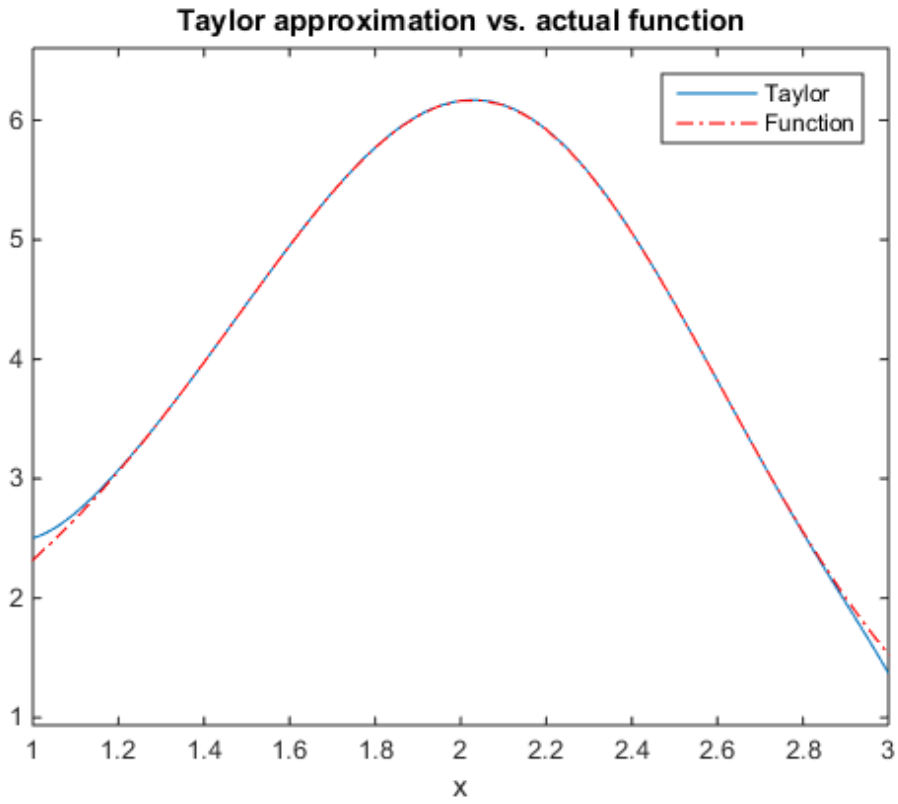
```
size(char(t))

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



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

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 =

laplace(y(t), t, s) - tau*(y(0) - s*laplace(y(t), t, s)) == a*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 =

laplace(y(t), t, s) + s*tau*laplace(y(t), t, s) == a*laplace(x(t), t, s)
```

To collect common terms, use `simplify`.

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

```
F =

(s*tau + 1)*laplace(y(t), t, s) == a*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 =

yS(s)*(s*tau + 1) == a*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) =

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 =

exp(-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 =

(a*exp(-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 =

(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 =

(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 + (9*a*exp(-s))/(2*tau - 2) - (a*exp(-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.

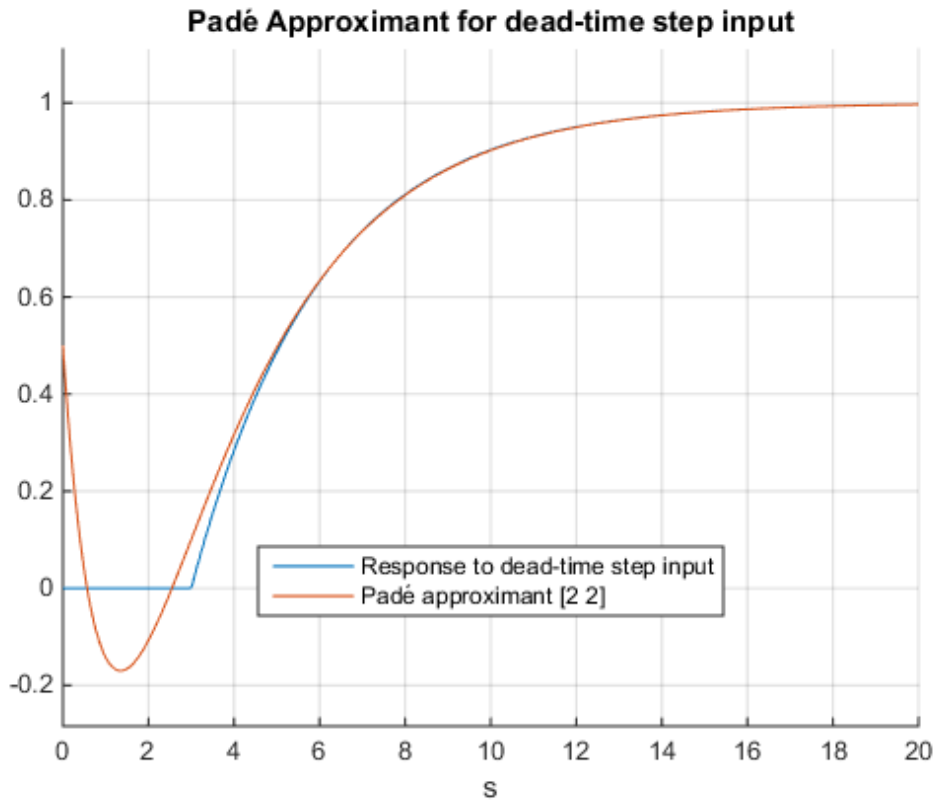
```
yPade22 = subs(yPade22,[a tau],[1 3])
```

```
yPade22 =
```

```
(9*exp(-s))/4 - (11*exp(-s/3))/4 + 1
```

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

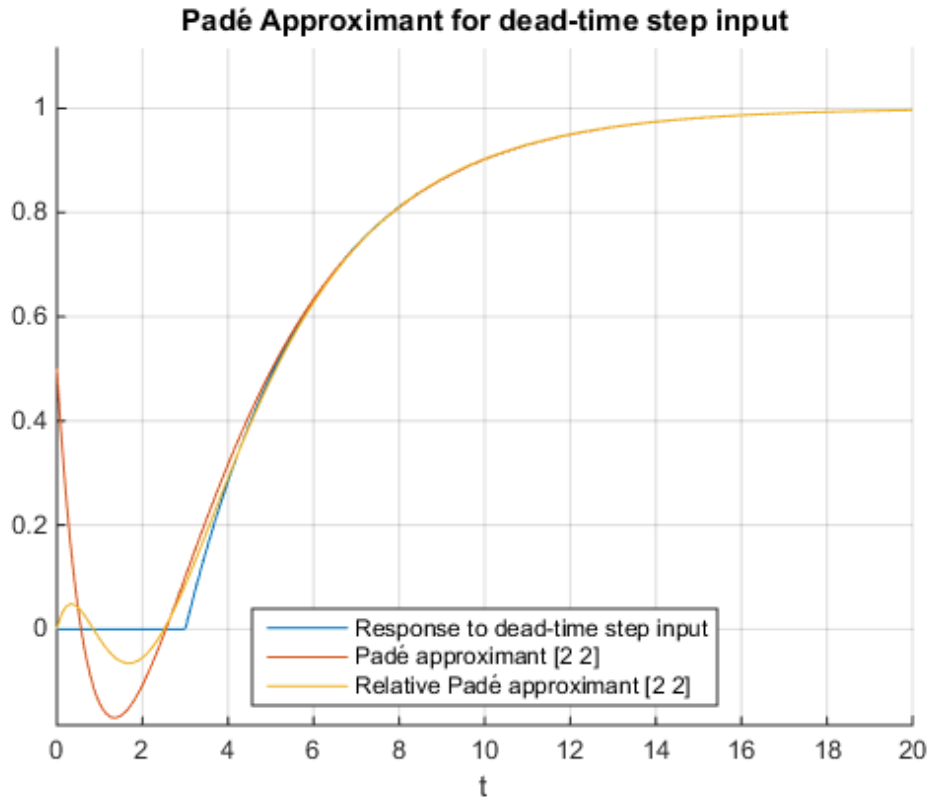
```
hold on
grid on
ezplot(y,[0 20])
ezplot(yPade22,[0 20])
title(['Pad' char(233) ' Approximant for dead-time step input'])
legend('Response to dead-time step input',...
       ['Pad' char(233) ' approximant [2 2]'],...
       'Location', 'Best')
```



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')
yPade22Rel = H(s)*stepPade22Rel
yPade22Rel = ilaplace(yPade22Rel)
yPade22Rel = subs(yPade22Rel,[a tau],[1 3])
ezplot(yPade22Rel,[0 20])
title(['Pad' char(233) ' Approximant for dead-time step input'])
legend('Response to dead-time step input',...
       ['Pad' char(233) ' approximant [2 2]'],...
       ['Relative Pad' char(233) ' approximant [2 2]'], 'Location', 'Best')
```

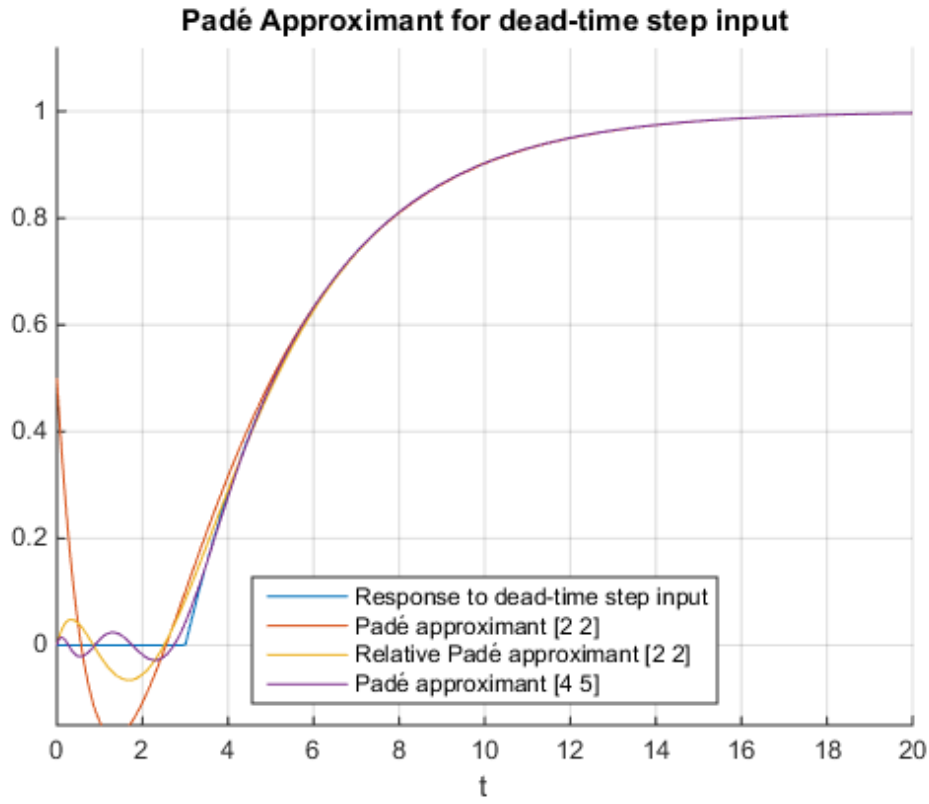
```
stepPade22Rel =

(3*s^2 - 6*s + 4)/(s*(3*s^2 + 6*s + 4))


yPade22Rel =

(a*(3*s^2 - 6*s + 4))/(s*(s*tau + 1)*(3*s^2 + 6*s + 4))


yPade22Rel =

a - (a*exp(-t/tau)*(4*tau^2 + 6*tau + 3))/(4*tau^2 - 6*tau + 3) + (12*a*tau*exp(-t)*(co


yPade22Rel =

(12*exp(-t)*(cos((3^(1/2)*t)/3) + (2*3^(1/2)*sin((3^(1/2)*t)/3))/3))/7 - (19*exp(-t/3)
```



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])
yPade45 = H(s)*stepPade45
yPade45 = subs(yPade45,[a tau],[1 3])
yPade45 = ilaplace(yPade45)
yPade45 = vpa(yPade45)
ezplot(yPade45,[0 20])
title(['Pad' char(233) ' Approximant for dead-time step input'])
legend('Response to dead-time step input',...
       ['Pad' char(233) ' approximant [2 2]'],...
       ['Relative Pad' char(233) ' approximant [2 2]'],...
```

```
        ['Pad' char(233) ' approximant [4 5]'], 'Location', 'Best')

stepPade45 =

(27*s^4 - 180*s^3 + 540*s^2 - 840*s + 560)/(s*(27*s^4 + 180*s^3 + 540*s^2 + 840*s + 560

yPade45 =

(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

yPade45 =

(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 + 8

yPade45 =

(294120*symsum((r3^2*exp(r3*t))/(12*(9*r3^3 + 45*r3^2 + 90*r3 + 70)), r3 in RootOf(s5^4

yPade45 =

exp(t*(- 1.930807068546914778929595950184 + 0.578156085956335834545982143280008*i))*(1.6
```



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

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

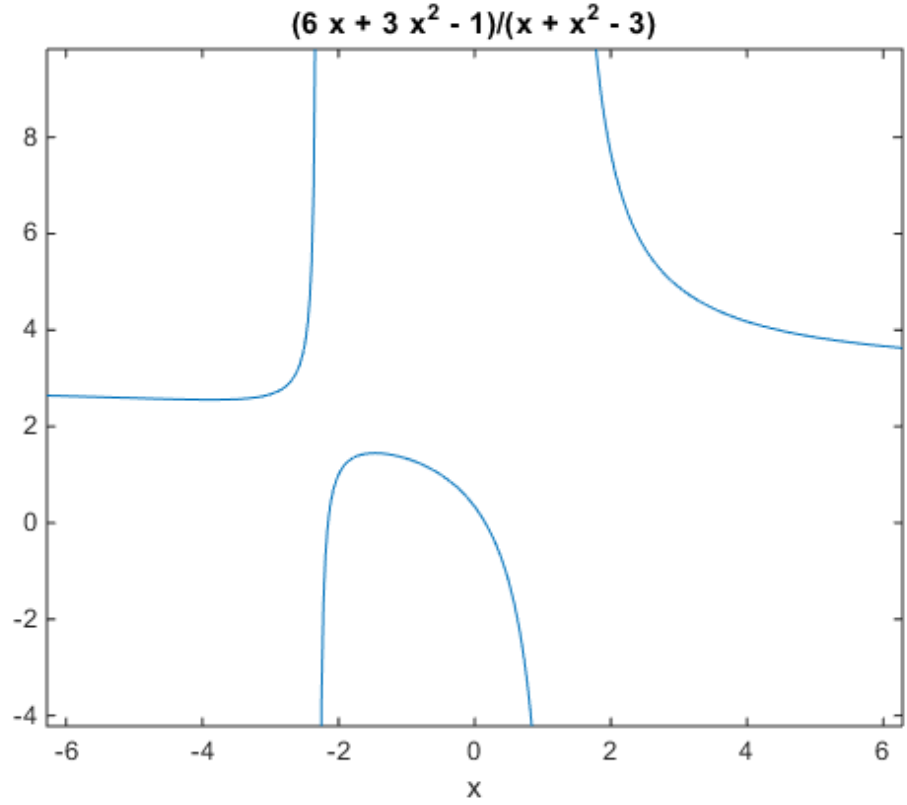
To create the function, enter the following commands:

```
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`

```
ezplot(f)
```



Find Asymptotes

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

```
limit(f, inf)
```

```
ans =
3
```

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

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

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

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

```
roots =
 - 13^(1/2)/2 - 1/2
   13^(1/2)/2 - 1/2
```

This tells you that vertical asymptotes are the lines

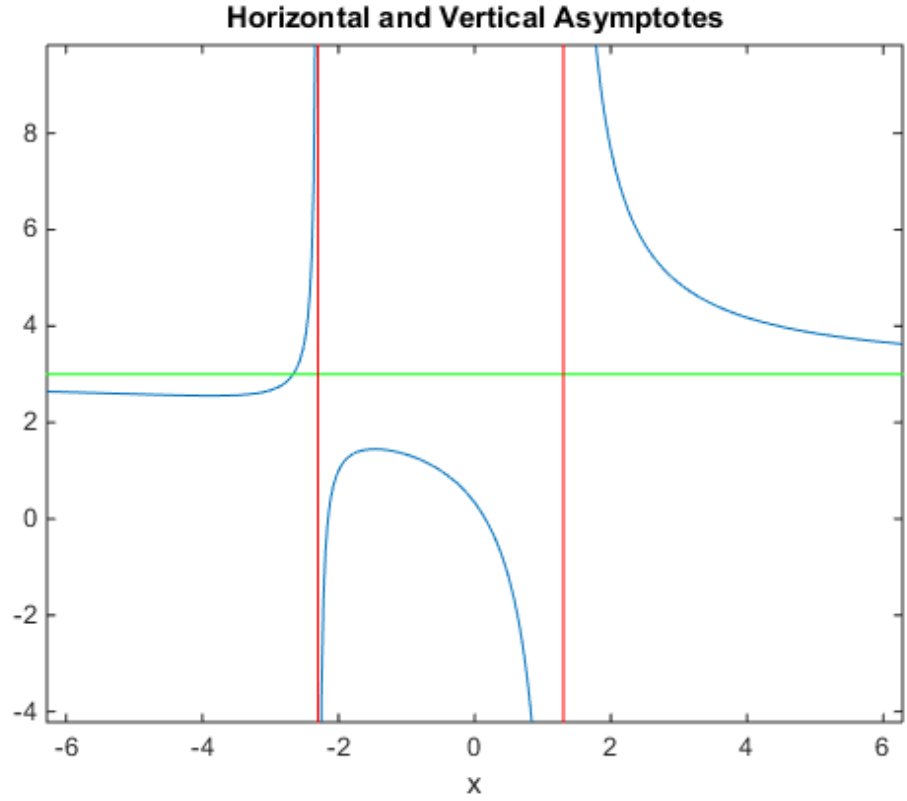
$$x = \frac{-1 + \sqrt{13}}{2},$$

and

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

You can plot the horizontal and vertical asymptotes with the following commands. Note that `roots` must be converted to `double` to use the `plot` command.

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



Find Maximum and Minimum

You can see from the graph that `f` has a local maximum somewhere between the points $x = -2$ and $x = 0$, and might have a local minimum between $x = -6$ and $x = -2$. To find the x-coordinates of the maximum and minimum, first take the derivative of `f`:

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

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

To simplify this expression, enter

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

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

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

```
pretty(f1)
```

which returns

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

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

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

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

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

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

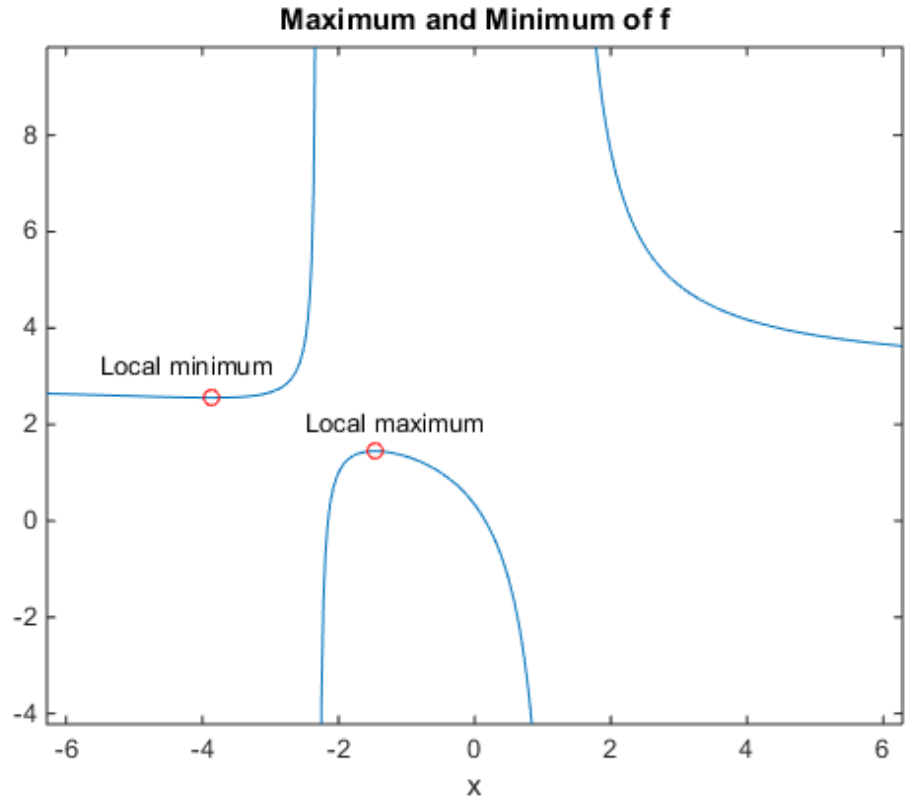
and a local maximum at

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

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

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

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



Find Inflection Point

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

```
f2 = diff(f1);
```

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

This returns

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

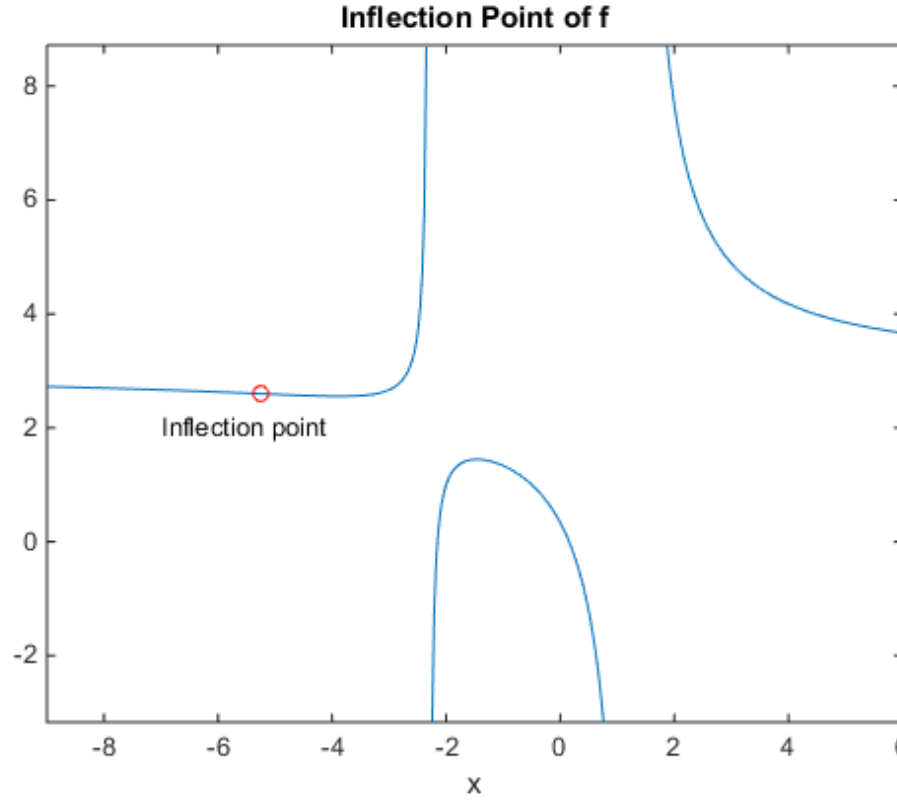
To see the symbolic expression for the inflection point, enter

```
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 `ezplot` extends the range of *x* values in the plot so that you see the inflection point more clearly, as shown in the following figure.

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


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

Simplifications

Here are three different symbolic expressions.

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

Here are their prettyprinted forms, generated by

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

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

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

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

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

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

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

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

collect

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

f	collect(f)

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

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

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

expand

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

f	expand(f)

`syms a x y`
`f = a*(x + y);`
	`expand(f)`
	
	`ans =`
	`a*x + a*y`

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

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

f	expand(f)

`syms a b`
`f = exp(a + b);`
	`expand(f)`
	
	`ans =`
	`exp(a)*exp(b)`

`syms x y`
`f = cos(x + y);`
	`expand(f)`
	
	`ans =`
	`cos(x)*cos(y) - sin(x)*sin(y)`

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

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

horner

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

f	horner(f)

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

`syms x`
`f = 1.1 + 2.2*x + 3.3*x`
	`horner(f)`
	
	`ans =`
	`x*((33*x)/10 + 11/5) + 11/10`

factor

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

```
factor(f)
```

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

f	factor(f)

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

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

`syms x`
`f = x^6 + 1;`
	`factor(f)`
	
	`ans =`
	`[ x^2 + 1, x^4 - x^2 + 1]`

As an aside at this point, `factor` can also factor symbolic objects containing integers. This is an alternative to using the MATLAB `factor` function. For example:

```
N = sym(902834092);
f = factor(N)
```

```
f =
[ 2, 2, 47, 379, 12671]
```

`factor` can also factor numbers larger than `flintmax` that the MATLAB `factor` cannot. For example, to perform prime factorization of a large number, place it in quotation marks to represent it accurately.

```
n = sym('41758540882408627201');
factor(n)
```

```
ans =
[ 479001599, 87178291199]
```

simplifyFraction

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

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

f	simplifyFraction(f)

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

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

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

simplify

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

f	simplify(f)

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

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

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

f	simplify(f)

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

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

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

You can also use the syntax `simplify(f, 'Steps', n)` where `n` is a positive integer that controls how many steps `simplify` takes. By default, `n = 1`. For example,

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

simplify(z)

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

simplify(z, 'Steps', 30)

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

Substitute with subexpr

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

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

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

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

Continuing with the example

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

returns

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

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

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

```
 sigma
```

which returns

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

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

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

```
u =
(a^3/27 + 1/4)^(1/2) - 1/2
r1 =
                                                  u^(1/3) - a/(3*u^(1/3))
 a/(6*u^(1/3)) - (3^(1/2)*(a/(3*u^(1/3)) + u^(1/3))*i)/2 - u^(1/3)/2
 (3^(1/2)*(a/(3*u^(1/3)) + u^(1/3))*i)/2 + a/(6*u^(1/3)) - u^(1/3)/2
```

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

Substitute with subs

Use this code to find the eigenvalues and eigenvectors of a circulant matrix A:

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

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

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

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

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

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

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

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

Simplify v:

```
v = simplify(v)
```

```
v =
```

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

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

```
S
```

```
S =
S
```

Assign this expression to `S`:

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

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

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

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

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

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

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

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

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

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

```
S
```

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

```
v
```

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

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

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

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

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

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

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

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

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

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

```
ans =
     1     2     3
```

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

```
S = subs(S)
```

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

```
v = subs(v)
```

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

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

Combine subs and double for Numeric Evaluations

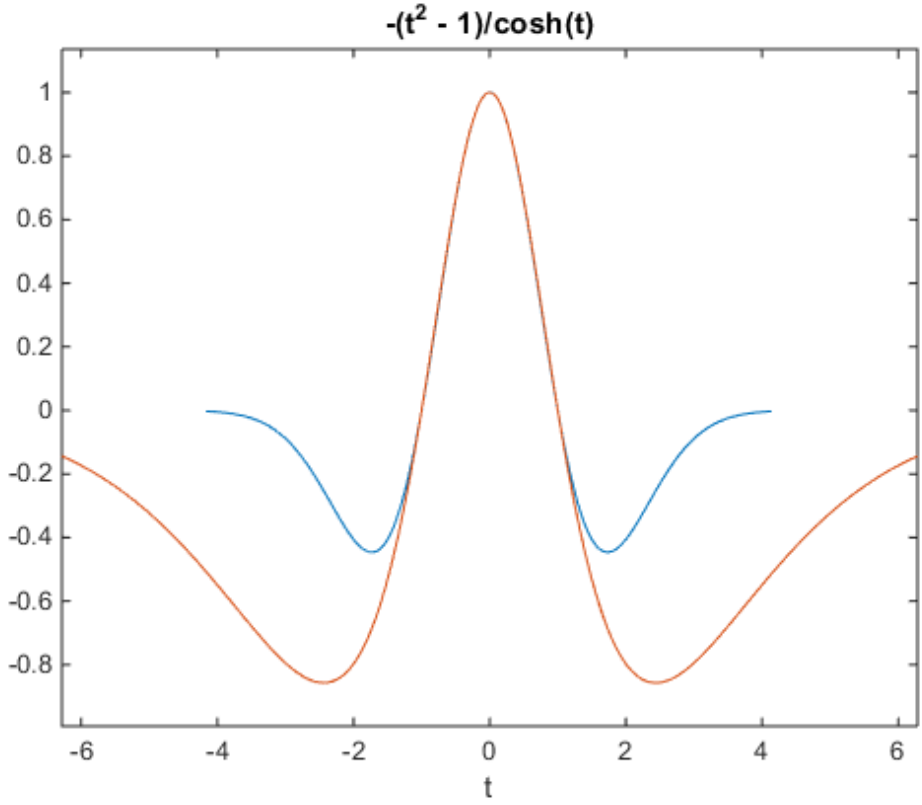
The `subs` command can be combined with `double` to evaluate a symbolic expression numerically. Suppose you have the following expressions

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

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

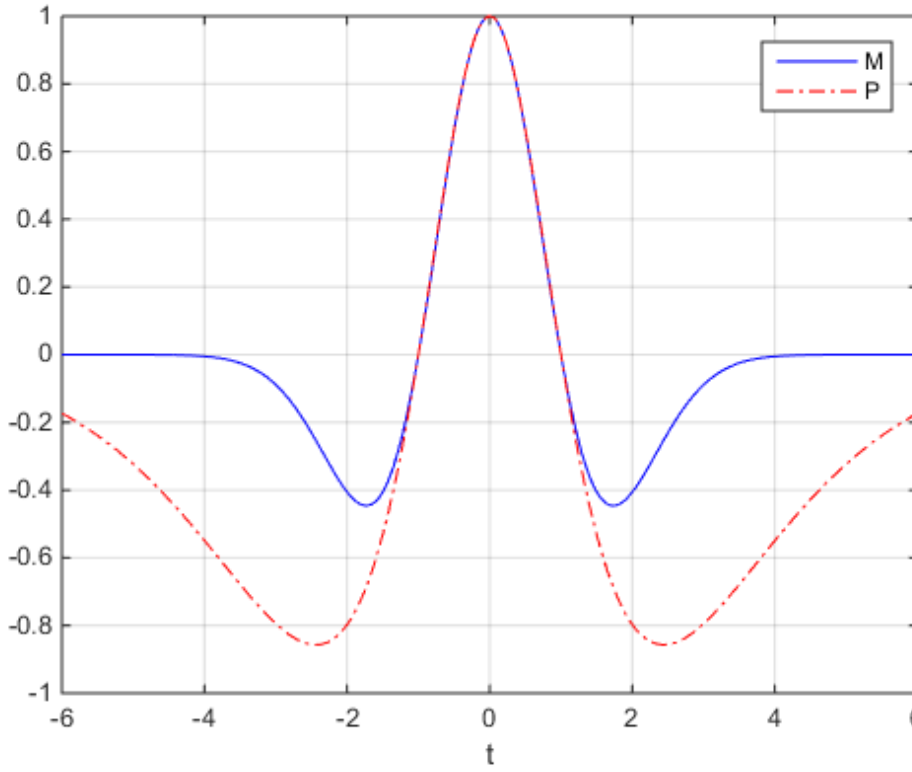
One approach is to type the following commands, but the resulting plot does not readily help you identify the curves.

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



Instead, combine `subs`, `double`, and `plot` to produce a multicolored graph that indicates the difference between `M` and `P`.

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


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

Choose the Arithmetic

Symbolic Math Toolbox operates on numbers using three types of arithmetic:

- Exact computations use exact symbolic numbers, such as `1/3`, `sqrt(2)`, or `pi`. By default, all computations involving symbolic objects (symbolic numbers, variables, expressions, or functions) use this arithmetic. The results are exact, so you do not need to worry about round-off errors. Exact computations are potentially the most expensive, in terms of both computer time and memory.
- Variable-precision arithmetic lets you control the number of significant digits. This is the recommended approach for numeric approximations in Symbolic Math Toolbox. To increase the accuracy, increase the number of digits. To speed up your computations and decrease memory usage, decrease the number of digits. You can set the number of digits to any value from 1 to 2^{29} using the `digits` function. To approximate a value using this kind of arithmetic, use the `vpa` function.
- Double-precision floating-point arithmetic uses the same precision as most numeric computations in MATLAB. To approximate a value with the double precision, use the `double` function. Internally, this arithmetic always uses the 64-bit (8-byte) floating-point representation provided by the particular computer hardware. The number of digits in printed output of MATLAB double quantities is determined by `format`. This arithmetic is recommended when you intend to use your computations on a computer that does not have a license for Symbolic Math Toolbox. Otherwise, exact symbolic numbers and variable-precision arithmetic are recommended.

Control Accuracy of Variable-Precision Computations

When you use variable-precision arithmetic, the accuracy of approximations depends on the value of the global variable `digits`. This variable determines the number of decimal digits for numerical computations. By default, the toolbox uses 32 significant decimal digits, which roughly corresponds to double-precision floating-point accuracy. For example, approximate this sum using the default number of digits:

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

```
ans =
0.83333333333333333333333333333333
```

Now, approximate the same sum with 5 and 50 decimal digits:

```
old = digits;

digits(5)
s5 = vpa(sym(1/3) + 1/2)

digits(50)
s50 = vpa(sym(1/3) + 1/2)

digits(old)
```

```
s5 =
0.83333

s50 =
0.83333333333333333333333333333333333333333333333333
```

To get the current `digits` setting, call `digits` without input arguments:

```
digits
```

```
Digits = 32
```

To change the accuracy for one operation without changing the current `digits` setting, use the `vpa` function with two input arguments. The second input argument must be an integer between 1 to 2^{29} specifying the accuracy of approximation. For example, approximate the value `pi` with 10 and 50 digits:

```
vpa(sym(pi), 10)
vpa(sym(pi), 50)
```

```
digits

ans =
3.141592654

ans =
3.14159265358979323846264338327950288419716939937 51

Digits = 32
```

Note that `digits` and `vpa` control the number of *significant* decimal digits. Thus, when you approximate the value `1/111` with 4-digit accuracy, the result has six digits after the decimal point. The first two of them are zeros:

```
vpa(sym(1/111), 4)

ans =
0.009009
```

Recognize and Avoid Round-Off Errors

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

- When possible, use symbolic computations. Switch to floating-point arithmetic only if you cannot obtain symbolic results.
- Numeric computations are sensitive to `digits`, which determines the numeric working precision. Increase the precision of numeric computations, and check if the result changes significantly.
- Compute the result symbolically, and then approximate it numerically. Also, compute the result using the floating-point parameters. Significant difference in these two results indicates that one or both approximations are incorrect.
- Plot the function or expression that you approximate.

Use Symbolic Computations When Possible

Performing computations symbolically 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))
```

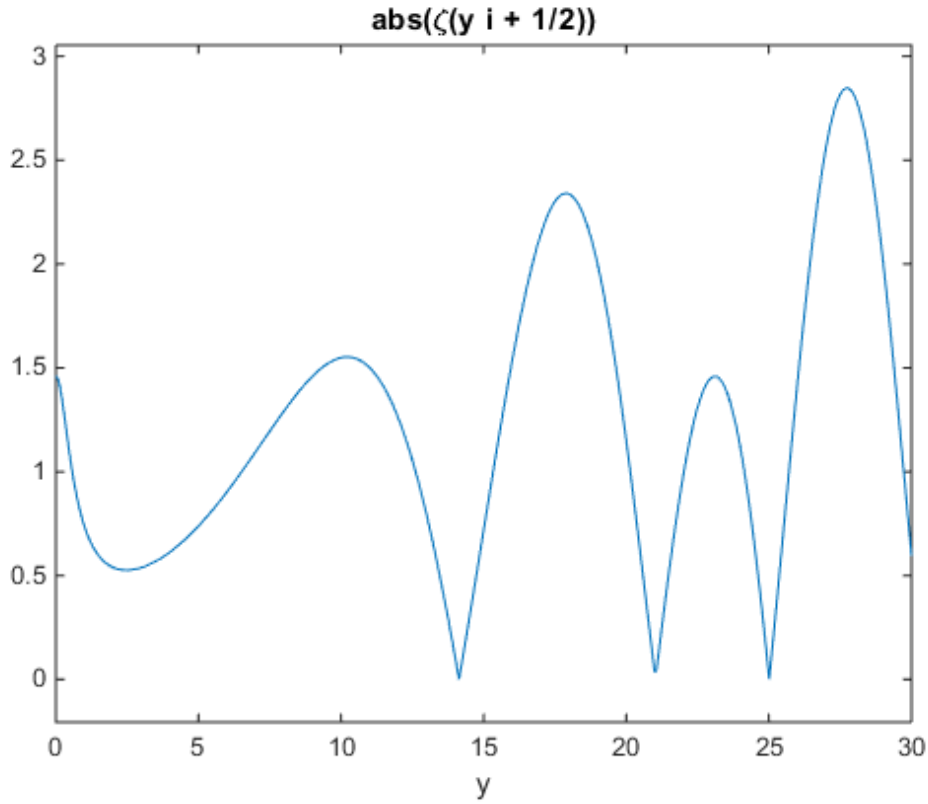
```
ans =
1/2
```

```
ans =
     1
```

Increase Precision

The Riemann hypothesis states that all nontrivial zeros of the Riemann Zeta function $\zeta(z)$ have the same real part $\#(z) = 1/2$. 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 $|\zeta(1/2 + iy)|$.

```
syms y
ezplot(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 + 0.000000000499989207306345*i
```

For further computations, restore the default number of digits:

```
digits(old)
```

Approximate Parameters and Approximate 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 +...
 1829478811390512975000/pi^12 - 557206975126367666100000/pi^14...
 + 1017414868369523902090312 5/pi^16 - 1060253389142977540073062500/pi^18...
 + 573066956831779360409490 28125/pi^20 - 133187103010706033170268875000/pi^22...
 + 849067781693250961460464 1578125/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 =
0.00000000000000000000006900145606917284206886223284139647379659723376116 1
```

Plot the Function or Expression

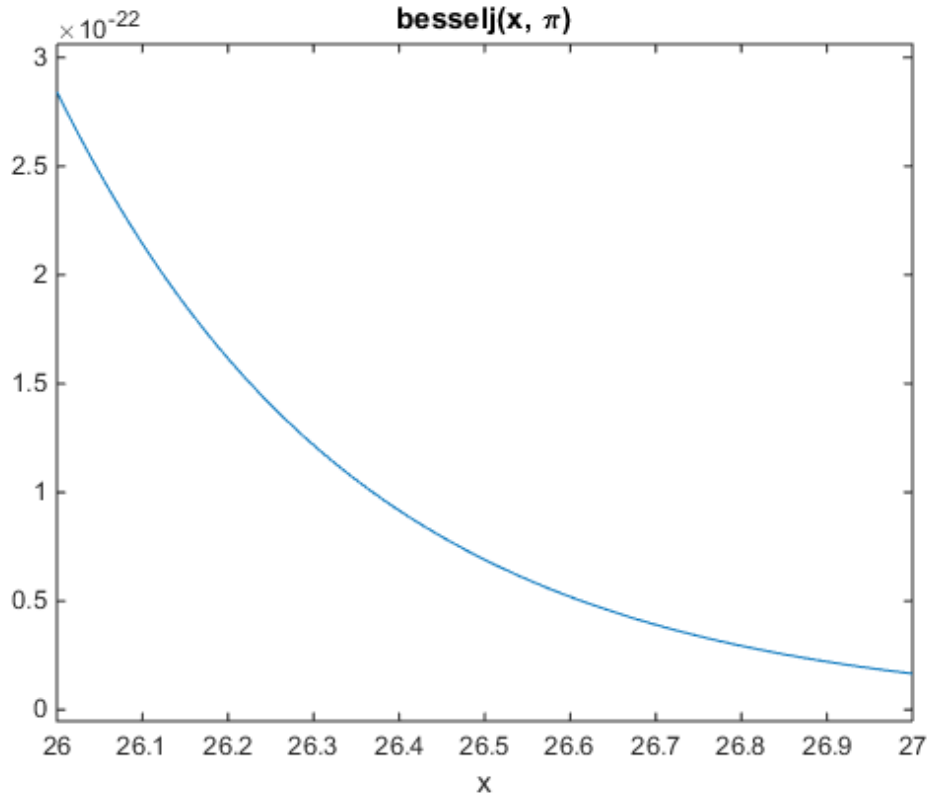
Plotting the results can help you recognize incorrect approximations. For example, the numeric approximation of the following Bessel function returns:

```
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
ezplot(besselj(x, sym(pi)), 26, 27)
```



Improve Performance of Numeric Computations

When you perform numeric computations, one of the most difficult tasks is finding the right balance between accuracy of computations and code performance. If you have Symbolic Math Toolbox, then the best approach is to use variable-precision arithmetic. Variable-precision arithmetic provides flexibility in terms of accuracy and performance, letting you choose the appropriate number of digits for your particular task. You can always convert the final results of your variable-precision computations to the `double` format, if that is needed for further tasks.

While increasing the number of significant decimal digits lets you perform numeric computations with better accuracy, decreasing that number might help you get the results in a reasonable amount of time. For example, compute the Riemann Zeta function of the elements of the `101`-by-`301` matrix `C`:

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

Computing the Zeta function of these elements directly takes a long time:

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

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

Computing the Zeta function of the same elements with 10-digit precision is much faster:

```
digits(10)
tic
D = zeta(vpa(C));
toc
```

```
Elapsed time is 113.792543 seconds.
```

For larger matrices, the difference in computation time can be more significant. For example, for the `1001`-by-`301` matrix `C`:

```
[X,Y] = meshgrid((0:0.00025:.75),(5:-0.005:0));
C = X + Y*i;
```

executing `D = zeta(vpa(C))` with 10-digit precision finishes in several minutes, while executing `D = zeta(C)` takes more than an hour.

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

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

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

Symbolic Math Toolbox commands

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

return

```
G =
```

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

You can simplify this expression using `simplify`:

```
G = simplify(G)
```

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

Next, the command

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

produces

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

You can rewrite `g` in terms of exponents:

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

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

Jordan Canonical Form

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

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

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

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

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

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

For example, let

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

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

Then

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

produces

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

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

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

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

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

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

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

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

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

Singular Value Decomposition

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 =
[  0.532543310273353384706833683602O4,  0.765768959488020529893040921799 52,...
                                        0.360548919520962147911898877283 53]
[ -0.825256896508494632225028536722 24,  0.375149652839654519931713386050 42,...
                                        0.422153754856514895224880319173 64]
[  0.188012439610432818399171141717 42, -0.522360640418974394474297842572 24,...
                                        0.831739552920751921784218743314 06]

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

V =
[  0.188012439610432818399171141717 42,  0.522360640418974394474297842572 24,...
                                        0.831739552920751921784218743314 06]
[ -0.825256896508494632225028536722 24, -0.375149652839654519931713386050 42,...
                                        0.422153754856514895224880319173 64]
[  0.532543310273353384706833683602O4, -0.765768959488020529893040921799 52,...
                                        0.360548919520962147911898877283 53]
```

Solve an 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 a Numeric or Symbolic Solver".

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 =
  -3.92699081698724154807830422909 94
 -0.78539816339744830961566084581988
   2.3561944901923449288469825374596
   5.4977871437821381673096259207391
```

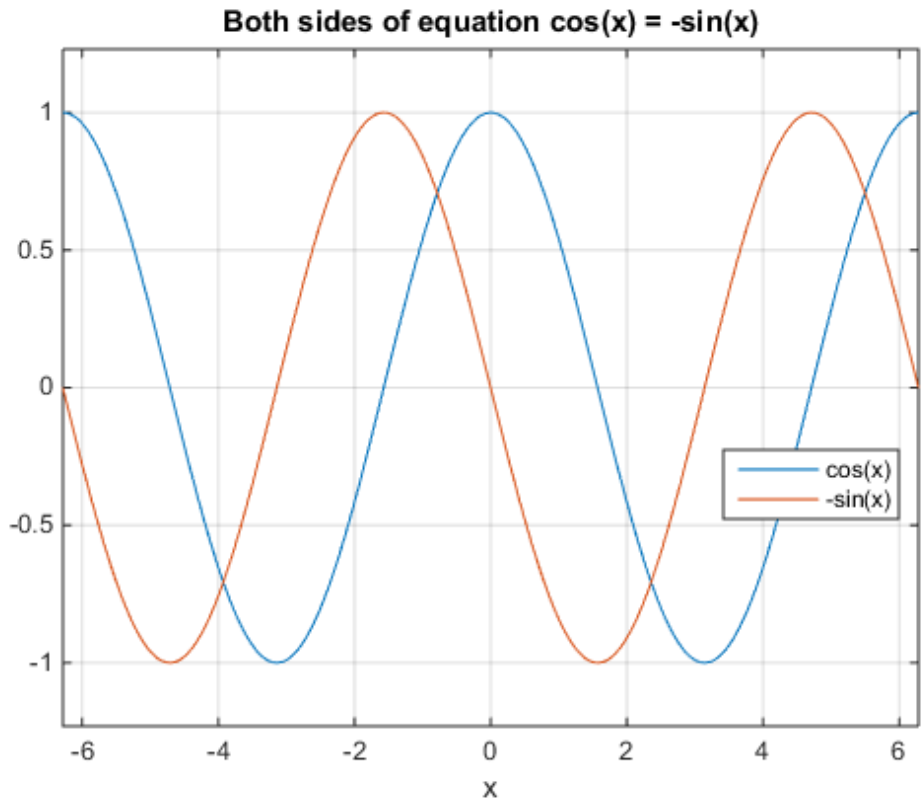
Visualize and Plot Solutions Returned by solve

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 `ezplot` and `scatter`.

Plot both sides of equation `cos(x) == -sin(x)`.

```
ezplot(cos(x))
hold on
grid on
ezplot(-sin(x))
title('Both sides of equation cos(x) = -sin(x)')
```

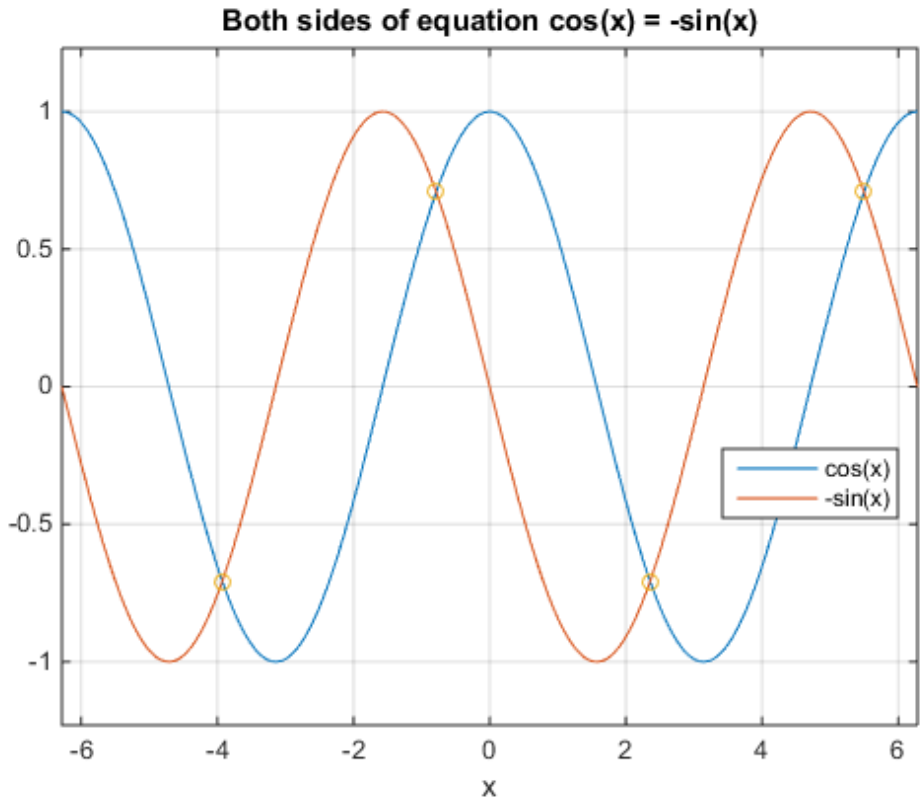
```
legend('cos(x)','-sin(x)','Location','Best')
```



Calculate the values of the functions at the values of `x`, and superimpose the solutions as points using `scatter`.

```
yvalues = cos(xvalues)
scatter(xvalues, yvalues)
```

```
yvalues =

  -0.707106781186547524400844362104 85
   0.707106781186547524400844362104 85
  -0.707106781186547524400844362104 85
   0.707106781186547524400844362104 85
```



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, "Resolve Complicated Solutions or Stuck Solver".

Select a 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 “Control Accuracy of Variable-Precision Computations”.

Solve a 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 a Numeric or Symbolic Solver".

In this section...

Handle the Output of solve

Suppose you have the system

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

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

```
syms x y alpha
```

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

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

The call returns the following.

```
solx =
     0
```

```
 alpha
soly =
 -2*alpha
        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 == alpha)
```

Four distinct solutions are produced.

```
solx =
 alpha/2 - (alpha^2 - 2)^(1/2)/2
 alpha/2 - (alpha^2 + 2)^(1/2)/2
 alpha/2 + (alpha^2 - 2)^(1/2)/2
 alpha/2 + (alpha^2 + 2)^(1/2)/2
soly =
 - alpha - (alpha^2 - 2)^(1/2)
 - alpha - (alpha^2 + 2)^(1/2)
   (alpha^2 - 2)^(1/2) - alpha
   (alpha^2 + 2)^(1/2) - alpha
```

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

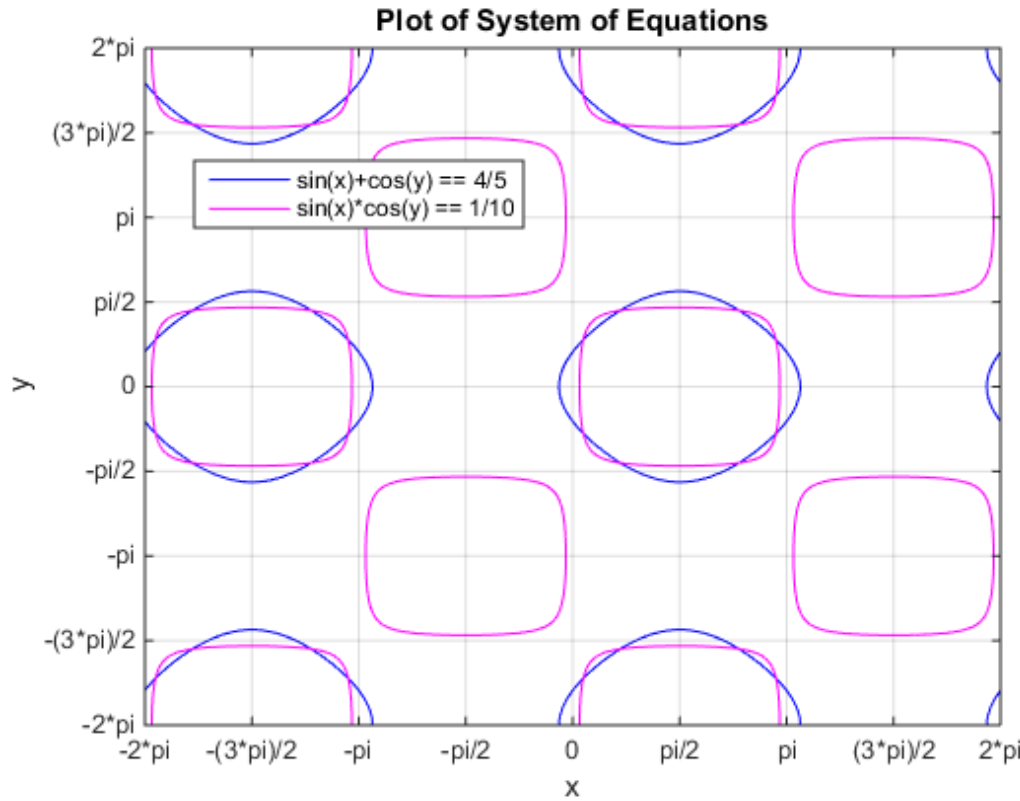
Consider the following system of equations:

$$\sin(x)+\cos(y)=\frac{4}{5}$$
$$\sin(x)\cos(y)=\frac{1}{10}$$

Visualize the system of equations using `ezplot`. 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 strings. 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;
h = ezplot(eqn1);
h.LineColor = 'blue';
hold on
grid on
```

```
g = ezplot(eqn2);
g.LineColor = 'magenta';
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')
```



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 =
             x: [2x1 sym]
             y: [2x1 sym]
    parameters: [1x2 sym]
    conditions: [2x1 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.

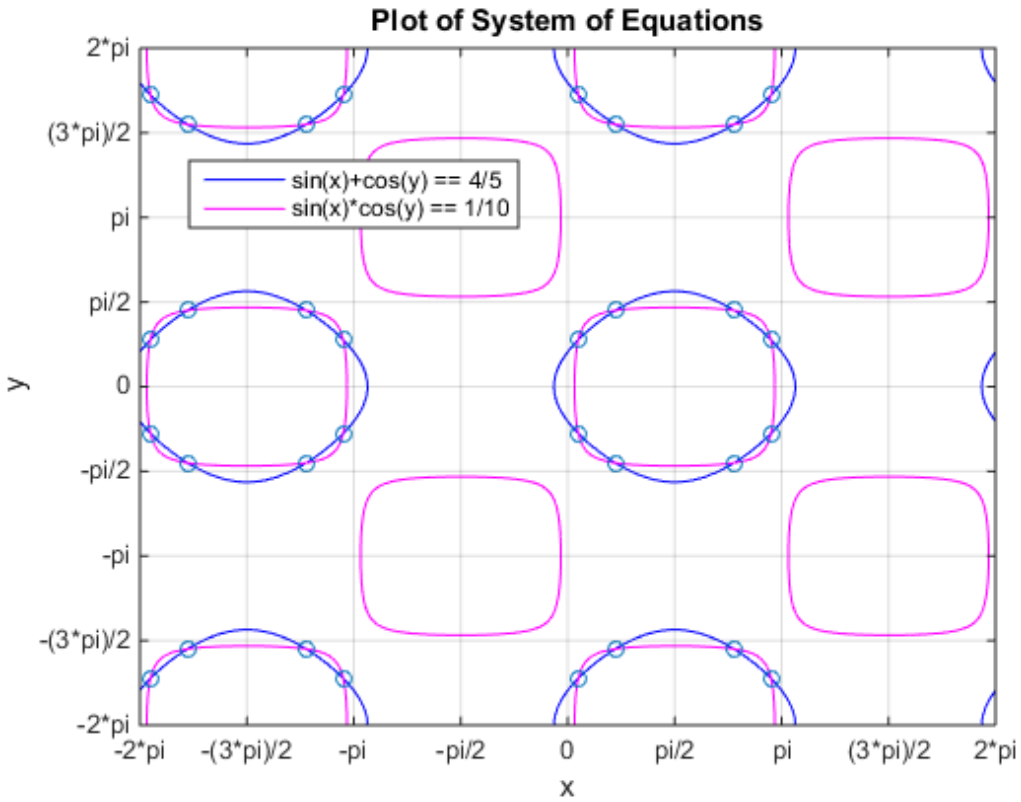
```
S.x
S.y
S.parameters
S.conditions

ans =
 z
 z
ans =
 z1
 z1
ans =
[ z, z1]
ans =
 (in((z - asin(6^(1/2)/10 + 2/5))/(2*pi), 'integer') |...
 in((z - pi + asin(6^(1/2)/10 + 2/5))/(2*pi), 'integer')) &...
 (in((z1 - acos(2/5 - 6^(1/2)/10))/(2*pi), 'integer') |...
 in((z1 + acos(2/5 - 6^(1/2)/10))/(2*pi), 'integer'))
 (in((z - asin(2/5 - 6^(1/2)/10))/(2*pi), 'integer') |...
 in((z - pi + asin(2/5 - 6^(1/2)/10))/(2*pi), 'integer')) &...
 (in((z1 - acos(6^(1/2)/10 + 2/5))/(2*pi), 'integer') |...
 in((z1 + acos(6^(1/2)/10 + 2/5))/(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)
```



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))
```

Solve the first solution of `x` for the parameter `z`.

```
solz(1,:) = solve(S.x(1)>-2*pi, S.x(1)<2*pi, S.parameters(1))
```

```
solz =
[ 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)]
```

Similarly, solve the first solution to `y` for `z1`.

```
solz1(1,:) = solve(S.y(1)>-2*pi, S.y(1)<2*pi, S.parameters(2))
```

```
solz1 =
[ 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)]
```

Clear the assumptions set by `S.conditions(1)` using `sym`. Call `asumptions` to check that the assumptions are cleared.

```
sym(S.parameters,'clear')
assumptions
```

```
ans =
[ z, z1]
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 `z` and `z1`.

```
solz(2,:) = solve(S.x(2)>-2*pi, S.x(2)<2*pi, S.parameters(1))
```

```
solz1(2,:) = solve(S.y(2)>-2*pi, S.y(2)<2*pi, S.parameters(2))
```

```
solz =
[ 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)]
[ asin(2/5 - 6^(1/2)/10), pi - asin(2/5 - 6^(1/2)/10),...
 asin(2/5 - 6^(1/2)/10) - 2*pi, - pi - asin(2/5 - 6^(1/2)/10)]
solz1 =
[ 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)]
[ 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)]
```

The first rows of `solz` and `solz1` 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 `z` and `z1`, use `subs` to substitute for `z` and `z1` in `S.x` and `S.y`.

```
solx(1,:) = subs(S.x(1), S.parameters(1), solz(1,:));
solx(2,:) = subs(S.x(2), S.parameters(1), solz(2,:))
soly(1,:) = subs(S.y(1), S.parameters(2), solz1(1,:));
soly(2,:) = subs(S.y(2), S.parameters(2), solz1(2,:))
```

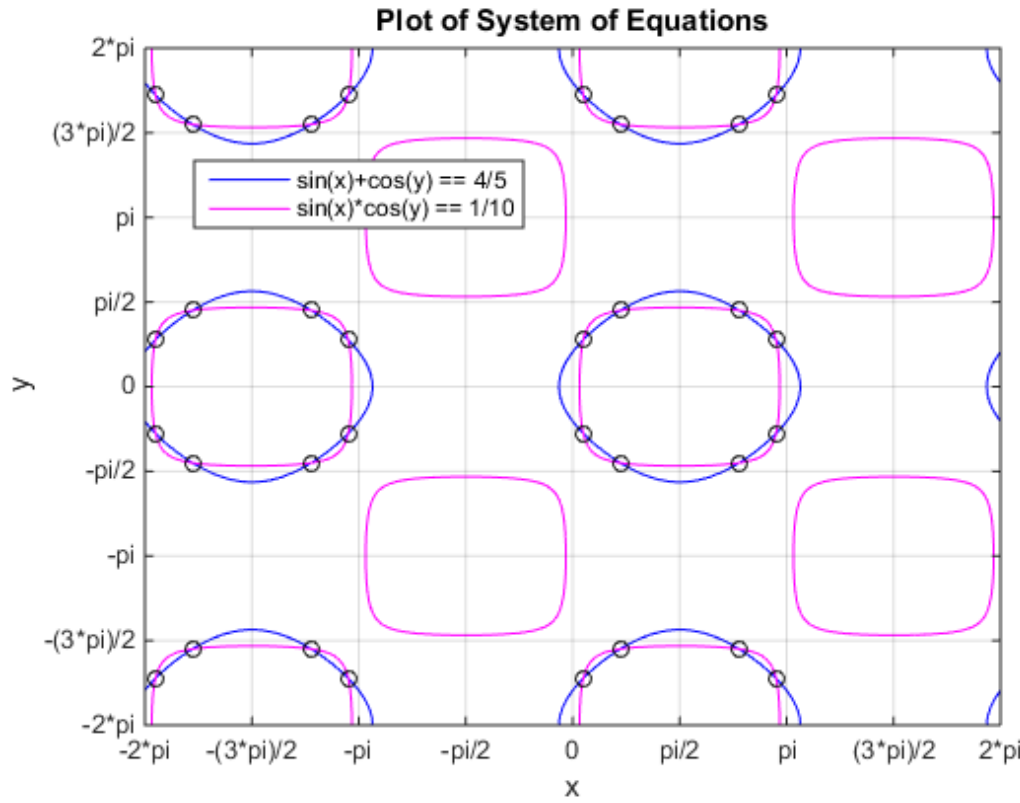
```
solx =
[ asin(6^(1/2)/10 + 2/5), pi - asin(6^(1/2)/10 + 2/5),...
 asin(6^(1/2)/10 + 2/5) - 2*pi, - pi - asin(6^(1/2)/10 + 2/5)]
[ asin(2/5 - 6^(1/2)/10), pi - asin(2/5 - 6^(1/2)/10),...
 asin(2/5 - 6^(1/2)/10) - 2*pi, - pi - asin(2/5 - 6^(1/2)/10)]
soly =
[ acos(2/5 - 6^(1/2)/10), acos(2/5 - 6^(1/2)/10) - 2*pi,...
 -acos(2/5 - 6^(1/2)/10), 2*pi - acos(2/5 - 6^(1/2)/10)]
[ 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)]
```

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), 'black')
```

```
        scatter(solx(2,i), soly(2,j), 'black')
    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 =
[ 0.70095651347102524787213653614929, 2.44063614011876799059050684713302,...
 -5.58222879370856122905315023040 97, -3.84254916706081848633477991942 88]
[ 0.15567910349205249963259154265761, 2.98591355009774073883005184062 19,...
 -6.12750620368753397729269522390 14, -3.29727175708184573809523492593 71]
ans =
[  1.41511722330284411959873014898 21, -4.86806808387674235732655661756 69,...
  -1.41511722330284411959873014898 21, 4.86806808387674235732655661756 69]
[ 0.86983981332387137135918515549046, -5.41334549385571510556610161106 85,...
 -0.86983981332387137135918515549046, 5.41334549385571510556610161106 85]
```

Simplify Complicated Results and Improve Performance

If results look complicated, `solve` is stuck, or if you want to improve performance, see, “Resolve Complicated Solutions or Stuck Solver”.

Resolve Complicated Solutions or Stuck Solver

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

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)*i)/4 - 5^(1/2)/4 - 1/4
   (2^(1/2)*(5 - 5^(1/2))^(1/2)*i)/4 - 5^(1/2)/4 - 1/4
   5^(1/2)/4 - (2^(1/2)*(5^(1/2) + 5)^(1/2)*i)/4 - 1/4
   5^(1/2)/4 + (2^(1/2)*(5^(1/2) + 5)^(1/2)*i)/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.

```
solve(x^(5/2) + 1/x^(5/2) == 1, x)

ans =
                                                                1/(1/2 - (3^(1/2)*i)/2)^(2/5)
                                                                1/((3^(1/2)*i)/2 + 1/2)^(2/5)
 -(5^(1/2)/4 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*i)/4 + 1/4)/(3^(1/2)*(-i/2) + 1/2)^(2/5)
 -((2^(1/2)*(5 - 5^(1/2))^(1/2)*i)/4 + 5^(1/2)/4 + 1/4)/(3^(1/2)*(-i/2) + 1/2)^(2/5)
  -(5^(1/2)/4 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*i)/4 + 1/4)/(3^(1/2)*(i/2) + 1/2)^(2/5)
  -((2^(1/2)*(5 - 5^(1/2))^(1/2)*i)/4 + 5^(1/2)/4 + 1/4)/(3^(1/2)*(i/2) + 1/2)^(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)*i)/2)^(2/5)
 1/((3^(1/2)*i)/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.

```
S = solve(log(1/x) + log(sym(5)) == 1/x + log(sym(3)), x)
```

```
S =
-1/lambertw(0, -exp(log(3) - log(5)))
```

Call `simplify` to simplify the result.

```
simplify(S)
```

```
ans =
exp(log(5/3) + lambertw(0, -3/5))
```

Tips

- To represent a number exactly, use `sym`. 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(133333/5)`, and `sym('133333/5')`.

  ```
  sym(13/5)
  sym(133333/5)
  sym('133333/5')
  ```

  ```
  ans =
  13/5
  ans =
  3665029596656435/137438953472
  ans =
  133333/5
  ```

  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 a 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

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1\\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2\\
&\ldots\\
&a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m
\end{aligned}
$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix,

$$
A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}
$$

and $\vec{b}$ is the vector containing the right sides of equations,

$$
\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}
$$

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.

$$2x + y + z = 2$$
$$-x + y - z = 3$$
$$x + 2y + 3z = -10$$

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;
```

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 the equations and a comma separated list 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.

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 a Numeric or Symbolic Solver". 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)
```

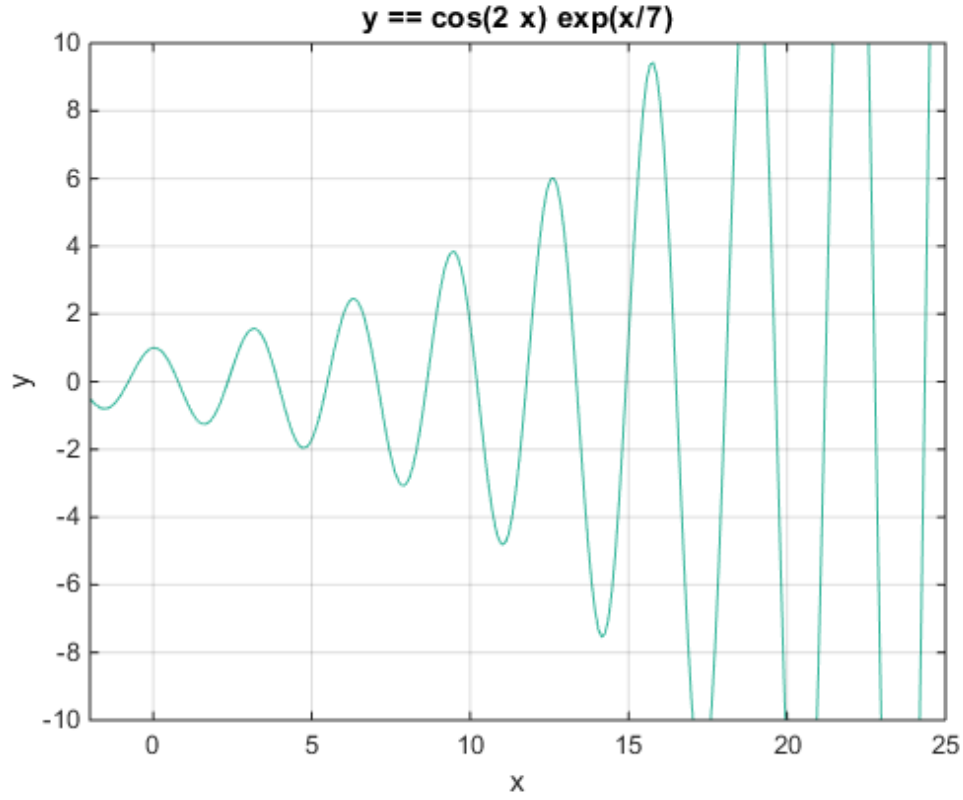
```
sol =
                                                 1.024024075905370294144831656333
 - 0.229747952261181639630985706107 24 + 0.967746155767440310739990106951 71*i
 - 0.229747952261181639630985706107 24 - 0.967746155767440310739990106951 71*i
 - 0.880806200517621496392056722983 26 - 0.504340588401275843763318065924 05*i
   0.765208781492784655617293267590 3 + 0.831873314310497132183672393171 21*i
 - 0.880806200517621496392056722983 26 + 0.504340588401275843763318065924 05*i
   0.765208781492784655617293267590 3 - 0.831873314310497132183672393171 21*i
```

`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 y
h = ezplot(y == exp(x/7)*cos(2*x),[-2, 25, -10, 10]);
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.

```
for i = 1:3
  vpasolve(exp(-x/20)*cos(2*x),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 the `vpasolve` page.

```
for i = 1:3
  vpasolve(exp(-x/20)*cos(2*x),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(exp(-x/20)*cos(2*x),x,10)
vpasolve(exp(-x/20)*cos(2*x),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(exp(-x/20)*cos(2*x),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(exp(-x/20)*cos(2*x),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(exp(-x/20)*cos(2*x),x,15,'random',true)
```

```
Warning: All variables have a starting value for the numeric...
 search. The option 'random' has no effect in this case.
> In sym.vpasolve at 166
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.

```
vpasolve(exp(x/7)*cos(2*x))
digitsOld = digits;
digits(64)
vpasolve(exp(x/7)*cos(2*x))
```

```
digits(digitsOld)

ans =
-7.068583470577034786540947612378881489443631148593988097193625333
ans =
-7.068583470577034786540947612378881489443631148593988097193625333
```

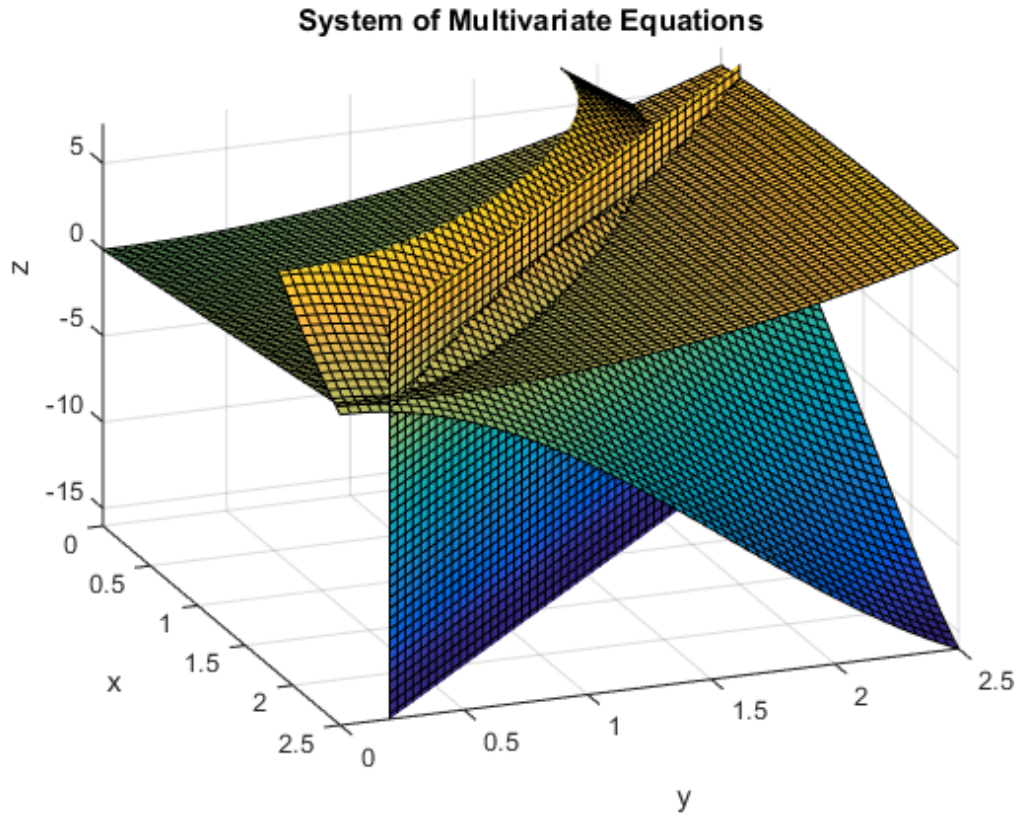
Solve Multivariate Equations Using Search Ranges

Consider the following system of equations.

$$10\left(\cos\left(x\right)+\cos\left(y\right)\right)=x+y^2-0.1x^2y$$

A plot of the equations for $0 \le x \le 3$ and $0 \le y \le 3$ shows that the three surfaces intersect in two points. To better visualize the plot, use `view`. To scale the colormap values, use `caxis`.

```
syms x y z
exp1 = 10*(cos(x)+cos(y));
exp2 = x+y^2-0.1*x^2*y;
exp3 = y+x-2.7;
ezsurf(exp1,[0, 2.5])
hold on
grid on
ezsurf(exp2,[0, 2.5])
x1 = @(s,t) s;
y1 = @(s,t) 2.7-s;
z1 = @(s,t) t;
ezsurf(x1,y1,z1,[0,2.5,-20,10])
title('System of Multivariate Equations')
view(69, 28)
caxis([-15 10])
```

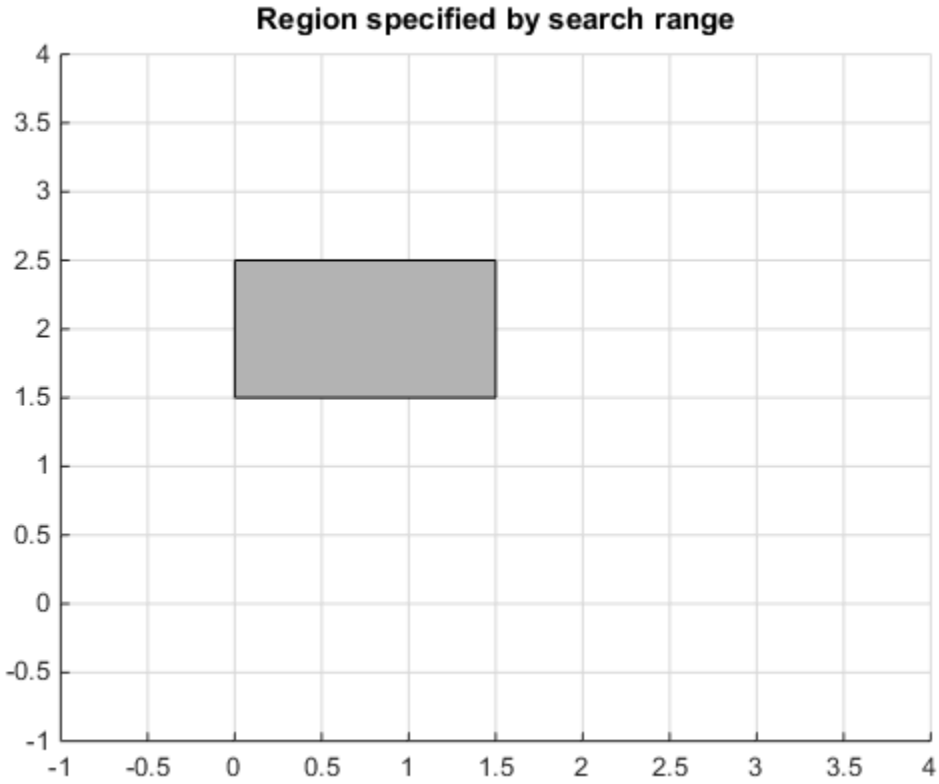


Use `vpasolve` to find a point where the surfaces intersect. The function `vpasolve` returns a structure. To access the solution, index into the structure.

```
sol = vpasolve([z == exp1, z == exp2, exp3 == 0]);
[sol.x sol.y sol.z]
```

```
ans =

[ 2.369747722454797920910133716017 4, 0.330252277545202079089866283982 61, 2.293354376823
```

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 $1.5 \le y \le 2.5$.

```
sol = vpasolve([z == exp1, z == exp2, 0 == exp3],[x y z],[0 1.5; 1.5 2.5;NaN NaN]);
[sol.x sol.y sol.z]
```

```
ans =
```

```
[ 0.91062661725633361176950031551069, 1.78937338274366638823049968448935, 3.96410157213
```

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. To omit a search range for `z`, set the search range to `[NaN NaN]`. Because `random` selects starting points randomly, the same solution might be found on successive calls. Call `vpasolve` repeatedly to ensure you find both solutions.

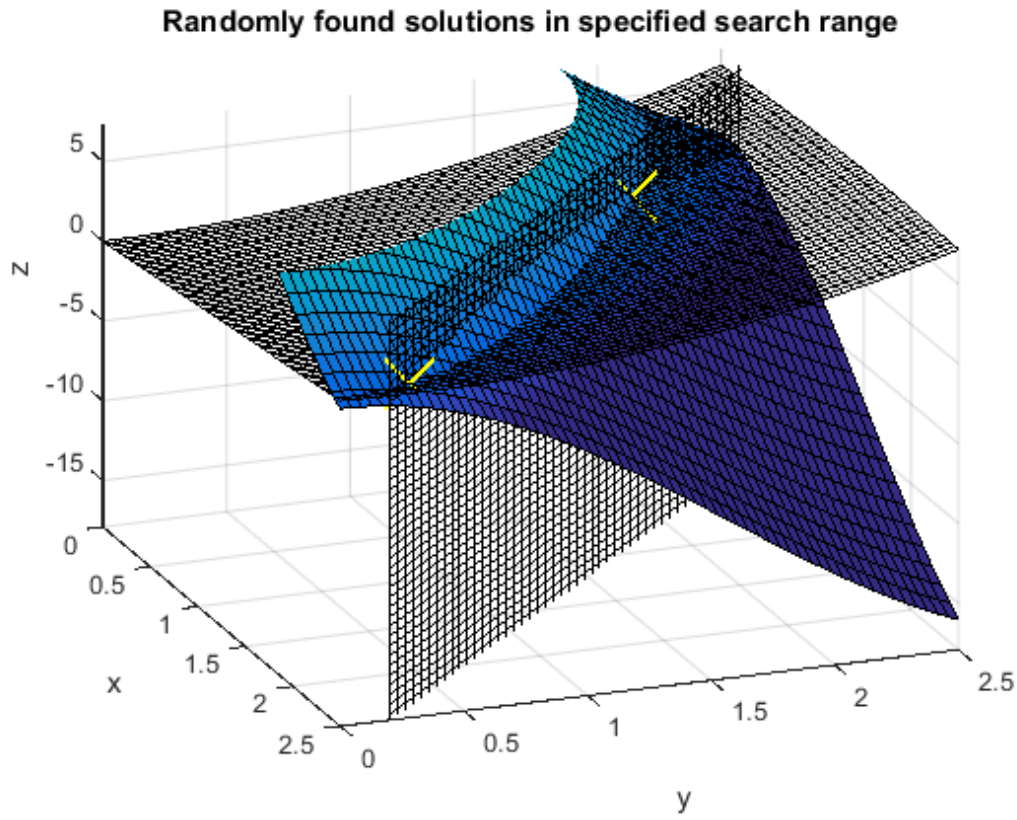
```
clear sol
for i = 1:5
    temp = vpasolve([z == exp1, z == exp2, exp3 == 0],[x y z],[0 3; 0 3;NaN NaN],...
        'random',true);
 sol(i,1) = temp.x;
 sol(i,2) = temp.y;
    sol(i,3) = temp.z;
end
sol
```

```
sol =

[ 0.91062661725633361176950031551069,  1.78937338274366638823049968448935, 3.96410157213
[  2.36974772245479792091013371601740, 0.33025227754520207908986628398261, 2.29335437682
[ 0.91062661725633361176950031551069,  1.78937338274366638823049968448935, 3.96410157213
[ 0.91062661725633361176950031551069,  1.78937338274366638823049968448935, 3.96410157213
[ 0.91062661725633361176950031551069,  1.78937338274366638823049968448935, 3.96410157213
```

Plot the equations using `ezsurf`. 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`.

```
clf
ax = axes;
ezsurf(exp1,[-3 2.5 0 2.5])
grid on
hold on
ezsurf(exp2,[0 2.5 0 2.5])
ezsurf(x1,y1,z1,[0,2.5,-20,10])
scatter3(sol(:,1),sol(:,2),sol(:,3),600,'yellow','X','LineWidth',2)
title('Randomly found solutions in specified search range')
```

```
cz = ax.Children;
alpha(cz(2),0)
alpha(cz(3),0)
caxis([0 20])
view(69,28)
```



`vpasolve` finds solutions at the intersection of the surfaces formed by the equations as shown.

Solve a Single Differential Equation

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

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

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

```
syms y(x)
```

For details, see “Create Symbolic Functions” on page 1-9.

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

The output from `dsolve` parallels the output from `solve`. That is, you can:

- Call `dsolve` with the number of output variables equal to the number of dependent variables.
- Place the output in a structure whose fields contain the solutions of the differential equations.

First-Order Linear ODE

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

```
syms y(t)
```

Now use `dsolve` to solve the equation:

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

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

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

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

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

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

Nonlinear ODE

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

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

results in

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

Second-Order ODE with Initial Conditions

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

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

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

Third-Order ODE

Solve this third-order ordinary differential equation:

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

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

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

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

```
u(x) =

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

More ODE Examples

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

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

Differential Equation	MATLAB Command

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

Solve a System of Differential Equations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Solve the system using `dsolve`:

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

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

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

Differential Algebraic Equations

A system of differential algebraic equations is a system of equations involving unknown functions of one independent variable (typically, the time variable `t`) and their derivatives. These functions are often called state variables. A general form of a DAE system is

$$F\left(\dot{x}(t), x(t), t\right) = 0$$

The number of equations $F = [F_1, \ldots F_n]$ must match the number of state variables $x(t) = [x_1(t), \ldots x_n(t)]$.

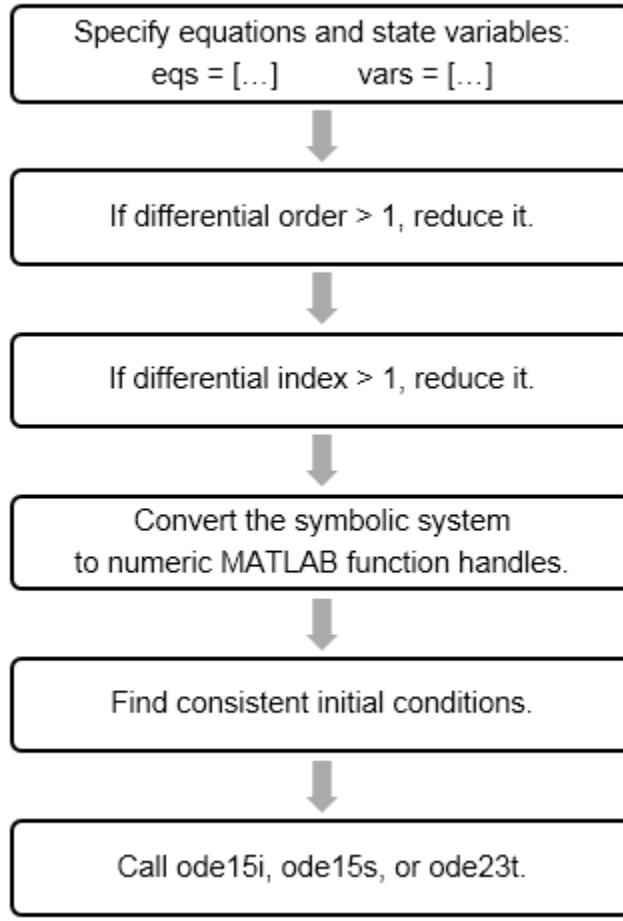
The differential order of a DAE system is the highest differential order of its equations. The differential order of a differential algebraic equation is the highest derivative of its state variables.

The differential index of a DAE system is the number of differentiations needed to reduce the system to a system of ordinary differential equations (ODEs).

Set Up Your DAE Problem

Often, you can solve a system of differential algebraic equations (DAEs) by converting it to a system of DAEs with differential index `1` or `0`, and then using MATLAB solvers, such as `ode15i`, `ode15s`, or `ode23t`. These solvers have their own requirements for the system of equations and initial conditions. Most DAE systems do not directly come in the form suitable for the MATLAB solvers, but you can convert them to a suitable form.

These preliminary steps help you set up the DAE system using the functions available in Symbolic Math Toolbox, and then convert the system to numeric function handles acceptable by MATLAB. After completing these steps, call `ode15i`, `ode15s`, or `ode23t` specifying the system by the MATLAB function handles, and also providing initial conditions.



Step 1: Equations and Variables

A system of differential algebraic equations includes equations, dependent variables (state variables), and an independent variable `t`. You can specify equations as symbolic equations (using the == operator) or as symbolic expressions. If you use symbolic expressions, the toolbox assumes that these expressions are equations with right sides equal to `0`.

For example, specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is the state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t) m r g;
eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

Alternatively, you can specify the same equations as symbolic expressions.

```
eqs= [m*diff(x(t), 2) - T(t)/r*x(t), ...
      m*diff(y(t), 2) - T(t)/r*y(t) + m*g, ...
      x(t)^2 + y(t)^2 - r^2];
```

Step 2: Differential Order

After specifying equations and state variables, see if the differential order of the system is `1`. For example, the differential order of the two-dimensional pendulum system is `2`.

If the system involves higher-order differential equations, use `reduceDifferentialOrder` to convert all higher-order equations to first-order equations by substituting derivatives with additional state variables. See "Reduce Differential Order of DAE Systems" on page 2-126.

Step 3: Differential Index

Check the differential index of the system. To be able to use the MATLAB solvers `ode15i`, `ode15s`, or `ode23t`, your DAE system must be of differential index `1` or `0`. (In the latter case, the system is a system of ordinary differential equations.) If the differential index of the system is `2` or higher, then reduce it by using `reduceDAEIndex` or `reduceDAEToODE`. See "Check and Reduce Differential Index" on page 2-128.

Step 4: MATLAB Function Handles

Convert the system to MATLAB functions acceptable by the MATLAB solvers. If you want to use the `ode15i` solver, then use `daeFunction` to convert the system to a MATLAB function handle. If you want to use the `ode15s` or `ode23t` solver, then use

`massMatrixForm` to extract the mass matrix and the right sides of the system of equations. Then, convert the resulting matrix and vector to MATLAB function handles by using `matlabFunction`. See "Convert DAE Systems to MATLAB Function Handles" on page 2-134.

Step 5: Consistent Initial Conditions

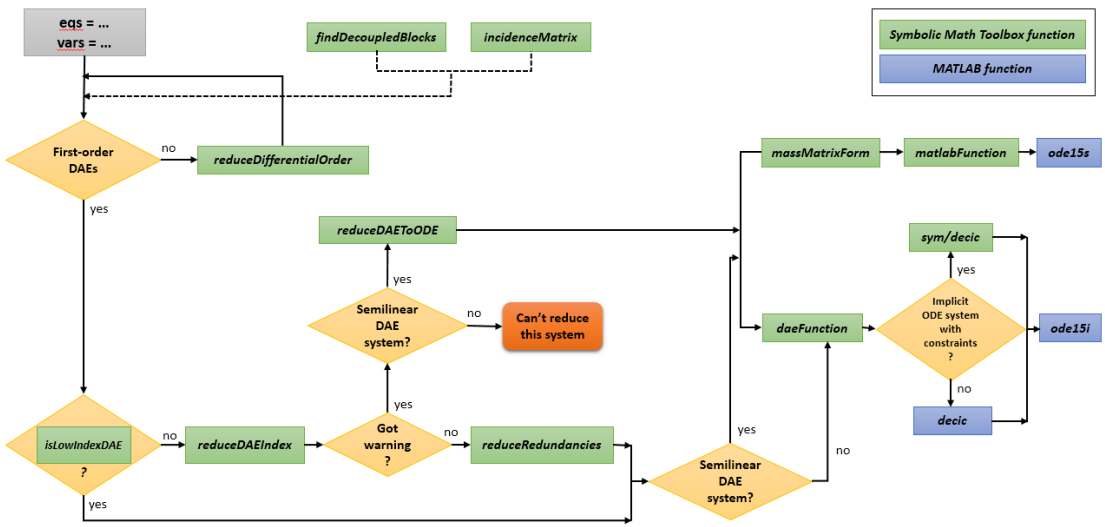
If you reduced the differential index of the system, then find consistent initial conditions for the new system. See "Find Consistent Initial Conditions" on page 2-147.

Step 6: ODE Solvers

Use one of the MATLAB solvers, `ode15i`, `ode15s`, or `ode23t`, to solve the system. See "Solve DAE System Using MATLAB ODE Solvers" on page 2-163.

Solving DAE Systems Flow Chart

This flow chart shows possible sequences of steps that you might need to take when solving a DAE system. The flow chart includes the functions that you might need to use. The process involves MATLAB functions as well as functions available in Symbolic Math Toolbox.



Reduce Differential Order of DAE Systems

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is the state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t) m r g;
eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

The first and second equations have second-order derivatives of the coordinates `x` and `y`. The third equation is an algebraic equation. Thus, the differential order of this DAE system is `2`. To visualize where the terms with the state variables and their derivatives appear in this DAE system, display the incidence matrix of the system. The system contains three equations and three state variables, so `incidenceMatrix` returns a `3`-by-`3` matrix of `1`s and `0`s. Here, `1`s correspond to the terms containing state variables or their derivatives.

```
M = incidenceMatrix(eqs, vars)
```

```
M =
     1     0     1
     0     1     1
     1     1     0
```

Before checking the differential index of the system or solving this DAE system, you must convert it to a first-order DAE system. For this, use the `reduceDifferentialOrder` function that substitutes the derivatives with new variables, such as `Dxt(t)` and `Dyt(t)`. You can call `reduceDifferentialOrder` with two or three output arguments. The syntax with three output arguments shows which derivatives correspond to new variables.

```
[eqs, vars, R] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)

R =
[ Dxt(t), diff(x(t), t)]
[ Dyt(t), diff(y(t), t)]
```

Display the incidence matrix of the new system. The index reduction process introduced two new variables and two new equations. As a result, `incidenceMatrix` now returns a 5-by-5 matrix of 1s and 0s.

```
M = incidenceMatrix(eqs, vars)

M =
     1     0     1     1     0
     0     1     1     0     1
     1     1     0     0     0
     1     0     0     1     0
     0     1     0     0     1
```

For the next step, see “Check and Reduce Differential Index” on page 2-128.

Check and Reduce Differential Index

The MATLAB solvers `ode15i`, `ode15s`, and `ode23t` can solve systems of ordinary differential equations or systems of differential algebraic equations of differential index `0` or `1`. Therefore, before you can solve a system of DAEs, you must check the differential index of the system. If the index is higher than `1`, the next step is to rewrite the system so that the index reduces to `0` or `1`.

In this section...

Reduce Differential Index to 1

Create a First-Order DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t) m r g;

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order differential system. Before checking the differential index, rewrite the system so that it contains only first-order differential equations and algebraic equations.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Once you have a first-order DAE system, use the `isLowIndexDAE` function to check the differential index of the system. If the index is `0` or `1`, then `isLowIndexDAE` returns `1` (logical true). In this case, skip the index reduction and go to the next step. If the differential index is `2` or higher, then `isLowIndexDAE` returns `0` (logical false). For this system of differential algebraic equations, `isLowIndexDAE` returns `0` (logical false).

```
isLowIndexDAE(eqs,vars)

ans =
     0
```

There are two index reduction functions available in Symbolic Math Toolbox. The `reduceDAEIndex` function tries to reduce the differential index by differentiating the original equations (Pantelides algorithm) and replacing the derivatives by new variables. The result contains the original equations (with the derivatives replaced by new variables) followed by the new equations. The vector of variables contains the original variables followed by variables generated by `reduceDAEIndex`.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,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)
```

Often, `reduceDAEIndex` introduces equations and variables that can be easily eliminated. You can simplify the system by eliminating redundant equations.

```
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                           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 `1`, which means that the differential index of the system is `0` or `1`.

```
isLowIndexDAE(DAEs,DAEvars)
```

```
ans =
     1
```

For the next step, see “Convert DAE Systems to MATLAB Function Handles” on page 2-134.

Reduce Differential Index to 0

Create a First-Order DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t) m r g;

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order differential system. Before checking the differential index, rewrite the system so that it contains only first-order differential equations and algebraic equations.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Once you have a first-order DAE system, use the `isLowIndexDAE` function to check the differential index of the system. If the index is `0` or `1`, then `isLowIndexDAE` returns `1` (logical true). In this case, skip the index reduction and go to the next step. If the

differential index is 2 or higher, then `isLowIndexDAE` returns `0` (logical false). For this system of differential algebraic equations, `isLowIndexDAE` returns `0` (logical false).

```
isLowIndexDAE(eqs,vars)
```

```
ans =
     0
```

The Pantelides algorithm used by `reduceDAEIndex` can underestimate the differential index of a system. After index reduction, the `reduceDAEIndex` function internally calls `isLowIndexDAE` to check the differential index of the new DAE system. If the reduced index is still 2 or higher, it issues the following warning:

```
Warning: The index of the reduced DAEs is larger...
than 1. [daetools::reduceDAEIndex]
```

Another index reduction function, `reduceDAEToODE`, reduces a DAE system to a system of implicit ordinary differential equations by using a structural algorithm based on Gaussian elimination of the mass matrix. This function only works on semilinear DAE systems, and it is typically slower than `reduceDAEIndex`. The main advantage of using `reduceDAEToODE` is that it reliably reduces semilinear DAE systems to ODE systems (DAEs of index `0`).

Use `reduceDAEToODE` to reduce the differential index of small semilinear DAE systems or semilinear DAE systems for which `reduceDAEIndex` fails to reduce the index to `1`.

For example, the system of equations for a two-dimensional pendulum is relatively small (five first-order equations in five variables). The `reduceDAEToODE` function reduces this system to a system of implicit ordinary differential equations as follows.

```
[ODEs,constraints] = reduceDAEToODE(eqs,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)
```

For the next step, see “Convert DAE Systems to MATLAB Function Handles” on page 2-134.

Convert DAE Systems to MATLAB Function Handles

At this step, your DAE system must be a first-order system of differential order `0` or `1`. The system is still a system of symbolic expressions and variables. Before you can use the MATLAB differential solvers, you must convert your DAE or ODE system to a suitable input for these solvers, that is, a MATLAB function handle.

There are two ways to convert a DAE or ODE system to a MATLAB function handle:

- To use the `ode15i` solver, convert a DAE or ODE system to a function handle by using `daeFunction.`.
- To use the `ode15s` or `ode23t` solver, find the mass matrix and vector containing the right sides of equations by using `massMatrixForm`. Then convert the result to function handles by using `matlabFunction`. Note that you can use this approach only with semilinear systems.

The following examples show how to convert your DAE or ODE system to function handles when you use different solvers.

In this section...
"When Solving DAEs with ode15i" on page 2-134
"When Solving ODEs with ode15i" on page 2-137
"When Solving DAEs with ode15s or ode23t" on page 2-140
"When Solving ODEs with ode15s or ode23t" on page 2-143

When Solving DAEs with ode15i

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
```

```
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                        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)
```

The `ode15i` solver requires its input argument to be a function handle that describes a DAE system as `F(t, y(t), y'(t)) = 0`. Thus, you must convert a DAE system to a function handle `F = F(y, y, yp)`, where `t` is a scalar, and `y` and `yp` are column vectors.

Once you have a first-order low-index DAE system consisting of a vector of equations `DAEs` and a vector of variables `DAEvars` that is ready for conversion to a MATLAB function handle, use `daeFunction` to convert the system. If a DAE system contains symbolic parameters (symbolic variables other than specified in the vector of state variables `DAEVars`), then specify these symbolic parameters as additional input arguments of `daeFunction`. For example, the two-dimensional pendulum model contains the variables `m`, `r`, and `g`. Call `daeFunction` and provide these variables as additional arguments.

```
f = daeFunction(DAEs, DAEvars, m, r, g);
```

Although `daeFunction` lets you create a function handle containing symbolic parameters without numeric values assigned to them, you cannot use these function handles as input arguments for the `ode15i` solver. Before calling the solvers, you must assign numeric values to all symbolic parameters.

```
m = 1.0;
r = 1.0;
g = 9.81;
```

The function handle `f` still contains symbolic parameters. Create the following purely numeric function handle `F` that you can pass to `ode15i`.

```
F = @(t, Y, YP) f(t, Y, YP, m, r, g);
```

If your DAE system does not contain any symbolic parameters, then `daeFunction` creates a function handle suitable for `ode15i`. For example, substitute the parameters `m = 1.0`, `r = 1.0`, and `g = 9.81` into the equations `DAEs`. Now the system does not contain symbolic variables other than specified in the vector of state variables `DAEvars`.

```
DAEs = subs(DAEs)
```

```
DAEs =
                                                  Dxtt(t) - T(t)*x(t)
                                          Dytt(t) - T(t)*y(t) + 981/100
                                                  x(t)^2 + y(t)^2 - 1
                                         2*Dxt(t)*x(t) + 2*Dyt(t)*y(t)
 2*Dxtt(t)*x(t) + 2*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                             Dytt(t) - diff(Dyt(t), t)
                                                Dyt(t) - diff(y(t), t)
```

Use `daeFunction` to create a function handle. The result is a function handle suitable for `ode15i`.

```
F = daeFunction(DAEs, DAEvars);
```

For the next step, see "Find Consistent Initial Conditions" on page 2-147.

When Solving ODEs with ode15i

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
```

```
       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)
```

Reduce the system to an ODE system (differential index `0`).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

The `ode15i` solver requires its input argument to be a function handle that describes an ODE system as `F(t, y(t), y'(t)) = 0`. Thus, you must convert an ODE system to a function handle `F = F(y, y, yp)`, where `t` is a scalar, and `y` and `yp` are column vectors.

Once you have a first-order ODE system consisting of a vector of equations `ODEs` and a vector of variables `vars` that is ready for conversion to a MATLAB function handle, use `daeFunction` to convert the system. If an ODE system contains symbolic parameters (symbolic variables other than specified in the vector of state variables `vars`), then specify these symbolic parameters as additional input arguments of `daeFunction`. For

example, the two-dimensional pendulum model contains the variables `m`, `r`, and `g`. Call `daeFunction` and provide these variables as additional arguments.

```
f = daeFunction(ODEs, vars, m, r, g);
```

Although `daeFunction` lets you create a function handle that contains symbolic parameters without numeric values assigned to them, you cannot use these function handles as input arguments for the `ode15i` solver. Before you call the solvers, you must assign numeric values to all symbolic parameters.

```
m = 1.0;
r = 1.0;
g = 9.81;
```

The function handle `f` still contains symbolic parameters. Create the following purely numeric function handle `F` that you can pass to `ode15i`.

```
F = @(t, Y, YP) f(t, Y, YP, m, r, g);
```

If your ODE system does not contain any symbolic parameters, then `daeFunction` creates a function handle suitable for `ode15i`. For example, substitute the parameters `m = 1.0`, `r = 1.0`, and `g = 9.81` into the equations `ODEs`. Now the system does not contain symbolic variables other than those specified in the vector of state variables `vars`.

```
ODEs = subs(ODEs)
```

```
ODEs =
                                Dxt(t) - diff(x(t), t)
                                Dyt(t) - diff(y(t), t)
                            diff(Dxt(t), t) - T(t)*x(t)
                  diff(Dyt(t), t) - T(t)*y(t) + 981/100
          - (4*T(t)*y(t) - 981/50)*diff(y(t), t) -...
                             4*Dxt(t)*diff(Dxt(t), t) -...
                             4*Dyt(t)*diff(Dyt(t), t) -...
             diff(T(t), t)*(2*x(t)^2 + 2*y(t)^2) -...
                               4*T(t)*x(t)*diff(x(t), t)
```

Use `daeFunction` to create a function handle suitable for `ode15i`.

```
F = daeFunction(ODEs, vars);
```

For the next step, see “Find Consistent Initial Conditions” on page 2-147.

When Solving DAEs with ode15s or ode23t

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                       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)
```

To use `ode15s` or `ode23t`, you need function two handles: one must represent the mass matrix of a DAE system, and the other must represent the vector containing the right side of the equations. If `M` is a mass matrix form and `F` is a vector containing the right side of equations, then `M(t,y(t))*y'(t) = F(t,y(t))`.

Once you have a first-order low-index semilinear DAE system consisting of a vector of equations and a vector of variables, use `massMatrixForm` to find the mass matrix `M` and vector `F` of the right side of the equations.

```
[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, 2*y(t), 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*Dxt(t)^2 - 2*Dyt(t)^2
```

```
                                    -Dytt(t)
                                     -Dyt(t)
```

The result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)
```

```
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, 2*y(t), 0, 0]
[ 0,  0, 0, 0,     -1, 0, 0]
[ 0, -1, 0, 0,      0, 0, 0]

F =
                         T(t)*x(t) - Dxtt(t)
               T(t)*y(t) - Dytt(t) - 981/100
                         1 - y(t)^2 - x(t)^2
             - 2*Dxt(t)*x(t) - 2*Dyt(t)*y(t)
 - 2*Dxtt(t)*x(t) - 2*Dxt(t)^2 - 2*Dyt(t)^2
                                    -Dytt(t)
                                     -Dyt(t)
```

The mass matrix and the vector containing the right side of the equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(DAEvars) 1])
```

```
tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
```

```
 Y6
 Y7
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, DAEvars, tempvars)
F = subs(F, DAEvars, tempvars)
```

```
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, 2*Y2, 0, 0]
[ 0,  0, 0, 0,   -1, 0, 0]
[ 0, -1, 0, 0,    0, 0, 0]

F =
                  Y1*Y3 - Y7
        Y2*Y3 - Y6 - 981/100
           - Y1^2 - Y2^2 + 1
          - 2*Y1*Y4 - 2*Y2*Y5
 - 2*Y4^2 - 2*Y5^2 - 2*Y1*Y7
                         -Y6
                         -Y5
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

For the next step, see “Find Consistent Initial Conditions” on page 2-147.

When Solving ODEs with ode15s or ode23t

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state

variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to an ODE system (differential index `0`).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

Find the mass matrix of the system and a vector containing the right side of the equations.

```
[M,F] = massMatrixForm(ODEs,vars)
```

```
M =
[           -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 result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)
```

```
M =
[           -1,                    0,                    0,         0,         0]
[            0,                   -1,                    0,         0,         0]
[            0,                    0,                    0,         1,         0]
[            0,                    0,                    0,         0,         1]
[ -4*T(t)*x(t), 981/50 - 4*T(t)*y(t), - 2*x(t)^2 - 2*y(t)^2, -4*Dxt(t), -4*Dyt(t)]

F =
             -Dxt(t)
             -Dyt(t)
           T(t)*x(t)
 T(t)*y(t) - 981/100
                   0
```

The mass matrix and the vector containing the right side of equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB

function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(vars) 1])
```

```
tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, vars, tempvars)
F = subs(F, vars, tempvars)
```

```
M =
[       -1,                 0,                  0,     0,     0]
[        0,                -1,                  0,     0,     0]
[        0,                 0,                  0,     1,     0]
[        0,                 0,                  0,     0,     1]
[ -4*Y1*Y3, 981/50 - 4*Y2*Y3, - 2*Y1^2 - 2*Y2^2, -4*Y4, -4*Y5]

F =
             -Y4
             -Y5
           Y1*Y3
 Y2*Y3 - 981/100
               0
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

For the next step, see “Find Consistent Initial Conditions” on page 2-147.

Find Consistent Initial Conditions

The next step after index reduction and conversion to function handles is searching for initial conditions that satisfy all equations of your new low-index DAE or ODE system. There are two functions that let you find consistent initial conditions:

- If you use `reduceDAEIndex` to reduce the differential index of the system to `1`, then use the MATLAB `decic` function to find consistent initial conditions for the new DAE system.
- If you use `reduceDAEToODE` to rewrite the system as a system of implicit ODEs, then use the `decic` function available in Symbolic Math Toolbox. As one of its input arguments, this function accepts algebraic constraints of the original system returned by `reduceDAEToODE` and returns consistent initial conditions that satisfy those constraints.

The following examples show how to find consistent initial conditions for your DAE or ODE system when you use different solvers.

In this section...
“When Solving DAEs with ode15i” on page 2-147
“When Solving ODEs with ode15i” on page 2-150
“When Solving DAEs with ode15s or ode23t” on page 2-153
“When Solving ODEs with ode15s or ode23t” on page 2-158

When Solving DAEs with ode15i

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth’s standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g
```

```
eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                        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)
```

Generate a Function Handle

Assign numeric values to `m`, `r`, and `g`. Then use `subs` to substitute these numeric values into `DAEs`.

```
m = 1.0;
r = 1.0;
g = 9.81;
DAEs = subs(DAEs);
```

Because you are going to solve this system with the `ode15i` solver, convert the system to a MATLAB function handle using `daeFunction`.

```
f = daeFunction(DAEs, DAEvars);
```

The vector of variables for the first-order DAE system of differential index 1 describing a two-dimensional pendulum is a `7-by-1` vector. Therefore, estimates for initial values of variables and their derivatives must also be `7-by-1` vectors.

```
DAEvars
```

```
DAEvars =
    x(t)
    y(t)
    T(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `7-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 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 the MATLAB `decic` function.

```
[y0, yp0] = decic(f, 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4828
   -0.8757
   -8.5909
         0
    0.0000
   -2.2866
   -4.1477

yp0 =
         0
    0.0000
         0
         0
   -2.2866
         0
         0
```

For the next step, see “Solve DAE System Using MATLAB ODE Solvers” on page 2-163.

When Solving ODEs with ode15i

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth’s standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to an ODE system (differential index 0).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

Generate a Function Handle

Assign numeric values to `m`, `r`, and `g`. Then use `subs` to substitute these numeric values into `ODEs` and `constraints`.

```
m = 1.0;
r = 1.0;
g = 9.81;
ODEs = subs(ODEs);
constraints = subs(constraints);
```

Because you are going to solve this system with the `ode15i` solver, convert the system to a MATLAB function handle using `daeFunction`.

```
f = daeFunction(ODEs, vars);
```

The vector of variables for the first-order ODE system describing a two-dimensional pendulum is a `5-by-1` vector, therefore, estimates for initial values of variables and their derivatives must also be `5-by-1` vectors.

```
vars
```

```
vars =
   x(t)
   y(t)
   T(t)
 Dxt(t)
 Dyt(t)
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `5-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0];
yp0est = zeros(5,1);
```

Create an option set that specifies numerical tolerances for the numerical search.

```
opt = odeset('RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

Find initial values consistent with the system of ODEs and with the algebraic constraints by using the `decic` function available in Symbolic Math Toolbox. 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. The zero elements in `[1,0,0,0,1]` correspond to those values in `y0est` for which `decic` solves the constraint equations.

```
[y0, yp0] = decic(ODEs, vars, constraints, 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
```

For the next step, see “Solve DAE System Using MATLAB ODE Solvers” on page 2-163.

When Solving DAEs with ode15s or ode23t

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth’s standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g
```

```
eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                        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)
```

Find Mass Matrix and Generate Function Handles

Find the mass matrix of the system and a vector containing the right sides of the equations.

```
[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, 2*y(t), 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*Dxt(t)^2 - 2*Dyt(t)^2
                                   -Dytt(t)
                                    -Dyt(t)
```

The result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)
```

```
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, 2*y(t), 0, 0]
[ 0,  0, 0, 0,     -1, 0, 0]
[ 0, -1, 0, 0,      0, 0, 0]

F =
                         T(t)*x(t) - Dxtt(t)
              T(t)*y(t) - Dytt(t) - 981/100
                         1 - y(t)^2 - x(t)^2
             - 2*Dxt(t)*x(t) - 2*Dyt(t)*y(t)
 - 2*Dxtt(t)*x(t) - 2*Dxt(t)^2 - 2*Dyt(t)^2
                                    -Dytt(t)
                                     -Dyt(t)
```

The mass matrix and the vector containing the right side of equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(DAEvars) 1])
```

```
tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
 Y6
 Y7
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, DAEvars, tempvars)
F = subs(F, DAEvars, tempvars)
```

```
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, 2*Y2, 0, 0]
[ 0,  0, 0, 0,   -1, 0, 0]
```

```
[ 0, -1, 0, 0,    0, 0, 0]

F =
                   Y1*Y3 - Y7
         Y2*Y3 - Y6 - 981/100
            - Y1^2 - Y2^2 + 1
          - 2*Y1*Y4 - 2*Y2*Y5
 - 2*Y4^2 - 2*Y5^2 - 2*Y1*Y7
                          -Y6
                          -Y5
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `7-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0; 0; 0];
yp0est = zeros(7,1);
```

Create an option set that contains the mass matrix `M` of the system, a vector `yp0est` of initial guesses for the derivatives, and also 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 `f` describing the DAE by `f(t,y,yp) = f(t,y,y') = 0`. In terms of `M` and `F`, this means `f(t,y,yp) = M(t,y)*yp - F(t,y)`.

```
[y0, yp0] = decic(@(t,y,yp) M(t,y)*yp - F(t,y), 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4828
   -0.8757
   -8.5909
```

```
         0
    0.0000
   -2.2866
   -4.1477

yp0 =
         0
    0.0000
         0
         0
   -2.2866
         0
         0
```

For the next step, see “Solve DAE System Using MATLAB ODE Solvers” on page 2-163.

When Solving ODEs with ode15s or ode23t

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)

eqs =
```

```
       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)
```

Reduce the system to an ODE system (differential index 0).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

Find Mass Matrix and Generate Function Handles

Find the mass matrix of the system and a vector containing the right sides of the equations.

```
[M,F] = massMatrixForm(ODEs,vars)
```

```
M =
[         -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 result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)
```

```
M =
[           -1,                   0,                   0,         0,         0]
[            0,                  -1,                   0,         0,         0]
[            0,                   0,                   0,         1,         0]
[            0,                   0,                   0,         0,         1]
[ -4*T(t)*x(t), 981/50 - 4*T(t)*y(t), - 2*x(t)^2 - 2*y(t)^2, -4*Dxt(t), -4*Dyt(t)]

F =
             -Dxt(t)
             -Dyt(t)
           T(t)*x(t)
 T(t)*y(t) - 981/100
                   0
```

The mass matrix and the vector containing the right side of equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(vars) 1])
```

```
tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, vars, tempvars)
F = subs(F, vars, tempvars)
```

```
M =
[       -1,                 0,                   0,     0,     0]
[        0,                -1,                   0,     0,     0]
[        0,                 0,                   0,     1,     0]
[        0,                 0,                   0,     0,     1]
[ -4*Y1*Y3, 981/50 - 4*Y2*Y3, - 2*Y1^2 - 2*Y2^2, -4*Y4, -4*Y5]

F =
             -Y4
             -Y5
           Y1*Y3
 Y2*Y3 - 981/100
               0
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `5-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0];
yp0est = zeros(5,1);
```

Before you proceed, substitute numeric values for `m`, `r`, and `g` into `ODEs`, `constraints`, and `y0est`.

```
m = 1.0;
r = 1.0;
g = 9.81;
ODEs = subs(ODEs);
constraints = subs(constraints);
y0est = subs(y0est);
```

Create an option set that contains the mass matrix `M` of the system and also 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 the `decic` function available in Symbolic Math Toolbox. 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. The zero elements in `[1,0,0,0,1]` correspond to those values in `y0est` for which `decic` solves the constraint equations.

```
[y0, yp0] = decic(ODEs, vars, constraints, 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
```

For the next step, see “Solve DAE System Using MATLAB ODE Solvers” on page 2-163.

Solve DAE System Using MATLAB ODE Solvers

At this step, you must have a MATLAB function handle representing your ODE or DAE system (of differential index `0` or `1`, respectively). You also must have two vectors specifying initial conditions for the variables of the system and their first derivatives.

`ode15i`, `ode15s`, and `ode23t` are the MATLAB differential equation solvers recommended for this workflow.

- If you have one function handle representing your DAE system (typically obtained via `daeFunction`), then use `ode15i`.
- If your DAE is semilinear, and you have function handles for the mass matrix and the right sides of equations of the DAE system, use `ode15s` or `ode23t`.

The following examples show the last step, which is solving DAE and ODE systems by using the MATLAB solvers.

In this section...
"Solve a DAE System with ode15i" on page 2-163
"Solve an ODE System with ode15i" on page 2-167
"Solve a DAE System with ode15s" on page 2-171
"Solve an ODE System with ode15s" on page 2-177

Solve a DAE System with `ode15i`

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g
```

```
eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                        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)
```

Generate a Function Handle

Assign numeric values to `m`, `r`, and `g`. Then use `subs` to substitute these numeric values into `DAEs`.

```
m = 1.0;
r = 1.0;
g = 9.81;
DAEs = subs(DAEs);
```

Because you are going to solve this system with the `ode15i` solver, convert the system to a MATLAB function handle using `daeFunction`.

```
f = daeFunction(DAEs, DAEvars);
```

Find Consistent Initial Conditions

The vector of variables for the first-order DAE system of differential index 1 describing a two-dimensional pendulum is a `7-by-1` vector. Therefore, estimates for initial values of variables and their derivatives must also be `7-by-1` vectors.

```
DAEvars
```

```
DAEvars =
    x(t)
    y(t)
    T(t)
  Dxt(t)
  Dyt(t)
 Dytt(t)
 Dxtt(t)
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and

$\sin(30°) \approx 0.8$, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `7-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 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 the MATLAB `decic` function.

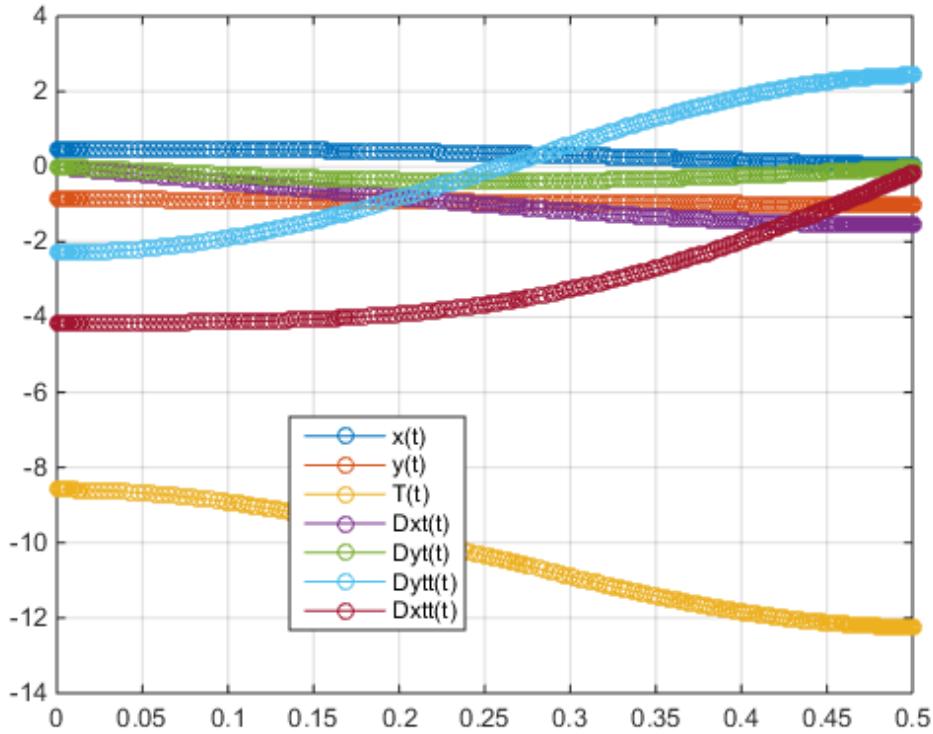
```
[y0, yp0] = decic(f, 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4828
   -0.8757
   -8.5909
         0
    0.0000
   -2.2866
   -4.1477

yp0 =
         0
    0.0000
         0
         0
   -2.2866
         0
         0
```

Solve the system integrating over the time span $0 \le t \le 0.5$. Add the grid lines and the legend to the plot.

```
ode15i(f, [0, 0.5], y0, yp0, opt)

for k = 1:numel(DAEvars)
  S{k} = char(DAEvars(k));
end
```

```
legend(S, 'Location', 'Best')
grid on
```



Solve an ODE System with `ode15i`

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state

variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)

eqs =
       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)
```

Reduce the system to an ODE system (differential index `0`).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

Generate a Function Handle

Assign numeric values to `m`, `r`, and `g`. Then use `subs` to substitute these numeric values into `ODEs` and `constraints`.

```
m = 1.0;
r = 1.0;
g = 9.81;
ODEs = subs(ODEs);
constraints = subs(constraints);
```

Because you are going to solve this system with the `ode15i` solver, convert the system to a MATLAB function handle using `daeFunction`.

```
f = daeFunction(ODEs, vars);
```

Find Consistent Initial Conditions

The vector of variables for the first-order ODE system describing a two-dimensional pendulum is a `5-by-1` vector, therefore, estimates for initial values of variables and their derivatives must also be `5-by-1` vectors.

```
vars
```

```
vars =
   x(t)
   y(t)
   T(t)
 Dxt(t)
 Dyt(t)
```

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `5-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0];
yp0est = zeros(5,1);
```

Create an option set that specifies numerical tolerances for the numerical search.

```
opt = odeset('RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

Find initial values consistent with the system of ODEs and with the algebraic constraints by using the `decic` function available in Symbolic Math Toolbox. 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. The zero elements in `[1,0,0,0,1]` correspond to those values in `y0est` for which `decic` solves the constraint equations.

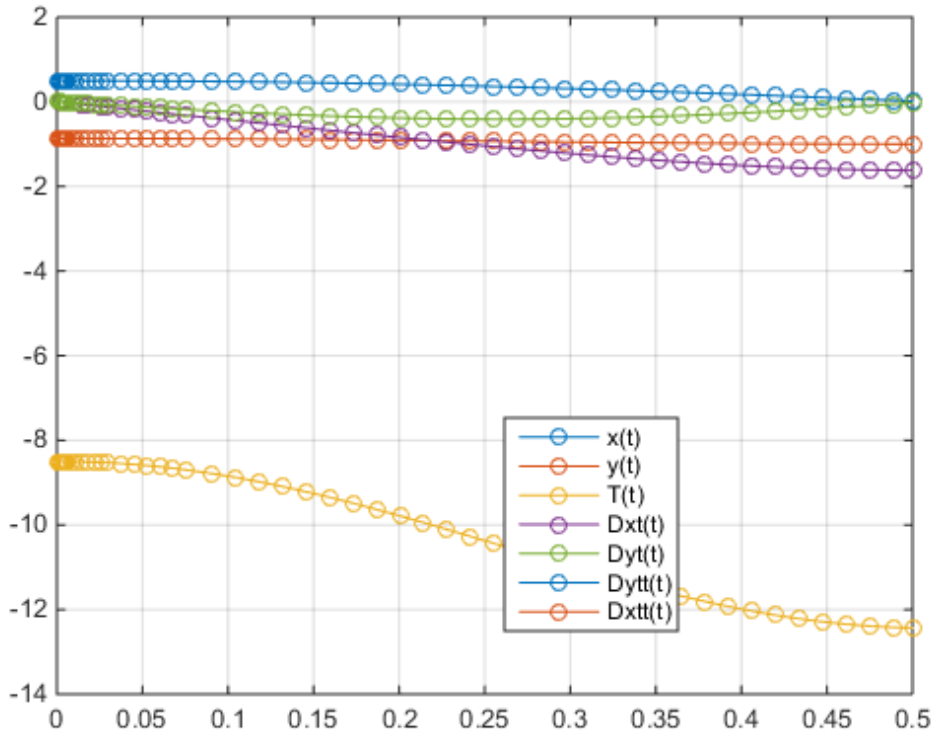
```
[y0, yp0] = decic(ODEs, vars, constraints, 0, y0est, [1,0,0,0,1], yp0est, opt)
```

```
y0 =
    0.5000
   -0.8660
   -8.4957
         0
         0

yp0 =
         0
         0
         0
   -4.2479
   -2.4525
```

Solve the system integrating over the time span $0 \le t \le 0.5$. Add the grid lines and the legend to the plot.

```
ode15i(f, [0, 0.5], y0, yp0, opt)

for k = 1:numel(vars)
  S{k} = char(vars(k));
```

```
end

legend(S, 'Location', 'Best')
grid on
```



Solve a DAE System with ode15s

Create a DAE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the

horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)
```

```
eqs =
       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)
```

Reduce the system to a DAE system of differential index `1`. Eliminate redundant equations.

```
[DAEs,DAEvars] = reduceDAEIndex(eqs,vars);
[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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
```

```
                                    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)
```

Find Mass Matrix and Generate Function Handles

Find the mass matrix of the system and a vector containing the right sides of the equations.

```
[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, 2*y(t), 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*Dxt(t)^2 - 2*Dyt(t)^2
                                    -Dytt(t)
                                     -Dyt(t)
```

The result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
```

```
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)

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, 2*y(t), 0, 0]
[ 0,  0, 0, 0,     -1, 0, 0]
[ 0, -1, 0, 0,      0, 0, 0]

F =
                         T(t)*x(t) - Dxtt(t)
               T(t)*y(t) - Dytt(t) - 981/100
                         1 - y(t)^2 - x(t)^2
             - 2*Dxt(t)*x(t) - 2*Dyt(t)*y(t)
 - 2*Dxtt(t)*x(t) - 2*Dxt(t)^2 - 2*Dyt(t)^2
                                    -Dytt(t)
                                     -Dyt(t)
```

The mass matrix and the vector containing the right side of equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(DAEvars) 1])

tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
 Y6
 Y7
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, DAEvars, tempvars)
F = subs(F, DAEvars, tempvars)
```

```
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, 2*Y2, 0, 0]
[ 0,  0, 0, 0,   -1, 0, 0]
[ 0, -1, 0, 0,    0, 0, 0]

F =
                  Y1*Y3 - Y7
        Y2*Y3 - Y6 - 981/100
           - Y1^2 - Y2^2 + 1
         - 2*Y1*Y4 - 2*Y2*Y5
 - 2*Y4^2 - 2*Y5^2 - 2*Y1*Y7
                         -Y6
                         -Y5
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

Find Consistent Initial Conditions

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values of the variables and their derivatives at the time `t0 = 0` as the following two `7-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0; 0; 0];
yp0est = zeros(7,1);
```

Create an option set that contains the mass matrix `M` of the system, a vector `yp0est` of initial guesses for the derivatives, and also specifies numerical tolerances for the numerical search.

```
opt = odeset('Mass', M, 'InitialSlope', yp0est,...
                'RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

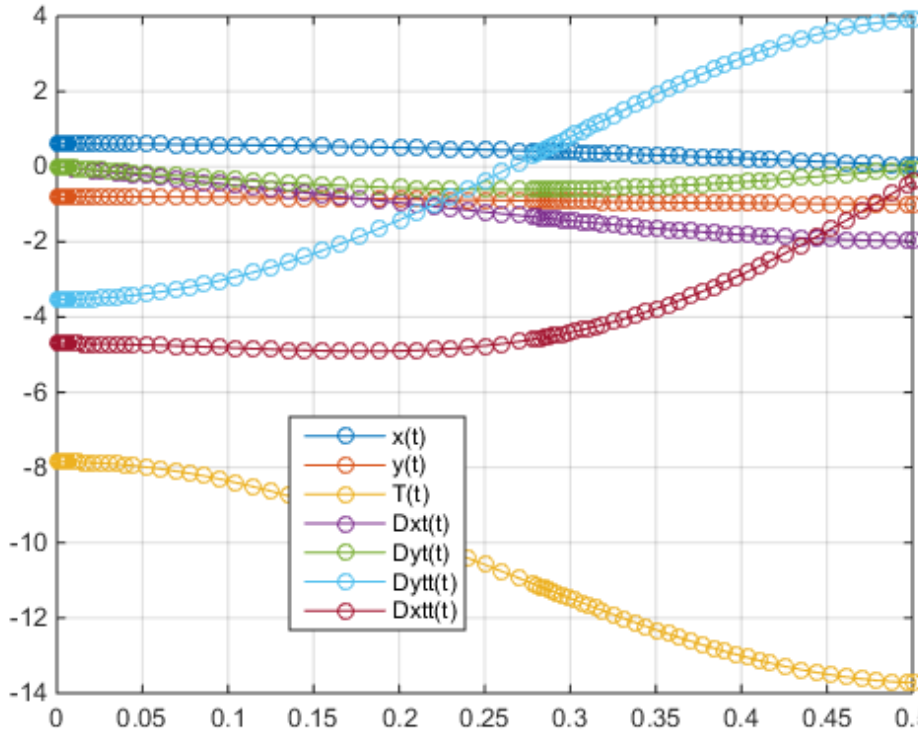
Find consistent initial values for the variables and their derivatives by using the MATLAB `decic` function. The first argument of `decic` must be a function handle `f` describing the DAE by `f(t,y,yp) = f(t,y,y') = 0`. In terms of `M` and `F`, this means `f(t,y,yp) = M(t,y)*yp - F(t,y)`.

```
[y0, yp0] = decic(@(t,y,yp) M(t,y)*yp - F(t,y), 0, y0est, [], yp0est, [], opt)
```

```
y0 =
    0.4828
   -0.8757
   -8.5909
         0
    0.0000
   -2.2866
   -4.1477

yp0 =
         0
    0.0000
         0
         0
   -2.2866
         0
         0
```

Solve the system integrating over the time span $0 \le t \le 0.5$. Add the grid lines and the legend to the plot.

```
ode15s(F, [0, 0.5], y0est, opt)

for k = 1:numel(DAEvars)
  S{k} = char(DAEvars(k));
end

legend(S, 'Location', 'Best')
grid on
```



Solve an ODE System with `ode15s`

Create an ODE System

Specify the system of equations that describes a two-dimensional pendulum. The functions `x(t)` and `y(t)` are the state variables of the system that describe the horizontal and vertical positions of the pendulum mass. The function `T(t)` is a state variable describing the force that keeps the mass from flying away. The variables `m`, `r`, and `g` are the mass, length of the rod, and Earth's standard surface gravity, respectively.

```
syms x(t) y(t) T(t);
```

```
syms m r g

eqs= [m*diff(x(t), 2) == T(t)/r*x(t), ...
      m*diff(y(t), 2) == T(t)/r*y(t) - m*g, ...
      x(t)^2 + y(t)^2 == r^2];

vars = [x(t); y(t); T(t)];
```

This is a second-order DAE system. Reduce it to a first-order system by using `reduceDifferentialOrder`.

```
[eqs, vars] = reduceDifferentialOrder(eqs, vars)

eqs =
       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)
```

Reduce the system to an ODE system (differential index `0`).

```
[ODEs, constraints] = reduceDAEToODE(eqs, 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)
```

Find Mass Matrix and Generate Function Handles

Find the mass matrix of the system and a vector containing the right sides of the equations.

```
[M,F] = massMatrixForm(ODEs,vars)
```

```
M =
[           -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 result contains the symbolic parameters `m`, `r`, and `g`. Assign numeric values to these parameters, and then use `subs` to substitute these numeric values into the mass matrix `M` and vector `F`.

```
m = 1.0;
r = 1.0;
g = 9.81;
M = subs(M)
F = subs(F)
```

```
M =
[           -1,                   0,                   0,        0,        0]
[            0,                  -1,                   0,        0,        0]
[            0,                   0,                   0,        1,        0]
[            0,                   0,                   0,        0,        1]
[ -4*T(t)*x(t), 981/50 - 4*T(t)*y(t), - 2*x(t)^2 - 2*y(t)^2, -4*Dxt(t), -4*Dyt(t)]

F =
             -Dxt(t)
             -Dyt(t)
           T(t)*x(t)
 T(t)*y(t) - 981/100
                   0
```

The mass matrix and the vector containing the right side of equations contain function calls, such as `y(t)`, `Dyt(t)`, and so on. Before you can convert `M` and `F` to MATLAB

function handles, you must replace these function calls with variables. You can use any valid MATLAB variable names. For example, generate the following vector of variables.

```
tempvars = sym('Y', [numel(vars) 1])
```

```
tempvars =
 Y1
 Y2
 Y3
 Y4
 Y5
```

Substitute the new variables into the mass matrix `M` and vector `F`. The resulting matrix `M` and vector `F` are ready for conversion to MATLAB function handles.

```
M = subs(M, vars, tempvars)
F = subs(F, vars, tempvars)
```

```
M =
[       -1,                 0,                   0,     0,     0]
[        0,                -1,                   0,     0,     0]
[        0,                 0,                   0,     1,     0]
[        0,                 0,                   0,     0,     1]
[ -4*Y1*Y3, 981/50 - 4*Y2*Y3, - 2*Y1^2 - 2*Y2^2, -4*Y4, -4*Y5]

F =
             -Y4
             -Y5
            Y1*Y3
 Y2*Y3 - 981/100
               0
```

Convert `M` and `F` to MATLAB function handles by using `matlabFunction`. Use two separate `matlabFunction` calls to convert `M` and `F`.

```
M = matlabFunction(M, 'vars', {t, tempvars});
F = matlabFunction(F, 'vars', {t, tempvars});
```

Find Consistent Initial Conditions

Suppose that the initial angular displacement of the pendulum is 30°, and the origin of the coordinates is at the suspension point of the pendulum. Since cos(30°) = 0.5 and sin(30°) ≈ 0.8, you can specify the starting points for the search for consistent values

of the variables and their derivatives at the time `t0 = 0` as the following two `5-by-1` vectors.

```
y0est = [0.5*r; -0.8*r; 0; 0; 0];
yp0est = zeros(5,1);
```

Before you proceed, substitute numeric values for `m`, `r`, and `g` into `ODEs`, `constraints`, and `y0est`.

```
m = 1.0;
r = 1.0;
g = 9.81;
ODEs = subs(ODEs);
constraints = subs(constraints);
y0est = subs(y0est);
```

Create an option set that contains the mass matrix `M` of the system and also specifies numerical tolerances for the numerical search.

```
opt = odeset('Mass', M, 'RelTol', 10.0^(-7), 'AbsTol' , 10.0^(-7));
```

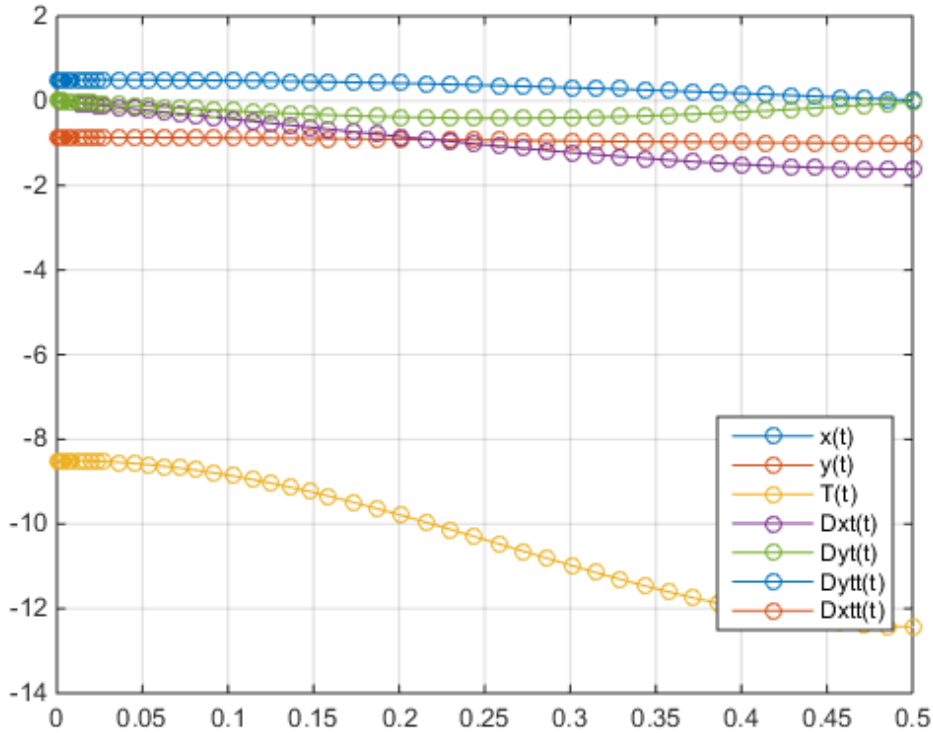
Find initial values consistent with the system of ODEs and with the algebraic constraints by using the `decic` function available in Symbolic Math Toolbox. The parameter `[1,0,0,0,1]` in this function call fixes the first and the last elements in `y0est`, so that `decic` does not change them during the numerical search. The zero elements in `[1,0,0,0,1]` correspond to those values in `y0est` for which `decic` solves the constraint equations.

```
[y0, yp0] = decic(ODEs,vars,constraints, 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
```

Solve the system integrating over the time span $0 \le t \le 0.5$. Add the grid lines and the legend to the plot.

```
ode15s(F, [0, 0.5], y0, opt)

for k = 1:numel(vars)
  S{k} = char(vars(k));
end

legend(S, 'Location', 'Best')
grid on
```



Compute Fourier and Inverse Fourier Transforms

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

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

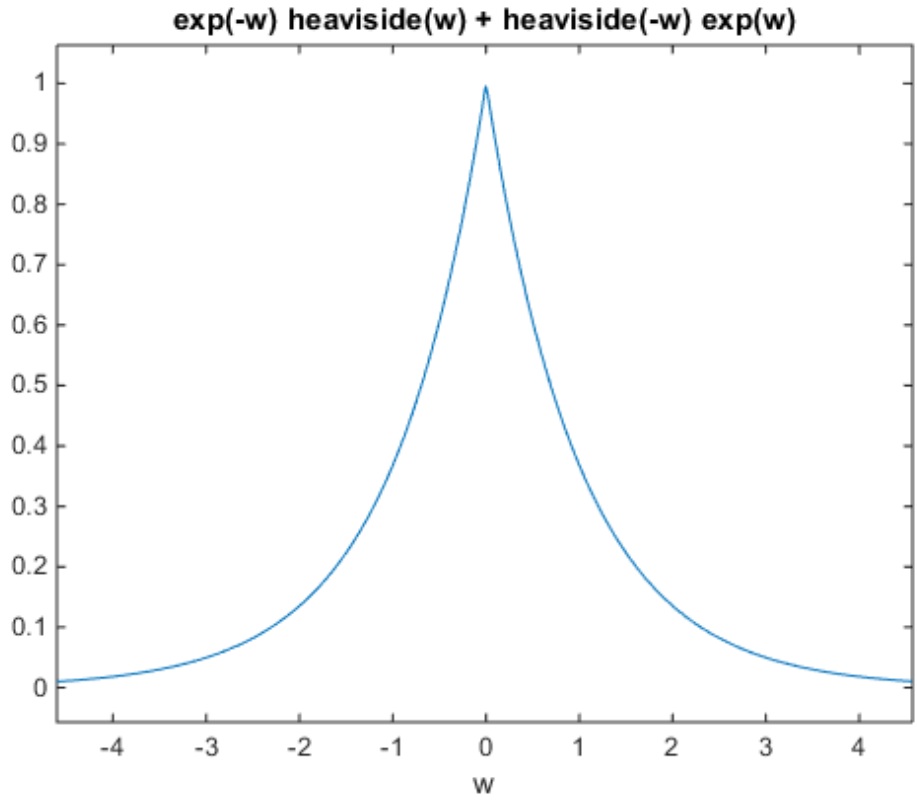
and the inverse Fourier transform (IFT) as

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

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

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

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

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

fcauchy = expand(fcauchy)

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


ezplot(fcauchy)
```

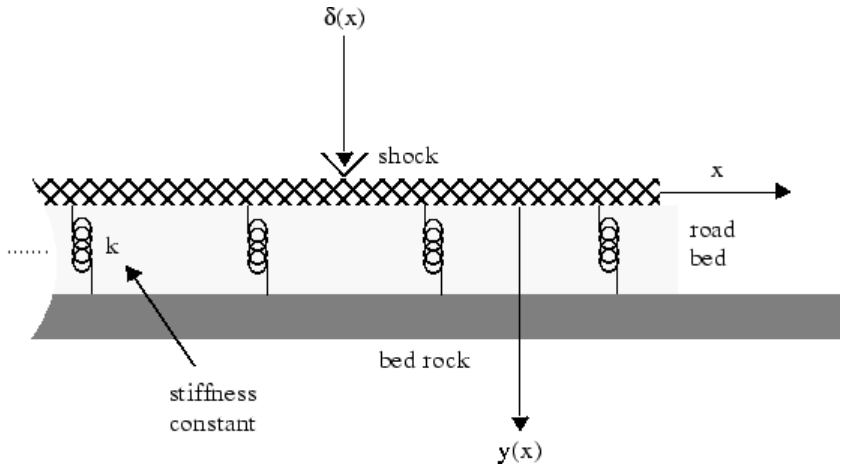


The Fourier transform is symmetric, since the original Cauchy density function is symmetric.

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

```
finvfcauchy = ifourier(fcauchy)

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

simplify(finvfcauchy)

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

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



The shock could be induced by a pneumatic hammer blow.

The differential equation idealizing this physical setting is

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

Here, E represents elasticity of the beam (railroad track), I is the "beam constant," and k is the spring (road bed) stiffness. The shock force on the right side of the differential equation is modeled by the Dirac Delta function $\delta(x)$. The Dirac Delta function has the following important property:

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

A definition of the Dirac Delta function is

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

where

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

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

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

To see a demonstration of this property, try this

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

which returns

```
ans =
w^4*fourier(y(x), x, w)
```

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

Applying the Fourier transform to the differential equation above yields the algebraic equation

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

or

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

where

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

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

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

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

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

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

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

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

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

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

and see

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

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

Notice that `g` contains the Heaviside distribution

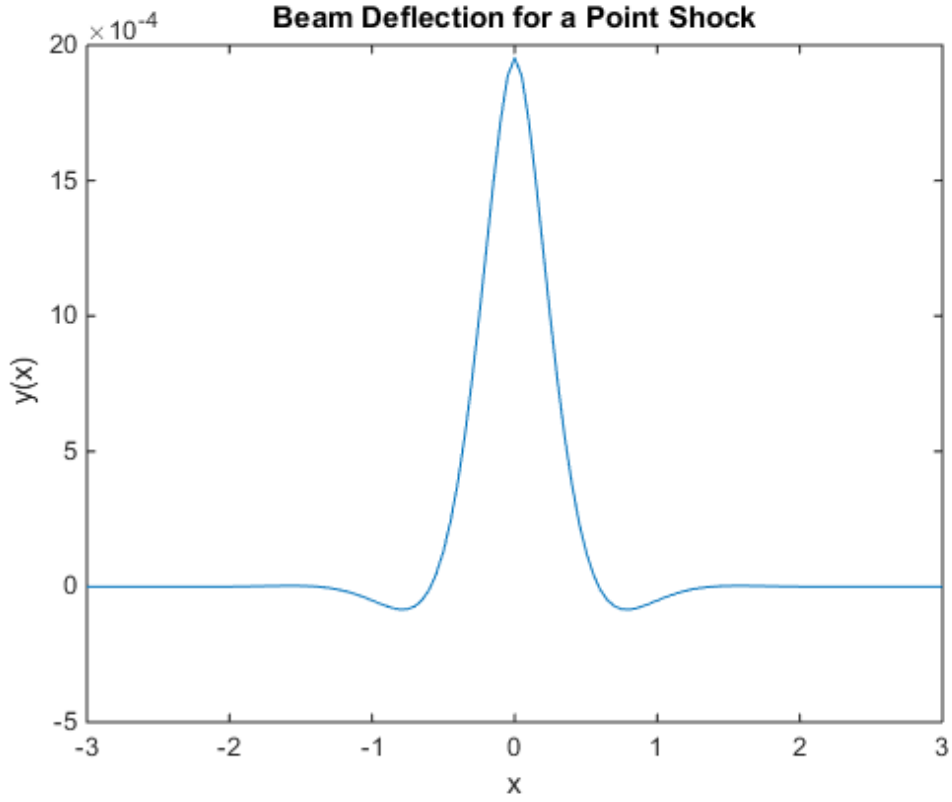
$$H(x) = \begin{cases} 1 & \text{for} \quad x > 0 \\ 0 & \text{for} \quad x < 0 \\ 1/2 & \text{for} \quad x = 0. \end{cases}$$

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

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

by the special property of the Dirac Delta function. To plot this function, substitute the domain of x into $y(x)$, using the `subs` command. The resulting graph shows that the impact of a blow on a beam is highly localized; the greatest deflection occurs at the point of impact and falls off sharply from there.

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


Beam Deflection for a Point Shock
×10^-4
20
15
10
5
0
-5
y(x)
-3
-2
-1
0
1
2
3
x

Compute Laplace and Inverse Laplace Transforms

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

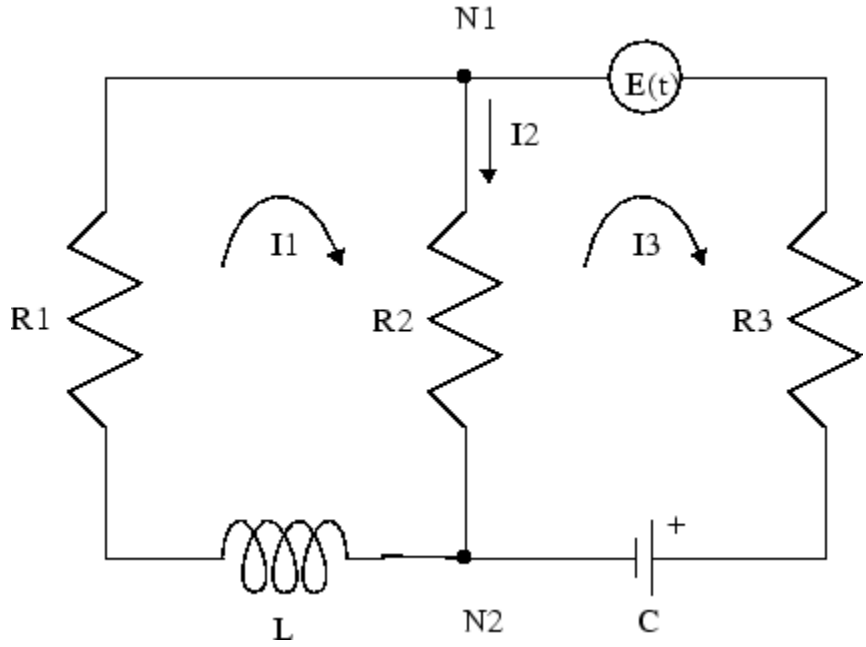
$$L[f](s) = \int_0^\infty f(t)e^{-ts}dt,$$

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

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

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

The Laplace transform has many applications including the solution of ordinary differential equations/initial value problems. Consider the resistance-inductor-capacitor (RLC) circuit below.



Let Rj and Ij, j = 1, 2, 3 be resistances (measured in ohms) and currents (amperes), respectively; L be inductance (henrys), and C be capacitance (farads); $E(t)$ be the electromotive force, and $Q(t)$ be the charge.

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

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

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

Solve this system of differential equations using `laplace`. First treat the R_j, L, and C as (unknown) real constants and then supply values later on in the computation.

```
clear E
syms R1 R2 R3 L C real
```

```
syms I1(t) Q(t) s
dI1(t) = diff(I1(t), t);
dQ(t) = diff(Q(t),t);
E(t) = sin(t);  % Voltage
eq1(t) = dI1(t) + R2*dQ(t)/L - (R2 - R1)*I1(t)/L;
eq2(t) = dQ(t) - (E(t) - Q/C)/(R2 + R3) - R2*I1(t)/(R2 + R3);
```

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

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

returns

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

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

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

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

returns

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

The substitution

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

returns

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

To solve for `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)`, make a final pair of substitutions. First, replace the strings `laplace(I1(t),t,s)` and `laplace(Q(t),t,s)` by the `sym` objects `LI1` and `LQ`, using

```
NI1 = subs(NI1,{laplace(I1(t),t,s),laplace(Q(t),t,s)},{LI1,LQ})
```

to obtain

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

Collecting terms

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

gives

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

A similar string substitution

```
NQ = ...
subs(NQ,{laplace(I1(t),t,s), laplace(Q(t),t,s)}, {LI1,LQ})
```

yields

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

which, after collecting terms,

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

gives

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

Now, solving for `LI1` and `LQ`

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

you obtain

```
LI1 =
(5*(60*s^3 + 56*s^2 + 59*s + 56))/((s^2 + 1)*(20*s^2 + 51*s + 20))

LQ =
(40*s^3 + 190*s^2 + 44*s + 195)/((s^2 + 1)*(20*s^2 + 51*s + 20))
```

To recover `I1` and `Q`, compute the inverse Laplace transform of `LI1` and `LQ`. Inverting `LI1`

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

produces

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

Inverting `LQ`

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

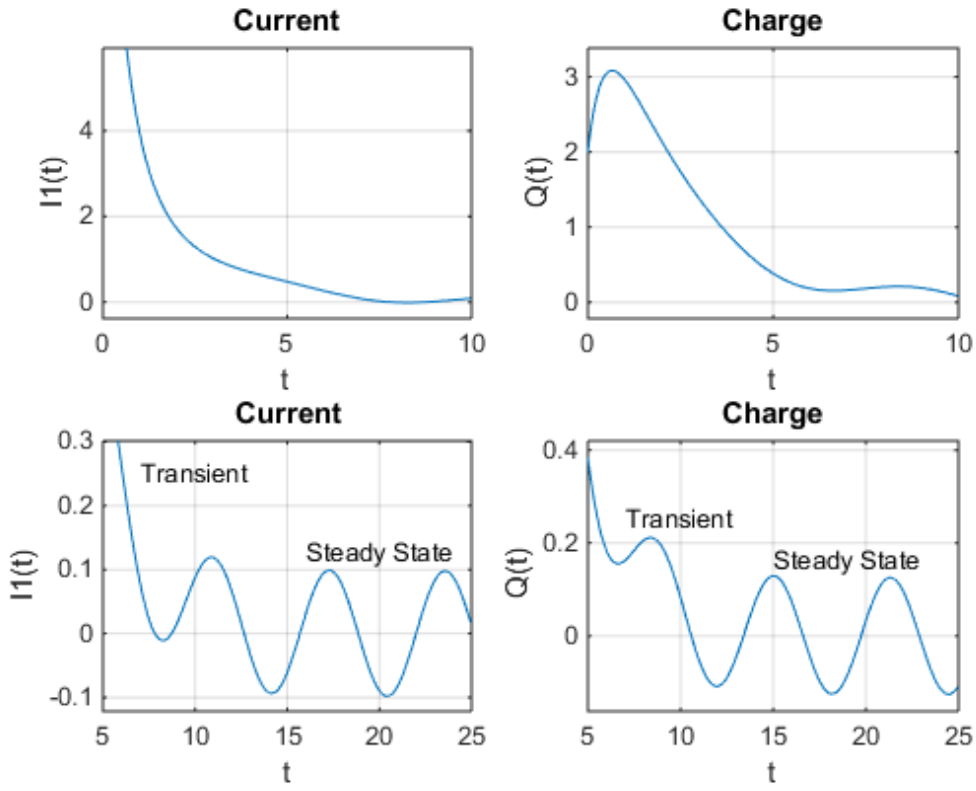
yields

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

Now plot the current `I1(t)` and charge `Q(t)` in two different time domains, $0 \le t \le 10$ and $5 \le t \le 25$. The following statements generate the desired plots.

```
subplot(2,2,1)
ezplot(I1,[0,10])
title('Current')
ylabel('I1(t)')
grid
subplot(2,2,2)
ezplot(Q,[0,10])
title('Charge')
ylabel('Q(t)')
grid
```

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



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

Compute Z-Transforms and Inverse Z-Transforms

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

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

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

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

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

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

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

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

```
syms p(n) z
eq = p(n + 2) - p(n + 1) - p(n);
Zeq = ztrans(eq, n, z)
```

to obtain

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

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

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

to obtain

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

Collecting terms

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

yields

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

Now solve for `Pz`

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

to obtain

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

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

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

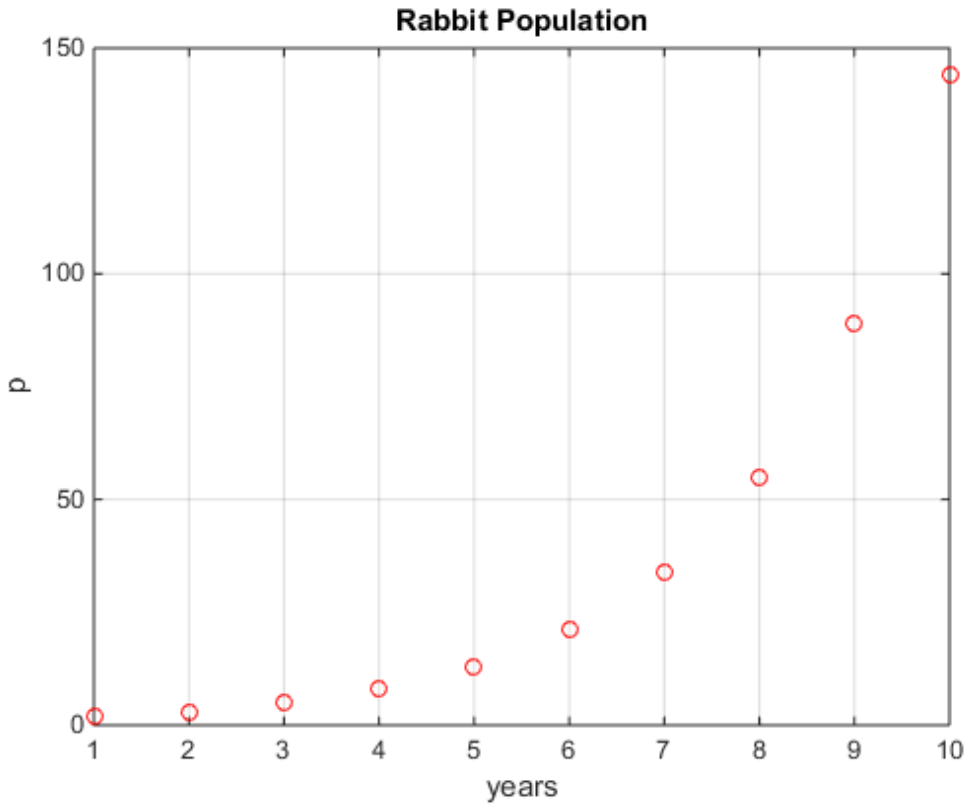
The result is a bit complicated, but explicit:

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

Finally, plot `p` to show the growth in rabbit population over time.

```
m = 1:10;
y = double(subs(p,n,m));
```

```
plot(m, real(y),'rO')
title('Rabbit Population')
xlabel('years')
ylabel('p')
grid on
```



References

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

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

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

Create Plots

In this section...

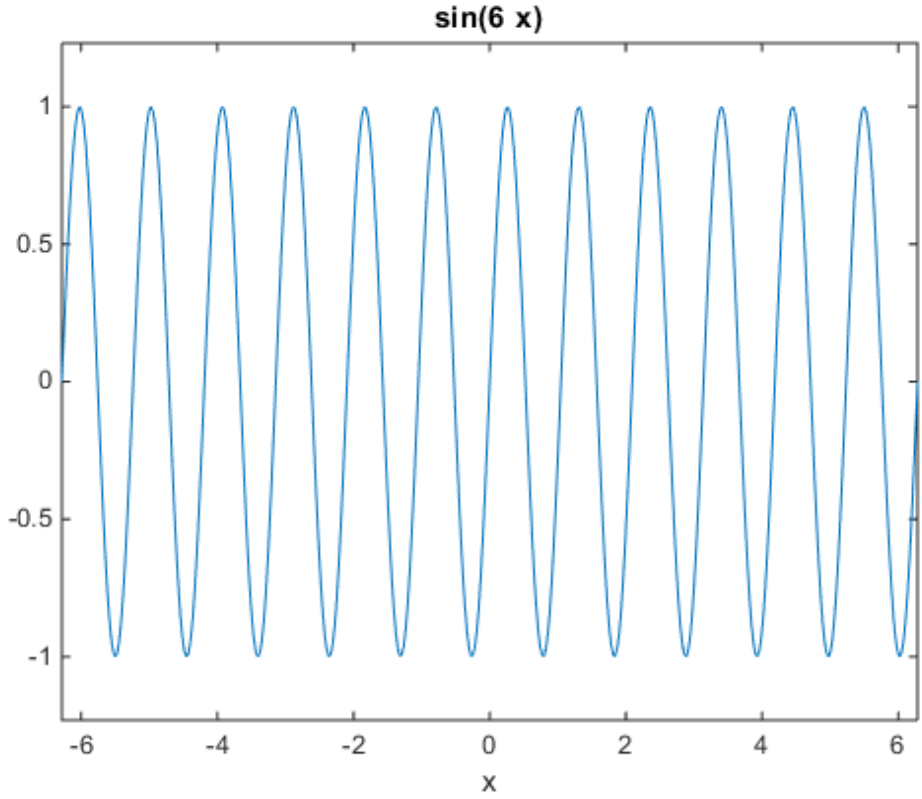
Plot with Symbolic Plotting Functions

MATLAB provides many techniques for plotting numerical data. Graphical capabilities of MATLAB include plotting tools, standard plotting functions, graphic manipulation and data exploration tools, and tools for printing and exporting graphics to standard formats. Symbolic Math Toolbox expands these graphical capabilities and lets you plot symbolic functions using:

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

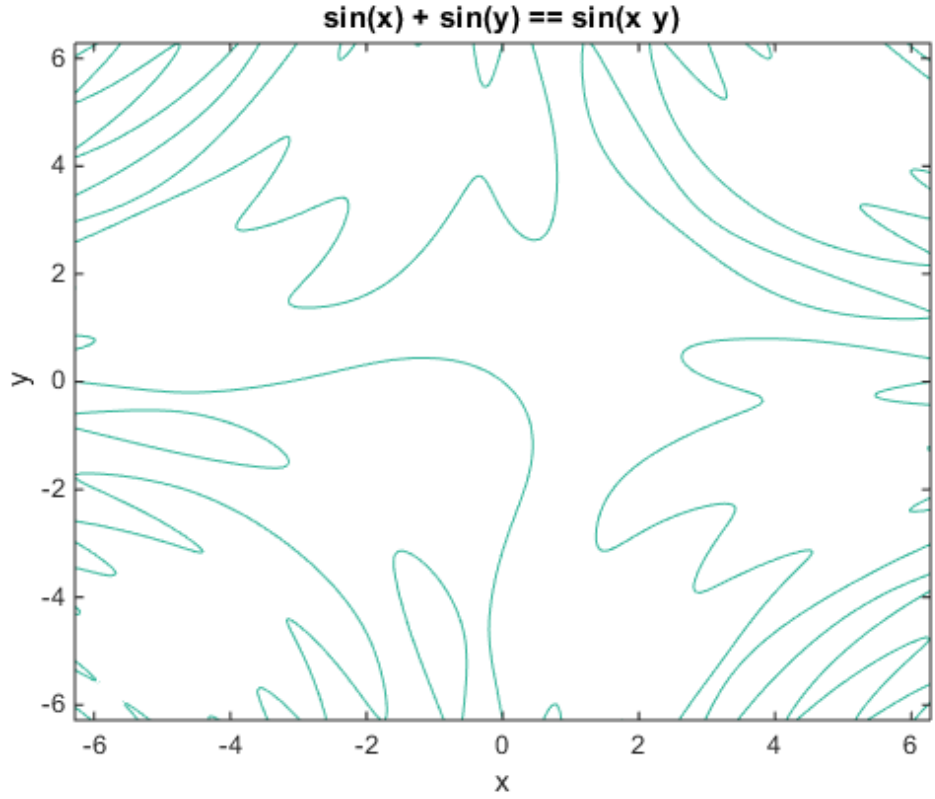
For example, plot the symbolic expression `sin(6x)` in Cartesian coordinates. By default, `ezplot` uses the range $-2\pi < x < 2\pi$:

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



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

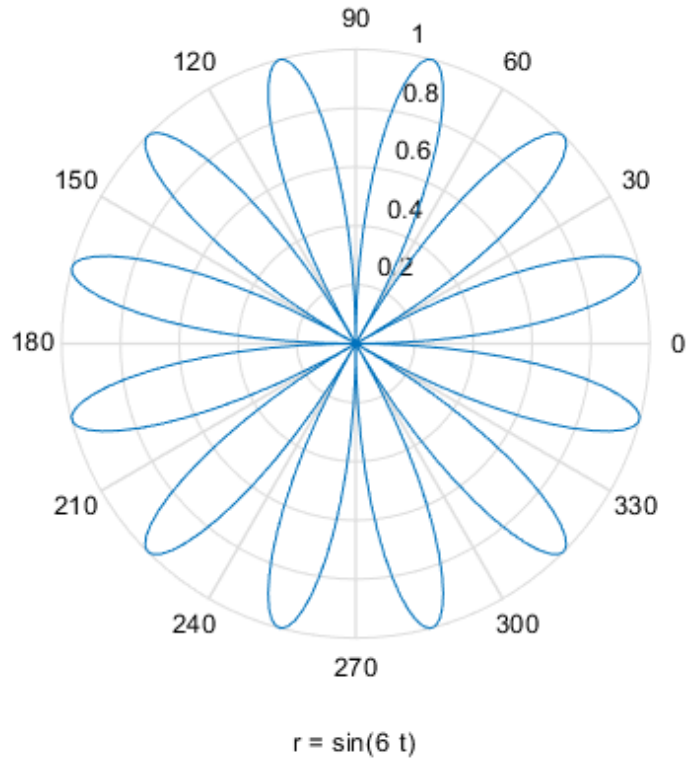
```
syms x y
ezplot(sin(x) + sin(y) == sin(x*y))
```



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

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

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



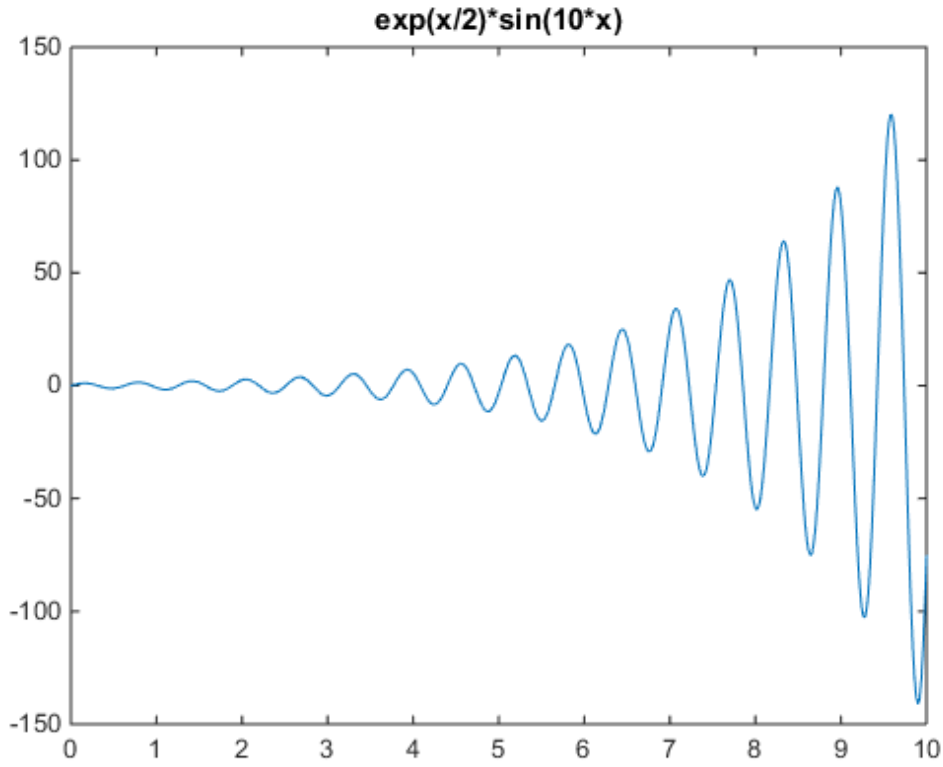
Plot with MATLAB Plotting Functions

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

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

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

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



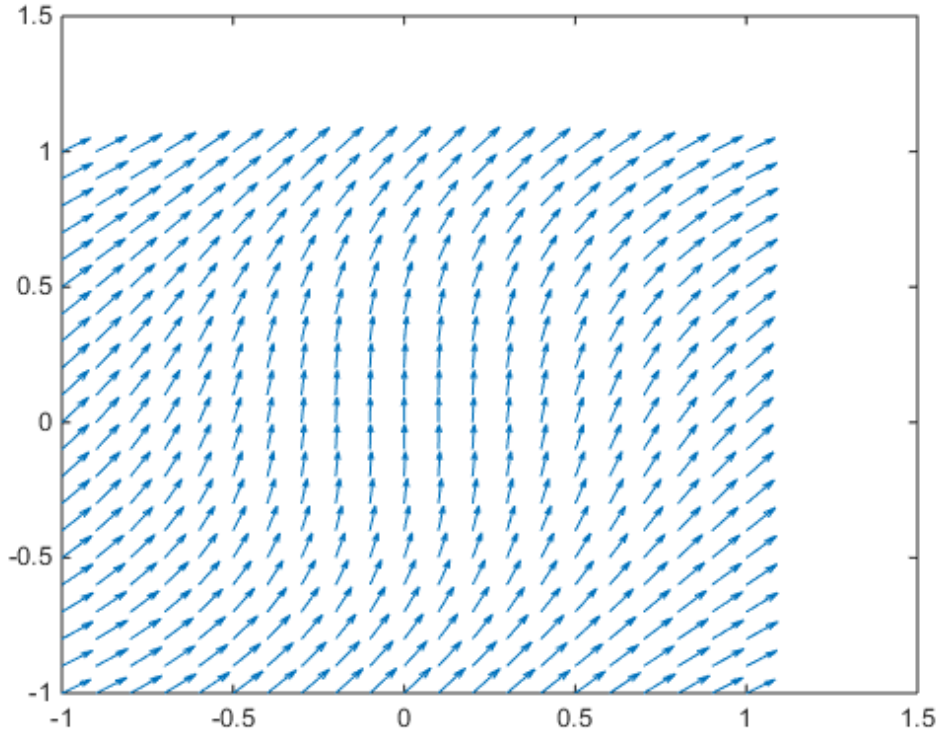
An alternative approach is to replace symbolic variables in an expression with numeric values by using the `subs` function. For example, in the following expressions u and v, substitute the symbolic variables x and y with the numeric values defined by `meshgrid`:

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

```
V = subs(v, [x y], {X,Y});
```

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

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

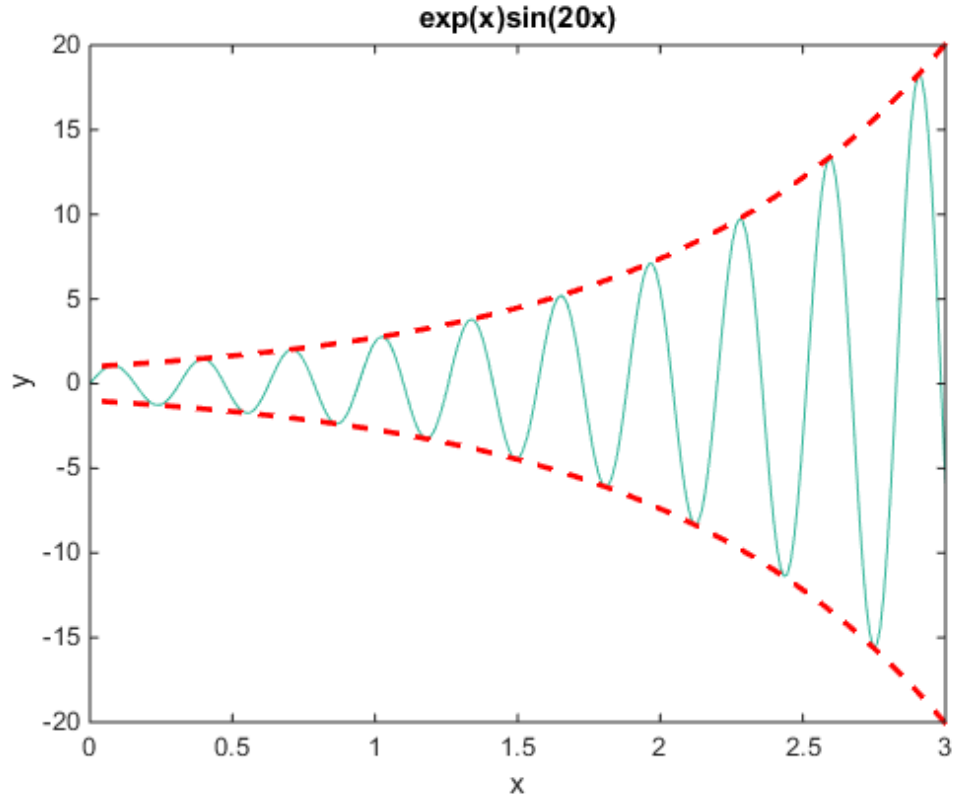


Plot Multiple Symbolic Functions in One Graph

To plot several symbolic functions in one graph, add them to the graph sequentially. To be able to add a new function plot to the graph that already contains a function plot,

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

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



Plot Multiple Symbolic Functions in One Figure

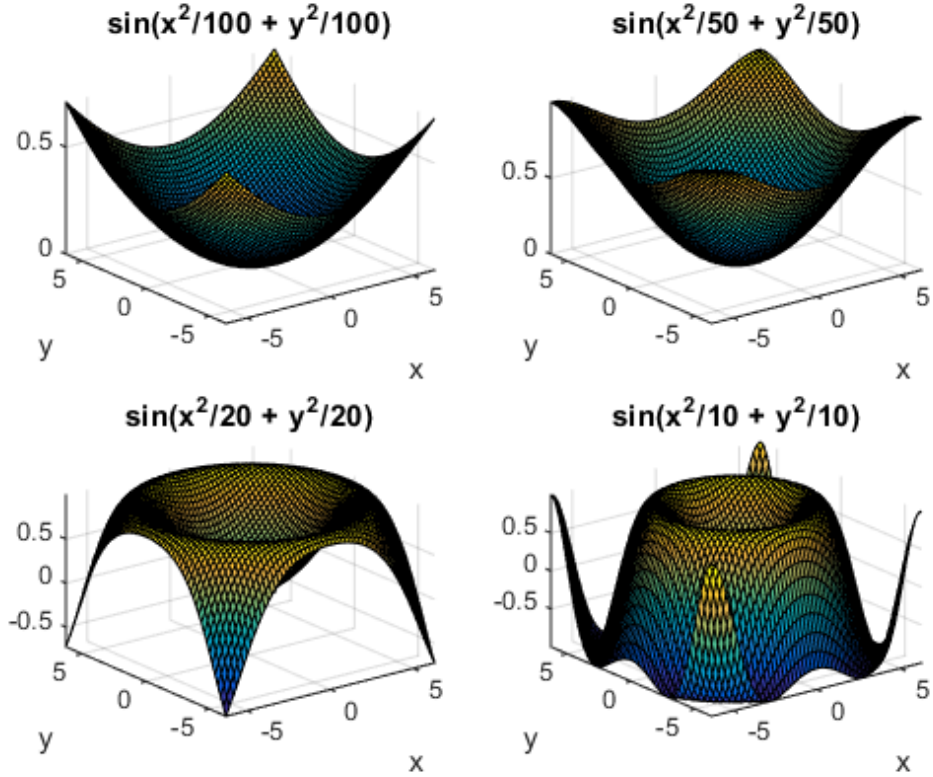
To display several function plots in one figure without overlapping, divide a figure window into several rectangular panes (tiles). Then, you can display each function plot in its own pane. For example, you can assign different values to symbolic parameters of a function, and plot the function for each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

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

This command partitions the figure window into an `m`-by-`n` matrix of small subplots and selects the subplot `p` for the current plot. MATLAB numbers the subplots along the

first row of the figure window, then the second row, and so on. For example, plot the expression `sin(x^2 + y^2)/a` for the following four values of the symbolic parameter `a`:

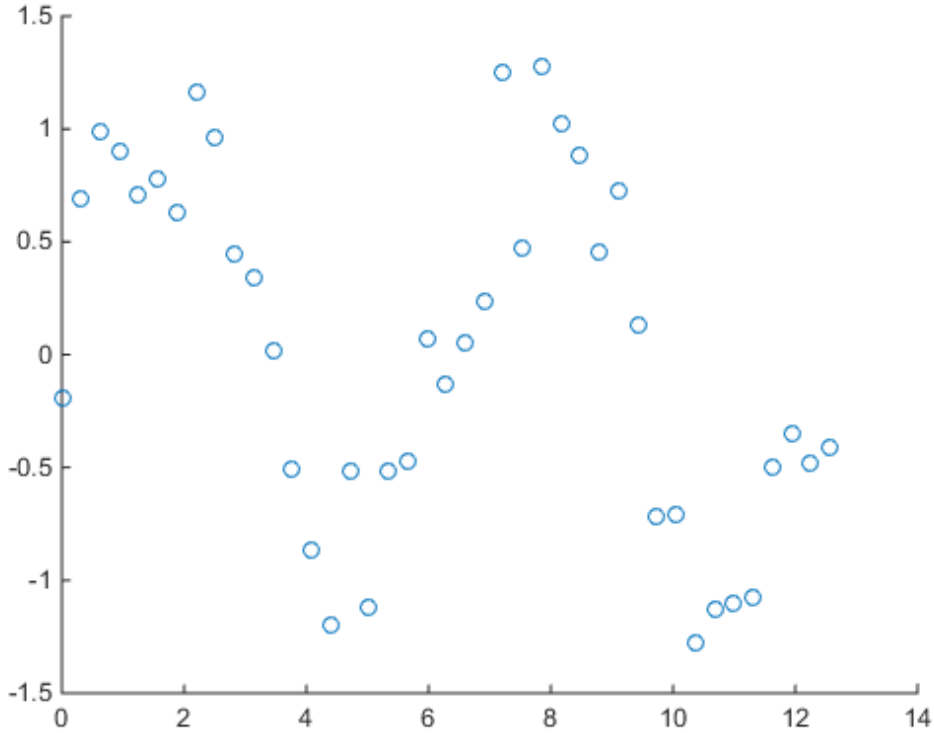
```
syms x y
z = x^2 + y^2;
subplot(2, 2, 1)
ezsurf(sin(z/100))
subplot(2, 2, 2)
ezsurf(sin(z/50))
subplot(2, 2, 3)
ezsurf(sin(z/20))
subplot(2, 2, 4)
ezsurf(sin(z/10))
```



Combine Symbolic Function Plots and Numeric Data Plots

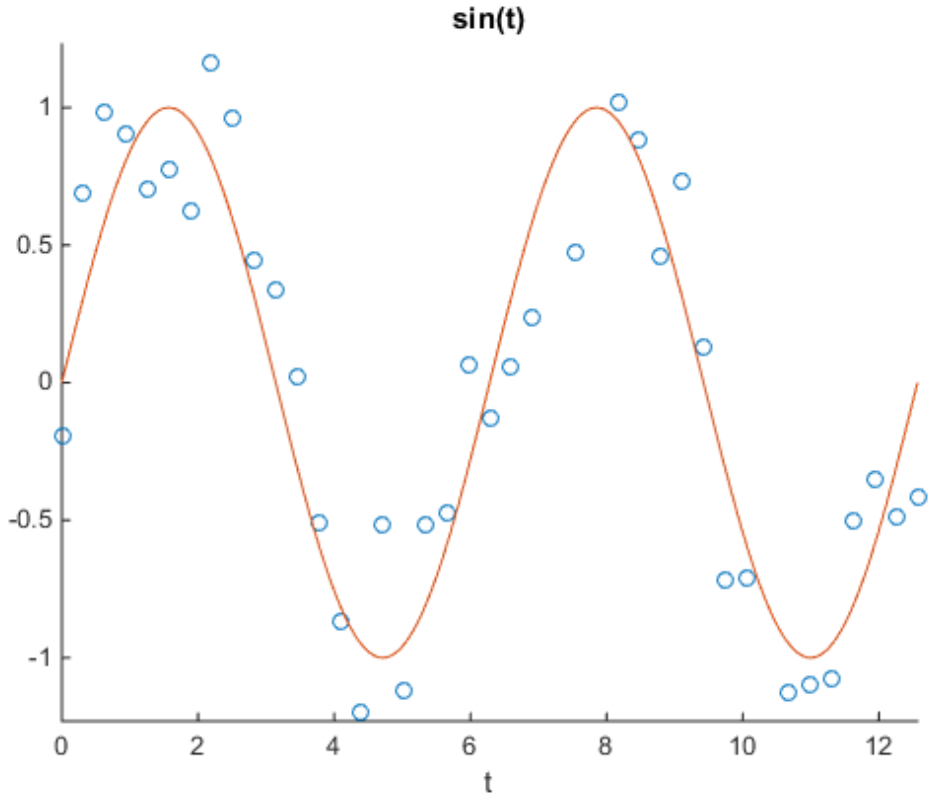
The combined graphical capabilities of MATLAB and the Symbolic Math Toolbox software let you plot numeric data and symbolic functions in one graph. Suppose, you have two discrete data sets, *x* and *y*. Use the `scatter` plotting function to plot these data sets as a collection of points with coordinates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

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



Now, suppose you want to plot the sine function on top of the scatter plot in the same graph. First, use the `hold on` command to retain the current plot in the figure. (Without this command, the symbolic plot that you are about to create replaces the numeric data plot.) Then, use `ezplot` to plot the sine function. To change the color or any other property of the plot, create the handle for the `ezplot` function call, and then use the `set` function:

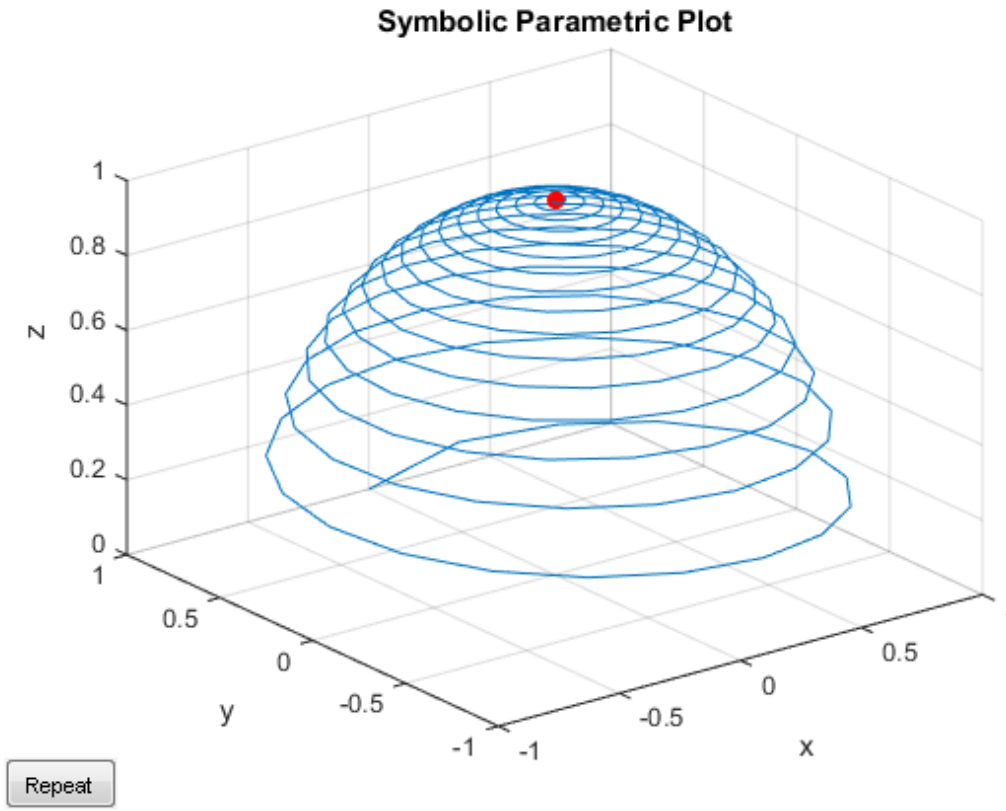
```
hold on
syms t
ezplot(sin(t), [0 4*pi])
hold off
```



MATLAB provides the plotting functions that simplify the process of generating spheres, cylinders, ellipsoids, and so on. The Symbolic Math Toolbox software lets you create

a symbolic function plot in the same graph with these volumes. For example, use the following commands to generate the spiral function plot wrapped around the top hemisphere. The `animate` option switches the `ezplot3` function to animation mode. The red dot on the resulting graph moves along the spiral:

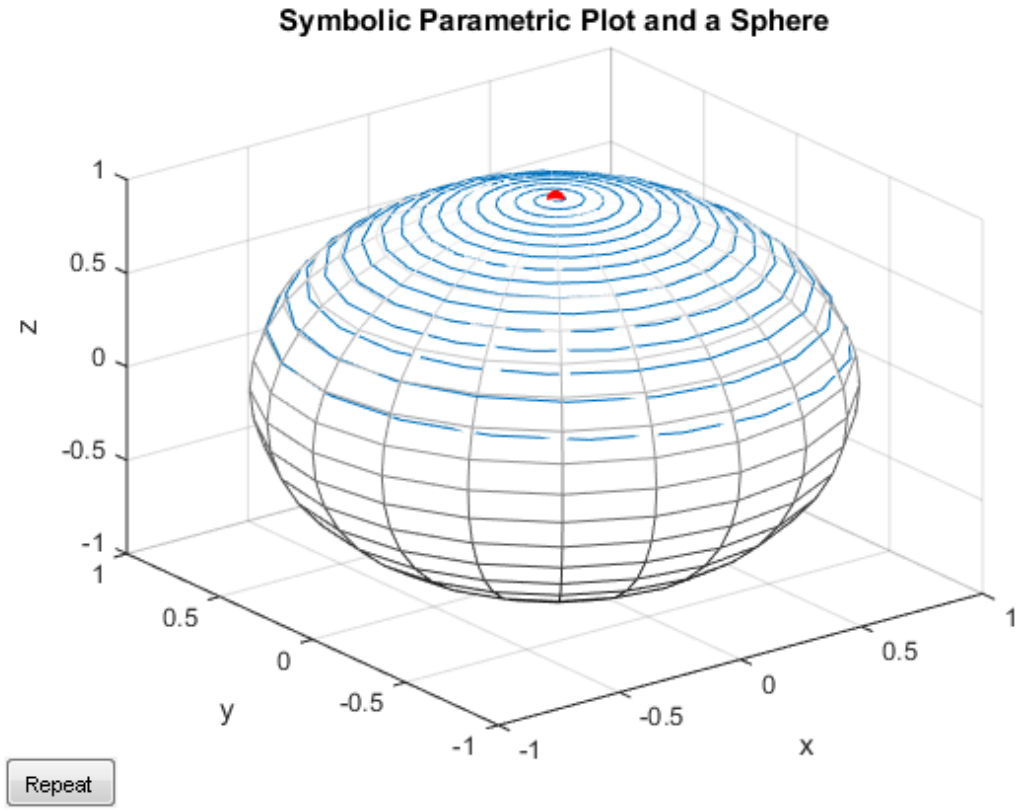
```
syms t
x = (1-t)*sin(100*t);
y = (1-t)*cos(100*t);
z = sqrt(1 - x^2 - y^2);
ezplot3(x, y, z, [0 1], 'animate')
title('Symbolic Parametric Plot')
```



Add the sphere with radius 1 and the center at (0, 0, 0) to this graph. The `sphere` function generates the required sphere, and the `mesh` function creates a mesh plot for

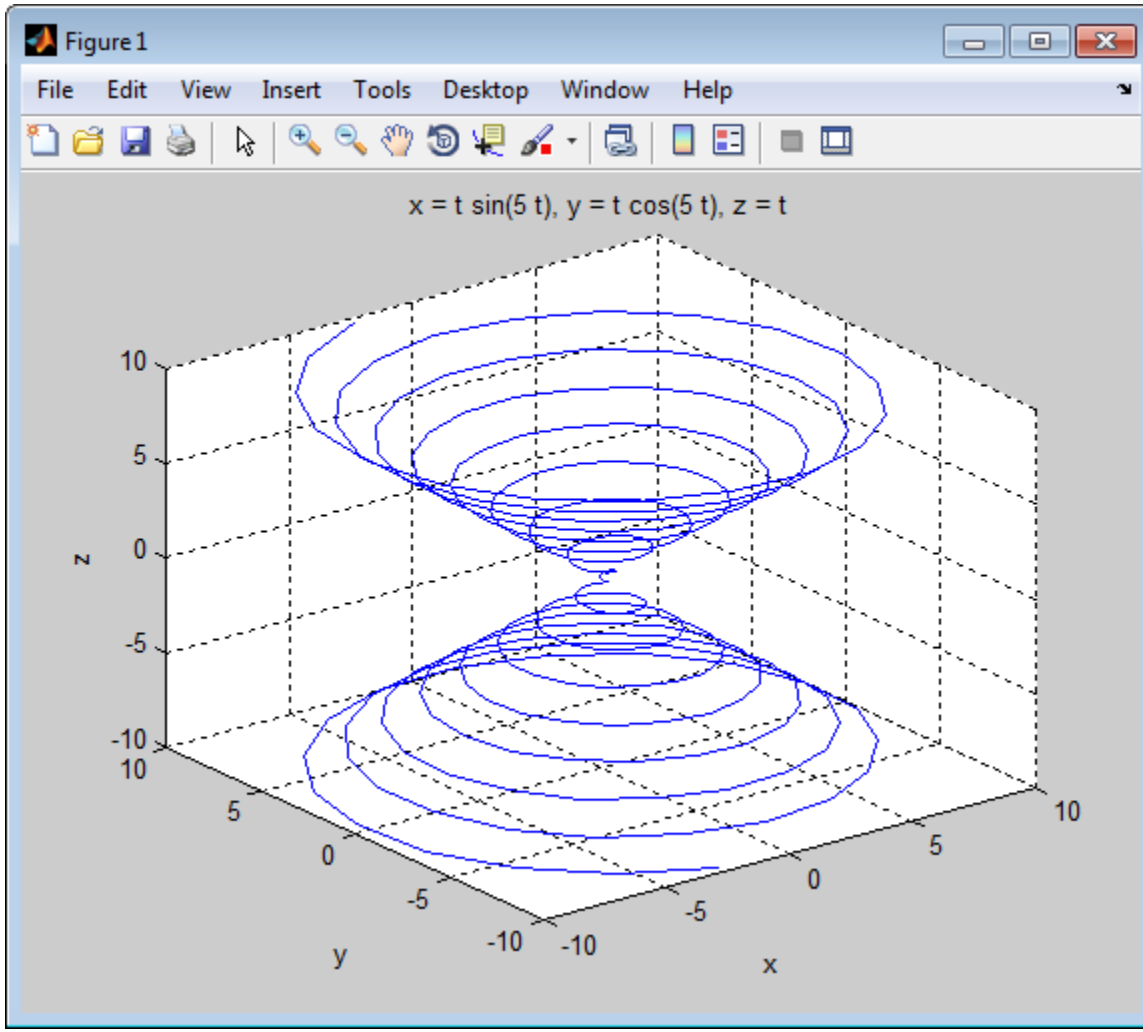
that sphere. Combining the plots clearly shows that the symbolic parametric function plot is wrapped around the top hemisphere:

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



Explore Function Plots

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



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

- Zoom in and out on particular parts of a graph (). Zooming allows you to see small features of a function plot. Zooming behaves differently for 2-D or 3-D views.
- Shift the view of the graph with the pan tool (). Panning is useful when you have zoomed in on a graph and want to move around the plot to view different portions.
- Rotate 3-D graphs (). Rotating 3-D graphs allows you to see more features of the surface and mesh function plots.
- Display particular data values on a graph and export them to MATLAB workspace variables ().

Edit Graphs

MATLAB supports the following two approaches for editing graphs:

- Interactive editing lets you use the mouse to select and edit objects on a graph.
- Command-line editing lets you use MATLAB commands to edit graphs.

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

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

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

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

If you prefer to work from the MATLAB command line or if you want to create a code file, you can edit graphs by using MATLAB plotting commands. For details, see "2-D and 3-D Plots". Also, you can combine the interactive and command-line editing approaches to achieve the look you want for the graphs you create.

Save Graphs

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

- Save a graph as a MATLAB FIG-file (a binary format). The FIG-file stores all information about a graph, including function plots, graph data, annotations, data tips, menus and other controls. You can open the FIG-file only with MATLAB.
- Export a graph to a different file format. When saving a graph, you can choose a file format other than FIG. For example, you can export your graphs to EPS, JPEG, PNG, BMP, TIFF, PDF, and other file formats. You can open the exported file in an appropriate application.
- Print a graph on paper or print it to file. To ensure the correct plot size, position, alignment, paper size and orientation, use Print Preview.
- Generate a MATLAB file from a graph. You can use the generated code to reproduce the same graph or create a similar graph using different data. This approach is useful for generating MATLAB code for work that you have performed interactively with the plotting tools.

For details, see “Printing and Saving”.

Generate C or Fortran Code

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

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

For example:

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

ans =
      t0 = (x**4*3.0D1)/(x*y**2+1.0D1)-x**3*(y**2+1.0D0)**2

ccode(z)

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

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

```
fortran(z, 'file', 'fortrantest')
```

generates a file named `fortrantest` in the current folder. `fortrantest` consists of the following:

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

Similarly, the command

```
ccode(z,'file','ccodetest')
```

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

```
  t16 = x*x;
  t17 = y*y;
```

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

`ccode` and `fortran` generate many intermediate variables. This is called *optimized* code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. Intermediate variables can make the resulting code more efficient by reusing intermediate expressions (such as `t12` in `fortrantest`, and `t16` in `ccodetest`). They can also make the code easier to read by keeping expressions short.

If you work in the MuPAD Notebook app, see `generate::C` and `generate::fortran`.

Generate MATLAB Functions

You can use `matlabFunction` to generate a MATLAB function handle that calculates numerical values as if you were substituting numbers for variables in a symbolic expression. Also, `matlabFunction` can create a file that accepts numeric arguments and evaluates the symbolic expression applied to the arguments. The generated file is available for use in any MATLAB calculation, whether or not the computer running the file has a license for Symbolic Math Toolbox functions.

If you work in the MuPAD Notebook app, see "Create MATLAB Functions from MuPAD Expressions" on page 3-47.

Generating a Function Handle

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

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

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

You can use this function handle to calculate numerically:

```
ht(.5,.5)

ans =
    0.6089
```

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

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

ans =
    0.6089    0.9954
```

Control the Order of Variables

`matlabFunction` generates input variables in alphabetical order from a symbolic expression. That is why the function handle in "Generating a Function Handle" on page 2-220 has `x` before `y`:

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

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

```
syms x y z
r = sqrt(x^2 + 3*y^2 + 5*z^2);
ht1 = matlabFunction(tanh(r), 'vars', [y x z])

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

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

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

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

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

Generate a File

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

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

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

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

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

t2 = x.^2;
t3 = tan(y);
t4 = t2.*x;
t5 = t.^2;
t6 = t5 + 1;
t7 = 1./y;
t8 = t6.*t7.*x;
t9 = t3 + t4;
t10 = 1./t9;
F = [-(t10.*(t3 - t4))./t6,t8; t8,- t10.*(3.*t3 - 3.*t2.*x) - 1];
```

`matlabFunction` generates many intermediate variables. This is called *optimized* code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. Intermediate variables can make the resulting code more efficient by reusing intermediate expressions (such as `t4`, `t6`, `t8`, `t9`, and `t10` in the calculation of `F`). Using intermediate variables can make the code easier to read by keeping expressions short.

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

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

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

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

Name Output Variables

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

```
syms x y t
z = (x^3 - tan(y))/(x^3 + tan(y));
w = z/(1 + t^2);
F = [w, (1 + t^2)*x/y; (1 + t^2)*x/y,3*z - 1];
```

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

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

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

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

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

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

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

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

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

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

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

Convert MuPAD Expressions

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

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

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

```
function f = new_function(x,y)
```

```
%NEW_FUNCTION
%    F = NEW_FUNCTION(X,Y)

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

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

Generate MATLAB Function Blocks

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

If you work in the MuPAD Notebook app, see "Create MATLAB Function Blocks from MuPAD Expressions" on page 3-50.

Generate and Edit a Block

Suppose, you want to create a model involving the symbolic expression `r = sqrt(x^2 + y^2)`. Before you can convert a symbolic expression to a MATLAB Function block, create an empty model or open an existing one:

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

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

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

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

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

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

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

Control the Order of Input Ports

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

```
syms x y
```

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

Name the Output Ports

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

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

Convert MuPAD Expressions

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

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

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

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

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

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

Generate Simscape Equations

Simscape software extends the Simulink product line with tools for modeling and simulating multidomain physical systems, such as those with mechanical, hydraulic, pneumatic, thermal, and electrical components. Unlike other Simulink blocks, which represent mathematical operations or operate on signals, Simscape blocks represent physical components or relationships directly. With Simscape blocks, you build a model of a system just as you would assemble a physical system. For more information about Simscape software see “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 app, see “Create Simscape Equations from MuPAD Expressions” on page 3-52.

Convert Algebraic and Differential Equations

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

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

The solution is:

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

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

```
simscapeEquation(s)
```

`simscapeEquation` generates the following code:

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

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

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

```
syms a x(t)
simscapeEquation(diff(x), -a^2*x)
```

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

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

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

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

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

```
simscapeEquation(s)
```

```
ans =
s == if (x ~= 0.0),
    -atan(1.0/x^2*(u-1.0))+atan(1.0/x^2*(u+1.0));
  else
    NaN;
  end
```

Clear the assumption that x is real:

```
syms x clear
```

Convert MuPAD Equations

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

Limitations

The equation section of a Simscape component file supports a limited number of functions. For details and the list of supported functions, see Simscape "equations". If a symbolic equation contains the functions that the equation section of a Simscape component file does not support. `simscapeEquation` cannot correctly convert these equations to Simscape equations. Such expressions do not trigger an error message. The following types of expressions are prone to invalid conversion:

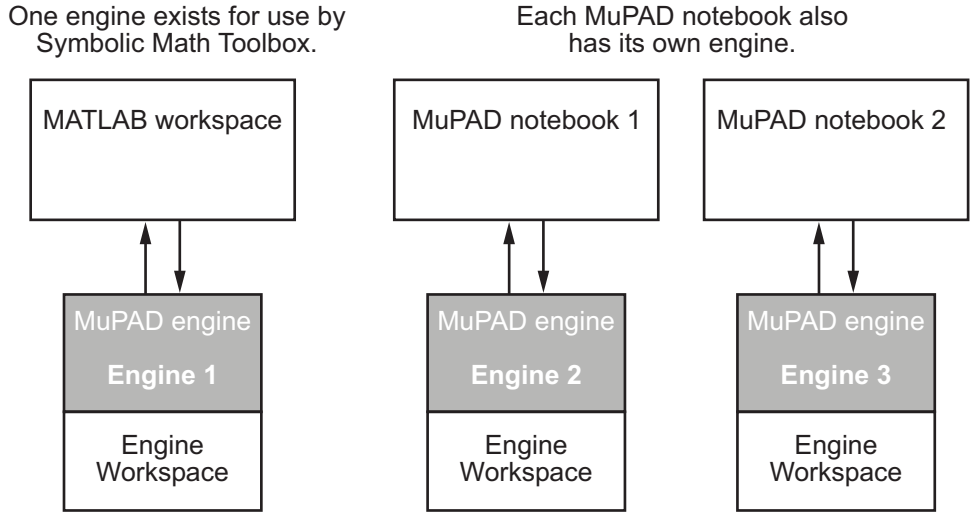
- Expressions with infinities
- Expressions returned by `evalin` and `feval`

MuPAD in Symbolic Math Toolbox

MuPAD Engines and MATLAB Workspace

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

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



The engine workspace associated with the MATLAB workspace is generally empty, except for assumptions you make about variables. For details, see “Clear Assumptions and Reset the Symbolic Engine” on page 3-43.

Create MuPAD Notebooks

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 Command Window using MuPAD notebooks as an auxiliary tool. This approach implies that you create a MuPAD notebook, and then execute it, transfer data and results, or close it from the MATLAB Command Window.
- Perform your computations and obtain the results in the MuPAD Notebook app. This approach implies that you use the MATLAB Command Window 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 Apps Tab

To create a new blank notebook:

1. On the MATLAB Toolstrip, click the **Apps** tab.
2. On the **Apps** tab, click the down arrow at the end of the **Apps** section.
3. Under **Math, Statistics and Optimization**, click the **MuPAD Notebook** button.

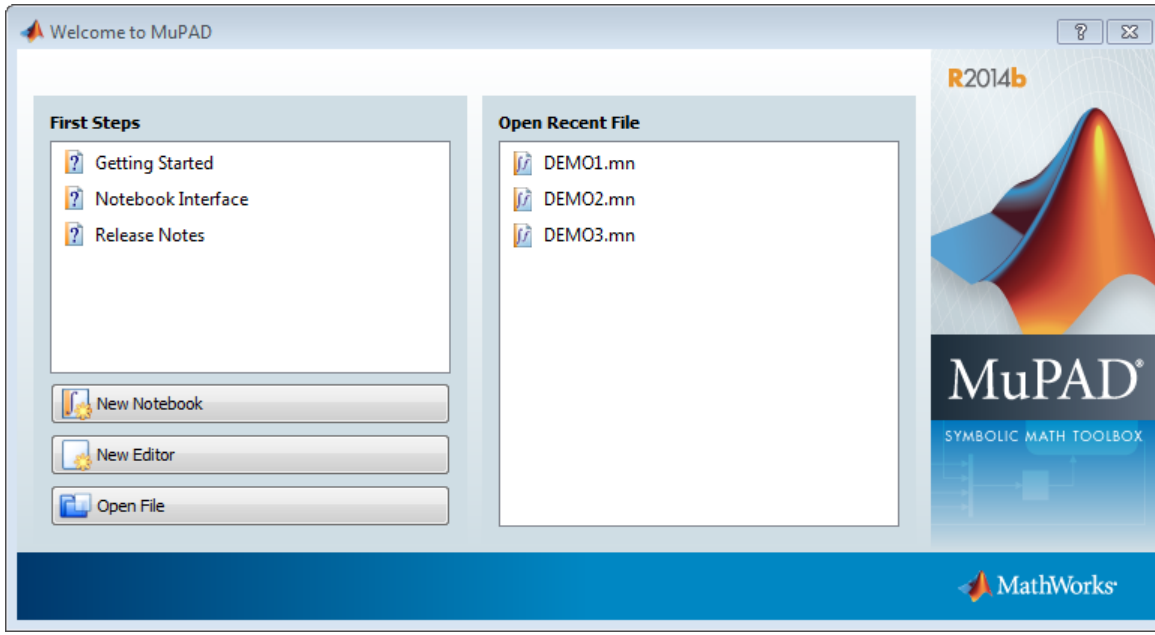
To create several MuPAD notebooks, click the **MuPAD Notebook** button repeatedly.

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 Command Window:

- 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

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 Command Window using MuPAD notebooks as an auxiliary tool. This approach implies that you open a MuPAD notebook, and then execute it, transfer data and results, or close it from the MATLAB Command Window. 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 implies that you use the MATLAB Command Window 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 Command Window 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

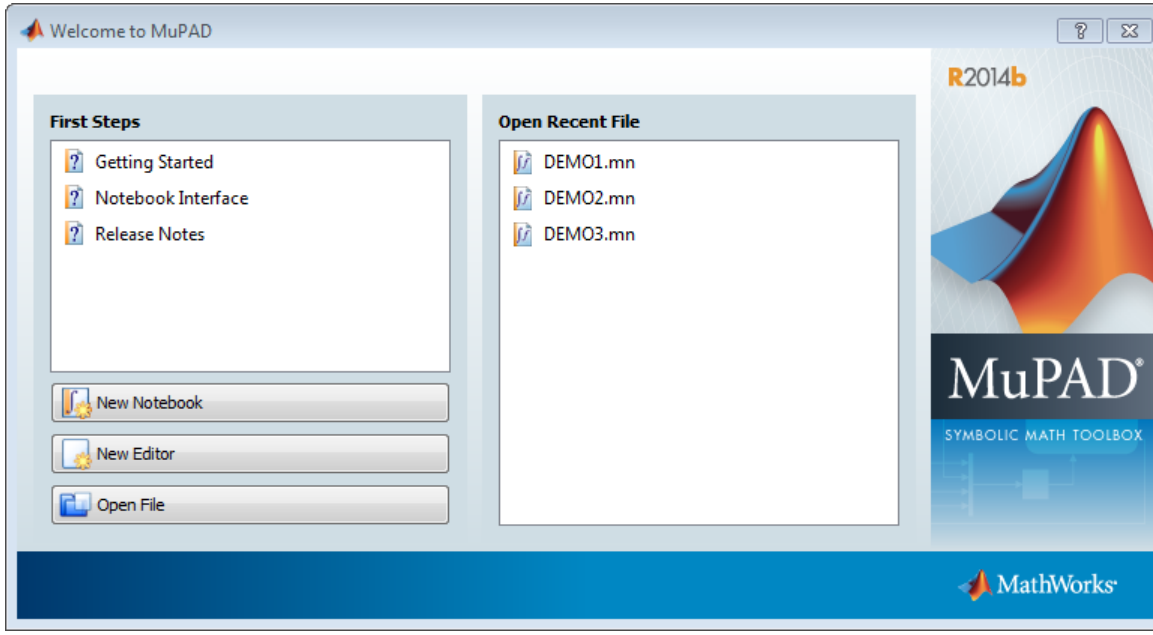
Open an existing MuPAD notebook file by using `open` in the MATLAB Command Window:

```
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 Command Window. 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 Command Window.

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

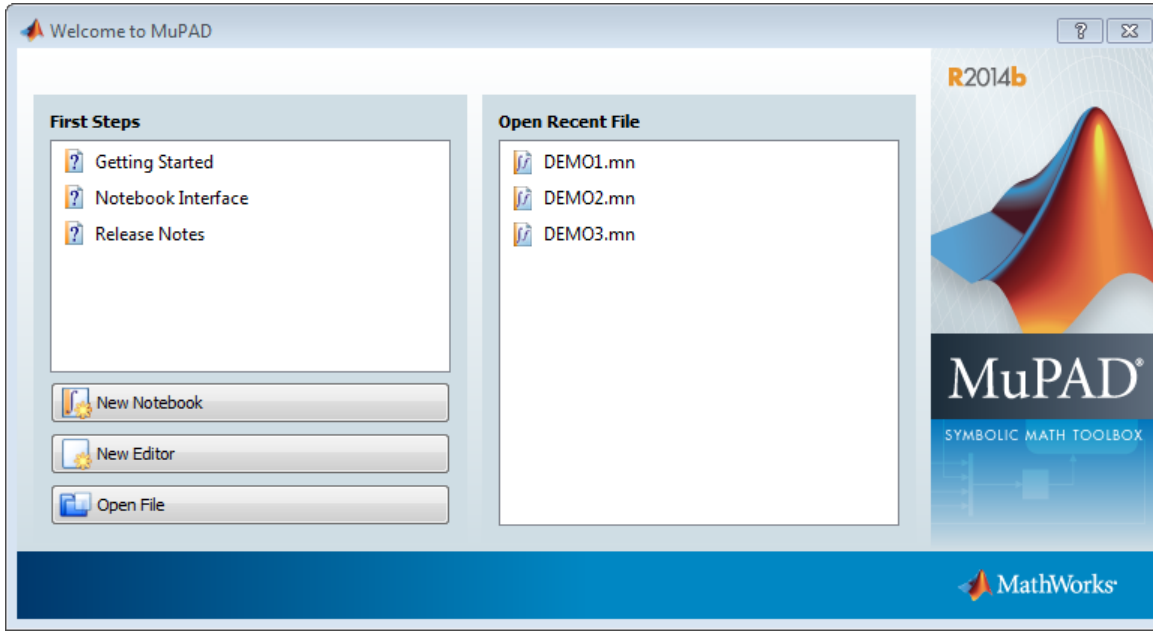
Open an existing MuPAD file by using `open` in the MATLAB Command Window:

```
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

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

When you open a saved MuPAD notebook file, the notebook displays the results (outputs), but the engine does not “remember” them. For example, suppose 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

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:

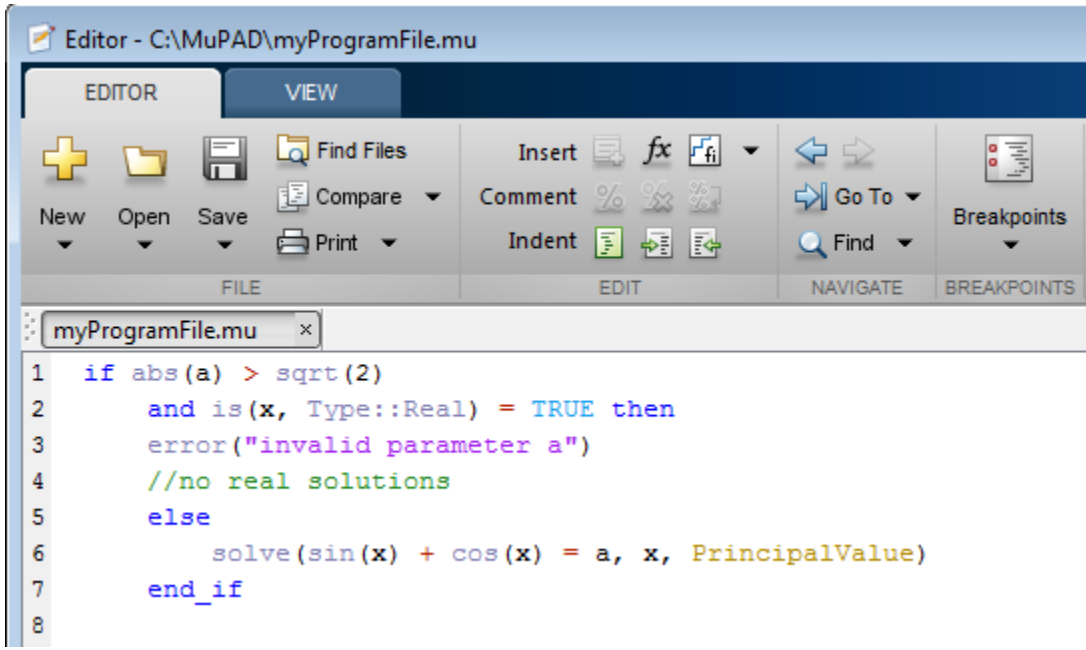
```
close(allMuPADNotebooks,'force')
```

Edit MuPAD Code in MATLAB Editor

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

Note: The MATLAB Editor cannot evaluate or debug MuPAD code.

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



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`.

Notebook Files and Program Files

The two main types of files in MuPAD are:

- Notebook files, or notebooks
- Program files

A *notebook file* has the extension `.mn` and lets you store the result of the work performed in the MuPAD Notebook app. 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 app. 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 3-39.

Source Code of the MuPAD Library Functions

You can display the source code of the MuPAD built-in library functions. If you work in the MuPAD Notebook app, enter `expose(name)`, where `name` is the library function name. The MuPAD Notebook app 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

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

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

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

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

Common Tasks in MATLAB and MuPAD Syntax

Task	MATLAB Syntax	MuPAD Syntax
Assignment	`=`	`:=`
List variables	`whos`	`anames(All, User)`
Numerical value of expression	`double(`*`expression`*`)`	`float(`*`expression`*`)`
Suppress output	`;`	`:`
Enter matrix	`[x11,x12,x13; x21,x22,x23]`	`matrix([[x11,x12,x13], [x21,x22,x23]])`
`{a,b,c}`	cell array	set
Auto-completion	**Tab**	**Ctrl+space bar**
Equality, inequality comparison	`==`, `~=`	`=`, `<>`

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

MATLAB vs. MuPAD Expressions

MATLAB Expression	MuPAD Expression
`Inf`	`infinity`

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

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

	Symbolic Math Toolbox Definition	MuPAD Definition
Exponential integral	$\mathrm{expint}(x) = -\mathrm{Ei}(-x) =$ $\int_x^{\infty} \frac{\exp(-t)}{t} dt \text{ for } x > 0 =$ $\mathrm{Ei}(1, x)$.	$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt \text{ for } x < 0.$ $\mathrm{Ei}(n, x) = \int_1^{\infty} \frac{\exp(-xt)}{t^n} dt.$ 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

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:

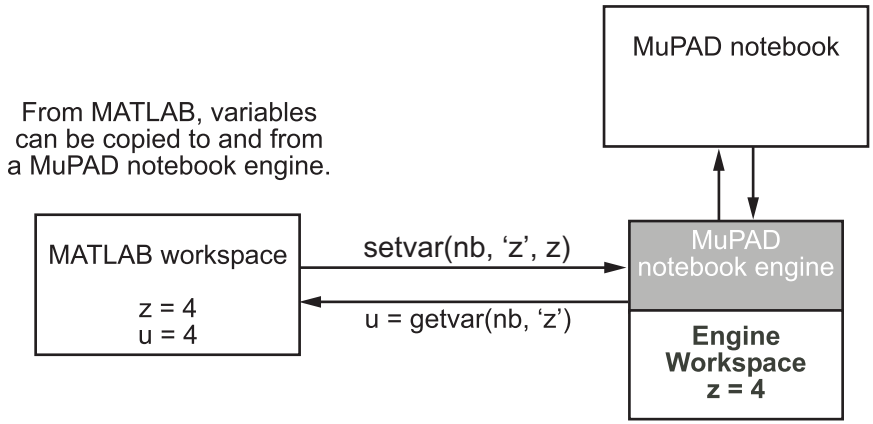
  ```
  MATLABvar = getVar(notebook_handle,'variable');
  ```

  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:

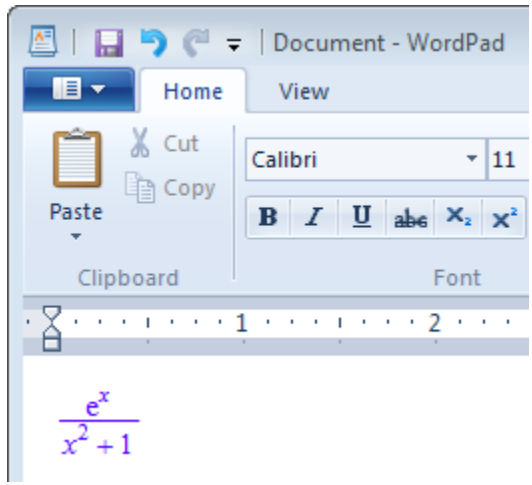
```
y := exp(x)/(1 + x^2)
```

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

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.



Reserved Variable and Function Names

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

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

Conflicts Caused by MuPAD Function Names

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

Conflicts Caused by Syntax Conversions

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

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

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

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

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

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

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

ans =
4
```

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

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

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

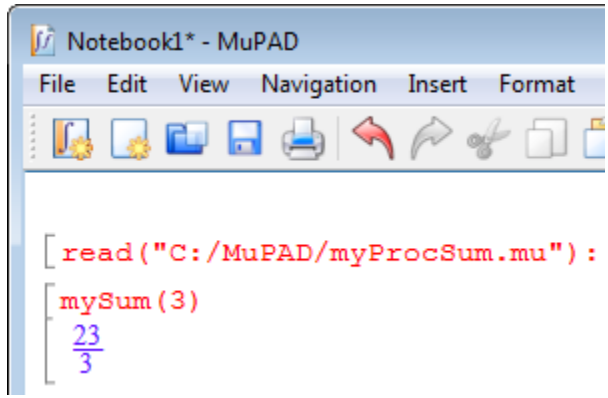
Save this code in the `myProcSum.mu` program file in the `C:/MuPAD` folder. Again, you must read the program file into the symbolic engine, and you cannot use `read` because the code defines an alias. Enter these commands in the MATLAB Command Window:

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

```
Error using mupadengine/feval (line 157)
MuPAD error: Error: The aliased identifier is invalid at this place.
[file C:/MuPAD/myProcSum.mu, line 4, col 7]
  Evaluating: read
  Reading File: C:/MuPAD/myProcSum.mu
```

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

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



Call Built-In MuPAD Functions from MATLAB

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 strings. For example, to find the tenth element in the Fibonacci sequence, enter:

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

```
z =
55
```

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

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

ans =
0
```

The numeric solver `fsolve` returns this result:

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

ans =
x == 3.069980123839465465438654874667 8
```

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

evalin vs. feval

The `evalin(symengine,...)` function causes the MuPAD engine to evaluate a string. Since the MuPAD engine workspace is generally empty, expressions returned by `evalin(symengine,...)` are not simplified or evaluated according to their definitions in the MATLAB workspace. For example:

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

ans =
cos(y)
```

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

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

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

ans =
```

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

Floating-Point Arguments of evalin and feval

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

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

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

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

```
format long
double(y)
```

```
ans =
    0.028005841168289
```

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

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

```
ans =
0.028005841168289177028337498391181
```

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

```
format short
```

Computations in MATLAB Command Window vs. MuPAD Notebook App

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

Working in the MATLAB Command Window lets you perform all symbolic computations using the familiar MATLAB language. The toolbox contains hundreds of MATLAB symbolic functions for common tasks, such as differentiation, integration, simplification, transforms, and equation solving. If your task requires a few specialized symbolic functions not available directly from this interface, you can use `evalin` or `feval` to call MuPAD functions. See “Call Built-In MuPAD Functions from MATLAB” on page 3-31.

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

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

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

Learning the MuPAD language and using the MuPAD Notebook app for your symbolic computations provides the following benefits.

Results Displayed in Typeset Math

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

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

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

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

Graphics and Animations

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

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

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

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

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

You can export animations as AVI files (on Windows systems), as animated GIF files, or as sequences of static images.

More Functionality in Specialized Mathematical Areas

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

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

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

More Options for Common Symbolic Functions

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

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

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

Possibility to Expand Existing Functionality

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

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

Use Your Own MuPAD Procedures

Write MuPAD Procedures

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

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

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

By default, a MuPAD procedure returns the result of the last executed command. You can force a procedure to return another result by using `return`. In both cases, a procedure returns only one result. To get multiple results from a procedure, 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 3-19. 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 app 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 Command Window, you still need to execute that procedure before calling it. See "Call Your Own MuPAD Procedures" on page 3-40.

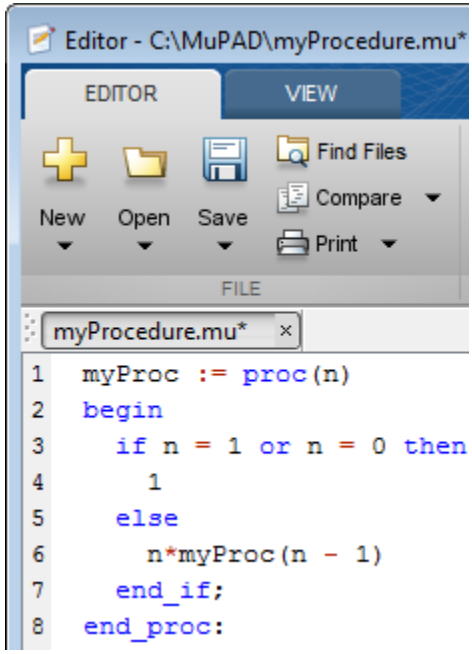
Use Startup Commands and Scripts

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.



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

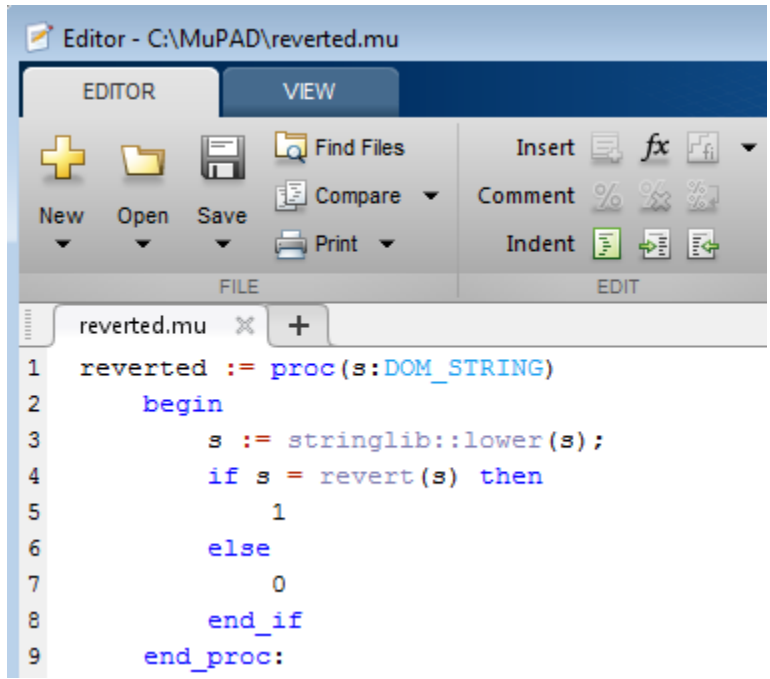
Return to the MATLAB Command Window. Before calling the procedure at the MATLAB command line, enter:

```
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 string 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 string. For example, this MuPAD procedure converts a string to lowercase and checks if reverting that string changes it.



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 string argument to the procedure, use double quotes inside single quotes.

```
feval(symengine, 'reverted', '"Abccba"')
```

```
ans =
1
```

Clear Assumptions and Reset the Symbolic Engine

The symbolic engine workspace associated with the MATLAB workspace is usually empty. The MATLAB workspace tracks the values of symbolic variables, and passes them to the symbolic engine for evaluation as necessary. However, the symbolic engine workspace contains all assumptions you make about symbolic variables, such as whether a variable is real, positive, integer, greater or less than some value, and so on. These assumptions can affect solutions to equations, simplifications, and transformations, as explained in "Effects of Assumptions on Computations" on page 3-45.

Note: These commands

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

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

If you make an assumption about the nature of a variable, for example, using the commands

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

or

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

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

```
syms x clear
```

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

If you reset the symbolic engine by entering the command

```
reset(symengine)
```

MATLAB no longer recognizes any symbolic variables that exist in the MATLAB workspace. Clear the variables with the `clear` command, or renew them with the `syms` or `sym` command.

This example shows how the MATLAB workspace and the symbolic engine workspace respond to a sequence of commands.

Step	Command	MATLAB Workspace	MuPAD Engine Workspace

1	`syms x positive`
	or
	`syms x;`
	`assume(x > 0)`
		`x`	`x > 0`

2	`clear x`	empty	`x > 0`
3	`syms x`	`x`	`x > 0`
4	`syms x clear`	`x`	empty

Check Assumptions Set On Variables

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

```
assumptions(x)
```

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

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

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

```
ans =
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`:

```
syms 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
 -i
  i
```

Now solve the same equation assuming that `x` is real:

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

```
ans =
 -1
```

```
  1
```

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

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

ans =
  1
```

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

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

ans =
[ 0 < x, in(x, 'real')]
ans =
1
```

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

```
syms x clear
assumptions(x)

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

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

Create MATLAB Functions from MuPAD Expressions

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

If you work in the MATLAB Command Window, see “Generate MATLAB Functions” on page 2-220.

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 Command Window:

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

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

   ```
   h = matlabFunction(S)
   ```

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

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

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

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

```
1     function S = cartesian(a,b,r)
2     %CARTESIAN
3     %    S = CARTESIAN(A,B,R)
4
5 -   t2 = sin(a);
6 -   t3 = cos(b);
7 -   S = [r.*t2.*t3;r.*t2.*sin(b);r.*t3];
```

Generate MATLAB Code in a MuPAD Notebook

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

1. In the MuPAD Notebook 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 string representing matrix `S`:

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

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

   ```
   1   function S = cartesian(r, a, b)
   2   %CARTESIAN Converts spherical coordinates
   3   % to Cartesian coordinates.
   4   %   Angles are measured in radians.
   5
   6 -   S = zeros(3,1);
   7 -   S(1,1) = r*cos(b)*sin(a);
   8 -   S(2,1) = r*sin(a)*sin(b);
   9 -   S(3,1) = r*cos(b);
   10 -  end
   ```

Create MATLAB Function Blocks from MuPAD Expressions

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

If you work in the MATLAB Command Window, see "Generate MATLAB Function Blocks" on page 2-225.

The MuPAD Notebook app 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 3-48.

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

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

```
notebook_handle = mupad;
```

In this notebook, create the following symbolic expression:

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

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

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

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

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

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

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

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

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

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

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

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

Create Simscape Equations from MuPAD Expressions

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

If you work in the MATLAB Command Window, see “Generate Simscape Equations” on page 2-227.

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 app 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(D(y)(t) - y(t)^2, y(t))
   ```

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

   ```
   simscapeEquation(s)
   ```

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

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

Functions — Alphabetical List

abs

Absolute value of real or complex value

Syntax

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

Description

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

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

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 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
```

More About

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

abs | angle | imag | real | sign | signIm

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]
```

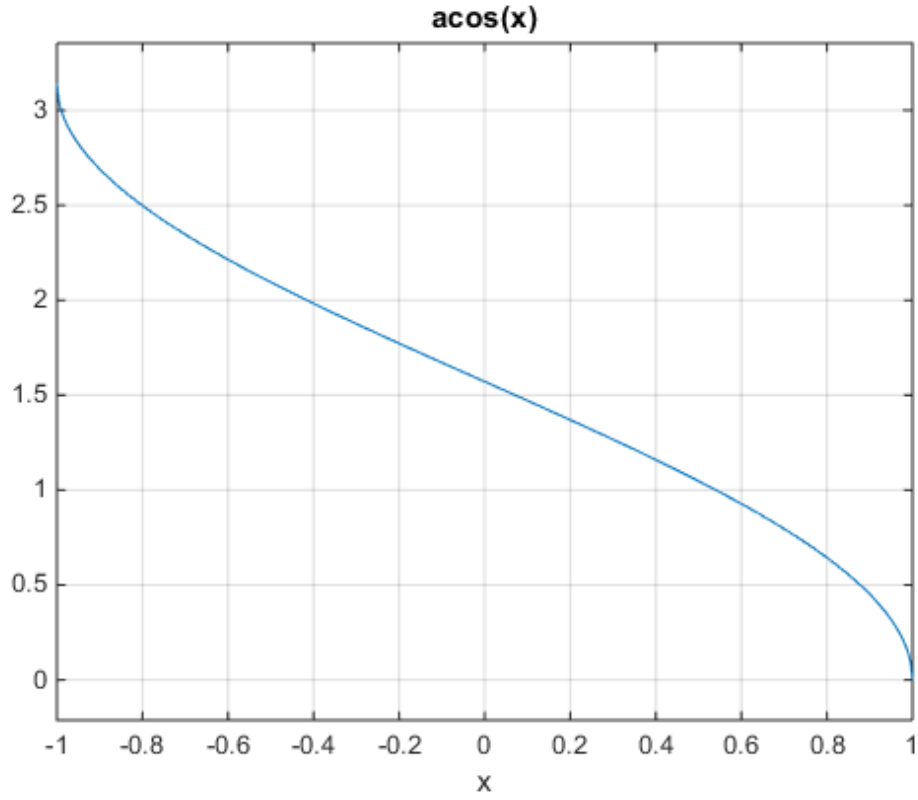
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 3.1415926535897932384626433832795,...
1.9106332362490185563277142050315,...
2.0943951023931954923084289221863,...
1.3181160716528179657456642546 46,...
1.0471975511965977461542144610932,...
0.5235987755982988730771072305 4658,...
0]
```

Plot the Inverse Cosine Function

Plot the inverse cosine function on the interval from -1 to 1.

```
syms x
ezplot(acos(x), [-1, 1])
grid on
```



Handle Expressions Containing the 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)*i)*i
```

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`

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*i, (pi*i)/2, acosh(1/6), (pi*i)/3, 0, acosh(2)]
```

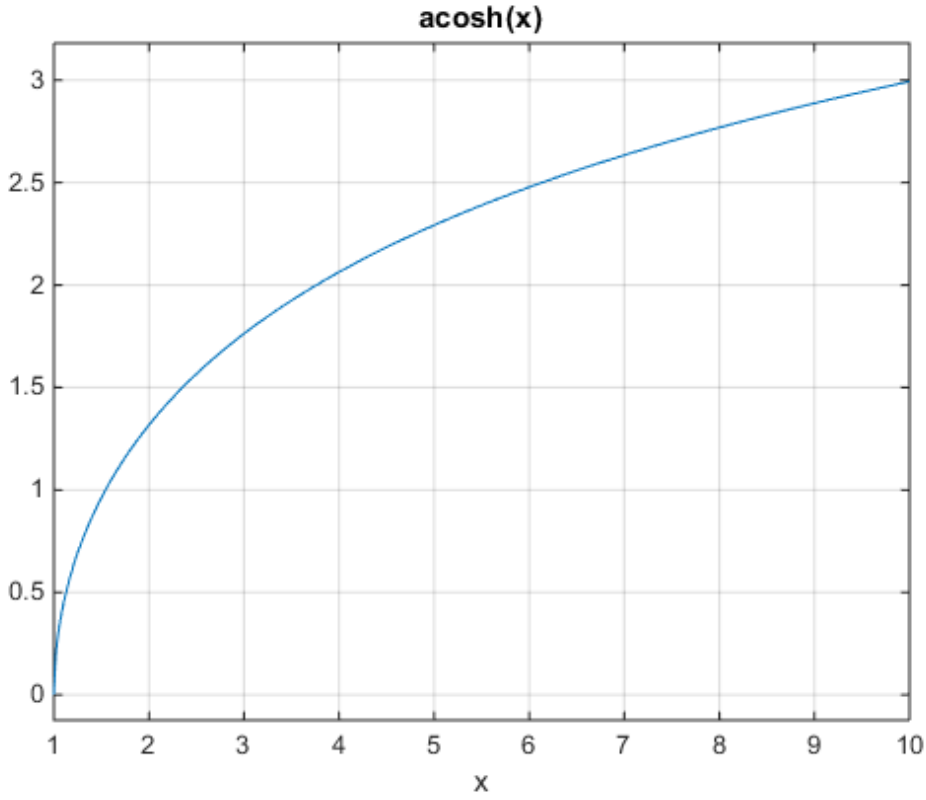
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 3.1415926535897932384626433832795*i,...
1.5707963267948966192313216916398*i,...
1.4033482475752072886780470855961*i,...
1.0471975511965977461542144610932*i,...
0,...
1.3169578969248167086250463473078]
```

Plot the Inverse Hyperbolic Cosine Function

Plot the inverse hyperbolic cosine function on the interval from 1 to 10.

```
syms x
ezplot(acosh(x), [1, 10]);
grid on
```



Handle Expressions Containing the 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:

```
syms x
diff(acosh(x), x)
diff(acosh(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 cosine function:

```
int(acosh(x), x)
```

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

Find the Taylor series expansion of `acosh(x)` for `x > 1`:

```
assume(x > 1)
taylor(acosh(x), x)
```

```
ans =
x^5*((3*i)/40) + x^3*(i/6) + x*i + pi*(-i/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^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

`acoth` | `acsch` | `asech` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

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]
```

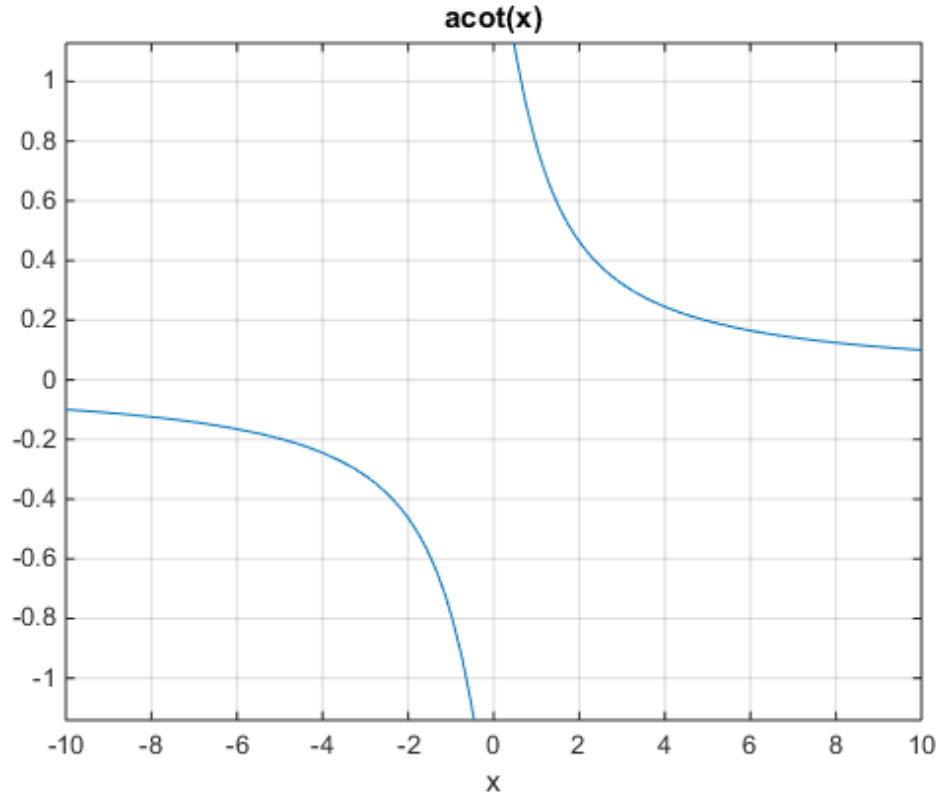
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.78539816339744830961566084581988,...
-1.24904577239825442582991707728 11,...
-1.04719755119659774615421446109 32,...
1.10714871779409050301706546017 85,...
0.78539816339744830961566084581988,...
0.52359877559829887307710723054658]
```

Plot the Inverse Cotangent Function

Plot the inverse cotangent function on the interval from -10 to 10.

```
syms x
ezplot(acot(x), [-10, 10]);
grid on
```



Handle Expressions Containing the 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 - i/x)*i)/2 - (log(i/x + 1)*i)/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`

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*i)/2, acoth(1/2), Inf, acoth(pi/2)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

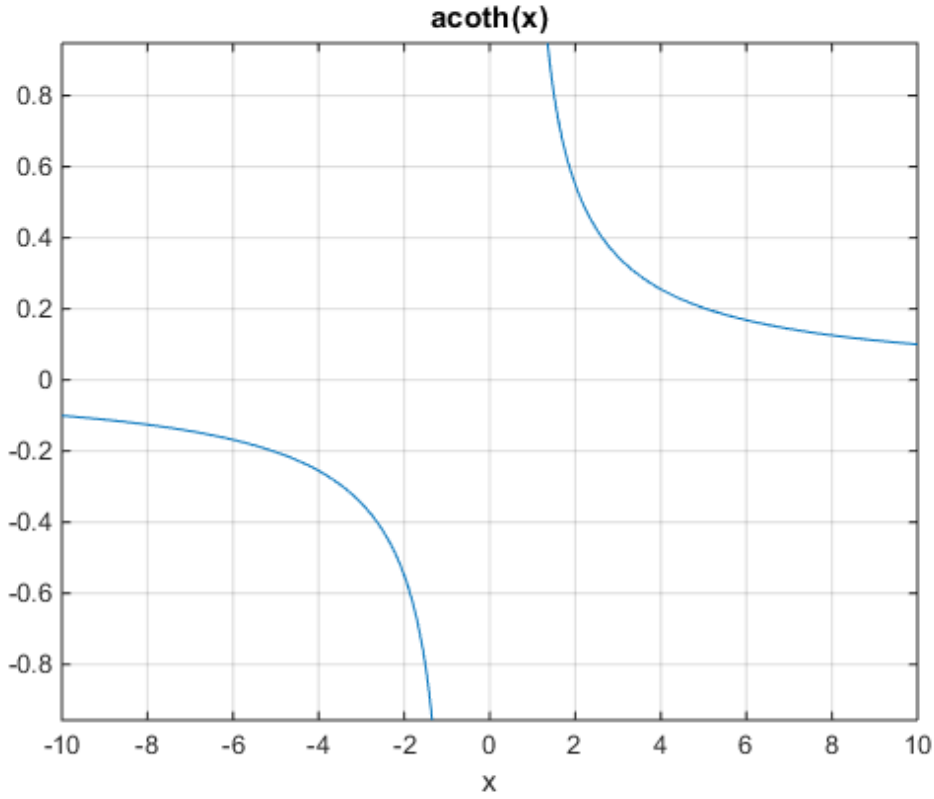
```
vpa(symA)
```

```
ans =
[ -0.752469267141927159162043478002 51,...
Inf,...
-1.570796326794896619231321691639 8*i,...
0.549306144334054845697622618461 26...
 - 1.570796326794896619231321691639 8*i,...
Inf,...
0.752469267141927159162043478002 51]
```

Plot the Inverse Hyperbolic Cotangent Function

Plot the inverse hyperbolic cotangent function on the interval from -10 to 10.

```
syms x
ezplot(acoth(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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*(-i/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`

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)]
```

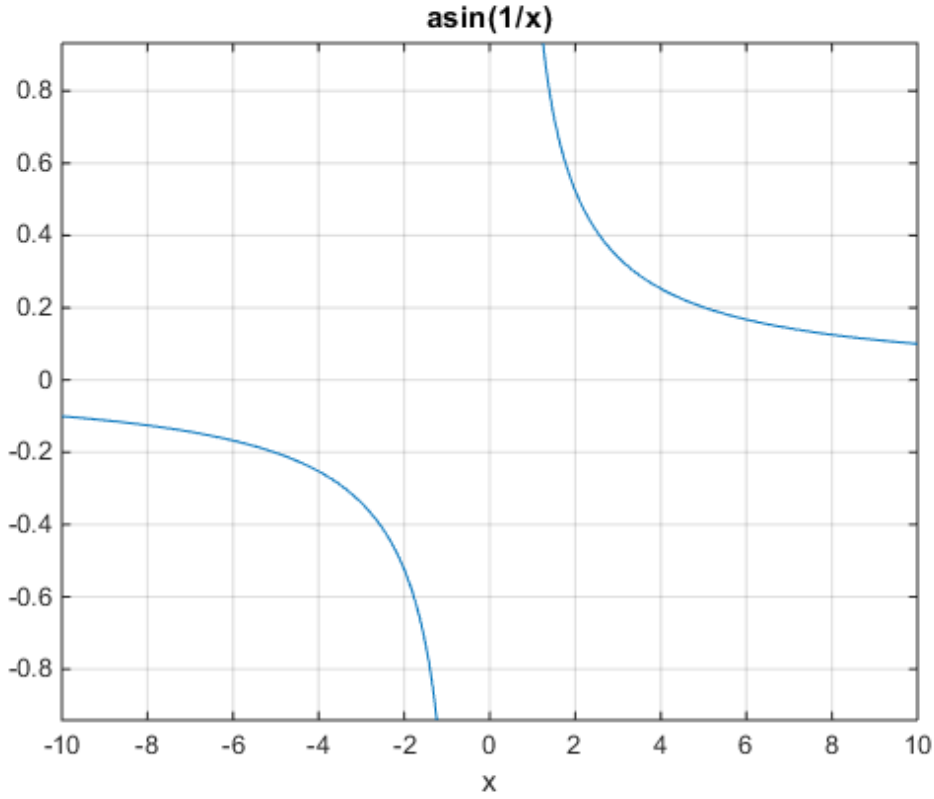
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.523598775598298873077107230546 58,...
Inf,...
1.047197551196597746154214461 0932,...
1.570796326794896619231321691 6398...
 - 1.316957896924816708625046347308*i,...
1.570796326794896619231321691 6398,...
0.201357920790330791455125552217 62]
```

Plot the Inverse Cosecant Function

Plot the inverse cosecant function on the interval from -10 to 10.

```
syms x
ezplot(acsc(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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) + acosh(x)*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(i/x + (1 - 1/x^2)^(1/2))*i
```

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`

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*i)/6, Inf, -(pi*i)/3, asinh(2), -(pi*i)/2, asinh(1/3)]
```

Use vpa to approximate symbolic results with floating-point numbers:

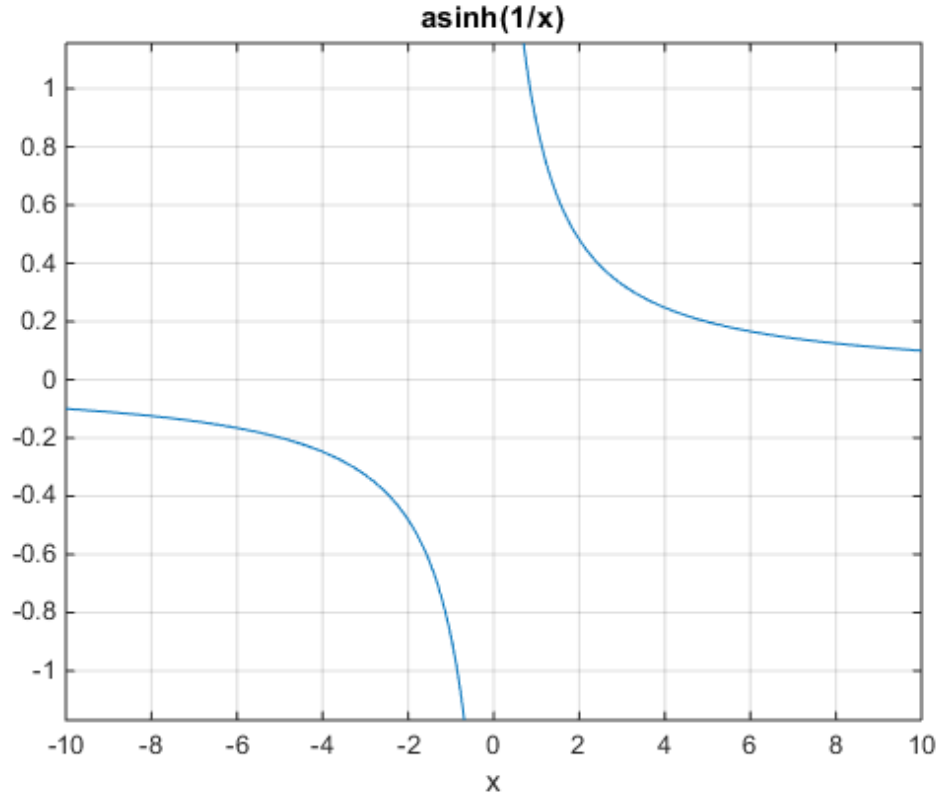
```
vpa(symA)
```

```
ans =
[ 0.523598775598298873077107230546 58*i,...
Inf,...
-1.047197551196597746154214461 0932*i,...
1.443635475178810342493276740 2731,...
-1.570796326794896619231321691 6398*i,...
0.327450150237258443322535259 98826]
```

Plot the Inverse Hyperbolic Cosecant Function

Plot the inverse hyperbolic cosecant function on the interval from -10 to 10.

```
syms x
ezplot(acsch(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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`

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 =
     1     1     1
     1     1     1
     1     1     1
```

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

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

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

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

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

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

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

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

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

More About

Adjoint of a Square Matrix

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

Cofactor of a Matrix

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

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

A_{ij} is the submatrix of A obtained from A by removing the i-th row and j-th column.

See Also

`det` | `inv` | `linalg::adjoint` | `rank`

airy

Airy function

Syntax

```
airy(x)
airy(0,x)
airy(1,x)
airy(2,x)
airy(3,x)
airy(n,x)
```

Description

`airy(x)` returns the Airy function of the first kind, Ai(*x*).

`airy(0,x)` is equivalent to `airy(x)`.

`airy(1,x)` returns the derivative of the Airy function of the first kind, Ai′(*x*).

`airy(2,x)` returns the Airy function of the second kind, Bi(*x*).

`airy(3,x)` returns the derivative of the Airy function of the second kind, Bi′(*x*).

`airy(n,x)` returns a vector or matrix of derivatives of the Airy function.

Input Arguments

`x`

Symbolic number, variable, expression, or function, or a vector or matrix of symbolic numbers, variables, expressions, or functions. If `x` is a vector or matrix, `airy` returns the Airy functions for each element of `x`.

`n`

Vector or matrix of numbers `0`, `1`, `2`, and `3`.

Examples

Solve this second-order differential equation. The solutions are the Airy functions of the first and the second kind.

```
syms y(x)
dsolve(diff(y, 2) - x*y == 0)

ans =
C2*airy(0, x) + C3*airy(2, x)
```

Verify that the Airy function of the first kind is a valid solution of the Airy differential equation:

```
syms x
simplify(diff(airy(0, x), x, 2) - x*airy(0, x)) == 0

ans =
     1
```

Verify that the Airy function of the second kind is a valid solution of the Airy differential equation:

```
simplify(diff(airy(2, x), x, 2) - x*airy(2, x)) == 0

ans =
     1
```

Compute the Airy functions for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[airy(1), airy(1, 3/2 + 2*i), airy(2, 2), airy(3, 1/101)]

ans =
   0.1353             0.1641 + 0.1523i   3.2981             0.4483
```

Compute the Airy functions for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `airy` returns unresolved symbolic calls.

```
[airy(sym(1)), airy(1, sym(3/2 + 2*i)), airy(2, sym(2)), airy(3, sym(1/101))]

ans =
[ airy(0, 1), airy(1, 3/2 + 2*i), airy(2, 2), airy(3, 1/101)]
```

For symbolic variables and expressions, `airy` also returns unresolved symbolic calls:

```
syms x y
[airy(x), airy(1, x^2), airy(2, x - y), airy(3, x*y)]

ans =
[ airy(0, x), airy(1, x^2), airy(2, x - y), airy(3, x*y)]
```

Compute the Airy functions for $x = 0$. The Airy functions have special values for this parameter.

```
airy(sym(0))

ans =
3^(1/3)/(3*gamma(2/3))

airy(1, sym(0))

ans =
-(3^(1/6)*gamma(2/3))/(2*pi)

airy(2, sym(0))

ans =
3^(5/6)/(3*gamma(2/3))

airy(3, sym(0))

ans =
(3^(2/3)*gamma(2/3))/(2*pi)
```

If you do not use `sym`, you call the MATLAB `airy` function that returns numeric approximations of these values:

```
[airy(0), airy(1, 0), airy(2, 0), airy(3, 0)]

ans =
    0.3550   -0.2588    0.6149    0.4483
```

Differentiate the expressions involving the Airy functions:

```
syms x y
diff(airy(x^2))
diff(diff(airy(3, x^2 + x*y -y^2), x), y)
```

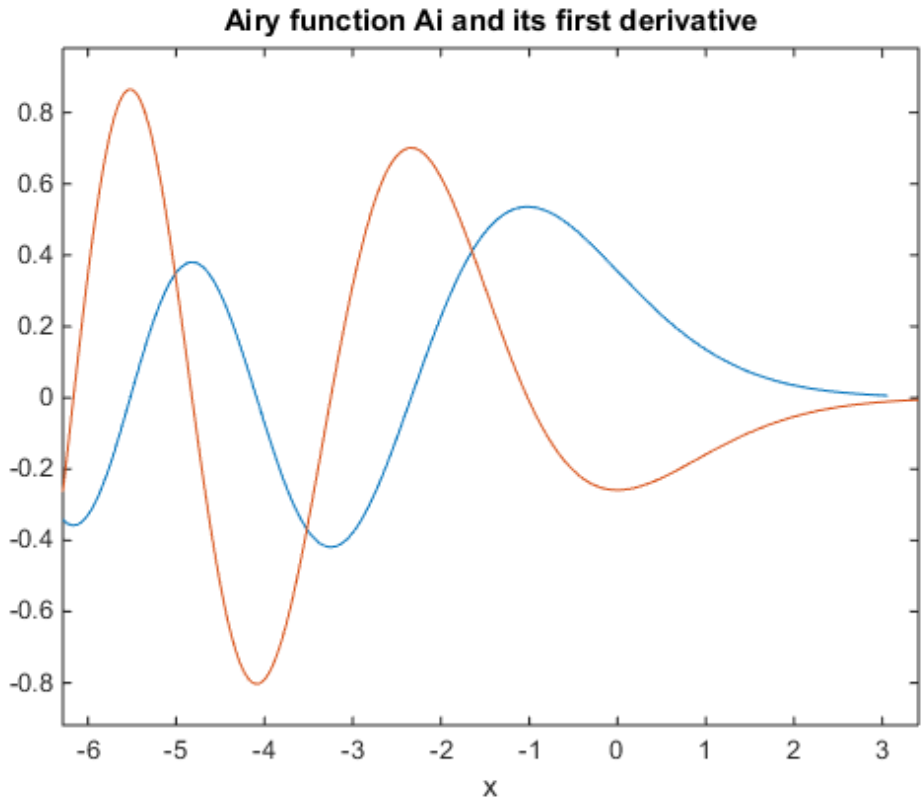
```
ans =
2*x*airy(1, x^2)

ans =
airy(2, x^2 + x*y - y^2)*(x^2 + x*y - y^2) +...
airy(2, x^2 + x*y - y^2)*(x - 2*y)*(2*x + y) +...
airy(3, x^2 + x*y - y^2)*(x - 2*y)*(2*x + y)*(x^2 + x*y - y^2)
```

Compute the Airy function of the first kind for the elements of matrix A:

```
syms x
A = [-1, 0; 0, x];
airy(A)

ans =
[            airy(0, -1), 3^(1/3)/(3*gamma(2/3))]
[ 3^(1/3)/(3*gamma(2/3)),             airy(0, x)]
```

Plot the Airy function Ai(x) and its derivative Ai'(x):

```
syms x
ezplot(airy(x))
hold on
ezplot(airy(1,x))

title('Airy function Ai and its first derivative')
hold off
```



More About

Airy Functions

The Airy functions Ai(x) and Bi(x) are linearly independent solutions of this differential equation:

$$\frac{\partial^2 y}{\partial x^2} - xy = 0$$

Tips

- Calling `airy` for a number that is not a symbolic object invokes the MATLAB `airy` function.
- When you call `airy` with two input arguments, 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, `airy(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.

References

Antosiewicz, H. A. “Bessel Functions of Fractional Order.” *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`besseli` | `besselj` | `besselk` | `bessely`

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 =
     1
```

Create this matrix of symbolic equations and inequalities:

```
syms x
M = [x == x, x == abs(x); abs(x) >= 0, x ~= 2*x]
```

```
M =
[      x == x, x == abs(x)]
[ 0 <= abs(x),    x ~= 2*x]
```

Use `all` to test equations and inequalities of this matrix. By default, `all` tests whether all elements of each column are valid for all possible values of variables. If all equations and inequalities in the column are valid (return logical `1`), then `all` returns logical `1` for that column. Otherwise, it returns logical `0` for the column. Thus, it returns `1` for the first column and `0` for the second column:

```
all(M)
```

```
ans =
     1     0
```

Create this matrix of symbolic equations and inequalities:

```
syms x
M = [x == x, x == abs(x); abs(x) >= 0, x ~= 2*x]
```

```
M =
[      x == x, x == abs(x)]
[ 0 <= abs(x),    x ~= 2*x]
```

For matrices and multidimensional arrays, `all` can test all elements along the specified dimension. To specify the dimension, use the second argument of `all`. For example, to test all elements of each column of a matrix, use the value 1 as the second argument:

```
all(M, 1)
```

```
ans =
     1     0
```

To test all elements of each row, use the value 2 as the second argument:

```
all(M, 2)
```

```
ans =
     0
     1
```

Test whether all elements of this vector return logical 1s. Note that `all` also converts all numeric values outside equations and inequalities to logical 1s and 0s. The numeric value 0 becomes logical 0:

```
syms x
all([0, x == x])
```

```
ans =
     0
```

All nonzero numeric values, including negative and complex values, become logical 1s:

```
all([1, 2, -3, 4 + i, x == x])
```

```
ans =
     1
```

More About

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` | `logical` | `not` | `or` | `xor`

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 a Handle to an 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 3-24

- “Close MuPAD Notebooks from MATLAB” on page 3-16

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`

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 `logical` or `isAlways`:

```
logical(x1)
isAlways(x2)
```

```
ans =
     1

ans =
     0
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)
```

```
s1 =
```

```
TRUE

s2 =
FALSE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)

ans =
     1

ans =
     0
```

The recommended approach to define a range of values is using `&`. Nevertheless, you can define a range of values of a variable as follows:

```
syms x
range = 0 < x < 1;
```

Now if you want to replace variable `x` with numeric values, use symbolic numbers instead of MATLAB double-precision numbers. To create a symbolic number, use `sym`

```
x1 = subs(range, x, sym(1/2))
x2 = subs(range, x, sym(10))

x1 =
(0 < 1/2) < 1

x2 =
(0 < 10) < 1
```

To evaluate these inequalities to logical `1` or `0`, use `isAlways`. Note that `logical` cannot resolve such inequalities.

```
isAlways(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

More About

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.

See Also

`all` | `any` | `isAlways` | `logical` | `not` | `or` | `xor`

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]
```

Alternatives

For real `X` and `Y` such that `Z = X + Y*i`, the call `angle(Z)` is equivalent to `atan2(Y,X)`.

More About

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`.

See Also

`atan2` | `conj` | `imag` | `real` | `sign` | `signIm`

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 =
     1
```

Create this matrix of symbolic equations and inequalities:

```
syms x real
M = [x == 2*x, x == abs(x); abs(x) >= 0, x == 2*x]
```

```
M =
[    x == 2*x, x == abs(x)]
[ 0 <= abs(x),    x == 2*x]
```

Use any to test equations and inequalities of this matrix. By default, any tests whether any element of each column is valid for all possible values of variables. If at least one equation or inequality in the column is valid (returns logical 1), then any returns logical 1 for that column. Otherwise, it returns logical 0 for the column. Thus, it returns 1 for the first column and 0 for the second column:

```
any(M)
```

```
ans =
     1     0
```

Create this matrix of symbolic equations and inequalities:

```
syms x real
M = [x == 2*x, x == abs(x); abs(x) >= 0, x == 2*x]
```

```
M =
[    x == 2*x, x == abs(x)]
[ 0 <= abs(x),    x == 2*x]
```

For matrices and multidimensional arrays, any can test elements along the specified dimension. To specify the dimension, use the second argument of any. For example, to test elements of each column of a matrix, use the value 1 as the second argument:

```
any(M, 1)
```

```
ans =
     1     0
```

To test elements of each row, use the value 2 as the second argument:

```
any(M, 2)
```

```
ans =
     0
     1
```

Test whether any element of this vector returns logical 1. Note that `any` also converts all numeric values outside equations and inequalities to logical 1s and 0s. The numeric value 0 becomes logical 0:

```
syms x
any([0, x == x + 1])
```

```
ans =
     0
```

All nonzero numeric values, including negative and complex values, become logical 1s:

```
any([-4 + i, x == x + 1])
```

```
ans =
     1
```

More About

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` | `logical` | `not` | `or` | `xor`

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`

Arithmetic Operations

Perform arithmetic operations on symbols

Syntax

```
A+B
A-B
A*B
A.*B
A\B
A.\B
B/A
A./B
A^B
A.^B
A'
A.'
```

Description

`+`	Matrix addition. `A+B` adds `A` and `B`. `A` and `B` must have the same dimensions, unless one is scalar.
`-`	Matrix subtraction. `A-B` subtracts `B` from `A`. `A` and `B` must have the same dimensions, unless one is scalar.
`*`	Matrix multiplication. `A*B` is the linear algebraic product of `A` and `B`. The number of columns of `A` must equal the number of rows of `B`, unless one is a scalar.
`.*`	Array multiplication. `A.*B` is the entry-by-entry product of `A` and `B`. `A` and `B` must have the same dimensions, unless one is scalar.
`\`	Matrix left division. `A\B` solves the symbolic linear equations `A*X=B` for `X`. Note that `A\B` is roughly equivalent to `inv(A)*B`. Warning messages are produced if `X` does not exist or is not unique. Rectangular matrices `A` are allowed, but the equations must be consistent; a least squares solution is *not* computed.

`.\`	Array left division. `A.\B` is the matrix with entries `B(i,j)/A(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`/`	Matrix right division. `B/A` solves the symbolic linear equation `X*A=B` for `X`. Note that `B/A` is the same as `(A.'\B.').'`. Warning messages are produced if `X` does not exist or is not unique. Rectangular matrices `A` are allowed, but the equations must be consistent; a least squares solution is not computed.
`./`	Array right division. `A./B` is the matrix with entries `A(i,j)/B(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`^`	Matrix power. `A^B` raises the square matrix `A` to the integer power `B`. If `A` is a scalar and `B` is a square matrix, `A^B` raises `A` to the matrix power `B`, using eigenvalues and eigenvectors. `A^B`, where `A` and `B` are both matrices, is an error.
`.^`	Array power. `A.^B` is the matrix with entries `A(i,j)^B(i,j)`. `A` and `B` must have the same dimensions, unless one is scalar.
`'`	Matrix Hermitian transpose. If `A` is complex, `A'` is the complex conjugate transpose.
`.'`	Array transpose. `A.'` is the real transpose of `A`. `A.'` does not conjugate complex entries.

Examples

The following statements

```
syms a b c d
A = [a b; c d];
A*A/A
A*A-A^2
```

return

```
[ a, b]
[ c, d]

[ 0, 0]
[ 0, 0]
```

The following statements

```
syms b1 b2
A = sym('a%d%d', [2 2]);
B = [b1 b2];
X = B/A;
x1 = X(1)
x2 = X(2)
```

return

```
x1 =
-(a21*b2 - a22*b1)/(a11*a22 - a12*a21)

x2 =
(a11*b2 - a12*b1)/(a11*a22 - a12*a21)
```

See Also

`null` | `solve`

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)]
```

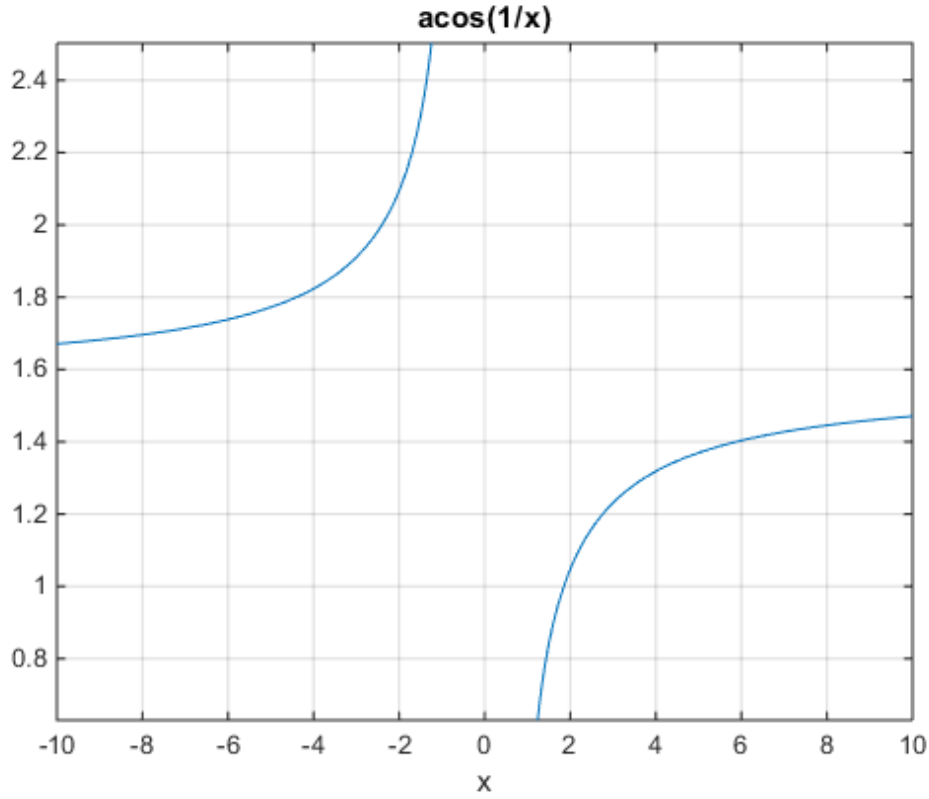
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 2.094395102393195492308428922186 3,...
Inf,...
0.523598775598298873077107230546 58,...
1.316957896924816708625046347308*i,...
0,...
1.369438406004565827776196139422 1]
```

Plot the Inverse Secant Function

Plot the inverse secant function on the interval from -10 to 10.

```
syms x
ezplot(asec(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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) - acosh(x)*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)*i)*i
```

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`

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*2*i)/3, Inf, (pi*i)/6, acosh(2), 0, acosh(1/3)]
```

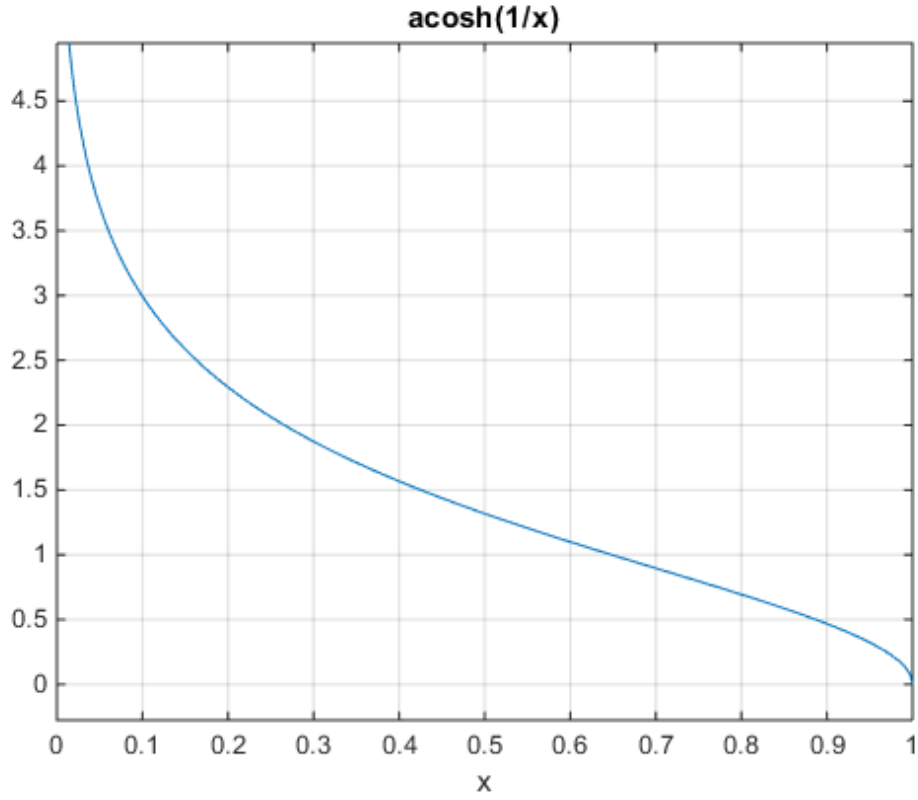
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 2.094395102393195492308428922186 3*i,...
Inf,...
0.523598775598298873077107230546 58*i,...
1.316957896924816708625046347308,...
0,...
1.230959417340774682134929178248*i]
```

Plot the Inverse Hyperbolic Secant Function

Plot the inverse hyperbolic secant function on the interval from 0 to 1.

```
syms x
ezplot(asech(x), [0, 1])
grid on
```



Handle Expressions Containing the 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:

```
syms x
diff(asech(x), x)
diff(asech(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 secant function:

```
int(asech(x), x)
```

```
ans =
x*acosh(1/x) + asin(x)*sign(x)
```

Find the Taylor series expansion of `asech(x)` around `x = Inf`:

```
taylor(asech(x), x, Inf)
```

```
ans =
(pi*i)/2 - i/x - i/(6*x^3) - (3*i)/(40*x^5)
```

Rewrite the inverse hyperbolic secant function in terms of the natural logarithm:

```
rewrite(asech(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` | `acsch` | `asinh` | `atanh` | `cosh` | `coth` | `csch` | `sech` | `sinh` | `tanh`

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]
```

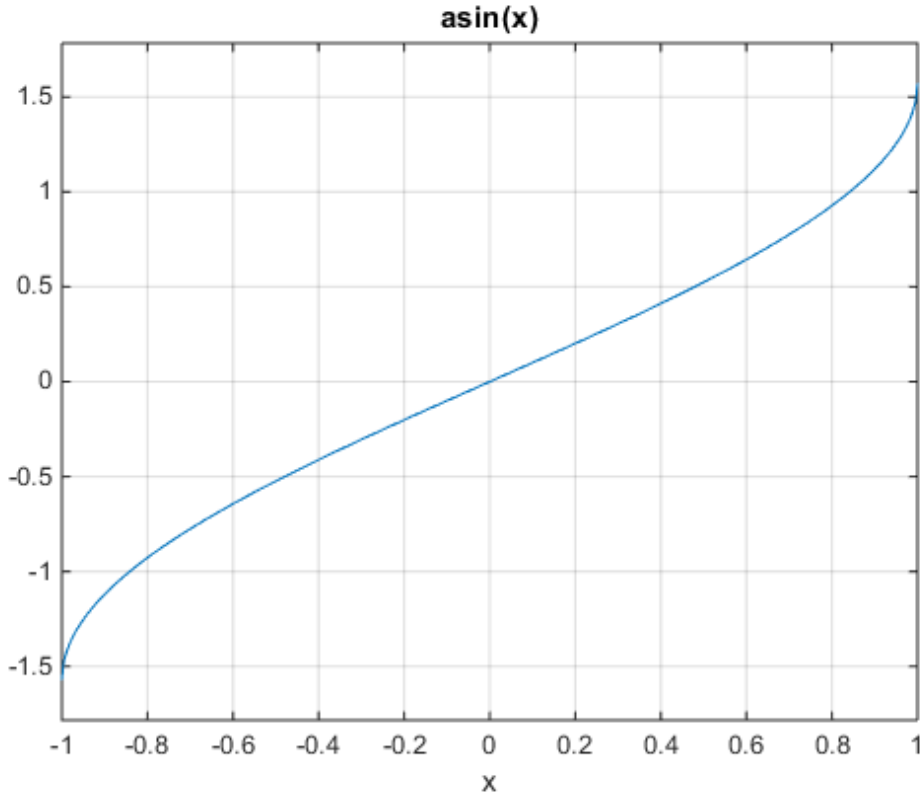
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.5707963267948966192313216916398,...
-0.33983690945412193709639251339176,...
-0.52359877559829887307710723054658,...
0.25268025514207865348565743699371,...
0.52359877559829887307710723054658,...
1.0471975511965977461542144610932,...
1.5707963267948966192313216916398]
```

Plot the Inverse Sine Function

Plot the inverse sine function on the interval from -1 to 1.

```
syms x
ezplot(asin(x), [-1, 1])
grid on
```



Handle Expressions Containing the 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(x*i + (1 - x^2)^(1/2))*i
```

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`

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*i)/2, 0, asinh(1/6), (pi*i)/6, (pi*i)/2, asinh(2)]
```

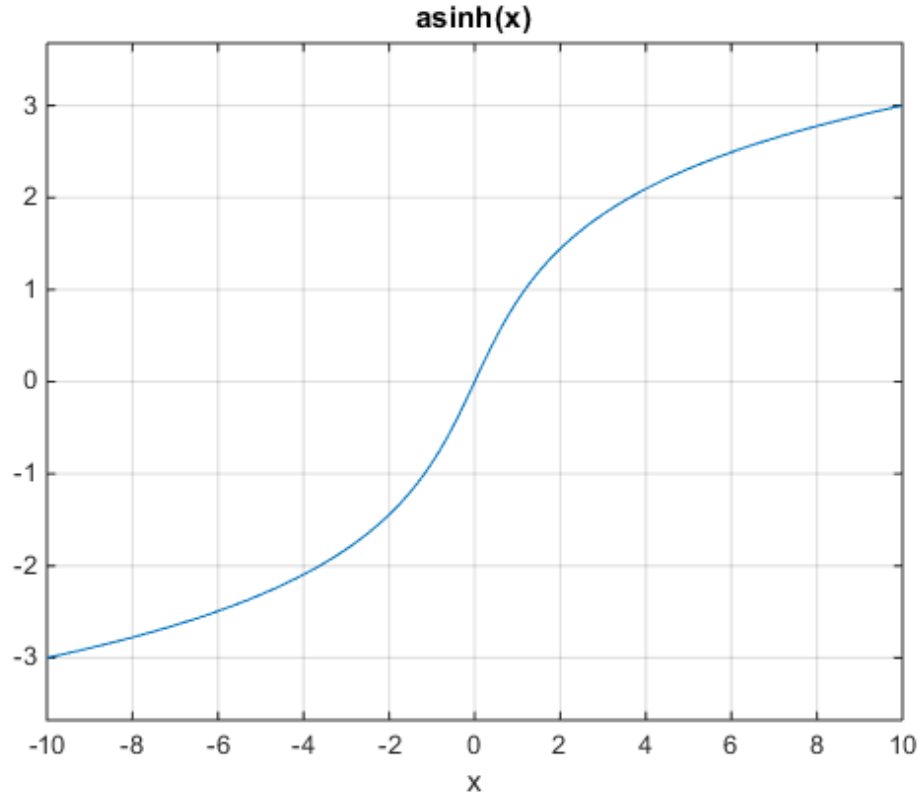
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.570796326794896619231321691639 8*i,...
0,...
0.16590455026930116978860971594885,...
0.523598775598298873077107230546 58*i,...
1.570796326794896619231321691639 8*i,...
1.443635475178810342493276740273 1]
```

Plot the Inverse Hyperbolic Sine Function

Plot the inverse hyperbolic sine function on the interval from -10 to 10.

```
syms x
ezplot(asinh(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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`

assume

Set assumption on symbolic object

Syntax

```
assume(condition)
assume(expr,set)
```

Description

`assume(condition)` states that condition is valid for all symbolic variables in condition. It also removes any assumptions previously made on these symbolic variables.

`assume(expr,set)` states that expr belongs to set. This new assumption replaces previously set assumptions on all variables in expr.

Examples

Assumptions on Integrand

Compute this 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:

```
syms a clear
int(x^a, x)

ans =
piecewise([a == -1, log(x)], [a ~= -1, x^(a + 1)/(a + 1)])
```

Assume Variable is Even or Odd

To assume the symbolic variable `x` is even, set the assumption that `x/2` is an integer. To assume `x` is odd, set the assumption 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. Find all odd numbers between `0` and `10` using `solve`.

```
assume((x-1)/2,'integer')
solve(x>0, x<10, x)

ans =
 1
 3
 5
 7
 9
```

Clear assumptions on `x` for further computations.

```
syms x clear
```

Assumptions on Parameters and Variables of Equation

Use assumptions on the symbolic parameter and variable in the kinematic equation for the free fall motion.

Calculate the time during which the object falls from a certain height by solving the kinematic equation for the free fall motion. If you do not consider the special case where no gravitational forces exist, you can assume that the gravitational acceleration `g` is positive:

```
syms g h t
assume(g > 0)
solve(h == g*t^2/2, t)
```

```
ans =
  (2^(1/2)*h^(1/2))/g^(1/2)
 -(2^(1/2)*h^(1/2))/g^(1/2)
```

You can also set assumptions on variables for which you solve an equation. When you set assumptions on such variables, the solver compares obtained solutions with the specified assumptions. This additional task can slow down the solver.

```
assume(t > 0)
solve(h == g*t^2/2, t)
```

```
Warning: The solutions are valid under the following
conditions: 0 < h. To include parameters and conditions in
the solution, specify the 'ReturnConditions' option.
> In solve>warnIfParams at 514
  In solve at 356
ans =
(2^(1/2)*h^(1/2))/g^(1/2)
```

The solver returns a warning that `h` must be positive. This follows as the object is above ground.

For further computations, clear the assumptions:

```
syms g t clear
```

Assumptions Used for Simplification

Use assumption to simplify the sine function.

Simplify this sine function:

```
syms n
simplify(sin(2*n*pi))

ans =
sin(2*pi*n)
```

Suppose `n` in this expression is an integer. Then you can simplify the expression further using the appropriate assumption:

```
assume(n,'integer')
simplify(sin(2*n*pi))

ans =
0
```

For further computations, clear the assumption:

```
syms n clear
```

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
int(1/abs(x^2 - 1), x)

ans =
-atanh(x)/sign(x^2 - 1)
```

If you know that $x^2 - 1 > 0$, set the appropriate assumption:

```
assume(x^2 - 1 > 0)
int(1/abs(x^2 - 1), x)

ans =
-atanh(x)
```

For further computations, clear the assumption:

```
syms x clear
```

Assumptions Reducing Number of Solutions

Use assumptions to restrict the returned solutions of an equation to a particular interval.

Solve this equation:

```
syms x
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6 + (365*x)/2 + 250/3, x)
```

```
ans =
   -5
   -1
 -1/3
  1/2
  100
```

Use `assume` to restrict the solutions to the interval –1 <= *x* <= 1:

```
assume(-1 <= x <= 1)
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6 + (365*x)/2 + 250/3, x)
```

```
ans =
   -1
 -1/3
  1/2
```

To set several assumptions simultaneously, use the logical operators `and`, `or`, `xor`, `not`, or their shortcuts. For example, all negative solutions less than `-1` and all positive solutions greater than `1`:

```
assume(x < -1 | x > 1)
solve(x^5 - (565*x^4)/6 - (1159*x^3)/2 - (2311*x^2)/6 + (365*x)/2 + 250/3, x)
```

```
ans =
  -5
 100
```

For further computations, clear the assumptions:

```
syms x clear
```

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 the auto-generated elements:

```
A = sym('A', [3 3])
```

```
A =
[ A1_1, A1_2, A1_3]
[ A2_1, A2_2, A2_3]
[ A3_1, A3_2, A3_3]
```

Suppose that all elements of this matrix represent rational numbers. Instead of setting an assumption on each element separately, you can set the assumption on the matrix:

```
assume(A,'rational')
```

To see the assumptions on the elements of A, use `assumptions`:

```
assumptions(A)
```

```
ans =
[ in(A3_1, 'rational'), in(A2_1, 'rational'), in(A1_1, 'rational'),...
 in(A3_2, 'rational'), in(A2_2, 'rational'), in(A1_2, 'rational'),...
 in(A3_3, 'rational'), in(A2_3, 'rational'), in(A1_3, 'rational')]
```

For further computations, clear the assumptions:

```
syms A clear
```

Input Arguments

`condition` — Assumption statement

symbolic expression | symbolic equation | relation | vector of symbolic expressions, equations, or relations | 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 | matrix

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, or real numbers

`'integer' | 'rational' | 'real'`

Set of integer, rational, or real numbers, specified as one of these strings: `'integer'`, `'rational'`, or `'real'`.

More About

Tips

- `assume` removes any assumptions previously set on the symbolic variables. To retain previous assumptions while adding a new one, use `assumeAlso`.
- When you delete a symbolic variable from the MATLAB workspace using `clear`, all assumptions that you set on that variable remain in the symbolic engine. If you later declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable and the value of the variable, use this command:

  ```
  syms x clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('x','clear')
  ```

- To delete all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace clearing all assumptions, use this command:

  ```
  clear all
  ```

- If 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`.) MATLAB projects complex numbers in inequalities to real axis. For example, `x > i` becomes `x > 0`, and `x <= 3 + 2*i` becomes `x <= 3`.
- The toolbox does not support assumptions on symbolic functions. Make assumptions on symbolic variables and expressions instead.
- When you create a new symbolic variable using `sym` and `syms`, you also can set an assumption that the variable is real or positive:

  ```
  a = sym('a','real');
  b = sym('b','real');
  ```

```
c = sym('c','positive');
```

or more efficiently

```
syms a b real
syms c positive
```

- “Default Assumption” on page 1-32

See Also

`and` | `assumeAlso` | `assumptions` | `clear all` | `in` | `isAlways` | `logical` | `not` | `or` | `sym` | `syms`

Related Examples

- “Set Assumptions” on page 1-32
- “Check Existing Assumptions” on page 1-33
- “Delete Symbolic Objects and Their Assumptions” on page 1-33

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: The solutions are valid under the following
conditions: x <= 1; x == 1. To include parameters and
conditions in the solution, specify the 'ReturnConditions'
option.
> In solve>warnIfParams at 514
  In solve at 356
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. Now add the assumption that `x < 1`. To add a new assumption without removing the previous one, use `assumeAlso`:

```
assumeAlso(x < 1)
```

Solve the same equation under the expanded set of assumptions:

```
s = solve(x^2 + y^2 == 1, y)
```

```
s =
(1 - x)^(1/2)*(x + 1)^(1/2)
```

For further computations, clear the assumptions:

```
syms x y clear
```

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`, you can add more assumptions on the same variable `n`. For example, assume also that n is and integer:

```
assumeAlso(n,'integer')
```

To see all assumptions currently valid for the variable `n`, use `assumptions`. In this case, `n` is a positive integer.

```
assumptions(n)
```

```
ans =
[ in(n, 'integer'), 0 < n]
```

For further computations, clear the assumptions:

```
syms 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 the auto-generated elements:

```
A = sym('A', [3 3])
```

```
A =
[ A1_1, A1_2, A1_3]
[ A2_1, A2_2, A2_3]
[ A3_1, A3_2, A3_3]
```

Suppose that all elements of this matrix represent rational numbers. Instead of setting an assumption on each element separately, you can set the assumption on the matrix:

```
assume(A,'rational')
```

Now, add the assumption that each element of A is greater than 1:

```
assumeAlso(A > 1)
```

To see the assumptions on the elements of A, use `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'),...
 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]
```

For further computations, clear the assumptions:

```
syms 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 the following error:

```
syms y
assume(y,'real')
assumeAlso(y == i)

Error using mupadmex
Error in MuPAD command: Inconsistent assumptions
detected. [property::_setgroup]
```

`assumeAlso` does not guarantee to detect contradicting assumptions. For example, you can assume that `y` is nonzero, and both `y` and `y*i` are real values:

```
syms y
assume(y ~= 0)
assumeAlso(y,'real')
assumeAlso(y*i,'real')
```

To see all assumptions currently valid for the variable `y`, use `assumptions`:

```
assumptions(y)

ans =
[ in(y, 'real'), y ~= 0, in(y*i, 'real')]
```

For further computations, clear the assumptions:

```
syms y clear
```

Input Arguments

`condition` — Assumption statement

symbolic expression | symbolic equation | relation | vector of symbolic expressions, equations, or relations | 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 | matrix

Expression to set assumption on, specified as a symbolic variable, expression, vector, or matrix. 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, or real numbers

`'integer' | 'rational' | 'real'`

Set of integer, rational, or real numbers, specified as one of these strings: `'integer'`, `'rational'`, or `'real'`.

More About

Tips

- `assumeAlso` keeps all assumptions previously set on the symbolic variables. To replace previous assumptions with the new one, use `assume`.
- When adding assumptions, always check that a new assumption does not contradict the existing assumptions. To see existing assumptions, use `assumptions`. Symbolic Math Toolbox does not guarantee to detect conflicting assumptions. Conflicting assumptions can lead to unpredictable and inconsistent results.
- When you delete a symbolic variable from the MATLAB workspace using `clear`, all assumptions that you set on that variable remain in the symbolic engine. If later you declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable and the value of the variable, use this command:

  ```
  syms x clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('x','clear')
  ```

- To clear all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace resetting all its assumptions, use this command:

  ```
  clear all
  ```

- If 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`.) MATLAB projects complex numbers in inequalities to real axis. For example, `x > i` becomes `x > 0`, and `x <= 3 + 2*i` becomes `x <= 3`.

- The toolbox does not support assumptions on symbolic functions. Make assumptions on symbolic variables and expressions instead.
- 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.

- "Default Assumption" on page 1-32

See Also

`and` | `assume` | `assumptions` | `clear all` | `in` | `isAlways` | `logical` | `not` | `or` | `sym` | `syms`

Related Examples

- "Set Assumptions" on page 1-32
- "Check Existing Assumptions" on page 1-33
- "Delete Symbolic Objects and Their Assumptions" on page 1-33

assumptions

Show assumptions set on symbolic variable

Syntax

```
assumptions(var)
assumptions
```

Description

`assumptions(var)` returns all assumptions set on variable var.

`assumptions` returns all assumptions set on all variables in MATLAB Workspace.

Examples

Assumptions on Several Variables

Show the assumptions set on variables `n` and `x` separately, and then show assumptions set on all variables.

Assume that the variable `n` is integer and the variable `x` is rational. In addition to that , assume that the product `n*x` belongs to the interval from `-100` to `100`:

```
syms n x
assume(n,'integer')
assume(x,'rational')
assumeAlso(-100 <= n*x <= 100)
```

To see the assumptions set on the variable `n`, enter:

```
assumptions(n)
```

```
ans =
[ -100 <= n*x, n*x <= 100, in(n, 'integer')]
```

The syntax `in(n, 'integer')` indicates `n` is an integer.

To see the assumptions set on the variable `x`, enter:

```
assumptions(x)
```

```
ans =
[ -100 <= n*x, n*x <= 100, in(x, 'rational')]
```

To see the assumptions set on all variables, use `assumptions` without any arguments:

```
assumptions
```

```
ans =
[ -100 <= n*x, n*x <= 100, in(n, 'integer'), in(x, 'rational')]
```

For further computations, clear the assumptions:

```
syms n x clear
```

Multiple Assumptions on One Variable

Show the assumptions set by using `syms` and `assume`.

Use `assumptions` to return all assumptions, including those set by the `syms` command:

```
syms x real
assumeAlso(x < 0)
assumptions(x)
```

```
ans =
[ x < 0, in(x, 'real')]
```

The syntax `in(x, 'real')` indicates `x` is `real`.

Input Arguments

`var` — Variable for which to show assumptions

symbolic variable | array of symbolic variables | vector of symbolic variables | matrix of symbolic variables

Variable for which to show assumptions, specified as a symbolic variable or array, vector, or matrix of symbolic variables.

More About

Tips

- When you delete a symbolic object from the MATLAB workspace by using `clear`, all assumptions that you set on that object remain in the symbolic engine. If later you declare a new symbolic variable with the same name, it inherits these assumptions.
- To clear all assumptions set on a symbolic variable `var` and the value of the variable, use this command:

  ```
  syms var clear
  ```

- To clear assumptions and keep the value of the variable, use this command:

  ```
  sym('var','clear')
  ```

- To clear all objects in the MATLAB workspace and close the MuPAD engine associated with the MATLAB workspace resetting all its assumptions, use this command:

  ```
  clear all
  ```

- "Default Assumption" on page 1-32

See Also

`and` | `assume` | `assumeAlso` | `clear` | `clear all` | `in` | `isAlways` | `logical` | `not` | `or` | `sym` | `syms`

Related Examples

- "Set Assumptions" on page 1-32
- "Check Existing Assumptions" on page 1-33
- "Delete Symbolic Objects and Their Assumptions" on page 1-33

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]
```

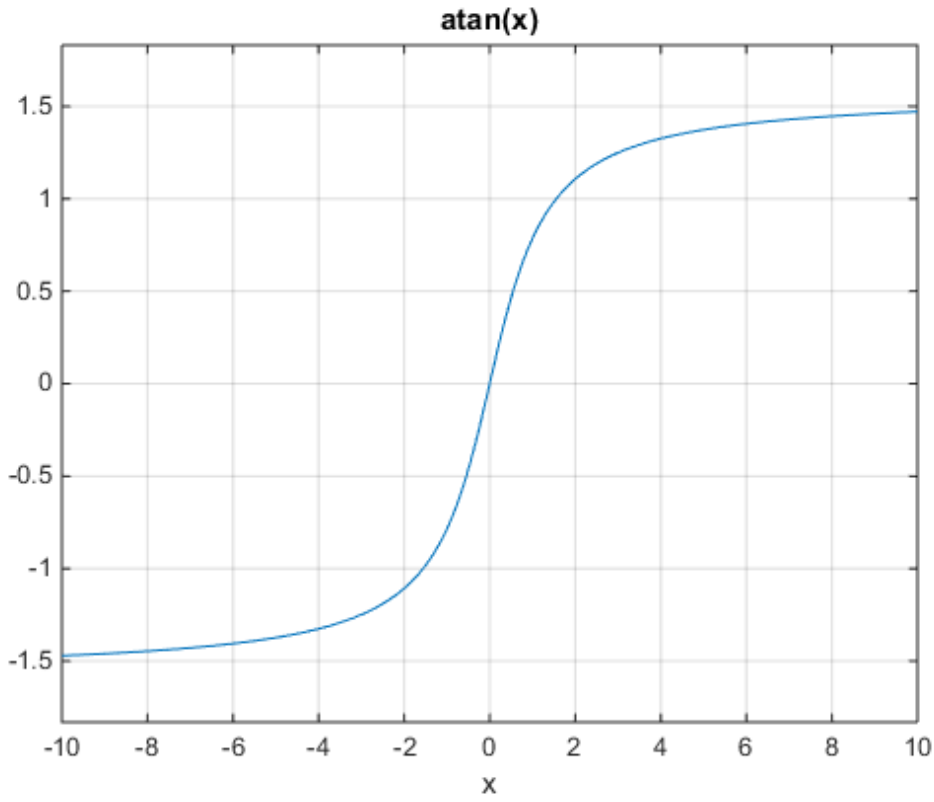
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.78539816339744830961566084581988,...
-0.32175055439664219340140461435866,...
-0.52359877559829887307710723054658,...
0.46364760900080611621425623146121,...
0.78539816339744830961566084581988,...
1.04719755119659774615421446109 32]
```

Plot the Inverse Tangent Function

Plot the inverse tangent function on the interval from -10 to 10.

```
syms x
ezplot(atan(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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*i)*i)/2 - (log(x*i + 1)*i)/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`

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]
```

Alternatives

For complex `Z = X + Y*i`, the call `atan2(Y,X)` is equivalent to `angle(Z)`.

More About

atan2 vs. atan

If $X \neq \mathbf{0}$ and $Y \neq \mathbf{0}$, then

$$\mathrm{atan2}(Y,X) = \mathrm{atan}(\frac{Y}{X}) + \frac{\pi}{2}\mathrm{sign}(Y)\left(1 - \mathrm{sign}(X)\right)$$

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`.

See Also

`angle` | `atan` | `conj` | `imag` | `real`

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*i)/4, 0, atanh(1/6), atanh(i/2), (pi*i)/4, atanh(2)]
```

Use vpa to approximate symbolic results with floating-point numbers:

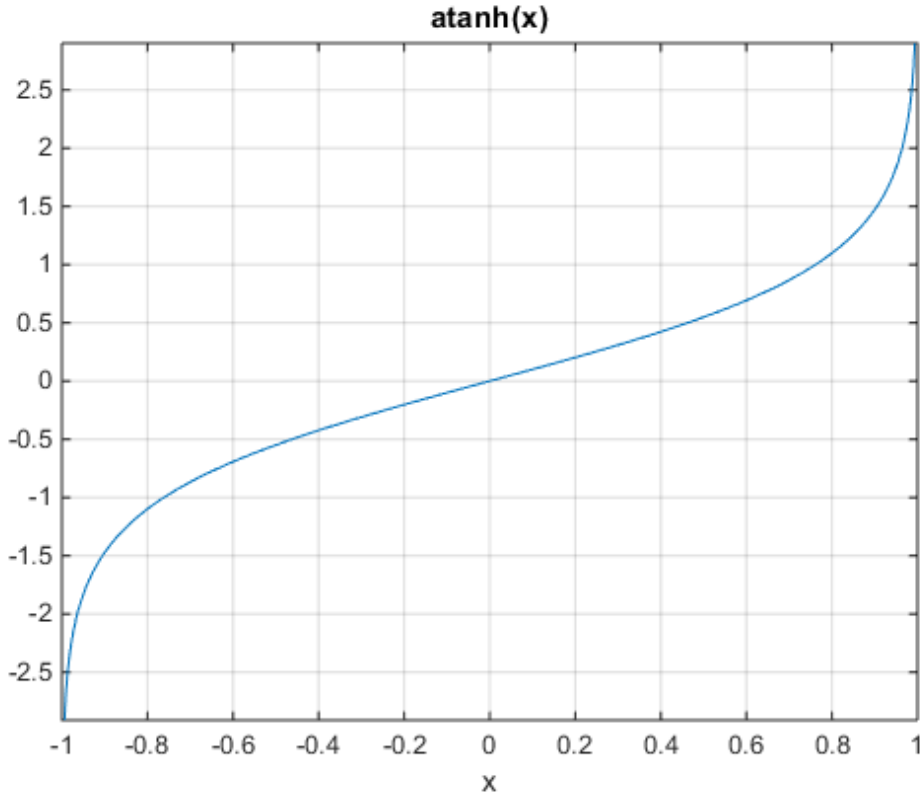
```
vpa(symA)
```

```
ans =
[ -0.785398163397448309615660845819 88*i,...
0,...
0.16823611831060646525229670510 85,...
0.46364760900080611621425623146121*i,...
0.785398163397448309615660845819 88*i,...
0.54930614433405484569762261846126 - 1.57079632679489661923132169163 98*i]
```

Plot the Inverse Hyperbolic Tangent Function

Plot the inverse hyperbolic tangent function on the interval from -1 to 1.

```
syms x
ezplot(atanh(x), [-1, 1])
grid on
```



Handle Expressions Containing the 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`

bernoulli

Bernoulli numbers and polynomials

Syntax

```
bernoulli(n)
bernoulli(n,x)
```

Description

`bernoulli(n)` returns the nth Bernoulli number.

`bernoulli(n,x)` returns the nth Bernoulli polynomial.

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:

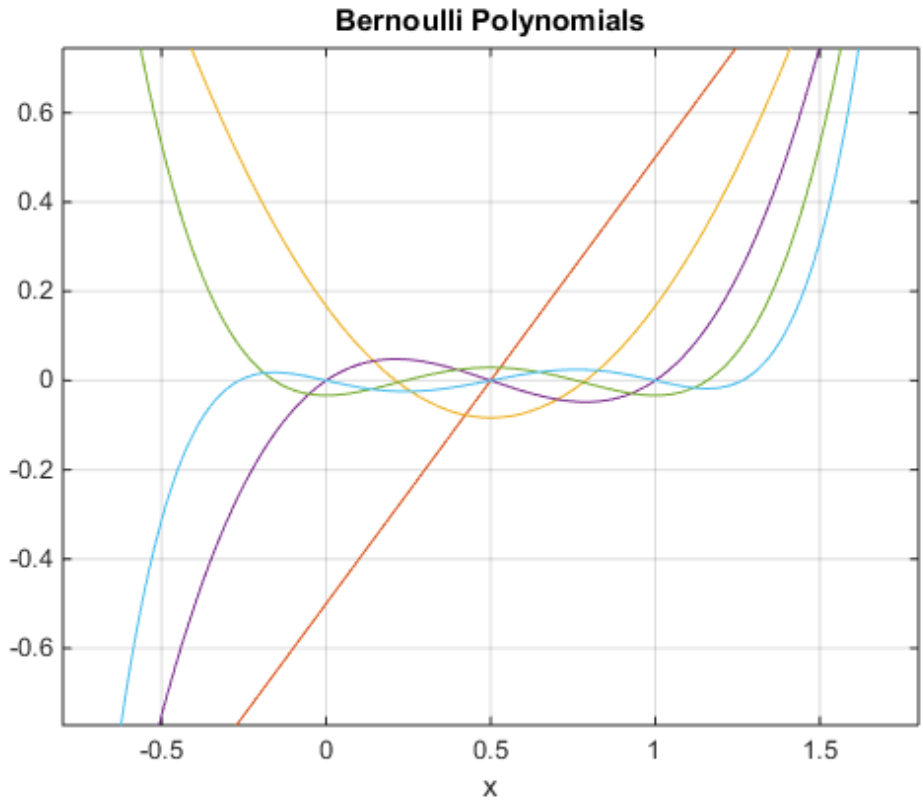
```
bernoulli(2, sym(1/3))
```

```
ans =
-1/18
```

Plot the Bernoulli Polynomials

Plot the first six Bernoulli polynomials.

```
syms x

for n = 0:5
  ezplot(bernoulli(n, x), [-0.8, 1.8]);
  hold on
end

title('Bernoulli Polynomials')
grid on
hold off
```



Handle Expressions Containing the 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.

More About

Bernoulli Polynomials

The Bernoulli polynomials are defined as follows:

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} \mathrm{bernoulli}(n, x)\frac{t^n}{n!}$$

Bernoulli Numbers

The Bernoulli numbers are defined as follows:

$$\mathrm{bernoulli}(n) = \mathrm{bernoulli}(n, 0)$$

See Also

`euler`

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 `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 the Sine Function Specified as a Function Handle

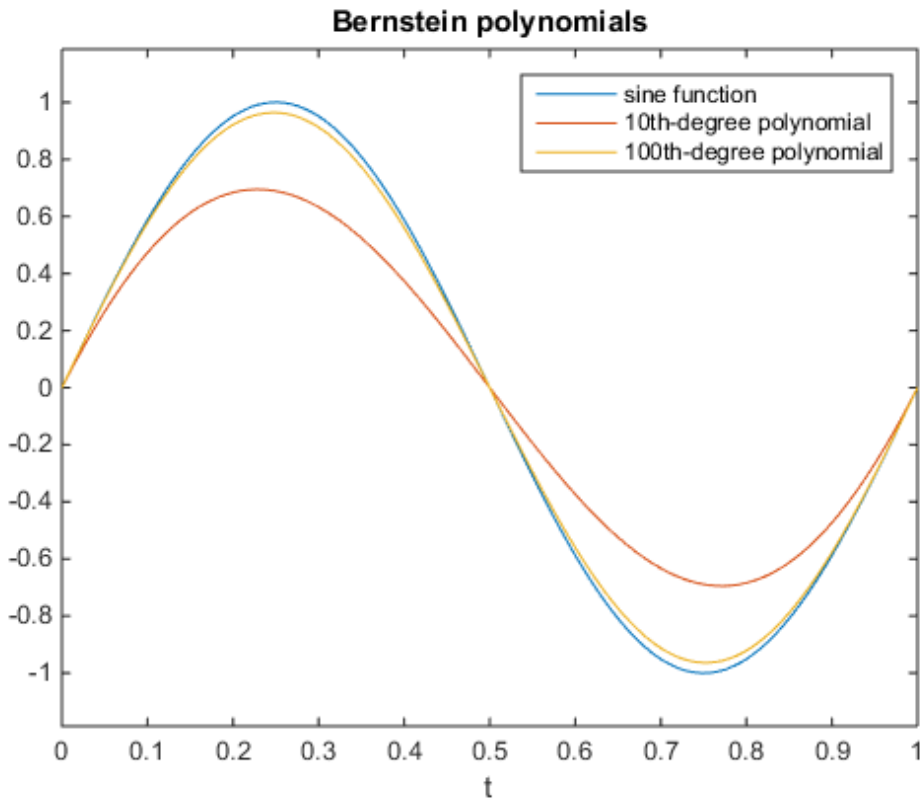
Approximate the sine function by the 10th- and 100th-degree Bernstein polynomials:

```
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:

```
ezplot(sin(2*pi*t),[0,1])
hold on
ezplot(b10,[0,1])
ezplot(b100,[0,1])

legend('sine function','10th-degree polynomial',...
                       '100th-degree polynomial')
title('Bernstein polynomials')
hold off
```



Approximation of the Exponential Function Specified as a 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 a Linear Ramp Specified as a 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:

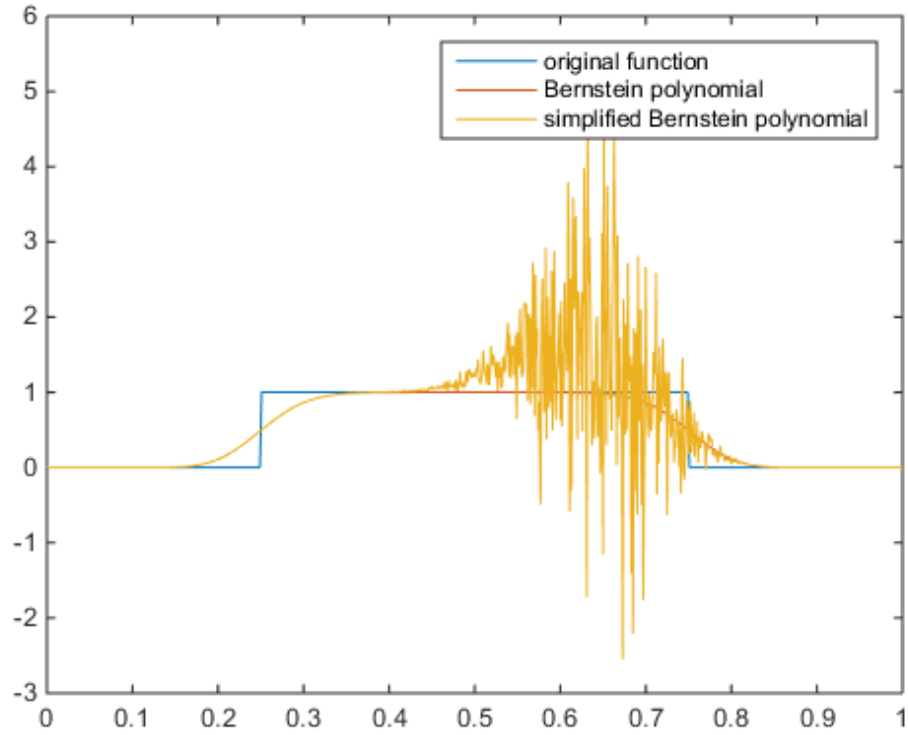
```
f = @(x)rectangularPulse(1/4,3/4,x);
b1 = bernstein(f, 100, sym('t'));
b2 = simplify(b1);
```

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.

More About

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

`bernstein` | `bernsteinMatrix` | `bernsteinMatrix` | `binomial` | `fact` | `formula` | `nchoosek` | `symsum` | `symvar`

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]
```

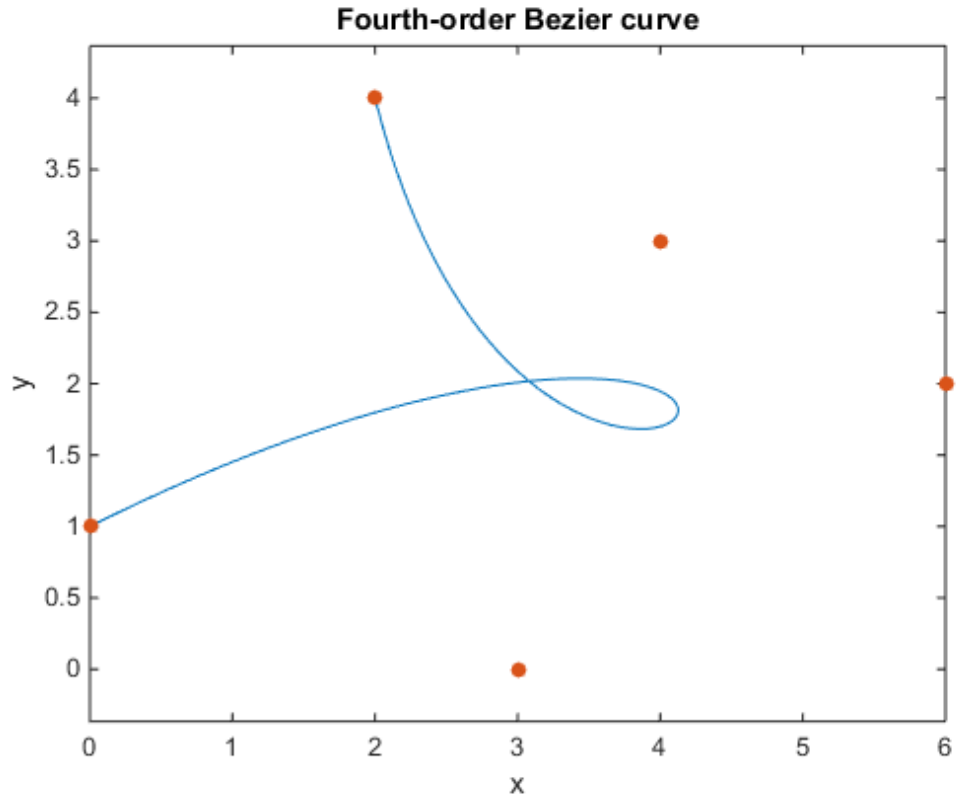
Construct the Bezier curve:

```
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:

```
ezplot(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]
```

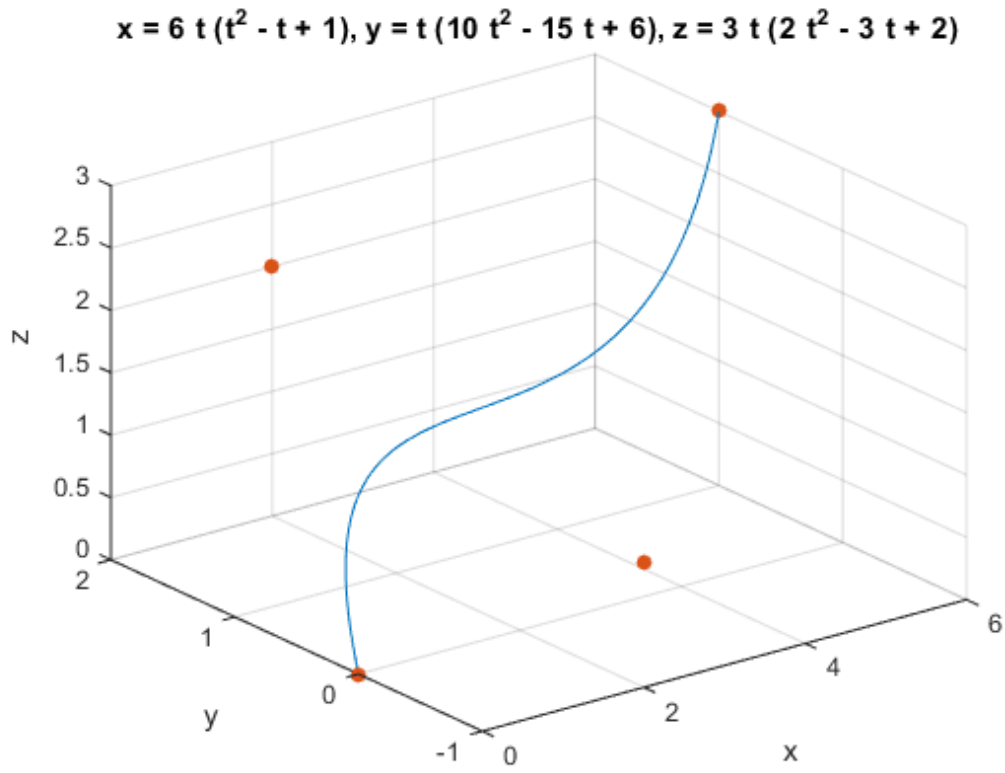
Construct the Bezier curve:

```
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:

```
ezplot3(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 the Evaluation Point Specified as a Vector

Construct the third-order Bezier curve with the evaluation point specified by the following 1-by-101 vector `t`:

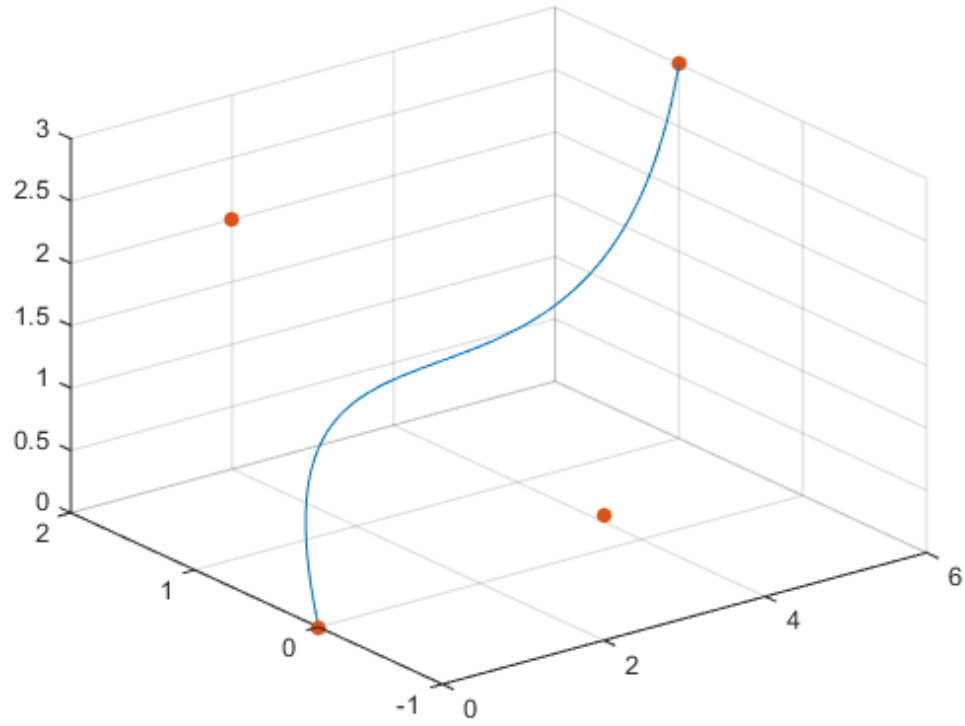
```
t = 0:1/100:1;
```

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` | `bernstein` | `bernsteinMatrix` | `binomial` | `fact` | `nchoosek` | `symsum` | `symvar`

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, $I_\nu(z)$.

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

Solve this second-order differential equation. The solutions are the modified Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) -(z^2 + nu^2)*w == 0)
```

```
ans =
C2*besseli(nu, z) + C3*besselk(nu, z)
```

Verify that the modified Bessel function of the first kind is a valid solution of the modified Bessel differential equation.

```
syms nu z
simplify(z^2*diff(besseli(nu, z), z, 2) + z*diff(besseli(nu, z), z)...
 - (z^2 + nu^2)*besseli(nu, z)) == 0
```

```
ans =
     1
```

Compute the modified Bessel functions of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besseli(0, 5), besseli(-1, 2), besseli(1/3, 7/4),  besseli(1, 3/2 + 2*i)]
```

```
ans =
  27.2399 + 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 + 2*i)]
```

For symbolic variables and expressions, `besseli` also returns unresolved symbolic calls:

```
syms x y
[besseli(x, y), besseli(1, x^2), besseli(2, x - y), besseli(x^2, x*y)]
```

```
ans =
[ besseli(x, y), besseli(1, x^2), besseli(2, x - y), besseli(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besseli` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besseli(1/2, x)
```

```
ans =
(2^(1/2)*sinh(x))/(pi^(1/2)*x^(1/2))
```

```
besseli(-1/2, x)
```

```
ans =
(2^(1/2)*cosh(x))/(pi^(1/2)*x^(1/2))
```

```
besseli(-3/2, x)
```

```
ans =
(2^(1/2)*(sinh(x) - cosh(x)/x))/(pi^(1/2)*x^(1/2))
```

```
besseli(5/2, x)
```

```
ans =
-(2^(1/2)*((3*cosh(x))/x - sinh(x)*(3/x^2 + 1)))/(pi^(1/2)*x^(1/2))
```

Differentiate the expressions involving the modified Bessel functions of the first kind:

```
syms x y
diff(besseli(1, x))
diff(diff(besseli(0, x^2 + x*y -y^2), x), y)
```

```
ans =
besseli(0, x) - besseli(1, x)/x
```

```
ans =
besseli(1, x^2 + x*y - y^2) +...
(2*x + y)*(besseli(0, x^2 + x*y - y^2)*(x - 2*y) -...
(besseli(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

Call `besseli` for the matrix `A` and the value 1/2. The result is a matrix of the modified Bessel functions `besseli(1/2, A(i,j))`.

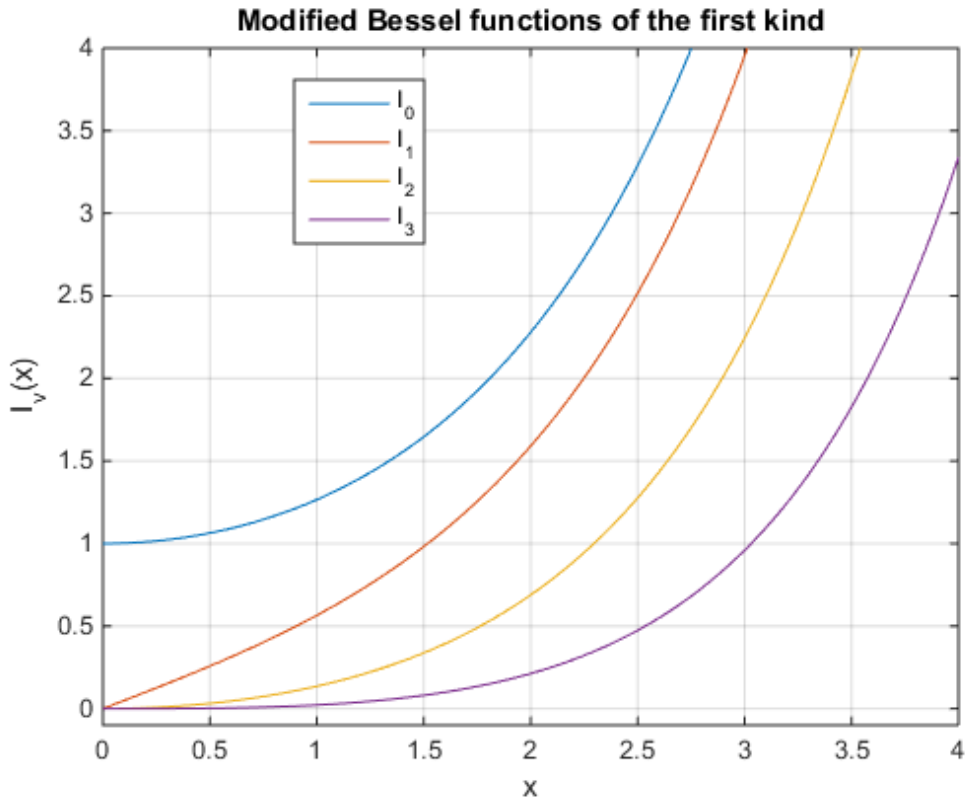
```
syms x
A = [-1, pi; x, 0];
besseli(1/2, A)
```

```
ans =
[         (2^(1/2)*sinh(1)*i)/pi^(1/2), (2^(1/2)*sinh(pi))/pi]
[ (2^(1/2)*sinh(x))/(pi^(1/2)*x^(1/2)),                    0]
```

Plot the modified Bessel functions of the first kind for v = 0, 1, 2, 3:

```
syms x y
for nu = [0, 1, 2, 3]
  ezplot(besseli(nu, x))
  hold on
```

```
end
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')
hold off
```



More About

Modified Bessel Functions of the First Kind

The modified Bessel differential equation

$$z^2\frac{d^2w}{dz^2}+z\frac{dw}{dz}-\left(z^2+\nu^2\right)w=0$$

has two linearly independent solutions. These solutions are represented by the modified Bessel functions of the first kind, $I_\nu(z)$, and the modified Bessel functions of the second kind, $K_\nu(z)$:

$$w(z)=C_1I_\nu(z)+C_2K_\nu(z)$$

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`

besselj

Bessel function of the first kind

Syntax

```
besselj(nu,z)
```

Description

`besselj(nu,z)` returns the Bessel function of the first kind, $J_v(z)$.

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

Solve this second-order differential equation. The solutions are the Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) +(z^2 - nu^2)*w == 0)
```

```
ans =
```

```
C2*besselj(nu, z) + C3*bessely(nu, z)
```

Verify that the Bessel function of the first kind is a valid solution of the Bessel differential equation:

```
syms nu z
simplify(z^2*diff(besselj(nu, z), z, 2) + z*diff(besselj(nu, z), z)...
 + (z^2 - nu^2)*besselj(nu, z)) == 0
```

```
ans =
     1
```

Compute the Bessel functions of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besselj(0, 5), besselj(-1, 2), besselj(1/3, 7/4),...
  besselj(1, 3/2 + 2*i)]
```

```
ans =
  -0.1776 + 0.0000i  -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 + 2*i)]
```

For symbolic variables and expressions, `besselj` also returns unresolved symbolic calls:

```
syms x y
[besselj(x, y), besselj(1, x^2), besselj(2, x - y), besselj(x^2, x*y)]
```

```
ans =
[ besselj(x, y), besselj(1, x^2), besselj(2, x - y), besselj(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besselj` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besselj(1/2, x)
```

```
ans =
(2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2))
```

```
besselj(-1/2, x)

ans =
(2^(1/2)*cos(x))/(pi^(1/2)*x^(1/2))

besselj(-3/2, x)

ans =
-(2^(1/2)*(sin(x) + cos(x)/x))/(pi^(1/2)*x^(1/2))

besselj(5/2, x)

ans =
-(2^(1/2)*((3*cos(x))/x - sin(x)*(3/x^2 - 1)))/(pi^(1/2)*x^(1/2))
```

Differentiate the expressions involving the Bessel functions of the first kind:

```
syms x y
diff(besselj(1, x))
diff(diff(besselj(0, x^2 + x*y -y^2), x), y)

ans =
besselj(0, x) - besselj(1, x)/x

ans =
- besselj(1, x^2 + x*y - y^2) -...
(2*x + y)*(besselj(0, x^2 + x*y - y^2)*(x - 2*y) -...
(besselj(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

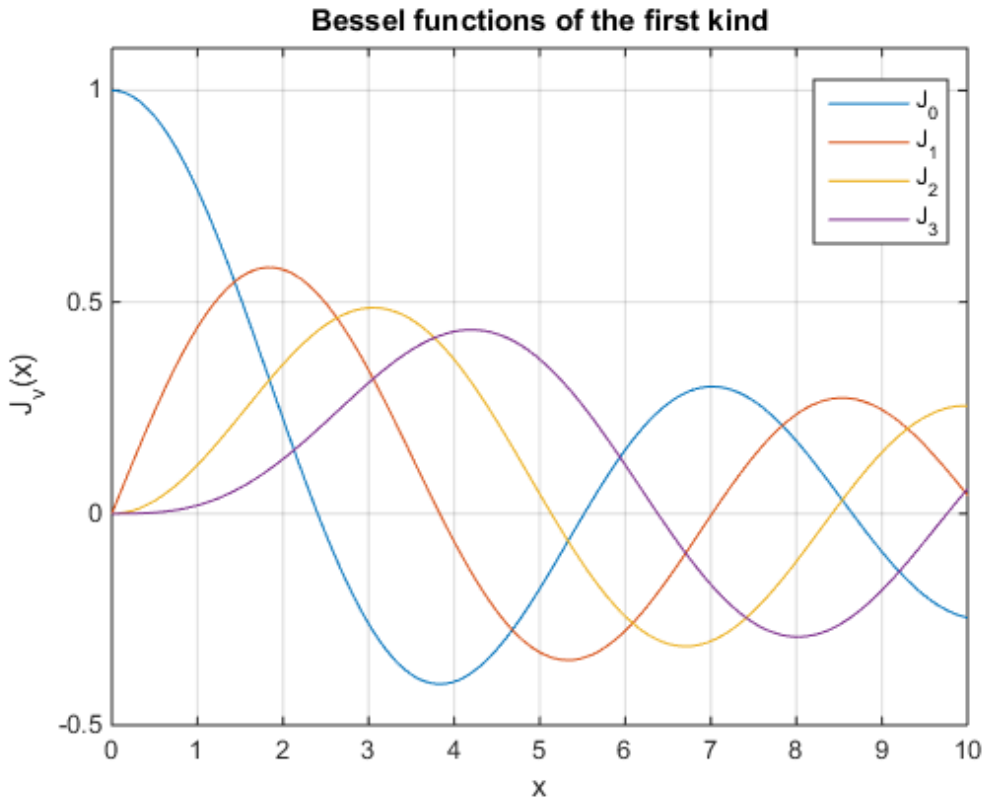
Call `besselj` for the matrix `A` and the value 1/2. The result is a matrix of the Bessel functions `besselj(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
besselj(1/2, A)

ans =
[         (2^(1/2)*sin(1)*i)/pi^(1/2), 0]
[ (2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2)), 0]
```

Plot the Bessel functions of the first kind for v = 0, 1, 2, 3:

```
syms x y
for nu = [0, 1, 2, 3]
  ezplot(besselj(nu, x), [0, 10])
  hold on
```

```
end
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')
hold off
```



More About

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, $Y_\nu(z)$:

$$w(z)=C_1J_\nu(z)+C_2Y_\nu(z)$$

This formula is the integral representation of the Bessel functions of the first kind:

$$J_\nu(z)=\frac{(z/2)^\nu}{\sqrt{\pi}\Gamma(\nu+1/2)}\int_0^\pi\cos\left(z\cos(t)\right)\sin(t)^{2\nu}\,dt$$

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`

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, $K_\nu(z)$.

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

Solve this second-order differential equation. The solutions are the modified Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) -(z^2 + nu^2)*w == 0)
```

```
ans =
C2*besseli(nu, z) + C3*besselk(nu, z)
```

Verify that the modified Bessel function of the second kind is a valid solution of the modified Bessel differential equation:

```
syms nu z
simplify(z^2*diff(besselk(nu, z), z, 2) + z*diff(besselk(nu, z), z)...
 - (z^2 + nu^2)*besselk(nu, z)) == 0

ans =
     1
```

Compute the modified Bessel functions of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[besselk(0, 5), besselk(-1, 2), besselk(1/3, 7/4),...
  besselk(1, 3/2 + 2*i)]

ans =
   0.0037 + 0.0000i   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 + 2*i)]
```

For symbolic variables and expressions, `besselk` also returns unresolved symbolic calls:

```
syms x y
[besselk(x, y), besselk(1, x^2), besselk(2, x - y), besselk(x^2, x*y)]

ans =
[ besselk(x, y), besselk(1, x^2), besselk(2, x - y), besselk(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besselk` rewrites the Bessel functions in terms of elementary functions:

```
syms x
besselk(1/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2))

besselk(-1/2, x)
```

```
ans =
(2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2))

besselk(-3/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x)*(1/x + 1))/(2*x^(1/2))

besselk(5/2, x)

ans =
(2^(1/2)*pi^(1/2)*exp(-x)*(3/x + 3/x^2 + 1))/(2*x^(1/2))
```

Differentiate the expressions involving the modified Bessel functions of the second kind:

```
syms x y
diff(besselk(1, x))
diff(diff(besselk(0, x^2 + x*y -y^2), x), y)

ans =
- besselk(1, x)/x - besselk(0, x)

ans =
(2*x + y)*(besselk(0, x^2 + x*y - y^2)*(x - 2*y) +...
(besselk(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2)) -...
besselk(1, x^2 + x*y - y^2)
```

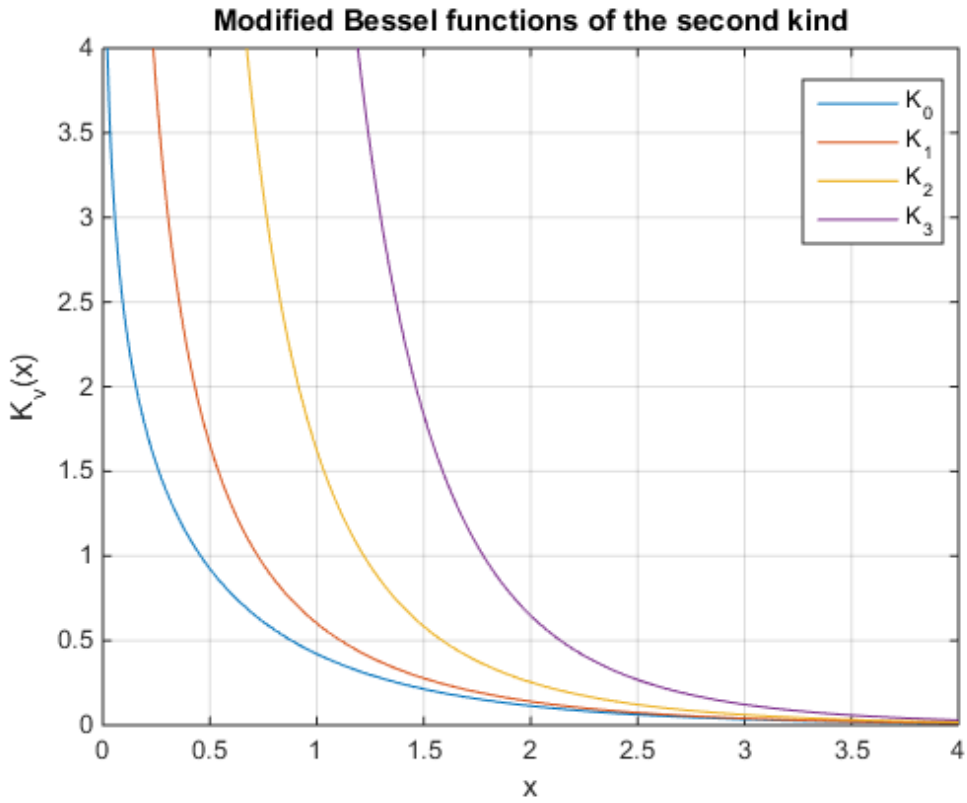
Call `besselk` for the matrix `A` and the value 1/2. The result is a matrix of the modified Bessel functions `besselk(1/2, A(i,j))`.

```
syms x
A = [-1, pi; x, 0];
besselk(1/2, A)

ans =
[         -(2^(1/2)*pi^(1/2)*exp(1)*i)/2, (2^(1/2)*exp(-pi))/2]
[ (2^(1/2)*pi^(1/2)*exp(-x))/(2*x^(1/2)),                  Inf]
```

Plot the modified Bessel functions of the second kind for v = 0, 1, 2, 3:

```
syms x y
for nu = [0, 1, 2, 3]
  ezplot(besselk(nu, x))
  hold on
```

```
end
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')
hold off
```



More About

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, $I_\nu(z)$, and the modified Bessel functions of the second kind, $K_\nu(z)$:

$$w(z)=C_1I_\nu(z)+C_2K_\nu(z)$$

The modified Bessel functions of the second kind are defined via the modified Bessel functions of the first kind:

$$K_\nu(z)=\frac{\pi/2}{\sin(\nu\pi)}\left(I_{-\nu}(z)-I_\nu(z)\right)$$

Here $I_\nu(z)$ are the modified Bessel functions of the first kind:

$$I_\nu(z)=\frac{(z/2)^\nu}{\sqrt{\pi}\Gamma(\nu+1/2)}\int_0^\pi e^{z\cos(t)}\sin(t)^{2\nu}\,dt$$

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`

bessely

Bessel function of the second kind

Syntax

```
bessely(nu,z)
```

Description

`bessely(nu,z)` returns the Bessel function of the second kind, $Y_\nu(z)$.

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

Solve this second-order differential equation. The solutions are the Bessel functions of the first and the second kind.

```
syms nu w(z)
dsolve(z^2*diff(w, 2) + z*diff(w) +(z^2 - nu^2)*w == 0)
```

```
ans =
```

```
C2*besselj(nu, z) + C3*bessely(nu, z)
```

Verify that the Bessel function of the second kind is a valid solution of the Bessel differential equation:

```
syms nu z
simplify(z^2*diff(bessely(nu, z), z, 2) + z*diff(bessely(nu, z), z)...
 + (z^2 - nu^2)*bessely(nu, z)) == 0

ans =
     1
```

Compute the Bessel functions of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[bessely(0, 5), bessely(-1, 2), bessely(1/3, 7/4), bessely(1, 3/2 + 2*i)]

ans =
  -0.3085 + 0.0000i   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 + 2*i)]
```

For symbolic variables and expressions, `bessely` also returns unresolved symbolic calls:

```
syms x y
[bessely(x, y), bessely(1, x^2), bessely(2, x - y), bessely(x^2, x*y)]

ans =
[ bessely(x, y), bessely(1, x^2), bessely(2, x - y), bessely(x^2, x*y)]
```

If the first parameter is an odd integer multiplied by 1/2, `besseli` rewrites the Bessel functions in terms of elementary functions:

```
syms x
bessely(1/2, x)

ans =
-(2^(1/2)*cos(x))/(pi^(1/2)*x^(1/2))

bessely(-1/2, x)
```

```
ans =
(2^(1/2)*sin(x))/(pi^(1/2)*x^(1/2))
```

```
bessely(-3/2, x)
```

```
ans =
(2^(1/2)*(cos(x) - sin(x)/x))/(pi^(1/2)*x^(1/2))
```

```
bessely(5/2, x)
```

```
ans =
-(2^(1/2)*((3*sin(x))/x + cos(x)*(3/x^2 - 1)))/(pi^(1/2)*x^(1/2))
```

Differentiate the expressions involving the Bessel functions of the second kind:

```
syms x y
diff(bessely(1, x))
diff(diff(bessely(0, x^2 + x*y -y^2), x), y)
```

```
ans =
bessely(0, x) - bessely(1, x)/x
```

```
ans =
- bessely(1, x^2 + x*y - y^2) -...
(2*x + y)*(bessely(0, x^2 + x*y - y^2)*(x - 2*y) -...
(bessely(1, x^2 + x*y - y^2)*(x - 2*y))/(x^2 + x*y - y^2))
```

Call `bessely` for the matrix `A` and the value 1/2. The result is a matrix of the Bessel functions `bessely(1/2, A(i,j))`.

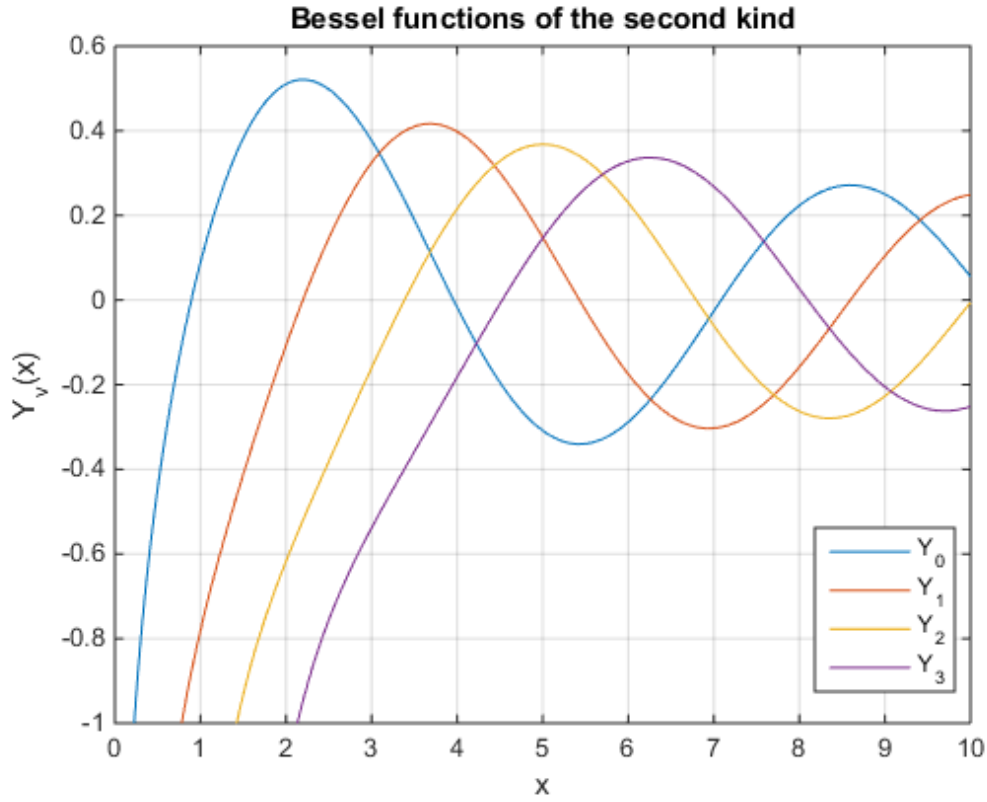
```
syms x
A = [-1, pi; x, 0];
bessely(1/2, A)
```

```
ans =
[          (2^(1/2)*cos(1)*i)/pi^(1/2), 2^(1/2)/pi]
[ -(2^(1/2)*cos(x))/(pi^(1/2)*x^(1/2)),        Inf]
```

Plot the Bessel functions of the second kind for v = 0, 1, 2, 3:

```
syms x y
for nu = [0, 1, 2, 3]
  ezplot(bessely(nu, x), [0, 10])
  hold on
end
```

```
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')
hold off
```



More About

Bessel Functions 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, $Y_\nu(z)$:

$$w(z)=C_1J_\nu(z)+C_2Y_\nu(z)$$

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:

$$J_\nu(z)=\frac{(z/2)^\nu}{\sqrt{\pi}\,\Gamma(\nu+1/2)}\int_0^\pi\cos\left(z\cos(t)\right)\sin(t)^{2\nu}\,dt$$

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`

beta

Beta function

Syntax

```
beta(x,y)
```

Description

`beta(x,y)` returns the beta function 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(i))/(2*gamma(3/2 + i)), gamma(i)^2/gamma(2*i),...
 (pi*(1/2 + i/2))/sinh(pi)]
```

Compute the beta function for negative parameters. If one or both arguments are negative numbers, convert these numbers to symbolic objects:

```
[beta(sym(-3), 2), beta(sym(-1/3), 2), beta(sym(-3), 4),  beta(sym(-3), -2)]
```

```
ans =
[ 1/6, -9/2, Inf, Inf]
```

Call `beta` for the matrix `A` and the value `1`. The result is a matrix of the beta functions `beta(A(i,j),1)`:

```
A = sym([1 2; 3 4]);
beta(A,1)
```

```
ans =
[   1, 1/2]
[ 1/3, 1/4]
```

Differentiate the beta function, then substitute the variable t with the value 2/3 and approximate the result using `vpa`:

```
syms t
u = diff(beta(t^2 + 1, t))
vpa(subs(u, t, 2/3), 10)
```

```
u =
beta(t, t^2 + 1)*(psi(t) + 2*t*psi(t^2 + 1) -...
psi(t^2 + t + 1)*(2*t + 1))

ans =
```

```
-2.836889094
```

Expand these beta functions:

```
syms x y
expand(beta(x, y))
expand(beta(x + 1, y - 1))
```

```
ans =
(gamma(x)*gamma(y))/gamma(x + y)

ans =
-(x*gamma(x)*gamma(y))/(gamma(x + y) - y*gamma(x + y))
```

More About

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(0, 0)` returns `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

`gamma` | `factorial` | `nchoosek` | `psi`

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 a 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 the 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

catalan

Catalan constant

Syntax

```
catalan
```

Description

`catalan` represents the Catalan constant. To get a floating-point approximation with the current precision set by `digits`, use `vpa(catalan)`.

Examples

Approximate the 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)
```

More About

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`

ccode

C code representation of symbolic expression

Syntax

```
ccode(s)
ccode(s,'file',fileName)
```

Description

`ccode(s)` returns a fragment of C that evaluates the symbolic expression `s`.

`ccode(s,'file',fileName)` writes an "optimized" C code fragment that evaluates the symbolic expression `s` to the file named `fileName`. "Optimized" means intermediate variables are automatically generated in order to simplify the code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
ccode(f)
```

return

```
t0 = x-(x*x)*(1.0/2.0)+(x*x*x)*(1.0/3.0)-(x*x*x*x)*(1.0/4.0)+...
(x*x*x*x*x)*(1.0/5.0);
```

The statements

```
H = sym(hilb(3));
ccode(H)
```

return

```
  H[0][0] = 1.0;
  H[0][1] = 1.0/2.0;
  H[0][2] = 1.0/3.0;
  H[1][0] = 1.0/2.0;
  H[1][1] = 1.0/3.0;
  H[1][2] = 1.0/4.0;
  H[2][0] = 1.0/3.0;
  H[2][1] = 1.0/4.0;
  H[2][2] = 1.0/5.0;
```

The statements

```
syms x
z = exp(-exp(-x));
ccode(diff(z,3),'file','ccodetest')
```

return a file named `ccodetest` containing the following:

```
  t2 = exp(-x);
  t3 = exp(-t2);
  t0 = t3*exp(x*(-2.0))*(-3.0)+t3*exp(x*(-3.0))+t2*t3;
```

See Also

`fortran` | `latex` | `matlabFunction` | `pretty`

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

`round` | `floor` | `fix` | `frac`

char

Convert symbolic objects to strings

Syntax

```
char(A)
```

Description

`char(A)` converts a symbolic scalar or a symbolic array to a string.

Input Arguments

A

Symbolic scalar or symbolic array.

Examples

Convert symbolic expressions to strings, and then concatenate the strings:

```
syms x
y = char(x^3 + x^2 + 2*x - 1);
name = [y, ' represents a polynomial expression']
```

```
name =
2*x + x^2 + x^3 - 1 represents a polynomial expression
```

Note that `char` changes the order of the terms in the resulting string.

Convert a symbolic matrix to a string:

```
A = sym(hilb(3))
char(A)
```

```
A =
```

```
[   1, 1/2, 1/3]
[ 1/2, 1/3, 1/4]
[ 1/3, 1/4, 1/5]

ans =
matrix([[1,1/2,1/3],[1/2,1/3,1/4],[1/3,1/4,1/5]])
```

More About

Tips

- `char` can change term ordering in an expression.

See Also

`sym` | `double` | `pretty`

charpoly

Characteristic polynomial of matrix

Syntax

```
charpoly(A)
charpoly(A,var)
```

Description

`charpoly(A)` returns a vector of the coefficients of the characteristic polynomial of A. If A is a symbolic matrix, `charpoly` returns a symbolic vector. Otherwise, it returns a vector of double-precision values.

`charpoly(A,var)` returns the characteristic polynomial of A in terms of var.

Input Arguments

`A`

Matrix.

`var`

Free symbolic variable.

Default: If you do not specify `var`, `charpoly` returns a vector of coefficients of the characteristic polynomial instead of returning the polynomial itself.

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
```

More About

Characteristic Polynomial of a Matrix

The characteristic polynomial of an n-by-n matrix A is the polynomial $p_A(x)$, such that

$$p_A(x) = \det(xI_n - A)$$

Here I_n is the n-by-n identity matrix.

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`

chebyshevT

Chebyshev polynomials of the first kind

Syntax

```
chebyshevT(n,x)
```

Description

`chebyshevT(n,x)` represents the nth degree Chebyshev polynomial of the first kind 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 =
0.96311412681708523377857128671 8
```

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 =
-32939057913375008974828134 72768.0
```

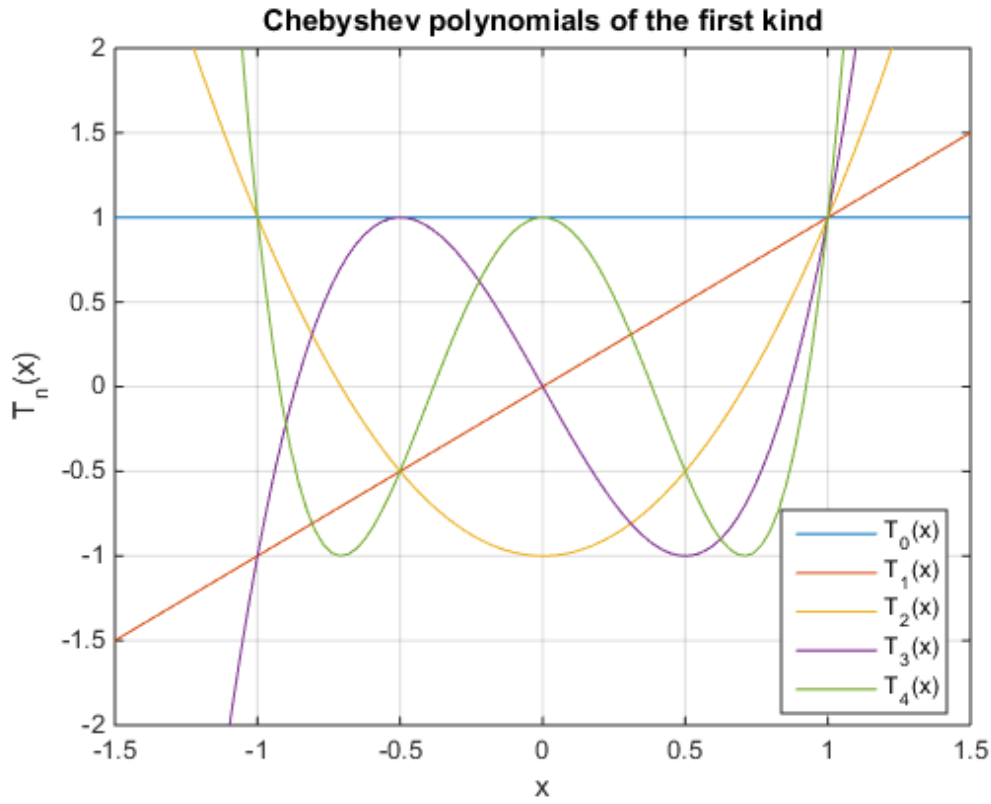
Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

```
subs(vpa(T500), x, sym(1/3))

ans =
120229243134934213275703836672O.0
```

Plot Chebyshev Polynomials of the First Kind

Plot these five Chebyshev polynomials of the first kind.

```
syms x y
for n = [0, 1, 2, 3, 4]
  ezplot(chebyshevT(n, x))
  hold on
end

hold off

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.

More About

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`

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 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 =
630806801959501609121108459520.0
```

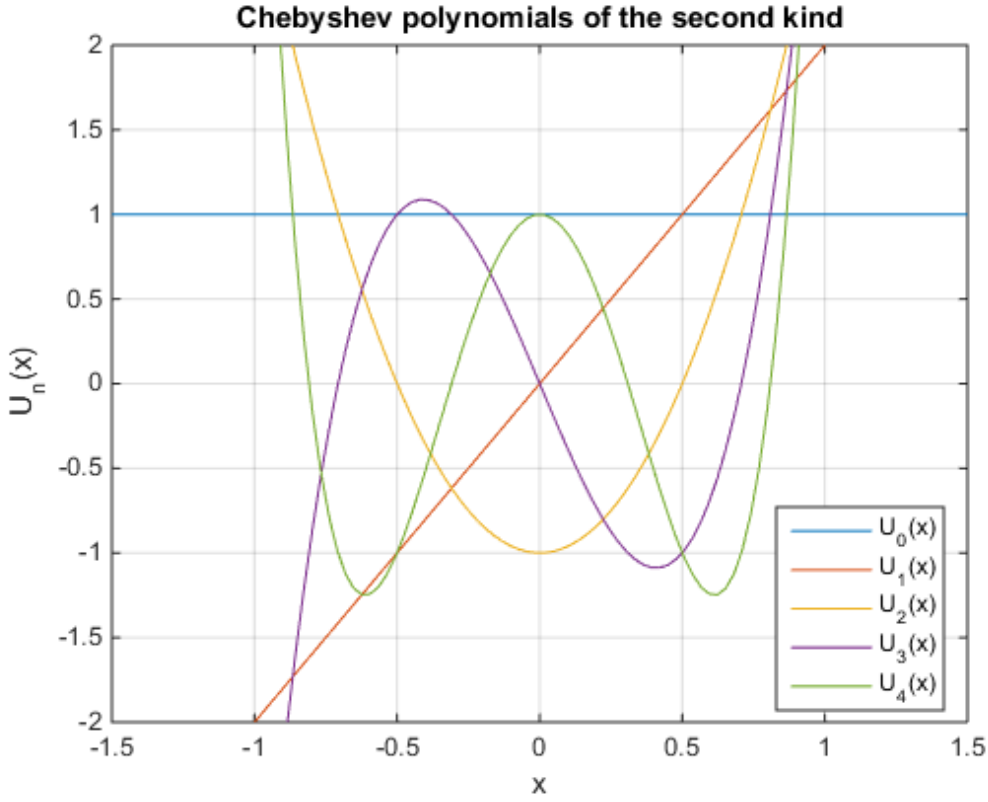
Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

```
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.

```
syms x y
for n = [0, 1, 2, 3, 4]
  ezplot(chebyshevU(n, x))
  hold on
end

hold off

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.

More About

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`

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
```

Create the symbolic expression using `sym`. With this approach, you do not create symbolic variables corresponding to the terms of the expression. Nevertheless, `children` finds the terms of the expression:

```
children(sym('a + b + c'))
```

```
ans =
[ a, b, c]
```

Find the child subexpressions of this equation. The child subexpressions of an equation are the left and right sides of that equation.

```
syms x y
children(x^2 + x*y == y^2 + 1)
```

```
ans =
[ x^2 + y*x, y^2 + 1]
```

Find the child subexpressions of this inequality. The child subexpressions of an inequality are the left and right sides of that inequality.

```
children(sin(x) < cos(x))
```

```
ans =
[ sin(x), cos(x)]
```

Call the `children` function for this matrix. The result is the cell array containing the child subexpressions of each element of the matrix.

```
syms x y
s = children([x + y, sin(x)*cos(y); x^3 - y^3, exp(x*y^2)])

s =
    [1x2 sym]    [1x2 sym]
    [1x2 sym]    [1x1 sym]
```

To access the contents of cells in the cell array, use braces:

```
s{1:4}

ans =
[ x, y]

ans =
[ x^3, -y^3]

ans =
[ cos(y), sin(x)]

ans =
x*y^2
```

More About

- "Create Symbolic Expressions" on page 1-8

See Also

`coeffs` | `numden` | `subs`

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. Otherwise, this syntax throws an error.

`[T,p] = chol(A)` computes the Cholesky factorization of A. This syntax does not error if A is not a Hermitian positive definite matrix. If A is a Hermitian positive definite matrix, then p is 0. Otherwise, T is `sym([])`, and p is a positive integer (typically, `p = 1`).

`[T,p,S] = chol(A)` returns a permutation matrix S, such that `T'*T = S'*A*S`, and the value `p = 0` if matrix A is Hermitian positive definite. Otherwise, it returns a positive integer p and an empty symbolic object `S = sym([])`.

`[T,p,s] = chol(A,'vector')` returns the permutation information as a vector s, such that `A(s,s) = T'*T`. If A is not recognized as a Hermitian positive definite matrix, then `p` is a positive integer and `s = sym([])`.

`___ = chol(A,'lower')` returns a lower triangular matrix T, such that `T*T' = A`.

`___ = chol(A,'noCheck')` skips checking whether matrix A is Hermitian positive definite. 'noCheck' lets you compute Cholesky factorization of a matrix that contains symbolic parameters without setting additional assumptions on those parameters.

`___ = chol(A,'real')` computes the Cholesky factorization of A using real arithmetic. In this case, `chol` computes a symmetric factorization `A = T.'*T` instead of a Hermitian factorization `A = T'*T`. This approach is based on the fact that if A is real and symmetric, then `T'*T = T.'*T`. Use `'real'` to avoid complex conjugates in the result.

`___ = chol(A,'lower','noCheck','real')` computes the Cholesky factorization of A with one or more of these optional arguments: 'lower', 'noCheck', and 'real'. These optional arguments can appear in any order.

`[T,p,s] = chol(A,'lower','vector','noCheck','real')` computes the Cholesky factorization of A and returns the permutation information as a vector s. You can use one or more of these optional arguments: 'lower', 'noCheck', and 'real'. These optional arguments can appear in any order.

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 =
     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)*i]

isAlways(A == T'*T,'Unknown','false')

ans =
     0     0
     0     0
```

Compute the Cholesky factorization of this matrix. To skip checking whether it is Hermitian positive definite, use `'noCheck'`. By default, `chol` computes a Hermitian factorization `A = T'*T`. Thus, the result contains complex conjugates.

```
syms a b
A = [a b; b a];
T = chol(A, 'noCheck')
```

```
T =
[ a^(1/2),                    conj(b)/conj(a^(1/2))]
[       0, (a*abs(a) - abs(b)^2)^(1/2)/abs(a)^(1/2)]
```

To avoid complex conjugates in the result, use `'real'`:

```
T = chol(A, 'noCheck', 'real')
```

```
T =
[ a^(1/2),             b/a^(1/2)]
[       0, ((a^2 - b^2)/a)^(1/2)]
```

When you use this flag, `chol` computes a symmetric factorization `A = T.'*T` instead of a Hermitian factorization `A = T'*T`. To make `isAlways` return logical `0` (`false`) for undecidable conditions, set `Unknown` to `false`.

```
isAlways(A == T.'*T)
```

```
ans =
     1     1
     1     1
```

```
isAlways(A == T'*T,'Unknown','false')
```

```
ans =
     0     0
     0     0
```

More About

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` | `linalg::factorCholesky` | `linalg::isHermitian` | `linalg::isPosDef` | `lu` | `qr` | `svd` | `transpose` | `vpa`

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`

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 a Particular 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 3-24
- "Close MuPAD Notebooks from MATLAB" on page 3-16

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`

coeffs

Coefficients of polynomial

Syntax

```
C = coeffs(p)
C = coeffs(p,var)
C = coeffs(p,vars)
[C,T] = coeffs( ___ )
```

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.

Examples

Coefficients of a Univariate Polynomial

Find the coefficients of this univariate polynomial:

```
syms x
c = coeffs(16*x^2 + 19*x + 11)
```

```
c =
```

```
[ 11, 19, 16]
```

Coefficients of a Multivariate Polynomial with Respect to a 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 a 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 a Univariate Polynomial

Find the coefficients and the corresponding terms of this univariate polynomial:

```
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 a 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]
```

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 vector | symbolic number | symbolic expression

Coefficients of polynomial, returned as a vector of symbolic numbers and expressions. If there is only one coefficient and one corresponding term, then C is returned as a scalar.

`T` — Terms of polynomial
symbolic vector | symbolic expression | symbolic number

Terms of polynomial, returned as a vector of symbolic expressions and numbers. If there is only one coefficient and one corresponding term, then T is returned as a scalar.

See Also
`poly2sym` | `sym2poly`

collect

Collect coefficients

Syntax

```
collect(P)
collect(P,var)
```

Description

`collect(P)` rewrites P in terms of the powers of the default variable determined by `symvar`.

`collect(P,var)` rewrites P in terms of the powers of the variable var. If P is a vector or matrix, this syntax regards each element of P as a polynomial in var.

Examples

Collect Coefficients in Terms of the Powers of a Default Variable

Collect the coefficients of this symbolic expression:

```
syms x
collect((exp(x) + x)*(x + 2))
```

```
ans =
x^2 + (exp(x) + 2)*x + 2*exp(x)
```

Because you did not specify the variable of a polynomial, `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 in Terms of the Powers of a Particular Variable

Rewrite this symbolic expression specifying the variables in terms of which you want to collect the coefficients:

```
syms x y
collect(x^2*y + y*x - x^2 - 2*x, x)
collect(x^2*y + y*x - x^2 - 2*x, y)

ans =
(y - 1)*x^2 + (y - 2)*x

ans =
(x^2 + x)*y - x^2 - 2*x
```

Collect Coefficients in Terms of the Powers of i and pi

Rewrite these expressions in terms of the powers of `i` and `pi`, respectively:

```
syms x y
collect(2*x*i - 3*i*y, i)
collect(x*pi*(pi - y) + x*(pi + i) + 3*pi*y, pi)

ans =
(2*x - 3*y)*i

ans =
x*pi^2 + (x + 3*y - x*y)*pi + x*i
```

Collect Coefficients for Each Element of a Matrix

If the argument is a vector or a matrix, then `collect` rewrites each element:

```
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]
```

Input Arguments

`P` — Input expression

symbolic expression | symbolic function | symbolic vector | symbolic matrix

Input expression, specified as a symbolic expression, function, vector, or matrix.

`var` — Variable in terms of which you collect coefficients

symbolic variable | symbolic expression

Variable in terms of which you collect the coefficients, specified as a symbolic variable or symbolic expression, such as `i` or `pi`.

See Also

`combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction` | `symvar`

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 a 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 the Product of the 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`

colspace

Column space of matrix

Syntax

```
B = colspace(A)
```

Description

`B = colspace(A)` returns a matrix whose columns form a basis for the column space of `A`. The matrix `A` can be symbolic or numeric.

Examples

Find the basis for the column space of this matrix:

```
A = sym([2,0;3,4;0,5])
B = colspace(A)
```

```
A =
[ 2, 0]
[ 3, 4]
[ 0, 5]

B =
[     1,   0]
[     0,   1]
[ -15/8, 5/4]
```

See Also

`null` | `size`

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. `combine` implements 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 or Exponent

Combine powers of the same base.

```
syms x y z
combine(x^y*x^z)
```

```
ans =
x^(y + z)
```

If you use numeric arguments, you must convert at least one numeric argument into a symbolic value using `sym` to prevent MATLAB from evaluating the expression.

```
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 the expression e^1 from being evaluated.

Powers of the Same Exponent

`combine` combines powers with the same exponents in certain cases.

```
combine(sqrt(sym(2))*sqrt(3))

ans =
6^(1/2)
```

`combine` does not usually combine powers with the same exponent. This is because the internal simplifier applies the same rules in the opposite direction.

```
syms x y
combine(y^5*x^5)

ans =
x^5*y^5
```

Terms with Inverse Tangent Function Calls

Combine these expressions in terms of the inverse tangent function by specifying the target argument as `atan`.

```
syms a b
assume(abs(a*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 Sine and Cosine Function Calls

Rewrite products of sine and cosine functions as a sum of sine and cosine functions with more complicated arguments by specifying the target argument as `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
```

Note that `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 Calls to the 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.

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)
```

In the complex plane the rule above does not hold. Thus, the action of `combine` has a dependence on properties of the variables appearing in the input.

```
syms a b
S = log(a) + log(b);
combine(S,'log')
```

```
ans =
log(a) + log(b)
```

To apply this rule you can set certain assumptions about the input variables.

```
syms a b
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, `combine` applies the rule if you relax the analytic constraints using `IgnoreAnalyticConstraints`.

```
syms a b
S = log(a) + log(b);
combine(S,'log','IgnoreAnalyticConstraints',true)
```

```
ans =
 log(a*b)
```

Multiple Input Expressions in One Call

To evaluate multiple expressions in one function call, use 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.

Tip S is the symbolic expression to rewrite. It accepts only symbolic data types as inputs. For example, `sin(3)` results in an error because `3` is not a symbolic object. Instead, use `sin(sym(3))`.

`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' | 'log' | 'sincos' | 'sinhcosh'`

Target function, specified as a string. The rewriting rules apply only to calls to this function. It can take the values of atan, exp, gamma, log, sincos, and sinhcosh.

Name-Value Pair Arguments

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`	Uses strict simplification rules.
`true`	Applies purely algebraic simplifications that generally not correct, but can give simpler results. For example, `log(a) + log(b) = log(a*b)`. This is most useful in simplifying expressions where direct use of the solver returns complicated results. Note that 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 rewriting rules applied to it, returned as a symbolic variable, number, expression, or as a vector or matrix of symbolic variables, numbers, or expressions.

More About

Algorithms

`combine` applies the following rewriting rules to the input expression `S`, depending on the value of the target argument `T`.

- When T = `atan` where x and y are such that `|xy| < 1`,

$$\mathrm{atan}(x)+\mathrm{atan}(y)=\mathrm{atan}\left(\frac{x+y}{1-xy}\right)$$

- When T = `exp`, `combine` applies the following rewriting rules where valid, reacting to the properties of the input arguments:

$$e^a e^b = e^{a+b}$$

$$(e^a)^b = e^{ab}$$

- When T = `gamma`:

$$a\Gamma(a)=\Gamma(a+1)$$

and,

$$\frac{\Gamma(a+1)}{\Gamma(a)}=a$$

For positive integers `n`,

$$\Gamma(-a)\Gamma(a)=-\frac{\pi}{\sin(\pi a)}$$

- 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 = `sincos`,

$$\sin\left(x\right)\sin\left(y\right) = \frac{\cos\left(x-y\right)}{2} - \frac{\cos\left(x+y\right)}{2}$$

`combine` applies similar rules for `sin(x) cos(y)` and `cos(x) cos(y)`.

`combine` applies rules recursively to powers of `sin` and `cos` with positive integral exponents.

- When T = `sinhcosh`:

$$\sinh\left(x\right)\sinh\left(y\right) = \frac{\cosh\left(x+y\right)}{2} - \frac{\cosh(x-y)}{2}$$

`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.

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

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`

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 +...
1.426811478813982694812838004 23e-38*i

condN1 =
748.0

condNf =
526.158821079719236517033364845

condNi =
748.0
```

Hilbert matrices are classic examples of ill-conditioned matrices.

More About

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`

conj

Symbolic complex conjugate

Syntax

```
conj(X)
```

Description

`conj(X)` is the complex conjugate of `X`.

For a complex `X`, `conj(X) = real(X) - i*imag(X)`.

See Also

`real` | `imag`

cos

Symbolic cosine function

Syntax

```
cos(X)
```

Description

`cos(X)` returns the cosine function 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)]
```

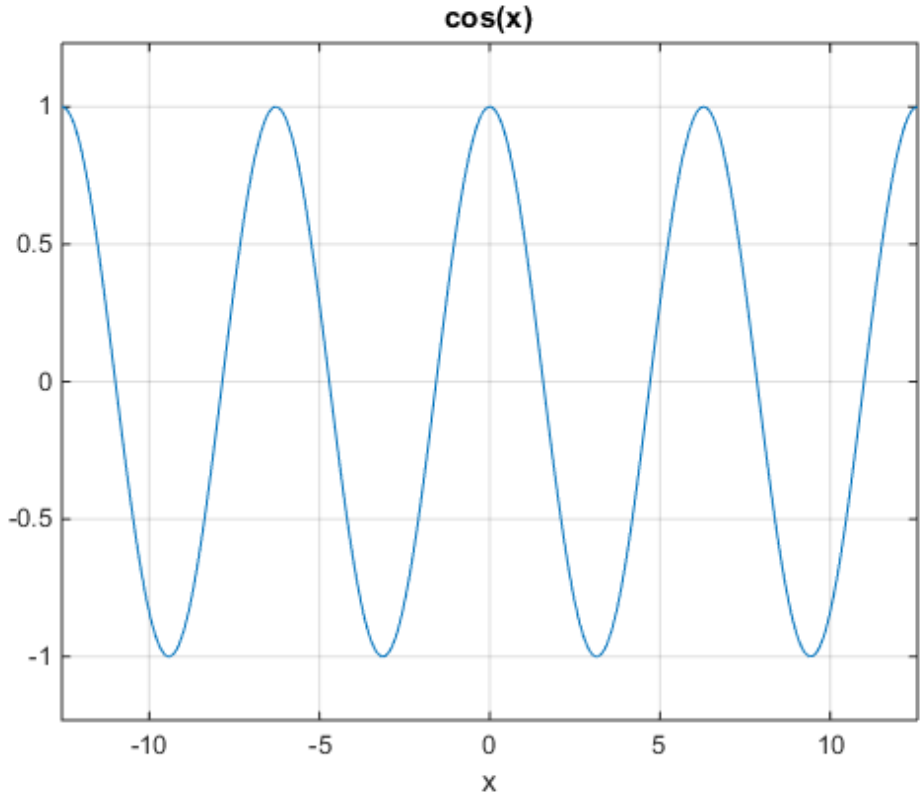
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.41614683654714238699756822950076,...
-1.0,...
0.86602540378443864676372317075294,...
-0.62348980185873353052500488400424,...
0.0044256979880507857483550247239416]
```

Plot the Cosine Function

Plot the cosine function on the interval from -4π to 4π .

```
syms x
ezplot(cos(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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*i)/2 + exp(x*i)/2
```

Input Arguments

X — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

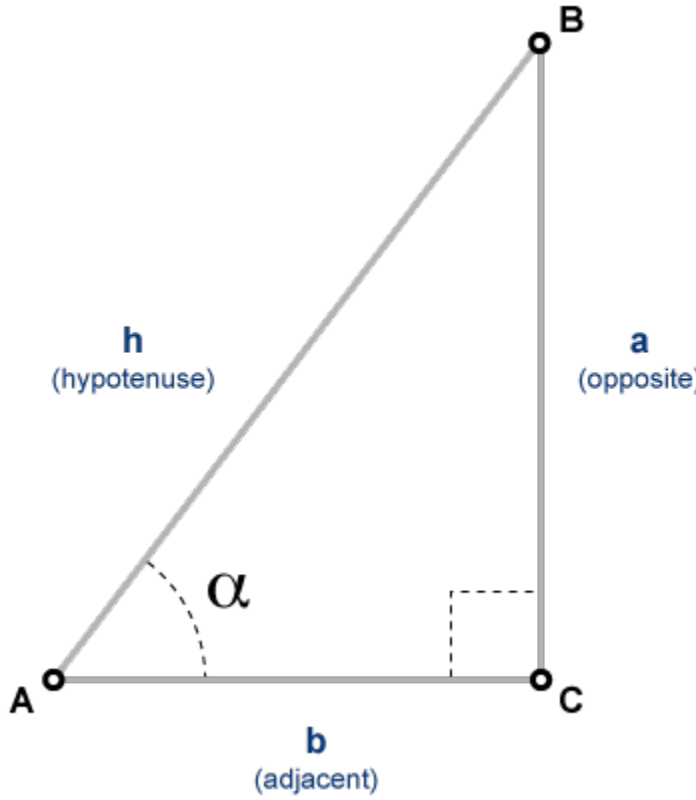
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

More About

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

$$\text{cosine}(\alpha) = \frac{e^{i\alpha} + e^{-i\alpha}}{2}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cot` | `csc` | `sec` | `sin` | `tan`

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*2*i)/7), 0]
```

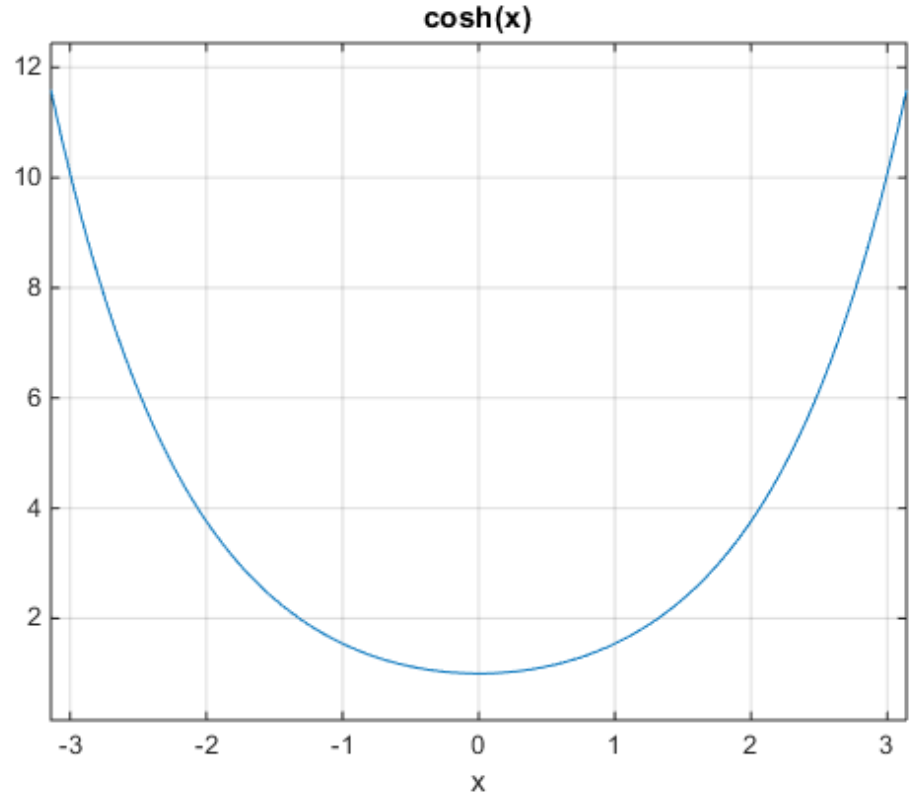
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 3.762195691083631459562213477737,...
-1.0,...
0.866025403784438646763723170752 94,...
-0.623489801858733530525004884004 24,...
0]
```

Plot the Hyperbolic Cosine Function

Plot the hyperbolic cosine function on the interval from $-\pi$ to π.

```
syms x
ezplot(cosh(x), [-pi, pi])
grid on
```



Handle Expressions Containing the 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`

coshint

Hyperbolic cosine integral function

Syntax

```
coshint(X)
```

Description

`coshint(X)` returns the hyperbolic cosine integral function 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 =
[ pi*i + coshint(1), -Inf, coshint(1/2), coshint(1), coshint(pi/2), coshint(pi)]
```

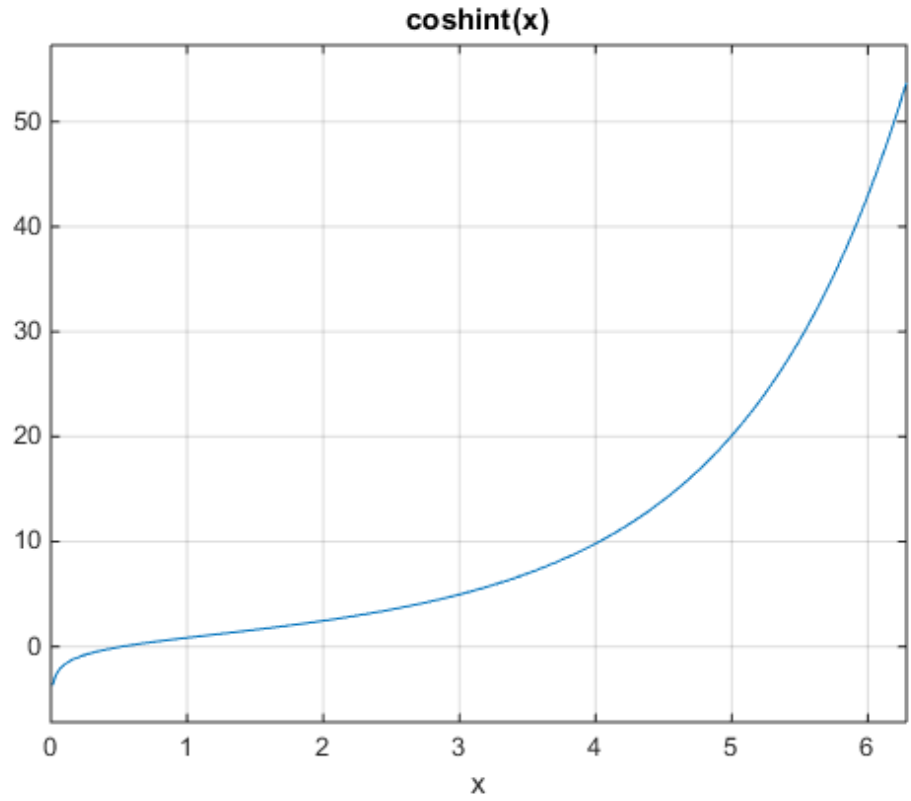
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.837866940980208240894678579435 76...
 + 3.14159265358979323846264338327 95*i,...
-Inf,...
-0.052776844956493615913136063326141,...
0.837866940980208240894678579435 76,...
1.712660736484428107995156989779 6,...
5.458734044216068198001487897779 8]
```

Plot the Hyperbolic Cosine Integral Function

Plot the hyperbolic cosine integral function on the interval from 0 to `2*pi` .

```
syms x
ezplot(coshint(x), [0, 2*pi])
grid on
```



Handle Expressions Containing the 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.

More About

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:

$$\gamma = \lim_{n \to \infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) - \log(n)$$

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`

cosint

Cosine integral function

Syntax

```
cosint(X)
```

Description

`cosint(X)` returns the cosine integral function 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 =
[ pi*i + cosint(1), -Inf, cosint(pi/2), cosint(pi), cosint(1)]
```

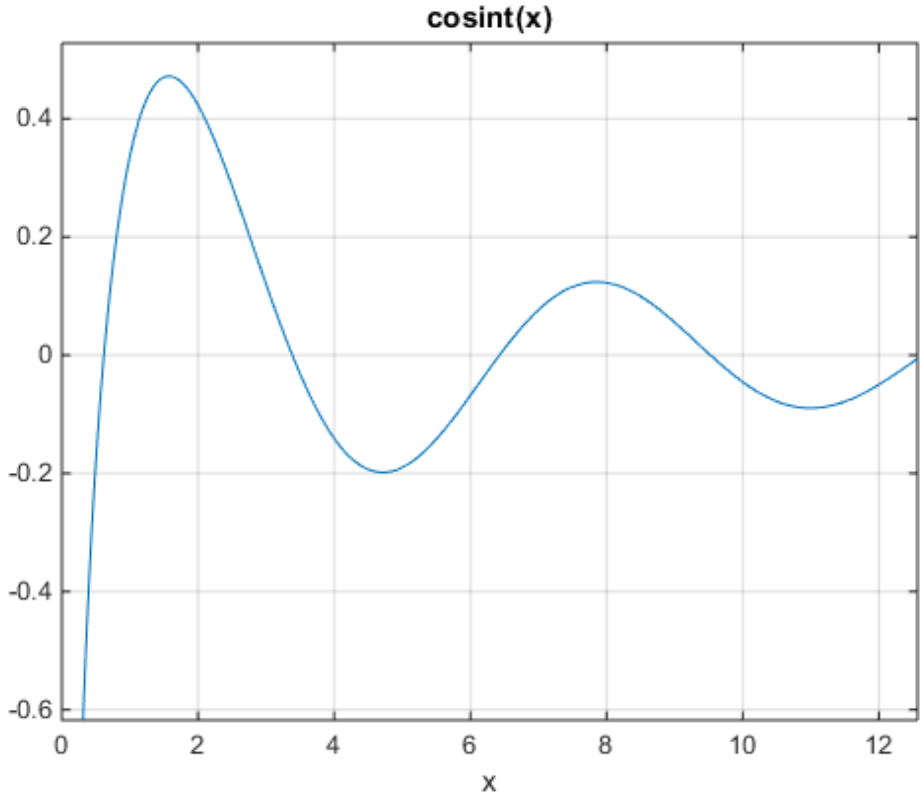
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.33740392290096813466264620388915...
 + 3.14159265358979323846264338327795*i,...
-Inf,...
0.47200065143956865077760610761413,...
0.07366791204642548599010096523015,...
0.33740392290096813466264620388915]
```

Plot the Cosine Integral Function

Plot the cosine integral function on the interval from 0 to `4*pi`.

```
syms x
ezplot(cosint(x), [0, 4*pi])
grid on
```



Handle Expressions Containing the 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.

More About

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:

$$\gamma = \lim_{n \to \infty} \left(\sum_{k=1}^{n} \frac{1}{k} \right) - \log(n)$$

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 | coshint | eulergamma | int | sinhint | sinint | ssinint

cot

Symbolic cotangent function

Syntax

```
cot(X)
```

Description

`cot(X)` returns the cotangent function 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)]
```

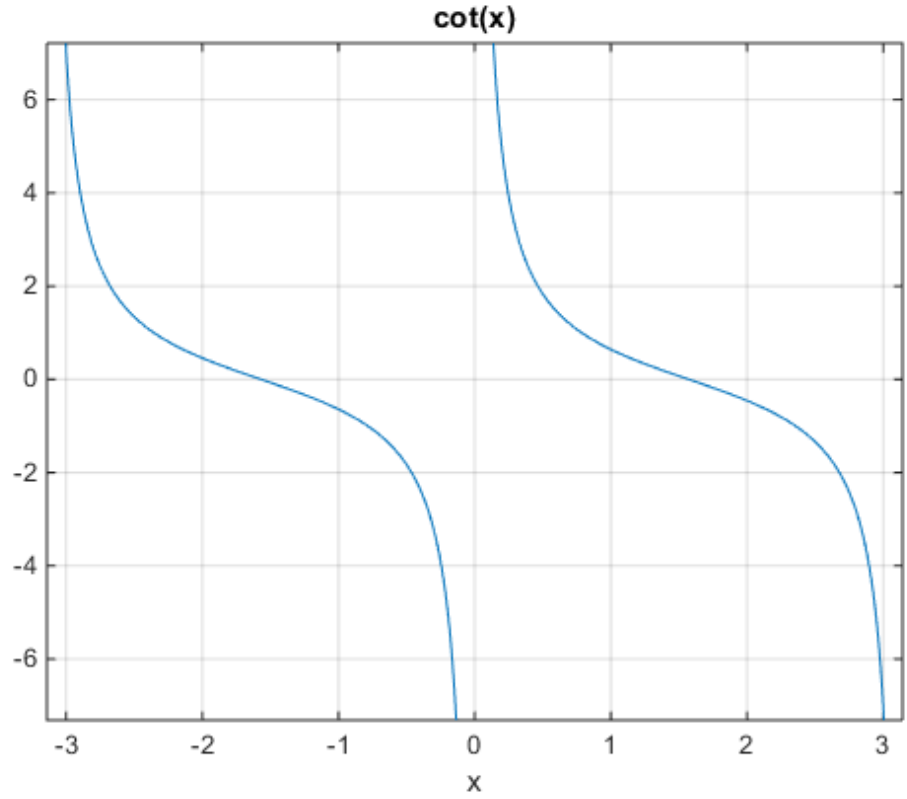
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.45765755436028576375027741043205,...
0,...
1.7320508075688772935274463415059,...
-0.79747338888240396141568825421443,...
-0.0044257413313241136855482762848043]
```

Plot the Cotangent Function

Plot the cotangent function on the interval from $-\pi$ to π.

```
syms x
ezplot(cot(x), [-pi, pi])
grid on
```



Handle Expressions Containing the 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 + (pi/2 - x)^3/3 + (2*(pi/2 - x)^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*2*i)*i + i)/(exp(x*2*i) - 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.

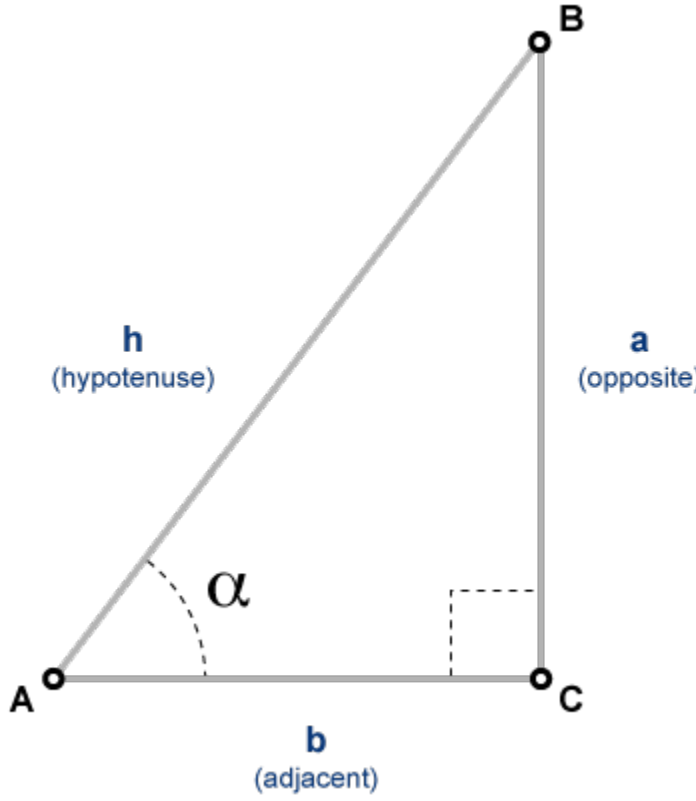
More About

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

$$\text{cotangent}(\alpha) = \frac{i\left(e^{i\alpha} + e^{-i\alpha}\right)}{\left(e^{i\alpha} - e^{-i\alpha}\right)}.$$

.

See Also

acos | acot | acsc | asec | asin | atan | cos | csc | sec | sin | tan

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)*i)/3, -3^(1/2)*i, -coth((pi*2*i)/7), 0]
```

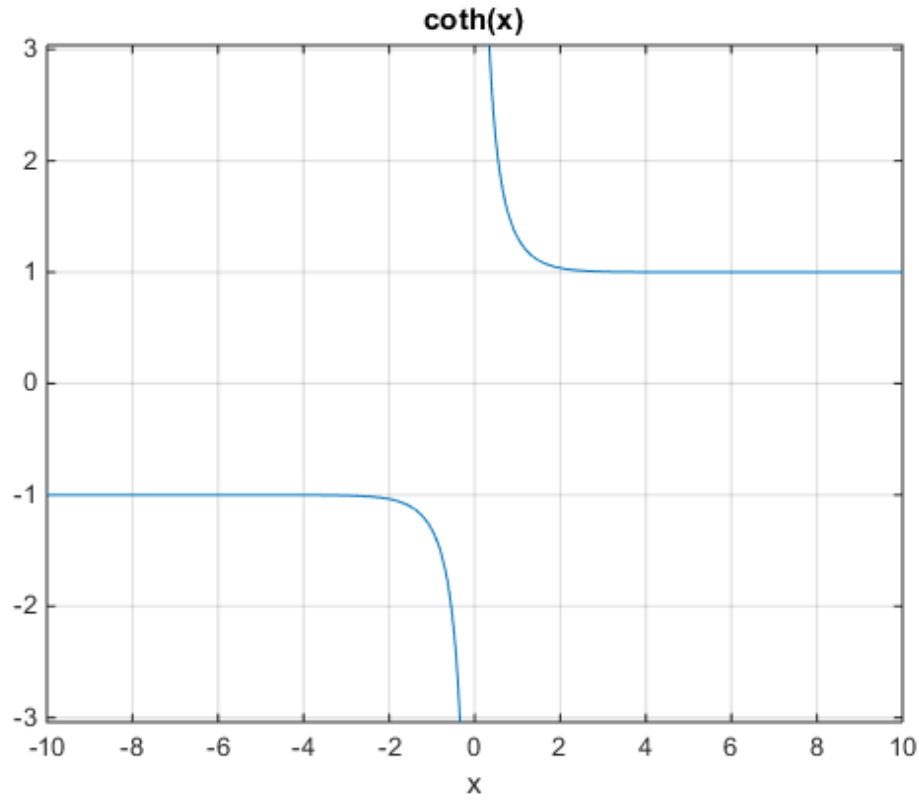
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.037314720727548095877809764767 8,...
0.577350269189625764509148780501 96*i,...
-1.732050807568877293527446341505 9*i,...
0.797473388882403961415688254214 43*i,...
0]
```

Plot the Hyperbolic Cotangent Function

Plot the hyperbolic cotangent function on the interval from -10 to 10.

```
syms x
ezplot(coth(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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*i)/2 + ((pi*i)/2 - x)^3/3 - (2*((pi*i)/2 - x)^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`

csc

Symbolic cosecant function

Syntax

```
csc(X)
```

Description

`csc(X)` returns the cosecant function 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)]
```

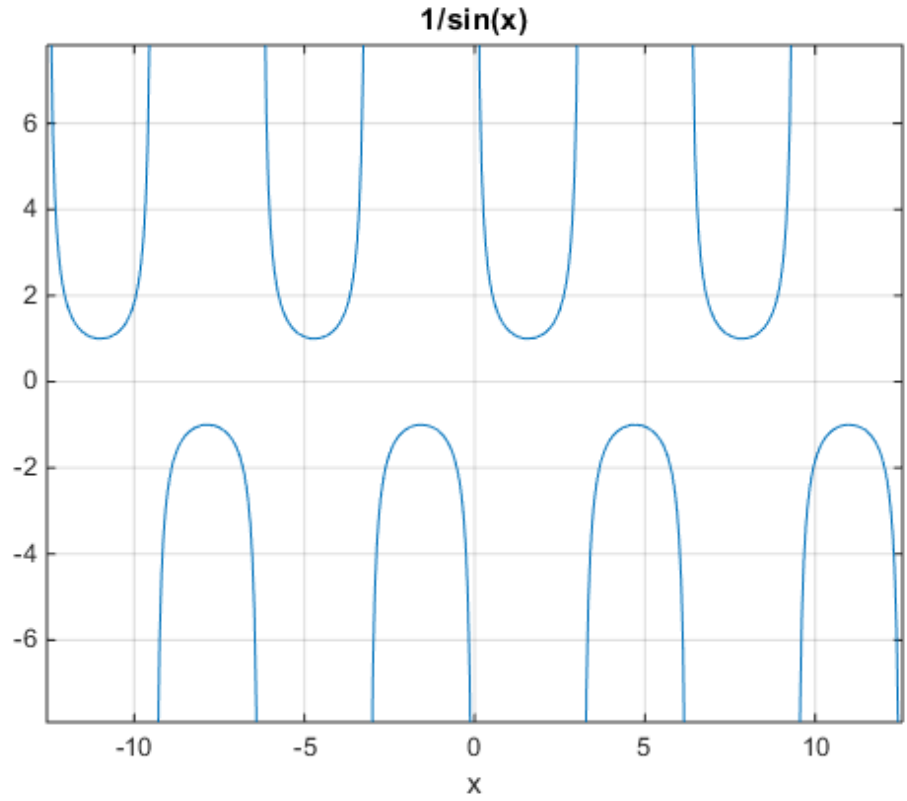
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.099750170294616466756697397026 3,...
-1.0,...
2.0,...
1.279048007689932605747850607271 4,...
-1.000009793545209131387464450355 1]
```

Plot the Cosecant Function

Plot the cosecant function on the interval from -4π to 4π.

```
syms x
ezplot(csc(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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 =
(pi/2 - x)^2/2 + (5*(pi/2 - x)^4)/24 + 1
```

Rewrite the cosecant function in terms of the exponential function:

```
rewrite(csc(x), 'exp')
```

```
ans =
1/((exp(-x*i)*i)/2 - (exp(x*i)*i)/2)
```

Input Arguments

X — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

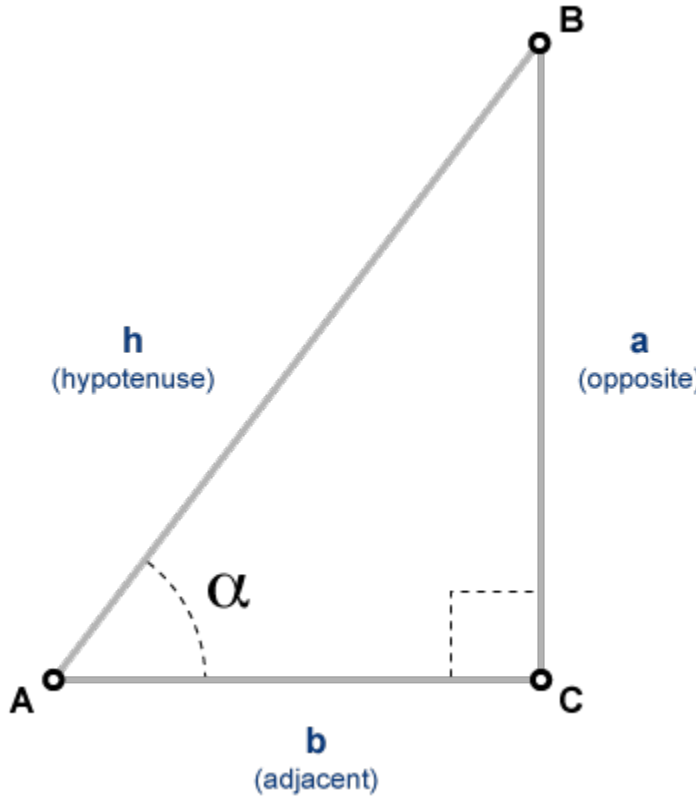
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

More About

Cosecant Function

The cosecant of an angle, α, defined with reference to a right angled triangle is

$$\operatorname{cosec}(\alpha) = \frac{1}{\sin(\alpha)} = \frac{\text{hypotenuse}}{\text{opposite side}} = \frac{h}{a}.$$



The cosecant of a complex angle, α, is

$$\text{cosecant}(\alpha) = \frac{2i}{e^{i\alpha} - e^{-i\alpha}}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sin` | `tan`

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), i, Inf, -(3^(1/2)*2*i)/3, 1/sinh((pi*2*i)/7), -i]
```

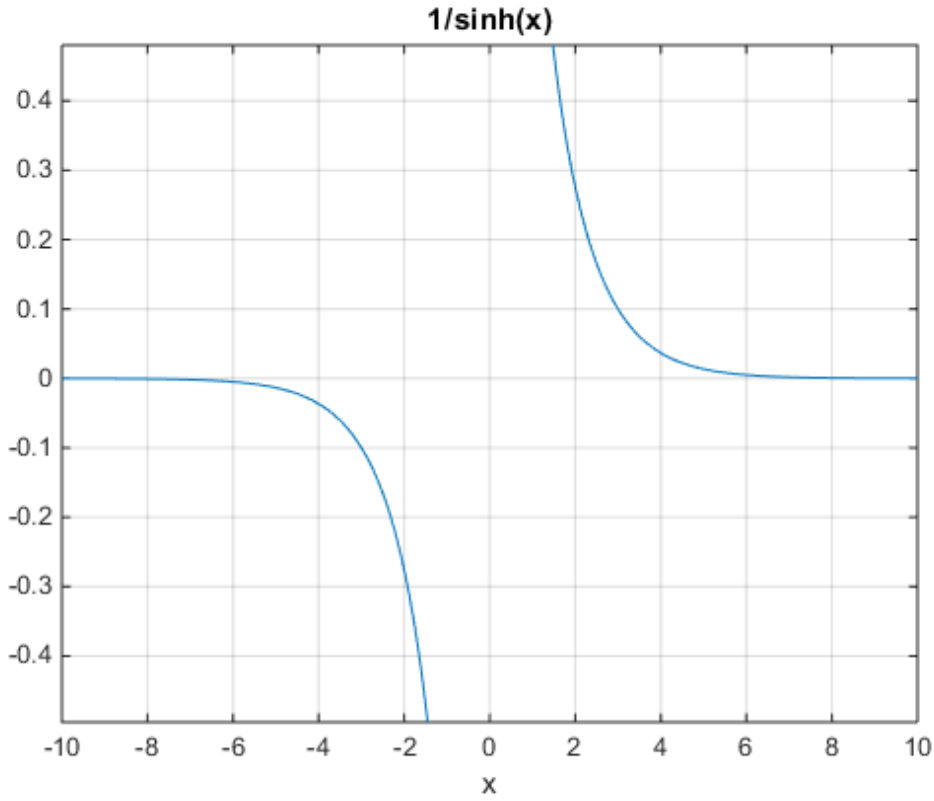
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.27572056477178320775835148216303,...
1.0*i,...
Inf,...
-1.1547005383792515290182975610039*i,...
-1.2790480076899326057478506072714*i,...
-1.0*i]
```

Plot the Hyperbolic Cosecant Function

Plot the hyperbolic cosecant function on the interval from -10 to 10.

```
syms x
ezplot(csch(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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 =
(((pi*i)/2 - x)^2*i)/2 - (((pi*i)/2 - x)^4*5*i)/24 - i
```

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`

cumprod

Symbolic cumulative product

Syntax

```
cumprod(A)
cumprod(A,dim)
```

Description

`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.

`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.

Examples

Cumulative Product of a Vector

Create vector `V` 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 a Symbolic Matrix

Create matrix A containing symbolic numbers and matrix B containing symbolic expressions:

```
syms x y
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
B = [x, 2*x + 1, 3*x + 2; 1/y, y, 2*y]
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

B =
[   x, 2*x + 1, 3*x + 2]
[ 1/y,       y,     2*y]
```

Compute the cumulative products of the elements of A and B. By default, `cumprod` returns the cumulative product of each column:

```
productA = cumprod(A)
productB = cumprod(B)
```

```
productA =
[ 0, 1,  2]
[ 0, 4, 10]
[ 0, 8, 30]

productB =
[   x,     2*x + 1,       3*x + 2]
[ x/y, y*(2*x + 1), 2*y*(3*x + 2)]
```

Cumulative Product of Each Row in a Symbolic Matrix

Create matrix A containing symbolic numbers and matrix B containing symbolic expressions:

```
syms x y
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
B = [x, 2*x + 1, 3*x + 2; 1/y, y, 2*y]
```

```
A =
[ 0, 1, 2]
```

```
[ 3, 4, 5]
[ 1, 2, 3]

B =
[   x, 2*x + 1, 3*x + 2]
[ 1/y,       y,     2*y]
```

Compute the cumulative product of each row of matrices `A` and `B`:

```
productA = cumprod(A,2)
productB = cumprod(B,2)
```

```
productA =
[ 0,  0,  0]
[ 3, 12, 60]
[ 1,  2,  6]

productB =
[   x, x*(2*x + 1), x*(2*x + 1)*(3*x + 2)]
[ 1/y,           1,                   2*y]
```

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.

`cumprod` returns A if `dim` is greater than `ndims(A)`.

See Also

`cumprod` | `cumsum` | `cumsum` | `int` | `symprod` | `symsum`

cumsum

Symbolic cumulative sum

Syntax

```
cumsum(A)
cumsum(A,dim)
```

Description

`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.

`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.

Examples

Cumulative Sum of a Vector

Create vector `V` 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 a Symbolic Matrix

Create matrix A containing symbolic numbers and matrix B containing symbolic expressions:

```
syms x y
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
B = [x, 2*x + 1, 3*x + 2; 1/y, y, 2*y]
```

```
A =
[ 0, 1, 2]
[ 3, 4, 5]
[ 1, 2, 3]

B =
[   x, 2*x + 1, 3*x + 2]
[ 1/y,       y,     2*y]
```

Compute the cumulative sums of elements of A and B. By default, `cumsum` returns the cumulative sum of each column:

```
sumA = cumsum(A)
sumB = cumsum(B)
```

```
sumA =
[ 0, 1,  2]
[ 3, 5,  7]
[ 4, 7, 10]

sumB =
[       x,     2*x + 1,       3*x + 2]
[ x + 1/y, 2*x + y + 1, 3*x + 2*y + 2]
```

Cumulative Sum of Each Row in a Symbolic Matrix

Create matrix A containing symbolic numbers and matrix B containing symbolic expressions:

```
syms x y
A = sym([0, 1, 2; 3, 4, 5; 1, 2, 3])
B = [x, 2*x + 1, 3*x + 2; 1/y, y, 2*y]
```

```
A =
[ 0, 1, 2]
```

```
[ 3, 4, 5]
[ 1, 2, 3]

B =
[   x, 2*x + 1, 3*x + 2]
[ 1/y,       y,     2*y]
```

Compute the cumulative sum of each row of matrices `A` and `B`:

```
sumA = cumsum(A,2)
sumB = cumsum(B,2)

sumA =
[ 0, 1,  3]
[ 3, 7, 12]
[ 1, 3,  6]

sumB =
[   x, 3*x + 1,   6*x + 3]
[ 1/y, y + 1/y, 3*y + 1/y]
```

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.

`cumsum` returns A if `dim` is greater than `ndims(A)`.

See Also

`cumprod` | `cumprod` | `cumsum` | `int` | `symprod` | `symsum`

curl

Curl of vector field

Syntax

```
curl(V,X)
curl(V)
```

Description

`curl(V,X)` returns the curl of the vector field V with respect to the vector X. The vector field V and the vector X are both three-dimensional.

`curl(V)` the curl of the vector field 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 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
```

More About

Curl of a Vector Field

The curl of the vector field $V = (V_1, V_2, V_3)$ with respect to the vector $X = (X_1, X_2, X_3)$ in Cartesian coordinates is the vector

$$curl(V) = \nabla \times V = \begin{pmatrix} \frac{\partial V_3}{\partial X_2} - \frac{\partial V_2}{\partial X_3} \\ \frac{\partial V_1}{\partial X_3} - \frac{\partial V_3}{\partial X_1} \\ \frac{\partial V_2}{\partial X_1} - \frac{\partial V_1}{\partial X_2} \end{pmatrix}$$

See Also

`diff` | `divergence` | `gradient` | `jacobian` | `hessian` | `laplacian` | `potential` | `vectorPotential`

daeFunction

Convert system of differential algebraic equations to MATLAB function handle

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 a 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 =
    @(t,in2,in3,param1,param2,param3)[in3(1,:)-param1.*in2(1,:)...
-param2.*in2(2,:).^2;-param3.^2+in2(1,:).^2+in2(2,:).^2]
```

You also can generate a file instead of generating a MATLAB function handle. If the file `myfile.m` already exists in the current folder, `daeFunction` replaces the existing function with the converted symbolic expression. You can open and edit the resulting file.

```
f = 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)

YP1 = in3(1,:);
x1 = in2(1,:);
x2 = in2(2,:);
t2 = x2.^2;
eqs = [YP1-param2.*t2-param1.*x1;t2-param3.^2+x1.^2];
```

Specify the parameter values, and create the reduced function handle `F` as follows.

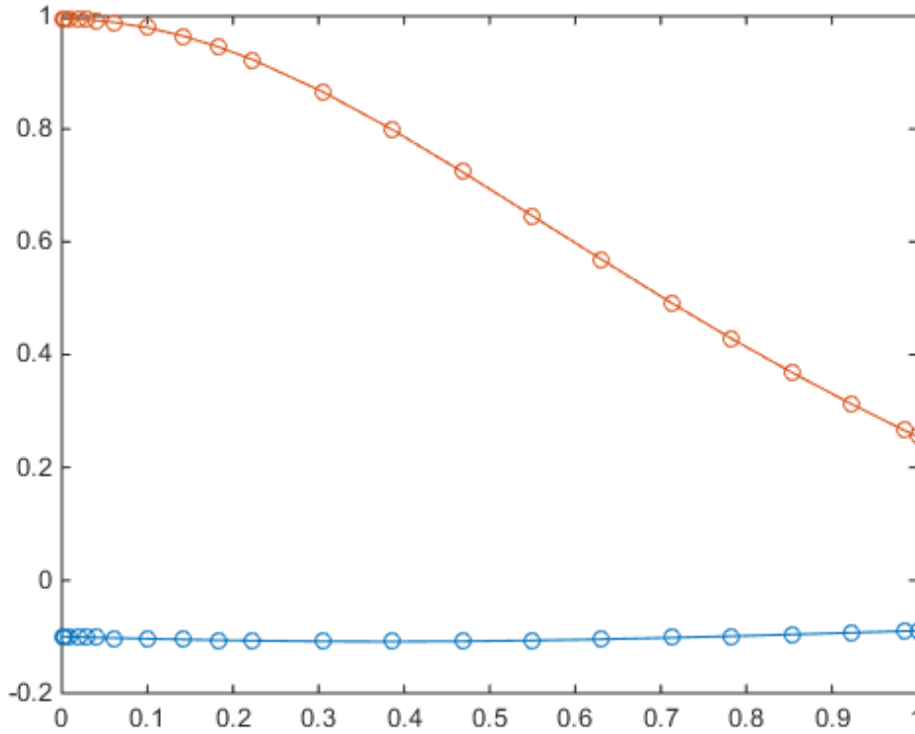
```
a = -0.6;
b = -0.1;
r = @(t) cos(t)/(1 + t^2);
F = @(t, Y, YP) f(t,Y,YP,a,b,r(t));
```

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)
```



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 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

Example: `daefunction(eqns,vars,'file','myfile')`

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'file'` — Path to file containing generated optimized code
string

Path to the file containing generated optimized code, specified as a string. The generated file can accept double or matrix arguments and evaluate the symbolic expression applied to the arguments. Optimized means intermediate variables are automatically generated to simplify or speed up the code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`.

If the value is an empty string, `daeFunction` generates an anonymous function. If the string does not end in `.m`, the function appends `.m`.

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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

dawson

Dawson integral

Syntax

```
dawson(X)
```

Description

`dawson(X)` represents the Dawson integral.

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]
```

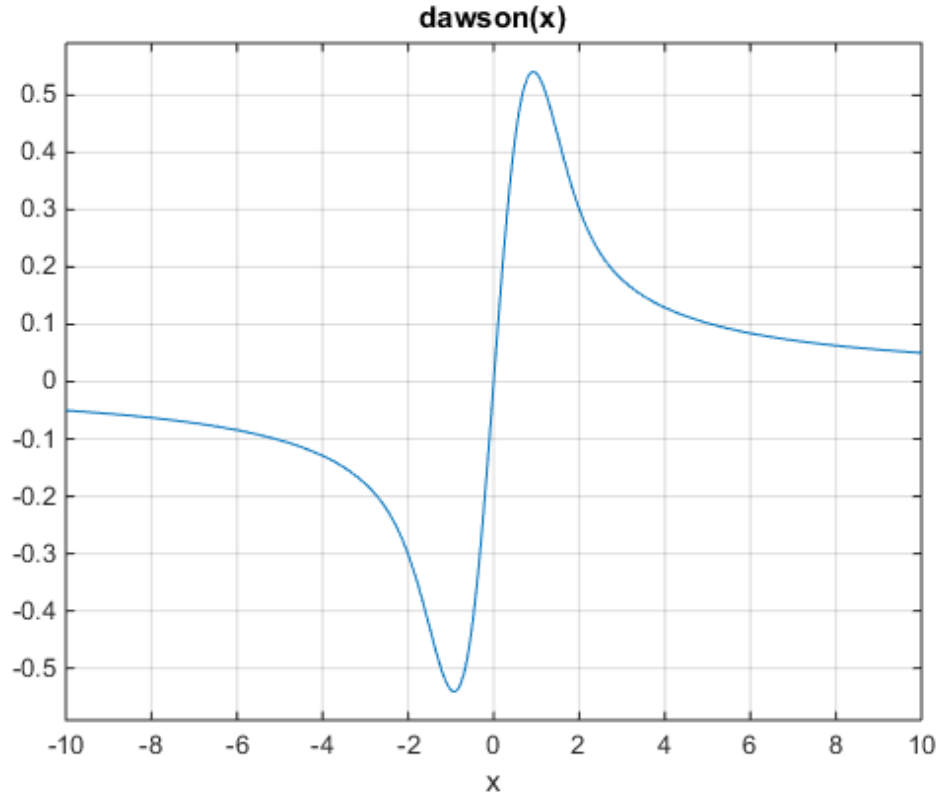
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0,...
-0.428249071085398625477190105151 75,...
-0.538079506912768419136387420407 56,...
0,...
0.301340388923791966034664439286 42,...
0]
```

Plot the Dawson Integral

Plot the Dawson integral on the interval from -10 to 10.

```
syms x
ezplot(dawson(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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.

More About

Dawson Integral

The Dawson integral, also called the Dawson function, is defined as follows:

$$\text{dawson}(x) = D(x) = e^{-x^2}\int_0^x e^{t^2}\,dt$$

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`

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 0s or 1s. 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 0s 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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

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

`rank` | `eig`

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`

diff

Differentiate symbolic expression or function

Syntax

```
diff(F)
diff(F,var)
diff(F,n)
diff(F,var,n)
diff(F,n,var)
diff(F,var1,...,varN)
```

Description

`diff(F)` differentiates F with respect to the variable determined by `symvar`.

`diff(F,var)` differentiates F with respect to the variable var.

`diff(F,n)` computes the nth derivative of F with respect to the variable determined by `symvar`.

`diff(F,var,n)` computes the nth derivative of F with respect to the variable var. This syntax is equivalent to `diff(F,n,var)`.

`diff(F,var1,...,varN)` differentiates F with respect to the variables var1,...,varN.

Examples

Differentiation of a Univariate Function

Find the first derivative of this univariate function:

```
syms x
f(x) = sin(x^2);
df = diff(f)
```

```
df(x) =
2*x*cos(x^2)
```

Differentiation with Respect to a 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 a 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 a Multivariate Expression with Respect to a 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 a Multivariate Expression with Respect to the 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 | string

Differentiation variable, specified as a symbolic variable or a string.

`var1,...,varN` — Differentiation variables

symbolic variables | strings

Differentiation variables, specified as symbolic variables or strings.

`n` — Differentiation order

nonnegative integer

Differentiation order, specified as a nonnegative integer.

More About

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.

See Also

`curl` | `divergence` | `gradient` | `hessian` | `int` | `jacobian` | `laplacian` | `symvar`

Related Examples

- “Differentiation” on page 2-3
- “Find Asymptotes, Critical and Inflection Points” on page 2-32

digits

Variable-precision accuracy

Syntax

```
digits
digits(d)
```

```
d1 = digits
d1 = digits(d)
```

Description

`digits` shows the number of significant decimal digits that MuPAD software uses to do variable-precision arithmetic (VPA). The default value is 32 digits.

`digits(d)` sets the current VPA accuracy to d significant decimal digits. The value d must be a positive integer greater than `1` and less than $2^{29} + 1$.

`d1 = digits` assigns the current setting of `digits` to variable d1.

`d1 = digits(d)` assigns the current setting of `digits` to variable d1 and sets VPA accuracy to d.

Examples

Default Accuracy of Variable-Precision Computations

By default, the minimum number of significant (nonzero) decimal digits is 32.

To obtain the current number of digits, use `digits` without input arguments:

```
digits
```

```
Digits = 32
```

To save the current setting, assign the result returned by `digits` to a variable:

```
CurrentDigits = digits

CurrentDigits =
    32
```

Control Accuracy of Variable-Precision Computations

`digits` lets you specify any number of significant decimal digits from `1` to $2^{29} + 1$.

Compute the ratio 1/3 and the ratio 1/3000 with four-digit accuracy:

```
old = digits(4);
vpa(1/3)
vpa(1/3000)

ans =
0.3333

ans =
0.0003333
```

Restore the default accuracy setting for further computations:

```
digits(old)
```

"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 =
0.00000000000000000000867361737988403547206008158444 03
```

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 =
0.09999999999999999991326382620116

ans =
0.00000000000000000000867361737988403547206008158444 03
```

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 =
3.14159265358979311599796346854 42

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 =
3.1415926535897932384626433832795028841971

ans =
3.1415926535897931159979634685441851615905

ans =
3.1415926535897931159979634685441851615905

ans =
3.1415926535897932384626433832795028841971 -...
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`

Related Examples

- “Control Accuracy of Variable-Precision Computations”
- “Recognize and Avoid Round-Off Errors”
- “Improve Performance of Numeric Computations”

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)*i, pi^2/6, dilog(1/4), pi^2/12 - log(2)^2/2, 0, -pi^2/12]
```

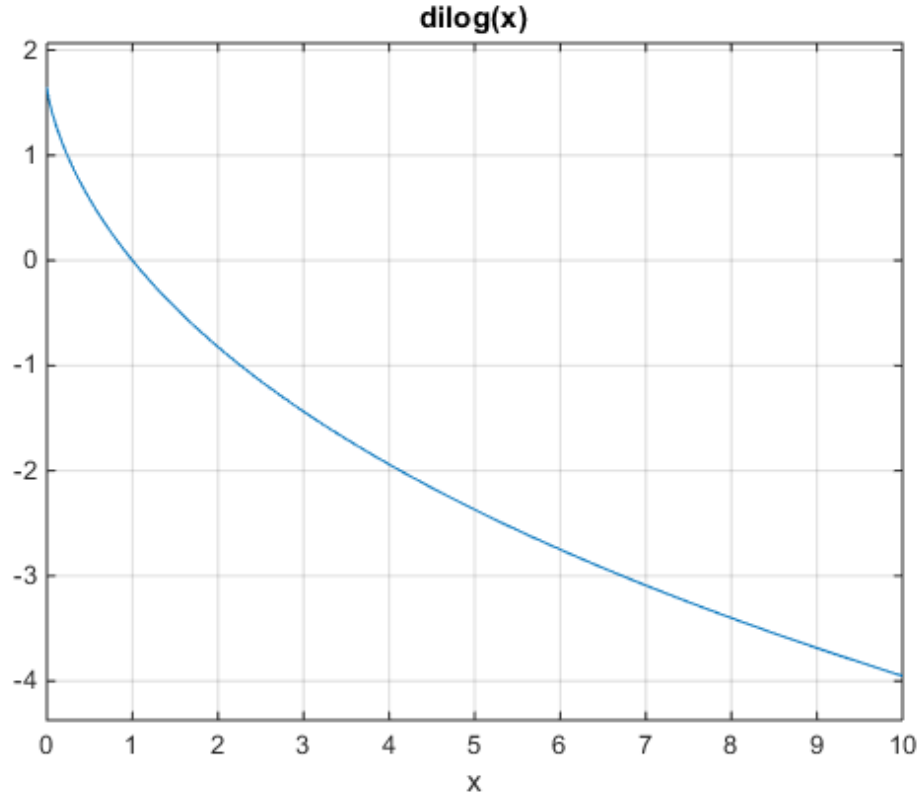
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 2.467401100272339654708622749969 - 2.177586090303602130500688898237 6*i,...
1.644934066848226436472415166646,...
0.978469392930306103743066666524 56,...
0.582240526465012505902656320159 68,...
0,...
-0.822467033424113218236207583323 01]
```

Plot the Dilogarithm Function

Plot the dilogarithm function on the interval from 0 to 10.

```
syms x
ezplot(dilog(x), [0, 10])
grid on
```



Handle Expressions Containing the 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.

More About

Dilogarithm Function

There are two common definitions of the dilogarithm function.

The implementation of the `dilog` function uses the following definition:

$$\mathrm{dilog}(x) = \int_1^x \frac{\ln(t)}{1-t}\,dt$$

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`

dirac

Dirac delta function

Syntax

```
dirac(x)
dirac(n,x)
```

Description

`dirac(x)` represents the Dirac delta function of x.

`dirac(n,x)` represents the nth 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 integral of the Dirac delta function is the Heaviside function.

```
int(dirac(x), x)

ans =
heaviside(x)
```

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 the 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.

More About

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`

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 a Symbolic Scalar

```
syms x
y = x^3 - exp(x);
disp(y)
```

```
x^3 - exp(x)
```

Display a Symbolic Matrix

```
A = sym('a%d%d',[3 3]);
disp(A)
```

```
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
```

Display a Symbolic Function

```
syms f(x)
f(x) = x+1;
disp(f)
```

```
x + 1
symbolic function inputs: x
```

Display a Sentence with Text and Symbolic Expressions

Display the sentence "Euler's formula is $e^{ix} = \cos(x) + i\sin(x)$".

To concatenate strings with symbolic expressions, convert the symbolic expressions to strings 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*i) = cos(x) + sin(x)*i.
```

Because `'` terminates the string, repeat it in `Euler''s` for MATLAB to interpret it as an apostrophe and not a string 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`

display

Display symbolic input

Syntax

```
display(X)
```

Description

`display(X)` displays the symbolic input X.

Examples

Display a Symbolic Scalar

```
syms x
y = x^3 - exp(x);
display(y)
```

```
y =
x^3 - exp(x)
```

Display a Symbolic Matrix

```
A = sym('a%d%d',[3 3]);
display(A)
```

```
A =
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
```

Display a Symbolic Function

```
syms f(x)
```

```
f(x) = x+1;
display(f)
```

```
f(x) =
x + 1
```

Display a Sentence with Text and Symbolic Expressions

Display the sentence "Euler's formula is $e^{ix} = \cos(x) + i\sin(x)$".

To concatenate strings with symbolic expressions, convert the symbolic expressions to strings 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*i) = cos(x) + sin(x)*i.
```

Because `'` terminates the string, you need to repeat it in `Euler''s` for MATLAB to interpret it as an apostrophe and not a string 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`

divergence

Divergence of vector field

Syntax

```
divergence(V,X)
```

Description

`divergence(V,X)` returns the divergence of the vector field V with respect to the vector X in Cartesian coordinates. Vectors V and X must have the same length.

Input Arguments

V

Vector of symbolic expressions or symbolic functions.

X

Vector with respect to which you compute the divergence.

Examples

Compute the divergence of the vector field $V(x, y, z) = (x, 2y^2, 3z^3)$ with respect to vector $X = (x, y, z)$ in Cartesian coordinates:

```
syms x y z
divergence([x, 2*y^2, 3*z^3], [x, y, z])
```

```
ans =
9*z^2 + 4*y + 1
```

Compute the divergence of the curl of this vector field. The divergence of the curl of any vector field is 0.

```
syms x y z
divergence(curl([x, 2*y^2, 3*z^3], [x, y, z]), [x, y, z])

ans =
0
```

Compute the divergence of the gradient of this scalar function. The result is the Laplacian of the scalar function:

```
syms x y z
f = x^2 + y^2 + z^2;
divergence(gradient(f, [x, y, z]), [x, y, z])

ans =
6
```

More About

Divergence of a Vector Field

The divergence of the vector field $V = (V_1,...,V_n)$ with respect to the vector $X = (X_1,...,X_n)$ in Cartesian coordinates is the sum of partial derivatives of V with respect to $X_1,...,X_n$:

$$div(V) = \nabla \cdot V = \sum_{i=1}^{n} \frac{\partial V_i}{\partial x_i}$$

See Also

`curl` | `diff` | `gradient` | `jacobian` | `hessian` | `laplacian` | `potential` | `vectorPotential`

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.

More About

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`

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.

double

Convert symbolic matrix to MATLAB numeric form

Syntax

```
r = double(S)
```

Description

`r = double(S)` converts the symbolic object `S` to a numeric object r.

Input Arguments

S

Symbolic constant, constant expression, or symbolic matrix whose entries are constants or constant expressions.

Output Arguments

r

If S is a symbolic constant or constant expression, `r` is a double-precision floating-point number representing the value of S. If S is a symbolic matrix whose entries are constants or constant expressions, `r` is a matrix of double precision floating-point numbers representing the values of the entries of S.

Examples

Find the numeric value for the expression $\frac{1+\sqrt{5}}{2}$:

```
double(sym('(1+sqrt(5))/2'))

1.6180
```

Find the numeric value for the entries of this matrix `T`:

```
a = sym(2*sqrt(2));
b = sym((1-sqrt(3))^2);
T = [a, b; a*b, b/a];
double(T)

ans =
    2.8284    0.5359
    1.5157    0.1895
```

Find the numeric value for this expression. By default, double uses a new upper limit of 664 digits for the working precision and returns the value 0:

```
x = sym('((exp(200) + 1)/(exp(200) - 1)) - 1');
double(x)

ans =
     0
```

To get a more accurate result, increase the precision of computations:

```
digits(1000)
double(x)

ans =
   2.7678e-87
```

More About

Tips

- The working precision for `double` depends on the input argument. It is also ultimately limited by 664 digits. If your computation requires a larger working precision, specify the number of digits explicitly using the `digits` function.

See Also

`sym` | `vpa`

Related Examples

- “Choose the Arithmetic”
- “Control Accuracy of Variable-Precision Computations”
- “Recognize and Avoid Round-Off Errors”
- “Improve Performance of Numeric Computations”

dsolve

Ordinary differential equation and system solver

Syntax

```
S = dsolve(eqn)
S = dsolve(eqn,cond)
S = dsolve(eqn,cond,Name,Value)
Y = dsolve(eqns)
Y = dsolve(eqns,conds)
Y = dsolve(eqns,conds,Name,Value)
[y1,...,yN] = dsolve(eqns)
[y1,...,yN] = dsolve(eqns,conds)
[y1,...,yN] = dsolve(eqns,conds,Name,Value)
```

Description

`S = dsolve(eqn)` solves the ordinary differential equation eqn. Here eqn is a symbolic equation containing `diff` to indicate derivatives. Alternatively, you can use a string with the letter `D` indicating derivatives. For example, `syms y(x); dsolve(diff(y) == y + 1)` and `dsolve('Dy = y + 1','x')` both solve the equation `dy/dx = y + 1` with respect to the variable `x`. Also, eqn can be an array of such equations or strings.

`S = dsolve(eqn,cond)` solves the ordinary differential equation eqn with the initial or boundary condition cond.

`S = dsolve(eqn,cond,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`Y = dsolve(eqns)` solves the system of ordinary differential equations eqns and returns a structure array that contains the solutions. The number of fields in the structure array corresponds to the number of independent variables in the system.

`Y = dsolve(eqns,conds)` solves the system of ordinary differential equations eqns with the initial or boundary conditions conds.

`Y = dsolve(eqns,conds,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`[y1,...,yN] = dsolve(eqns)` solves the system of ordinary differential equations eqns and assigns the solutions to the variables y1,...,yN.

`[y1,...,yN] = dsolve(eqns,conds)` solves the system of ordinary differential equations eqns with the initial or boundary conditions conds.

`[y1,...,yN] = dsolve(eqns,conds,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Input Arguments

eqn

Symbolic equation, string representing an ordinary differential equation, or array of symbolic equations or strings.

When representing `eqn` as a symbolic equation, you must create a symbolic function, for example `y(x)`. Here `x` is an independent variable for which you solve an ordinary differential equation. Use the `==` operator to create an equation. Use the `diff` function to indicate differentiation. For example, to solve $d^2y(x)/dx^2 = x*y(x)$, enter:

```
syms y(x)
dsolve(diff(y, 2) == x*y)
```

When representing `eqn` as a string, use the letter `D` to indicate differentiation. By default, `dsolve` assumes that the independent variable is `t`. Thus, `Dy` means `dy/dt`. You can specify the independent variable. The letter `D` followed by a digit indicates repeated differentiation. Any character immediately following a differentiation operator is a dependent variable. For example, to solve `y''(x) = x*y(x)`, enter:

```
dsolve('D2y = x*y','x')
```

or

```
dsolve('D2y == x*y','x')
```

cond

Equation or string representing an initial or boundary condition. If you use equations, assign expressions with `diff` to some intermediate variables. For example, use `Dy`, `D2y`, and so on as intermediate variables:

```
Dy = diff(y);
D2y = diff(y, 2);
```

Then define initial conditions using symbolic equations, such as `y(a) == b` and `Dy(a) == b`. Here `a` and `b` are constants.

If you represent initial and boundary conditions as strings, you do not need to create intermediate variables. In this case, follow the same rules as you do when creating an equation eqn as a string. For example, specify `'y(a) = b'` and `'Dy(a) = b'`. When using strings, you can use = or == in equations.

`eqns`

Symbolic equations or strings separated by commas and representing a system of ordinary differential equations. Each equation or string represents an ordinary differential equation.

`conds`

Symbolic equations or strings separated by commas and representing initial or boundary conditions or both types of conditions. Each equation or string represents an initial or boundary condition. If the number of the specified conditions is less than the number of dependent variables, the resulting solutions contain arbitrary constants `C1`, `C2`, `...`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoreAnalyticConstraints'`

By default, the solver applies the purely algebraic simplifications to the expressions on both sides of equations. These simplifications might not be generally valid. Therefore, by default the solver does not guarantee general correctness and completeness of the results. To solve ordinary differential equations without additional assumptions, set `IgnoreAnalyticConstraints` to `false`. The results obtained with `IgnoreAnalyticConstraints` set to `false` are correct for all values of the arguments.

If you do not set `IgnoreAnalyticConstraints` to `false`, always verify results returned by the `dsolve` command.

Default: `true`

`'MaxDegree'`

Do not use explicit formulas that involve radicals when solving polynomial equations of degrees larger than the specified value. This value must be a positive integer smaller than 5.

Default: 2

Output Arguments

`S`

Symbolic array that contains solutions of an equation. The size of a symbolic array corresponds to the number of the solutions.

`Y`

Structure array that contains solutions of a system of equations. The number of fields in the structure array corresponds to the number of independent variables in a system.

`y1,...,yN`

Variables to which the solver assigns the solutions of a system of equations. The number of output variables or symbolic arrays must equal the number of independent variables in a system. The toolbox sorts independent variables alphabetically, and then assigns the solutions for these variables to output variables or symbolic arrays.

Examples

Solve these ordinary differential equations. Use `==` to create an equation, and `diff` to indicate differentiation:

```
syms a x(t)
dsolve(diff(x) == -a*x)
```

```
ans =
C2*exp(-a*t)
```

```
syms f(t)
dsolve(diff(f) == f + sin(t))

ans =
C4*exp(t) - sin(t)/2 - cos(t)/2
```

Solve this ordinary differential equation with the initial condition `y(0) = b`:

```
syms a b y(t)
dsolve(diff(y) == a*y, y(0) == b)
```

Specifying the initial condition lets you eliminate arbitrary constants, such as `C1`, `C2`,`...`:

```
ans =
b*exp(a*t)
```

Solve this ordinary differential equation with the initial and boundary conditions. To specify a condition that contains a derivative, assign the derivative to a variable:

```
syms a y(t)
Dy = diff(y);
dsolve(diff(y, 2) == -a^2*y, y(0) == 1, Dy(pi/a) == 0)
```

Because the equation contains the second-order derivative d^2y/dt^2, specifying two conditions lets you eliminate arbitrary constants in the solution:

```
ans =
exp(-a*t*i)/2 + exp(a*t*i)/2
```

Solve this system of ordinary differential equations:

```
syms x(t) y(t)
z = dsolve(diff(x) == y, diff(y) == -x)
```

When you assign the solution of a system of equations to a single output, `dsolve` returns a structure containing the solutions:

```
z =
    y: [1x1 sym]
    x: [1x1 sym]
```

To see the results, enter `z.x` and `z.y`:

```
z.x
```

```
ans =
C12*cos(t) + C11*sin(t)
```

```
z.y
```

```
ans =
C11*cos(t) - C12*sin(t)
```

By default, the solver applies a set of purely algebraic simplifications that are not correct in general, but that can produce simple and practical solutions:

```
syms a y(t)
dsolve(diff(y) == a/sqrt(y) + y, y(a) == 1)
```

```
ans =
(exp((3*t)/2 - (3*a)/2 + log(a + 1)) - a)^(2/3)
```

To obtain complete and generally correct solutions, set the value of `IgnoreAnalyticConstraints` to `false`:

```
dsolve(diff(y) == a/sqrt(y) + y, y(a) == 1, 'IgnoreAnalyticConstraints', false)
```

```
Warning: The solutions are subject to the following
conditions:
PI/2 < angle(-a) in(C23, 'integer')
.
> In dsolve at 219
ans =
(- a + exp((3*t)/2 - (3*a)/2 + log(a + 1) + C23*pi*2*i))^(2/3)
```

If you apply algebraic simplifications, you can get explicit solutions for some equations for which the solver cannot compute them using strict mathematical rules:

```
syms y(t)
dsolve(sqrt(diff(y)) == sqrt(y) + log(y))
```

```
Warning: Explicit solution could not be found; implicit solution returned.

ans =
                                                                  4*lambertw(0, 1/2)^2
                                                                 4*lambertw(0, -1/2)^2
 solve(int(1/(log(y) + y^(1/2))^2, y, 'IgnoreAnalyticConstraints', true) - t - C28 == 0, y)
```

versus

```
dsolve(sqrt(diff(y)) == sqrt(y) + log(y), 'IgnoreAnalyticConstraints', false)
```

```
Warning: Explicit solution could not be found; implicit solution returned.
```

```
ans =
solve(C32 + t - int(1/(log(y) + y^(1/2))^2, y), y)...
intersect solve(signIm(log(y(t))*i + y(t)^(1/2)*i) == 1, y(t))
```

When you solve a higher-order polynomial equation, the solver sometimes uses `RootOf` to return the results:

```
syms a y(x)
dsolve(diff(y) == a/(y^2 + 1))
```

```
Warning: Explicit solution could not be found; implicit solution returned.

ans =
RootOf(z^3 + 3*z - 3*a*x - 3*C36, z)
```

To get an explicit solution for such equations, try calling the solver with `MaxDegree`. The option specifies the maximum degree of polynomials for which the solver tries to return explicit solutions. The default value is 2. By increasing this value, you can get explicit solutions for higher-order polynomials. For example, increase the value of `MaxDegree` to 4 and get explicit solutions instead of `RootOf` for this equation:

```
s = dsolve(diff(y) == a/(y^2 + 1), 'MaxDegree', 4);
pretty(s)
```

```
/                    1              \
|               #1 - --              |
|                    #1              |
|                                    |
|          /  1      \               |
| sqrt(3) | -- + #1 | i              |
|          \ #1      /       1    #1 |
| --------------------- + ---- - -- |
|            2             2 #1    2 |
|                                    |
|                 /  1      \        |
|        sqrt(3) | -- + #1 | i      |
|   1            \ #1      /     #1 |
| ---- - ---------------------- - -- |
\ 2 #1             2              2 /
```

```
where

         /                     /              2     \ \1/3
         | 3 C40   3 a x       | 9 (C40 + a x)      | |
```

```
   #1 == | ----- + ----- + sqrt| -------------- + 1 | |
         \   2       2         \        4           / /
```

If `dsolve` can find neither an explicit nor an implicit solution, then it issues a warning and returns the empty `sym`:

```
syms y(x)
dsolve(exp(diff(y)) == 0)
```

```
 Warning: Explicit solution could not be found.

ans =
[ empty sym ]
```

Returning the empty symbolic object does not prove that there are no solutions.

Solve this equation specifying it as a string. By default, `dsolve` assumes that the independent variable is `t`:

```
dsolve('Dy^2 + y^2 == 1')
```

```
ans =
               1
              -1
 cosh(C49 + t*i)
 cosh(C45 - t*i)
```

Now solve this equation with respect to the variable `s`:

```
dsolve('Dy^2 + y^2 == 1','s')
```

```
ans =
               1
              -1
 cosh(C57 + s*i)
 cosh(C53 - s*i)
```

More About

Tips

- The names of symbolic variables used in differential equations should not contain the letter `D` because `dsolve` assumes that `D` is a differential operator and any character immediately following `D` is a dependent variable.

- If `dsolve` cannot find a closed-form (explicit) solution, it attempts to find an implicit solution. When `dsolve` returns an implicit solution, it issues this warning:

  ```
  Warning: Explicit solution could not be found;
  implicit solution returned.
  ```

- If `dsolve` can find neither an explicit nor an implicit solution, then it issues a warning and returns the empty `sym`. In this case, try to find a numeric solution using the MATLAB `ode23` or `ode45` function. In some cases, the output is an equivalent lower-order differential equation or an integral.

Algorithms

If you do not set the value of `IgnoreAnalyticConstraints` to `false`, the solver applies these rules to the expressions on both sides of an equation:

- $\log(a) + \log(b) = \log(a \cdot b)$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$(a \cdot b)^c = a^c \cdot b^c.$$

- $\log(a^b) = b \cdot \log(a)$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$(a^b)^c = a^{b \cdot c}.$$

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:

  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

- The solver can multiply both sides of an equation by any expression except `0`.
- The solutions of polynomial equations must be complete.

- "Solve a Single Differential Equation"
- "Solve a System of Differential Equations"

See Also

`linsolve` | `ode23` | `ode45` | `odeToVectorField` | `solve` | `syms` | `vpasolve`

ei

One-argument exponential integral function

Syntax

```
ei(x)
```

Description

`ei(x)` returns the one-argument exponential integral defined as follows:

$$\mathrm{ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt$$

Examples

Exponential Integral for Floating-Point and Symbolic Numbers

Compute the exponential integrals for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ei(-2), ei(-1/2), ei(1), ei(sqrt(2))]
```

```
s =
   -0.0489   -0.5598    1.8951    3.0485
```

Compute the exponential integrals for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ei` returns unresolved symbolic calls.

```
s = [ei(sym(-2)), ei(sym(-1/2)), ei(sym(1)), ei(sqrt(sym(2)))]
```

```
s =
[ ei(-2), ei(-1/2), ei(1), ei(2^(1/2))]
```

Use `vpa` to approximate this result with the 10 digits accuracy:

```
vpa(s, 10)

ans =
[ -0.04890051071, -0.5597735948, 1.895117816, 3.048462479]
```

Branch Cut at the Negative Real Axis

Compute the exponential integrals for these numbers. The negative real axis is a branch cut. The exponential integral has a jump of height $2\pi i$ when crossing this cut:

```
[ei(-1), ei(-1 + 10^(-10)*i), ei(-1 - 10^(-10)*i)]

ans =
  -0.2194 + 0.0000i  -0.2194 + 3.1416i  -0.2194 - 3.1416i
```

Derivatives of the Exponential Integral

Compute the first, second, and third derivatives of the one-argument exponential integral:

```
syms x
diff(ei(x), x)
diff(ei(x), x, 2)
diff(ei(x), x, 3)

ans =
exp(x)/x

ans =
exp(x)/x - exp(x)/x^2

ans =
exp(x)/x - (2*exp(x))/x^2 + (2*exp(x))/x^3
```

Limits of the Exponential Integral

Compute the limits of this one-argument exponential integral:

```
syms x
limit(ei(2*x^2/(1+x)), x, -Inf)
limit(ei(2*x^2/(1+x)), x, 0)
limit(ei(2*x^2/(1+x)), x, Inf)
```

```
ans =
0

ans =
-Inf

ans =
Inf
```

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.

More About

Tips

- The one-argument exponential integral is singular at `x = 0`. The toolbox uses this special value: `ei(0) = -Inf`.

References

[1] Gautschi, W., and W. F. Gahill "Exponential Integral and Related Functions." *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* (M. Abramowitz and I. A. Stegun, eds.). New York: Dover, 1972.

See Also

`Ei` | `expint` | `expint` | `int` | `vpa`

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
 13.126280930709218802525643085949
  -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]
```

More About

- “Eigenvalues” on page 2-71

See Also

`charpoly` | `svd` | `vpa` | `jordan`

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 and second kinds.

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]
```

Alternatives

You can use `ellipticK` and `ellipticE` to compute elliptic integrals of the first and second kinds separately.

More About

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.

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 | ellipticE | ellipticK | ellipticK | vpa

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.

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

Compute the complementary complete elliptic integrals of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticCE(0), ellipticCE(pi/4),...
 ellipticCE(1), ellipticCE(pi/2)]
```

```
s =
    1.0000    1.4828    1.5708    1.7753
```

Compute the complementary complete elliptic integrals of the second kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticCE` returns unresolved symbolic calls.

```
s = [ellipticCE(sym(0)), ellipticCE(sym(pi/4)),...
 ellipticCE(sym(1)), ellipticCE(sym(pi/2))]

s =
[ 1, ellipticCE(pi/4), pi/2, ellipticCE(pi/2)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.0, 1.482786927, 1.570796327, 1.775344699]
```

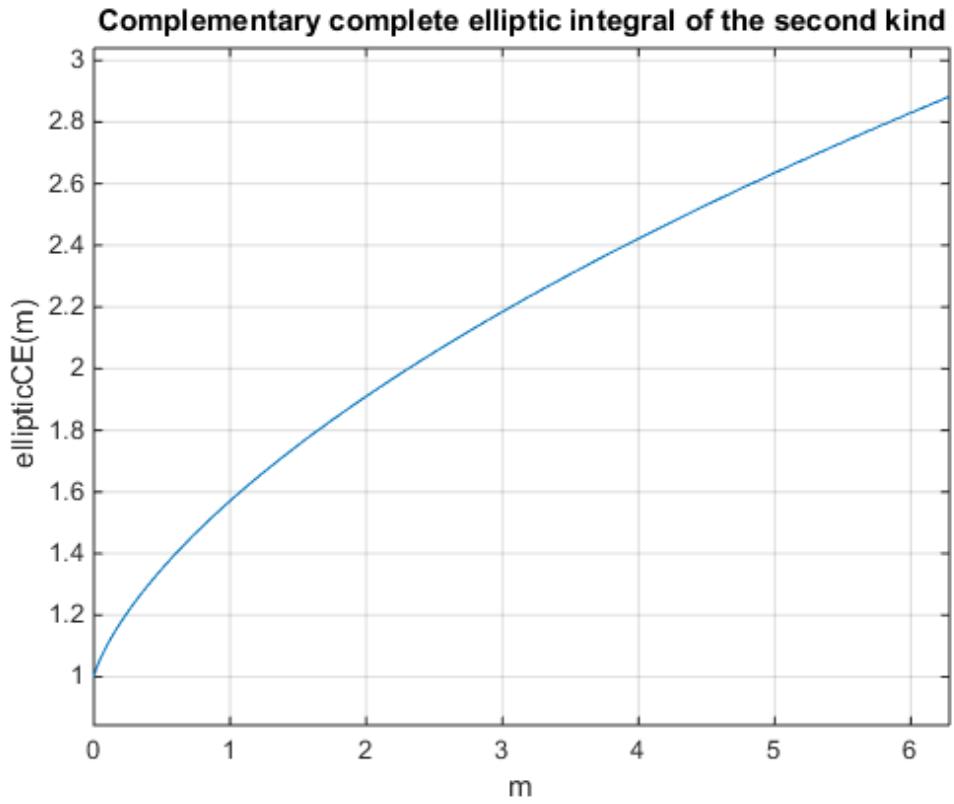
Differentiate these expressions involving the complementary complete elliptic integral of the second kind:

```
syms m
diff(ellipticCE(m))
diff(ellipticCE(m^2), m, 2)

ans =
ellipticCE(m)/(2*m - 2) - ellipticCK(m)/(2*m - 2)

ans =
(2*ellipticCE(m^2))/(2*m^2 - 2) -...
(2*ellipticCK(m^2))/(2*m^2 - 2) +...
2*m*(((2*m*ellipticCK(m^2))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m*(m^2 - 1)))/(2*m^2 - 2) +...
(2*m*(ellipticCE(m^2)/(2*m^2 - 2) -...
ellipticCK(m^2)/(2*m^2 - 2)))/(2*m^2 - 2) -...
(4*m*ellipticCE(m^2))/(2*m^2 - 2)^2 +...
(4*m*ellipticCK(m^2))/(2*m^2 - 2)^2)
```

Here, `ellipticCK` represents the complementary complete elliptic integral of the first kind.

Plot the complementary complete elliptic integral of the second kind:

```
syms m
ezplot(ellipticCE(m))
title('Complementary complete elliptic integral of the second kind')
ylabel('ellipticCE(m)')
grid on
```



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)]
```

More About

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:

$$E(m) = E\left(\frac{\pi}{2} \middle| m\right) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\, d\theta$$

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` | `ellipticCE` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

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.

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

Compute the complementary complete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticCK(1/2), ellipticCK(pi/4), ellipticCK(1), ellipticCK(inf)]
```

```
s =
    1.8541    1.6671    1.5708       NaN
```

Compute the complete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticCK` returns unresolved symbolic calls.

```
s = [ellipticCK(sym(1/2)), ellipticCK(sym(pi/4)),...
```

```
 ellipticCK(sym(1)), ellipticCK(sym(inf))]

s =
[ ellipticCK(1/2), ellipticCK(pi/4), pi/2, ellipticCK(Inf)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.854074677, 1.667061338, 1.570796327, NaN]
```

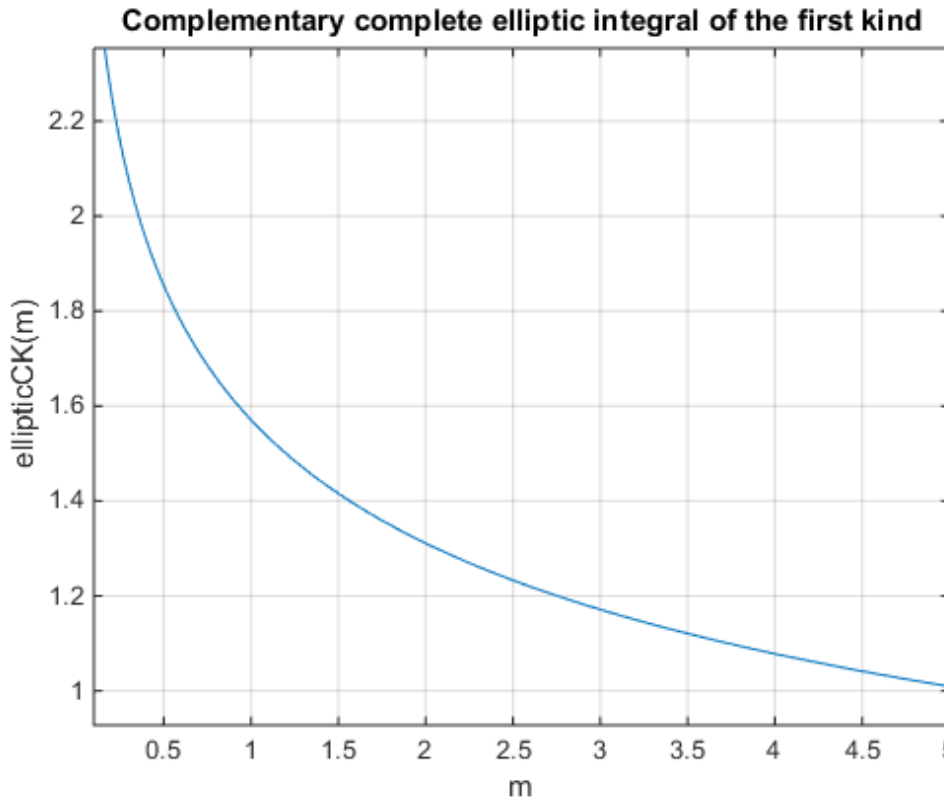
Differentiate these expressions involving the complementary complete elliptic integral of the first kind:

```
syms m
diff(ellipticCK(m))
diff(ellipticCK(m^2), m, 2)

ans =
ellipticCE(m)/(2*m*(m - 1)) - ellipticCK(m)/(2*m - 2)

ans =
(2*(ellipticCE(m^2)/(2*m^2 - 2) -...
ellipticCK(m^2)/(2*m^2 - 2)))/(m^2 - 1) -...
(2*ellipticCE(m^2))/(m^2 - 1)^2 -...
(2*ellipticCK(m^2))/(2*m^2 - 2) +...
(8*m^2*ellipticCK(m^2))/(2*m^2 - 2)^2 +...
(2*m*((2*m*ellipticCK(m^2))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m*(m^2 - 1))))/(2*m^2 - 2) -...
ellipticCE(m^2)/(m^2*(m^2 - 1))
```

Here, `ellipticCE` represents the complementary complete elliptic integral of the second kind.

Plot the complementary complete elliptic integral of the first kind:

```
syms m
ezplot(ellipticCK(m), [0.1, 5])
title('Complementary complete elliptic integral of the first kind')
ylabel('ellipticCK(m)')
grid on
hold off
```



Call `ellipticCK` for this symbolic matrix. When the input argument is a matrix, `ellipticCK` computes the complementary complete elliptic integral of the first kind for each element.

```
ellipticCK(sym([pi/6 pi/4; pi/3 pi/2]))
```

```
ans =
[ ellipticCK(pi/6), ellipticCK(pi/4)]
[ ellipticCK(pi/3), ellipticCK(pi/2)]
```

More About

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` | `ellipticCK` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

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.

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.

More About

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:

$$\Pi(n,m) = \Pi\left(n;\frac{\pi}{2}\,|\,m\right) = \int_0^{\pi/2} \frac{1}{(1-n\sin^2\theta)\sqrt{1-m\sin^2\theta}}\,d\theta$$

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` | `ellipticCPi` | `ellipticE` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

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.

`ellipticE(phi,m)` returns the incomplete elliptic integral of the second kind.

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

Compute the complete elliptic integrals of the second kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticE(-10.5), ellipticE(-pi/4),...
 ellipticE(0),  ellipticE(1)]
```

```
s =
    3.7096    1.8443    1.5708    1.0000
```

Compute the complete elliptic integral of the second kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticE` returns unresolved symbolic calls.

```
s = [ellipticE(sym(-10.5)), ellipticE(sym(-pi/4)),...
 ellipticE(sym(0)),  ellipticE(sym(1))]

s =
[ ellipticE(-21/2), ellipticE(-pi/4), pi/2, 1]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 3.70961391, 1.844349247, 1.570796327, 1.0]
```

Differentiate these expressions involving elliptic integrals of the second kind:

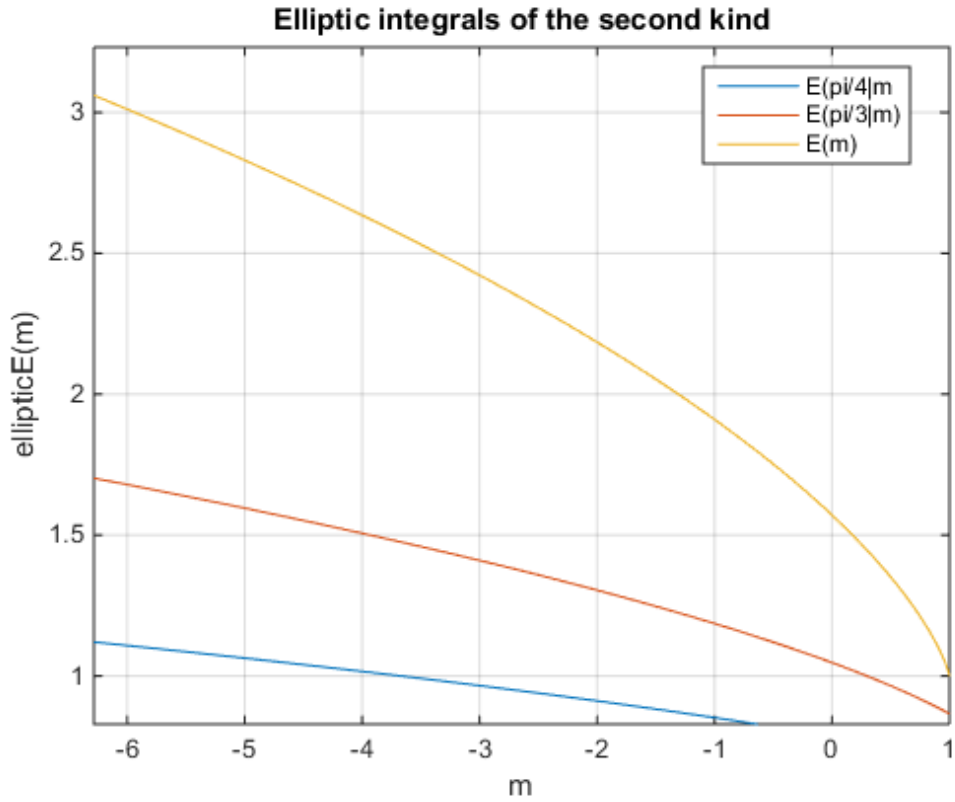
```
syms m
diff(ellipticE(pi/3, m))
diff(ellipticE(m^2), m, 2)

ans =
ellipticE(pi/3, m)/(2*m) - ellipticF(pi/3, m)/(2*m)

ans =
2*m*((ellipticE(m^2)/(2*m^2) -...
ellipticK(m^2)/(2*m^2))/m - ellipticE(m^2)/m^3 +...
ellipticK(m^2)/m^3 + (ellipticK(m^2)/m +...
ellipticE(m^2)/(m*(m^2 - 1)))/(2*m^2)) +...
ellipticE(m^2)/m^2 - ellipticK(m^2)/m^2
```

Here, `ellipticK` and `ellipticF` represent the complete and incomplete elliptic integrals of the first kind, respectively.

Plot the incomplete elliptic integrals `ellipticE(phi,m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticE(m)`:

```
syms m
ezplot(ellipticE(pi/4, m))
hold on
```

```
ezplot(ellipticE(pi/3, m))
ezplot(ellipticE(m))

title('Elliptic integrals of the second kind')
ylabel('ellipticE(m)')
legend('E(pi/4|m', 'E(pi/3|m)', 'E(m)', 'Location', 'Best')
grid on
hold off
```



Call `ellipticE` for this symbolic matrix. When the input argument is a matrix, `ellipticE` computes the complete elliptic integral of the second kind for each element.

```
ellipticE(sym([1/3 1; 1/2 0]))
```

```
ans =
```

```
[ ellipticE(1/3),    1]
[ ellipticE(1/2), pi/2]
```

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

More About

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)`.

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` | `ellipticPi` | `vpa`

ellipticF

Incomplete elliptic integral of the first kind

Syntax

```
ellipticF(phi,m)
```

Description

`ellipticF(phi,m)` returns the incomplete elliptic integral of the first kind.

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

Compute the incomplete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticF(pi/3, -10.5), ellipticF(pi/4, -pi),...
 ellipticF(1, -1),  ellipticF(pi/2, 0)]
```

```
s =
```

```
    0.6184    0.6486    0.8964    1.5708
```

Compute the incomplete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticF` returns unresolved symbolic calls.

```
s = [ellipticF(sym(pi/3), -10.5), ellipticF(sym(pi/4), -pi),...
ellipticF(sym(1), -1),  ellipticF(pi/6, sym(0))]
```

```
s =
[ ellipticF(pi/3, -21/2), ellipticF(pi/4, -pi), ellipticF(1, -1), pi/6]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)
```

```
ans =
[ 0.6184459461, 0.6485970495, 0.8963937895, 0.5235987756]
```

Differentiate this expression involving the incomplete elliptic integral of the first kind:

```
syms m
diff(ellipticF(pi/4, m))
```

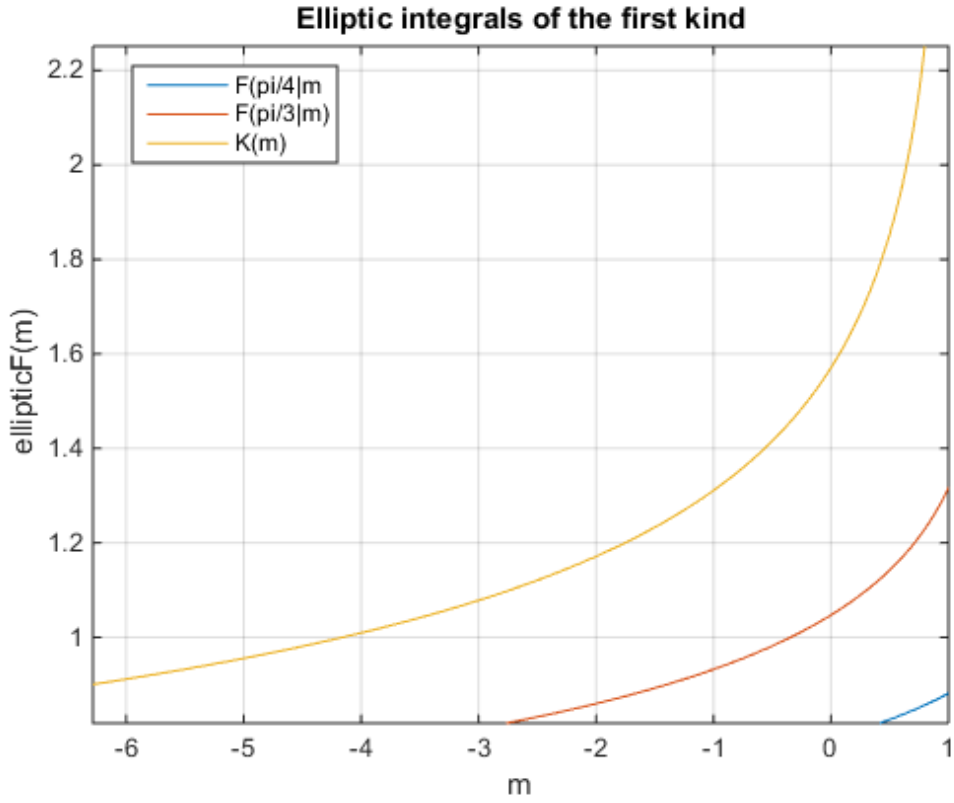
```
ans =
1/(4*(1 - m/2)^(1/2)*(m - 1)) - ellipticF(pi/4, m)/(2*m) -...
ellipticE(pi/4, m)/(2*m*(m - 1))
```

Here, `ellipticE` represents the incomplete elliptic integral of the second kind.

Plot the incomplete elliptic integrals `ellipticF(phi, m)` for `phi = pi/4` and `phi = pi/3`. Also plot the complete elliptic integral `ellipticK(m)`:

```
syms m
ezplot(ellipticF(pi/4, m))
hold on
ezplot(ellipticF(pi/3, m))
ezplot(ellipticK(m))

title('Elliptic integrals of the first kind')
ylabel('ellipticF(m)')
legend('F(pi/4|m', 'F(pi/3|m)', 'K(m)', 'Location', 'Best')
grid on
```

```
hold off
```



More About

Incomplete Elliptic Integral of the First Kind

The complete elliptic integral of the first kind is defined as follows:

$$F(\varphi \mid m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - m\sin^2\theta}}\, d\theta$$

Note that some definitions use the elliptical modulus k or the modular angle α instead of the parameter m. They are related as $m = k^2 = \sin^2\alpha$.

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` | `ellipticF` | `ellipticK` | `ellipticPi` | `vpa`

ellipticK

Complete elliptic integral of the first kind

Syntax

```
ellipticK(m)
```

Description

`ellipticK(m)` returns the complete elliptic integral of the first kind.

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

Compute the complete elliptic integrals of the first kind for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [ellipticK(1/2), ellipticK(pi/4), ellipticK(1),  ellipticK(-5.5)]
```

```
s =
    1.8541    2.2253       Inf    0.9325
```

Compute the complete elliptic integrals of the first kind for the same numbers converted to symbolic objects. For most symbolic (exact) numbers, `ellipticK` returns unresolved symbolic calls.

```
s = [ellipticK(sym(1/2)), ellipticK(sym(pi/4)),...
```

```
 ellipticK(sym(1)),  ellipticK(sym(-5.5))]

s =
[ ellipticK(1/2), ellipticK(pi/4), Inf, ellipticK(-11/2)]
```

Use `vpa` to approximate this result with floating-point numbers:

```
vpa(s, 10)

ans =
[ 1.854074677, 2.225253684, Inf, 0.9324665884]
```

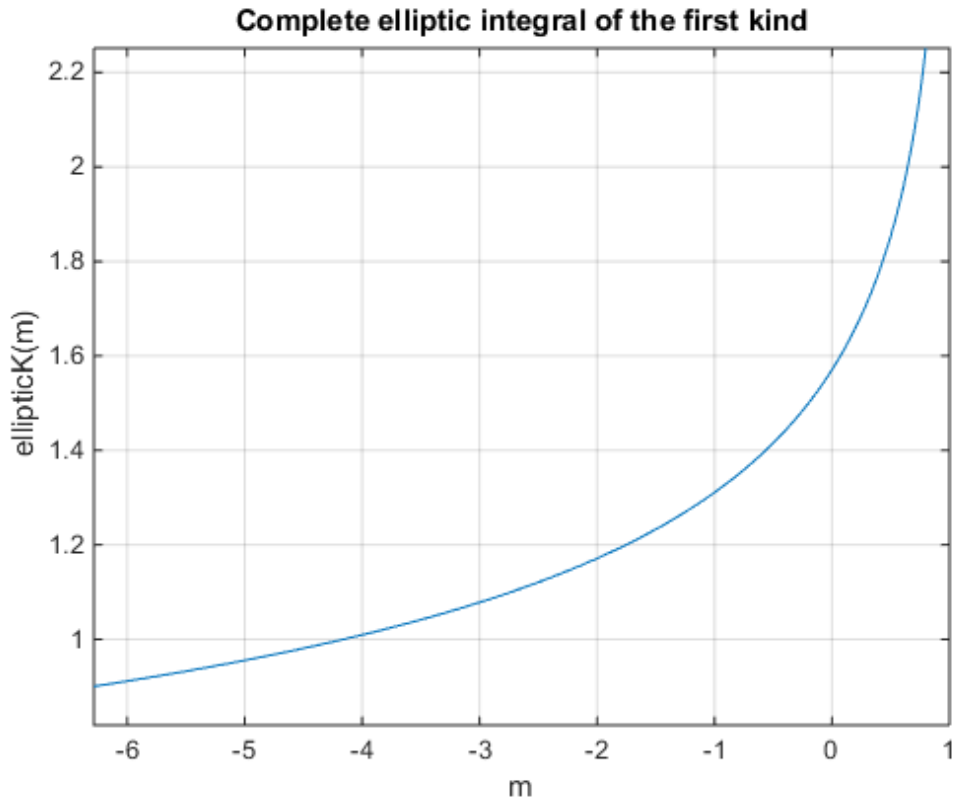
Differentiate these expressions involving the complete elliptic integral of the first kind:

```
syms m
diff(ellipticK(m))
diff(ellipticK(m^2), m, 2)

ans =
- ellipticK(m)/(2*m) - ellipticE(m)/(2*m*(m - 1))

ans =
(2*ellipticE(m^2))/(m^2 - 1)^2 - (2*(ellipticE(m^2)/(2*m^2) -...
ellipticK(m^2)/(2*m^2)))/(m^2 - 1) + ellipticK(m^2)/m^2 +...
(ellipticK(m^2)/m + ellipticE(m^2)/(m*(m^2 - 1)))/m +...
ellipticE(m^2)/(m^2*(m^2 - 1))
```

Here, `ellipticE` represents the complete elliptic integral of the second kind.

Plot the complete elliptic integral of the first kind:

```
syms m
ezplot(ellipticK(m))
title('Complete elliptic integral of the first kind')
ylabel('ellipticK(m)')
grid on
```



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]
```

Alternatives

You can use `ellipke` to compute elliptic integrals of the first and second kinds in one function call.

More About

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.

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` | `ellipticPi` | `vpa`

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.

`ellipticPi(n,phi,m)` returns the incomplete elliptic integral of the third kind.

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.

More About

Incomplete Elliptic Integral of the Third Kind

The incomplete elliptic integral of the third kind is defined as follows:

$$\Pi(n;\varphi \mid m) = \int_0^{\varphi} \frac{1}{(1-n\sin^2\theta)\sqrt{1-m\sin^2\theta}}\,d\theta$$

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:

$$\Pi(n,m) = \Pi\left(n;\frac{\pi}{2} \mid m\right) = \int_0^{\pi/2} \frac{1}{(1-n\sin^2\theta)\sqrt{1-m\sin^2\theta}}\,d\theta$$

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` | `ellipticPi` | `vpa`

eq

Define equation

Note: In previous releases, `eq` evaluated equations and returned logical `1` or `0`. Now it returns unevaluated equations letting you create equations that you can pass to `solve`, `assume`, and other functions. To obtain the same results as in previous releases, wrap equations in `logical` or `isAlways`. For example, use `logical(A == B)`.

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` creates a symbolic equation.

`eq(A,B)` is equivalent to `A == B`.

Input Arguments

A

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

B

Number (integer, rational, floating-point, complex, or symbolic), symbolic variable or expression, or array of numbers, symbolic variables or expressions.

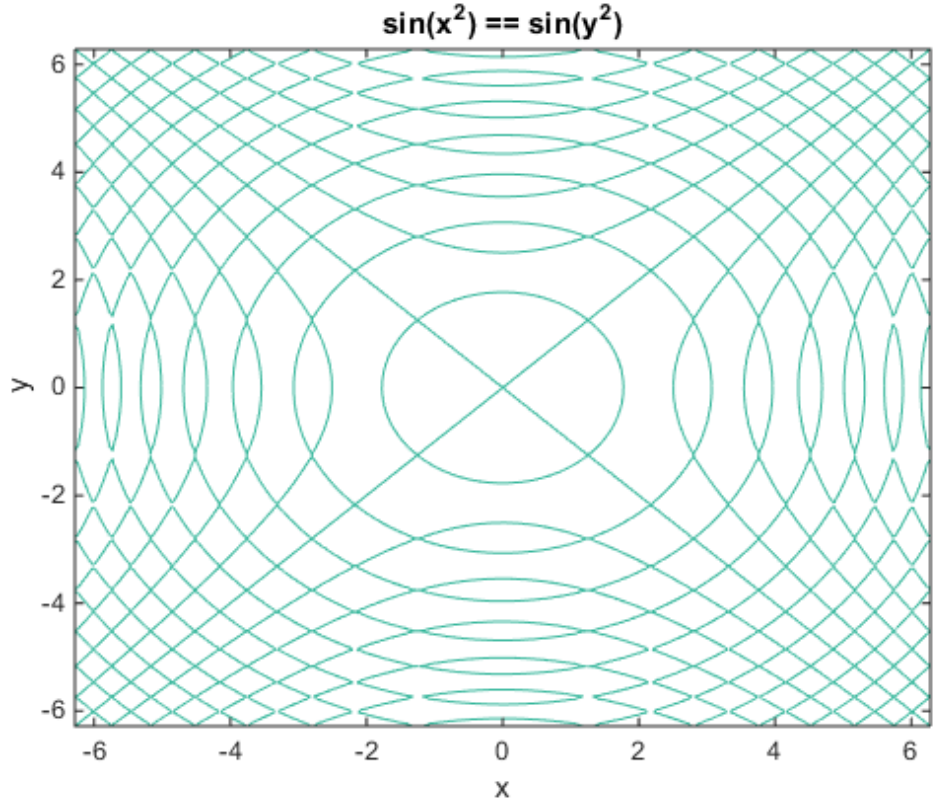
Examples

Solve this trigonometric equation. To define the equation, use the relational operator `==`.

```
syms x
solve(sin(x) == cos(x), x)

ans =
pi/4
```

Plot this trigonometric equation. To define the equation, use the relational operator ==.

```
syms x y
ezplot(sin(x^2) == sin(y^2))
```



Check the equality of two symbolic matrices. Because the elements of both matrices are numbers, == returns logical 1s and 0s:

```
A = sym(hilb(3));
B = sym([1, 1/2, 5; 1/2, 2, 1/4; 1/3, 1/8, 1/5]);
A == B

ans =
     1     1     0
     1     0     1
     1     0     1
```

If you use == to compare a matrix and a scalar, then == expands the scalar into a matrix of the same dimensions as the input matrix:

```
A = sym(hilb(3));
B = sym(1/2);
A == B

ans =
     0     1     0
     1     0     0
     0     0     0
```

If the input arguments are symbolic variables or expression, == does not return logical 1s and 0s. Instead, it creates equations:

```
syms x
x + 1 == x + 1
sin(x)/cos(x) == tan(x)

ans =
x + 1 == x + 1

ans =
sin(x)/cos(x) == tan(x)
```

To test the equality of two symbolic expressions, use `logical` or `isAlways`. Use `logical` when expressions on both sides of the equation do not require simplification or transformation:

```
logical(x + 1 == x + 1)

ans =
     1
```

Use `isAlways` when expressions need to be simplified or transformed or when you use assumptions on variables:

```
isAlways(sin(x)/cos(x) == tan(x))

ans =
     1
```

More About

Tips

- If A and B are both numbers, then `A == B` compares A and B and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A == B` returns a symbolic equation. You can use that equation as an argument for such functions as `solve`, `assume`, `ezplot`, and `subs`.
- If both A and B are arrays, then these arrays must have the same dimensions. `A == B` returns an array of equations `A(i,j,...)==B(i,j,...)`
- If one input is scalar and the other an array, then `==` expands the scalar into an array of the same dimensions as the input array. In other words, if A is a variable (for example, `x`), and B is an *m*-by-*n* matrix, then A is expanded into *m*-by-*n* matrix of elements, each set to `x`.

- "Solve Equations" on page 1-25
- "Set Assumptions" on page 1-32

See Also

`ge` | `gt` | `isAlways` | `le` | `logical` | `lt` | `ne` | `solve`

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.
```

More About

Matrix Representation of a System of Linear Equations

A system of linear equations

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ &\ldots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \end{aligned}$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

and $\vec{b}$ is the vector containing the right sides of equations:

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

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`

Related Examples

- "Solve a System of Differential Equations"

erf

Error function

Syntax

```
erf(X)
```

Description

`erf(X)` represents the error function 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*i
```

Special Values of the 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*i, -Inf*i]
```

Handling Expressions That Contain the 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:

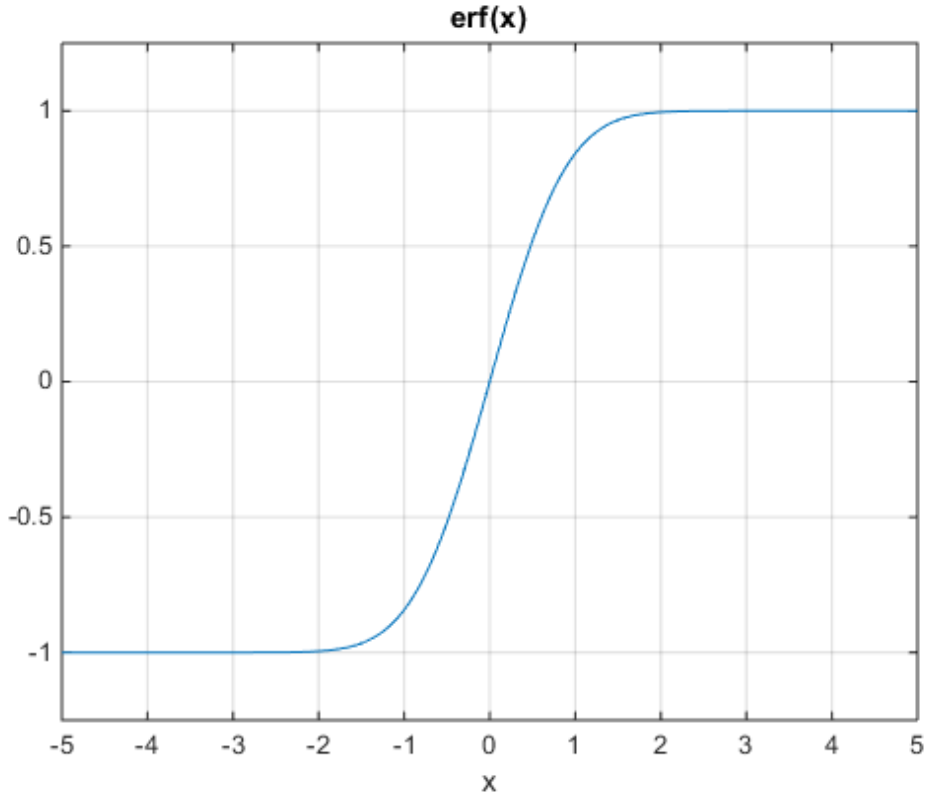
```
int(erf(x), x)
int(erf(log(x)), x)
```

```
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 the Error Function

Plot the error function on the interval from -5 to 5.

```
syms x
ezplot(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.

More About

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`

erfc

Complementary error function

Syntax

```
erfc(X)
erfc(K,X)
```

Description

`erfc(X)` represents the complementary error function of X, that is,`erfc(X) = 1 - erf(X)`.

`erfc(K,X)` represents the iterated integral of the complementary error function 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)
 Inf*i + 1
```

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 the 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))]
```

```
[ 1 - Inf*i, Inf*i + 1]
```

Handling Expressions That Contain the 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)
```

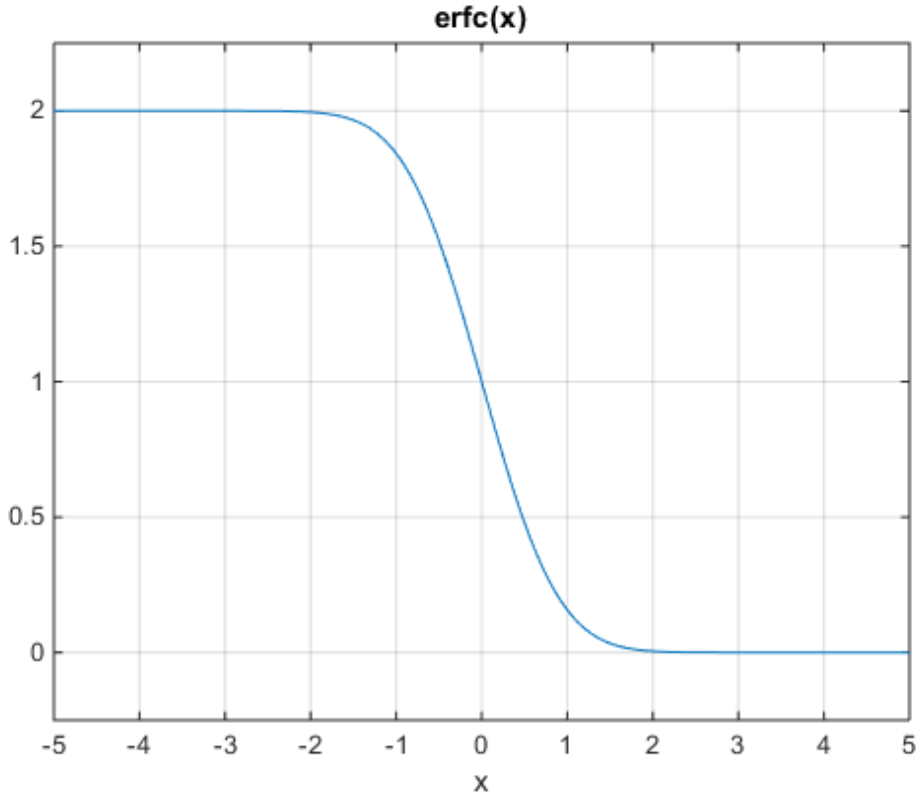
Compute the integrals of these expressions:

```
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 the Complementary Error Function

Plot the complementary error function on the interval from -5 to 5.

```
syms x
ezplot(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.

More About

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 the 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`

erfcinv

Inverse complementary error function

Syntax

```
erfcinv(X)
```

Description

`erfcinv(X)` computes the inverse complementary error function 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 Complex Numbers

To compute the inverse complementary error function for complex numbers, first convert them to symbolic numbers.

Compute the inverse complementary error function for complex numbers. Use `sym` to convert complex numbers to symbolic objects:

```
[erfcinv(sym(2 + 3*i)), erfcinv(sym(1 - i))]

ans =
[ erfcinv(2 + 3*i), -erfcinv(1 + i)]
```

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, erfcinv(1 + i)]
[ -erfcinv(5/3),              0]

ans =
         -Inf
 erfcinv(Inf)
```

Special Values of the 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 the 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
```

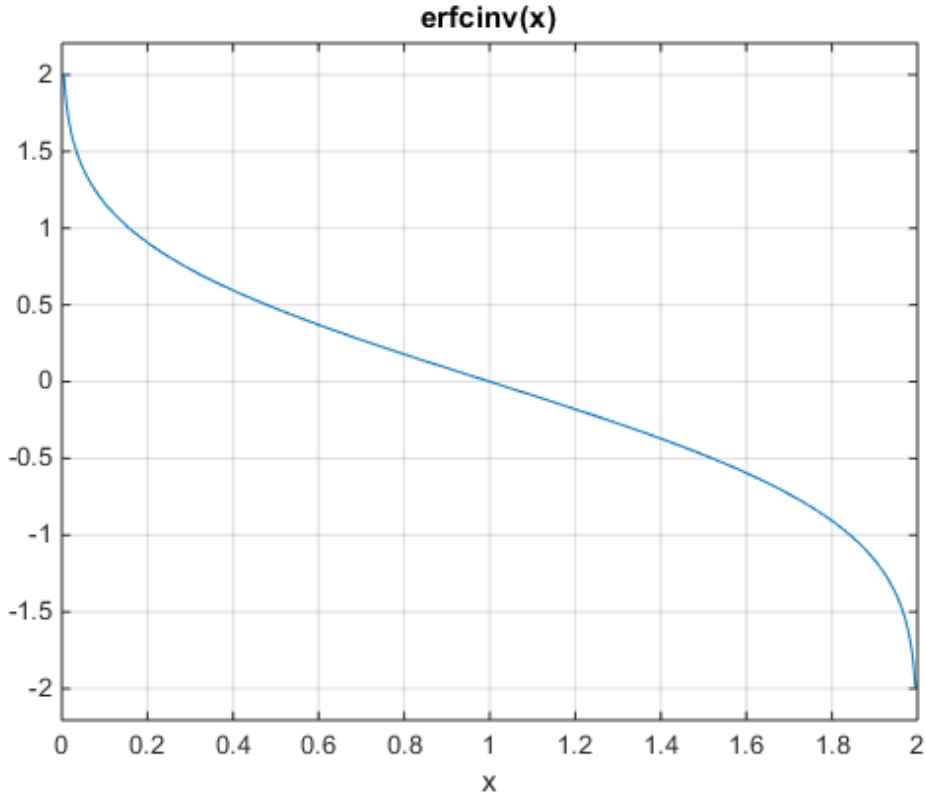
Compute the integral of the inverse complementary error function:

```
int(erfcinv(x), x)

ans =
exp(-erfcinv(x)^2)/pi^(1/2)
```

Plot the Inverse Complementary Error Function

Plot the inverse complementary error function on the interval from 0 to 2.

```
syms x
ezplot(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.

More About

Inverse Complementary Error Function

The inverse complementary error function is defined as $\text{erfc}^{-1}(x)$, such that $\text{erfc}(\text{erfc}^{-1}(x)) = x$. 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 $x > 2$, the MATLAB `erfcinv` function returns `NaN`. The symbolic `erfcinv` function returns unresolved symbolic calls for such numbers. To call the symbolic `erfcinv` function, convert its argument to a symbolic object using `sym`.

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`

erfi

Imaginary error function

Syntax

```
erfi(x)
```

Description

`erfi(x)` returns the "Imaginary Error Function" on page 4-394 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)
      -i
```

Special Values of Imaginary Error Function

Compute the imaginary error function for $x = 0$, $x = \infty$, and $x = -\infty$. Use `sym` to convert `0` and infinities to symbolic objects. The imaginary error function has special values for these parameters:

```
[erfi(sym(0)), erfi(sym(inf)), erfi(sym(-inf))]

ans =
[ 0, Inf, -Inf]
```

Compute the imaginary error function for complex infinities. Use `sym` to convert complex infinities to symbolic objects:

```
[erfi(sym(i*inf)), erfi(sym(-i*inf))]

ans =
[ i, -i]
```

Handling Expressions That Contain the 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)
```

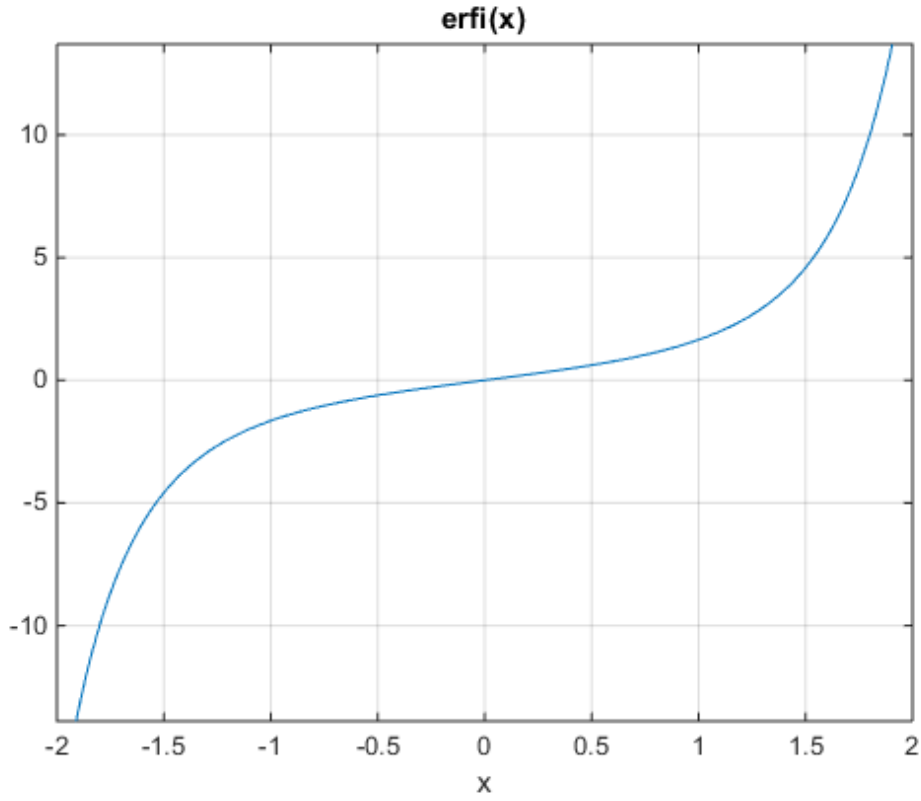
Compute the integrals of these expressions:

```
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 the Imaginary Error Function

Plot the imaginary error function on the interval from -2 to 2.

```
syms x
ezplot(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.

More About

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`

erfinv

Inverse error function

Syntax

```
erfinv(X)
```

Description

`erfinv(X)` computes the inverse error function 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 Complex Numbers

To compute the inverse error function for complex numbers, first convert them to symbolic numbers.

Compute the inverse error function for complex numbers. Use `sym` to convert complex numbers to symbolic objects:

```
[erfinv(sym(2 + 3*i)), erfinv(sym(1 - i))]
```

```
ans =
[ erfinv(2 + 3*i), erfinv(1 - i)]
```

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, erfinv(1 + i)]
[ erfinv(1/3),           Inf]

ans =
        -Inf
 erfinv(Inf)
```

Special Values of the 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 the 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:

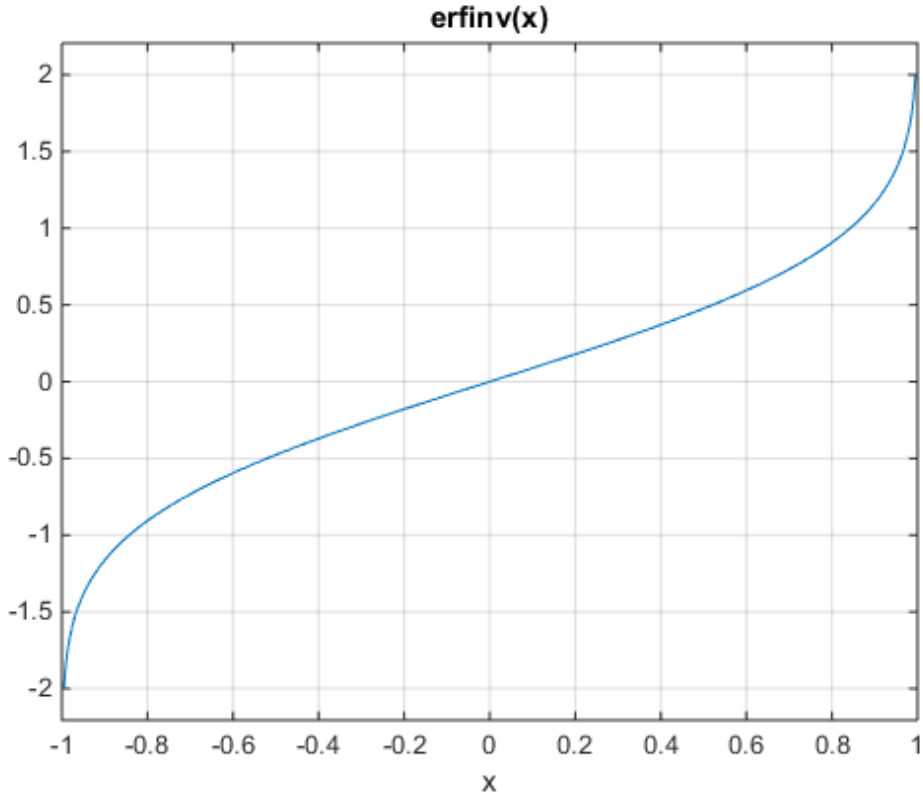
```
int(erfinv(x), x)
```

```
ans =
-exp(-erfinv(x)^2)/pi^(1/2)
```

Plot the Inverse Error Function

Plot the inverse error function on the interval from -1 to 1.

```
syms x
ezplot(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.

More About

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 $x > 1$, the MATLAB `erfinv` function returns `NaN`. The symbolic `erfinv` function returns unresolved symbolic calls for such numbers. To call the symbolic `erfinv` function, convert its argument to a symbolic object using `sym`.

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`

euler

Euler numbers and polynomials

Syntax

```
euler(n)
euler(n,x)
```

Description

`euler(n)` returns the nth Euler number.

`euler(n,x)` returns the nth Euler polynomial.

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))
```

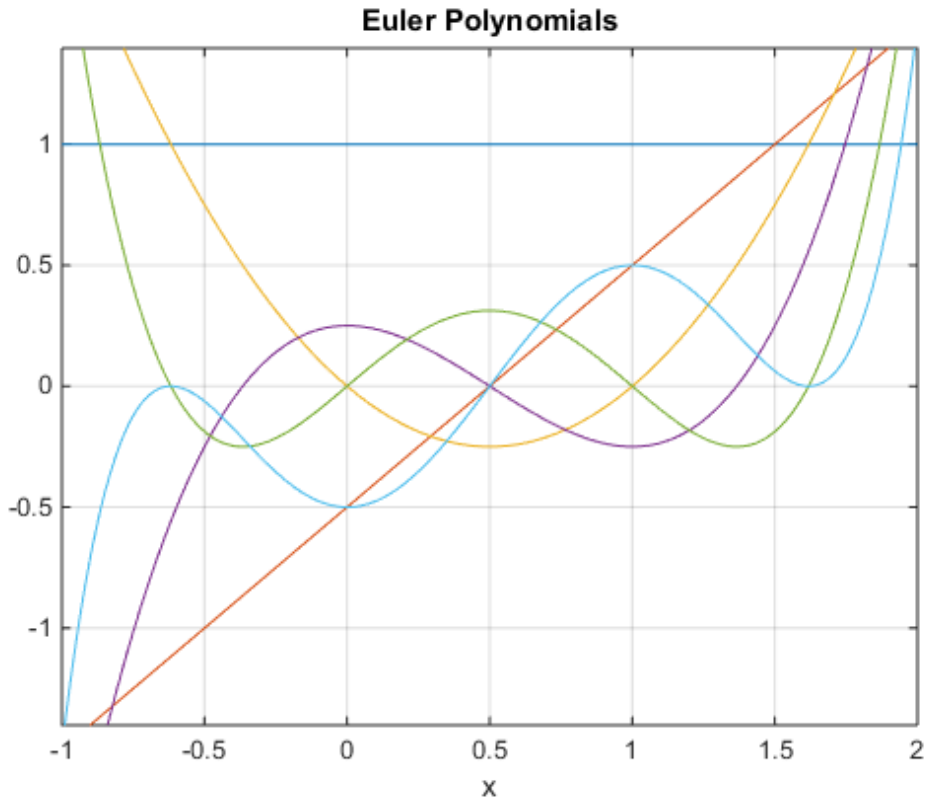
```
ans =
-2/9
```

Plot the Euler Polynomials

Plot the first six Euler polynomials.

```
syms x

for n = 0:5
```

```
  ezplot(euler(n, x), [-1, 2])
  hold on
end

title('Euler Polynomials')
grid on
hold off
```



Handle Expressions Containing the 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.

More About

Euler Polynomials

The Euler polynomials are defined as follows:

$$\frac{2e^{xt}}{e^t+1} = \sum_{n=0}^{\infty} \mathrm{euler}(n,x)\frac{t^n}{n!}$$

Euler Numbers

The Euler numbers are defined as follows:

$$\mathrm{euler}(n) = 2^n\,\mathrm{euler}\left(n,\frac{1}{2}\right)$$

See Also

`bernoulli`

eulergamma

Euler-Mascheroni constant

Syntax

```
eulergamma
```

Description

`eulergamma` represents the Euler-Mascheroni constant. To get a floating-point approximation with the current precision set by `digits`, use `vpa(eulergamma)`.

Examples

Approximate the Euler-Mascheroni Constant

Get a floating-point approximation of the Euler-Mascheroni constant with the default number of digits and with the 10-digit precision.

Use `vpa` to approximate the Euler-Mascheroni constant with the default 32–digit precision:

```
vpa(eulergamma)
```

```
ans =
0.57721566490153286060651209008240
```

Set the number of digits to 10 and approximate the Euler-Mascheroni constant:

```
old = digits(10);
vpa(eulergamma)
```

```
ans =
0.5772156649
```

Restore the default number of digits:

```
digits(old)
```

More About

Euler-Mascheroni Constant

The Euler-Mascheroni constant is defined as follows:

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \log(n)$$

See Also

`coshint`

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 string.

Input Arguments

`MuPAD_expression`

String containing a MuPAD expression.

Output Arguments

`result`

Symbolic object or string containing a MuPAD error message.

`status`

Integer indicating the error status. If MuPAD_expression executes without errors, the error status is 0.

Examples

Compute the discriminant of the following polynomial:

```
evalin(symengine,'polylib::discrim(a*x^2+b*x+c,x)')
```

```
ans =
 b^2 - 4*a*c
```

Try using `polylib::discrim` to compute the discriminant of the following nonpolynomial expression:

```
[result, status] = evalin(symengine,'polylib::discrim(a*x^2+b*x+c*ln(x),x)')
```

```
result =
Error: An arithmetical expression is expected. [normal]

status =
     2
```

Alternatives

`feval` lets you evaluate MuPAD expressions with arguments. When using `feval`, you must explicitly specify the arguments of the MuPAD expression.

More About

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.

- "Evaluations in Symbolic Computations"
- "Level of Evaluation"

See Also

`feval` | `read` | `symengine`

Related Examples

- “Call Built-In MuPAD Functions from MATLAB”

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 a 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 3-24
- “Close MuPAD Notebooks from MATLAB” on page 3-16

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`

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)
```

More About

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:

  - $\log(e^x) = x$
  - $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
  - $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

- "Simplifications" on page 2-40

See Also

`collect` | `combine` | `factor` | `horner` | `numden` | `rewrite` | `simplify` | `simplifyFraction`

expint

Exponential integral function

Syntax

```
expint(x)
expint(n,x)
```

Description

`expint(x)` returns the one-argument exponential integral function defined as follows:

$$\mathrm{expint}(x) = \int_x^\infty \frac{e^{-t}}{t}\,dt$$

`expint(n,x)` returns the two-argument exponential integral function defined as follows:

$$\mathrm{expint}(n,x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt$$

Examples

One-Argument Exponential Integral for Floating-Point and Symbolic Numbers

Compute the exponential integrals for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
s = [expint(1/3), expint(1), expint(-2)]
```

```
s =
```

```
   0.8289 + 0.0000i   0.2194 + 0.0000i  -4.9542 - 3.1416i
```

Compute the exponential integrals for the same numbers converted to symbolic objects. For positive values `x`, `expint(x)` returns `-ei(-x)`. For negative values `x`, it returns `- pi*i - ei(-x)`.

```
s = [expint(sym(1)/3), expint(sym(1)), expint(sym(-2))]
```

```
s =
[ -ei(-1/3), -ei(-1), - pi*i - ei(2)]
```

Use `vpa` to approximate this result with the 10-digit accuracy:

```
vpa(s, 10)
```

```
ans =
[ 0.8288877453, 0.2193839344, - 4.954234356 - 3.141592654*i]
```

Two-Argument Exponential Integral for Floating-Point and Symbolic Numbers

When computing two-argument exponential integrals, convert numbers to symbolic objects:

```
s = [expint(2, sym(1)/3), expint(sym(1), Inf), expint(-1, sym(-2))]
```

```
s =
[ expint(2, 1/3), 0, -exp(2)/4]
```

Use `vpa` to approximate this result with the 25- digit accuracy:

```
vpa(s, 25)
```

```
ans =
[ 0.44023539545759370505220 18, 0, -1.8472640247326625568076 07]
```

Two-Argument Exponential Integral with a Nonpositive First Argument

Compute these two-argument exponential integrals. If `n` is a nonpositive integer, then `expint(n, x)` returns an explicit expression in the form `exp(-x)*p(1/x)`, where `p` is a polynomial of degree `1 - n`.

```
syms x
```

```
expint(0, x)
expint(-1, x)
expint(-2, x)

ans =
exp(-x)/x

ans =
exp(-x)*(1/x + 1/x^2)

ans =
exp(-x)*(1/x + 2/x^2 + 2/x^3)
```

Derivatives of the Exponential Integral

Compute the first, second, and third derivatives of the one-argument exponential integral:

```
syms x
diff(expint(x), x)
diff(expint(x), x, 2)
diff(expint(x), x, 3)

ans =
-exp(-x)/x

ans =
exp(-x)/x + exp(-x)/x^2

ans =
- exp(-x)/x - (2*exp(-x))/x^2 - (2*exp(-x))/x^3
```

Compute the first derivatives of the two-argument exponential integral:

```
syms n x
diff(expint(n, x), x)
diff(expint(n, x), n)

ans =
-expint(n - 1, x)

ans =
- hypergeom([1 - n, 1 - n], [2 - n, 2 - n], -x)/(n - 1)^2 -...
(pi*x^(n - 1)*(psi(n) - log(x) + pi*cot(pi*n)))/(sin(pi*n)*gamma(n))
```

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.

More About

Tips

- `expint(x)` is uniquely defined for positive numbers. It is approximated for the rest of the complex plane.
- Calling `expint` for numbers that are not symbolic objects invokes the MATLAB `expint` function. This function accepts one argument only. To compute the two-argument exponential integral, use `sym` to convert the numbers to symbolic objects, and then call `expint` for those symbolic objects. You can approximate the results with floating-point numbers using `vpa`.
- The following values of the exponential integral differ from those returned by the MATLAB `expint` function: `expint(sym(Inf)) = 0`, `expint(-sym(Inf)) = -Inf`, `expint(sym(NaN)) = NaN`.
- For positive `x`, `expint(x) = -ei(-x)`. For negative `x`, `expint(x) = -pi*i - ei(-x)`.
- If one input argument is a scalar and the other one is a vector or a matrix, `expint(n,x)` expands the scalar into a vector or matrix of the same size as the other argument with all elements equal to that scalar.

See Also

`Ei` | `ei` | `expint` | `vpa`

expm

Matrix exponential

Syntax

```
R = expm(A)
```

Description

`R = expm(A)` computes the matrix exponential 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.

See Also

`eig` | `funm` | `jordan` | `logm` | `sqrtm`

ezcontour

Contour plotter

Syntax

```
ezcontour(f)
ezcontour(f,domain)
ezcontour(...,n)
```

Description

`ezcontour(f)` plots the contour lines of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezcontour(f,domain)` plots $f(x,y)$ over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezcontour(u^2 - v^3, [0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezcontour(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour` automatically adds a title and axis labels.

Examples

Plot the Contour Lines of a Symbolic Expression

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

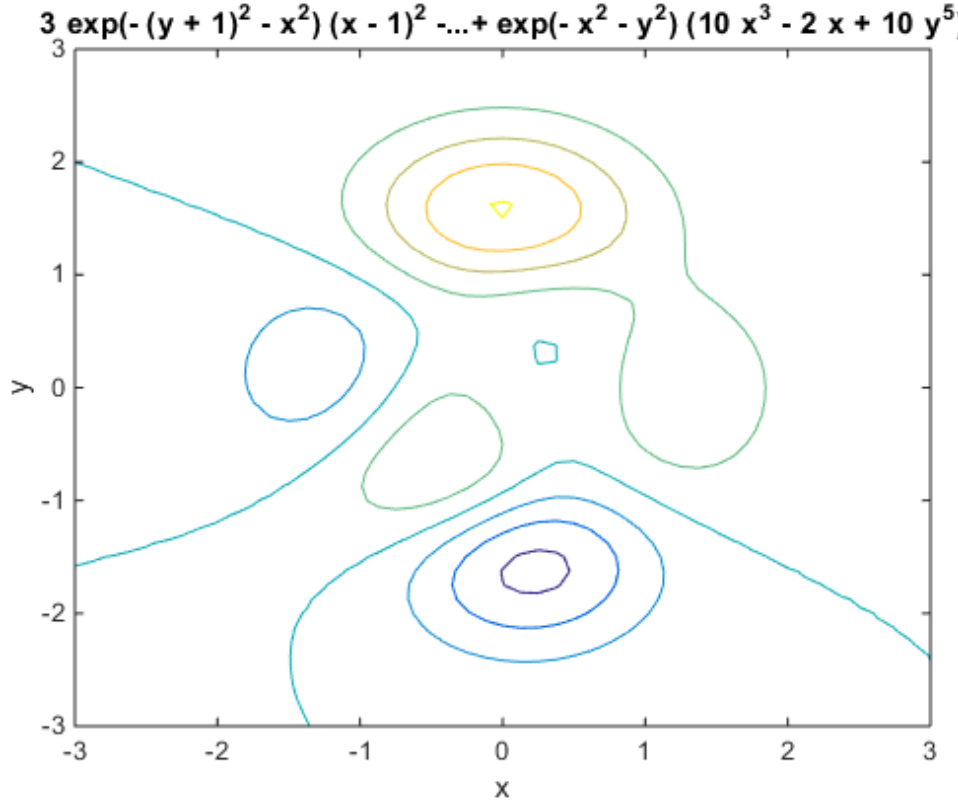
`ezcontour` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
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 string.

See Also

`contour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc`

ezcontourf

Filled contour plotter

Syntax

```
ezcontour(f)
ezcontour(f,domain)
ezcontourf(...,n)
```

Description

`ezcontour(f)` plots the contour lines of $f(x,y)$, where `f` is a `sym` that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezcontour(f,domain)` plots $f(x,y)$ over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezcontourf(u^2 - v^3, [0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezcontourf(...,n)` plots f over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf` automatically adds a title and axis labels.

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}.$$

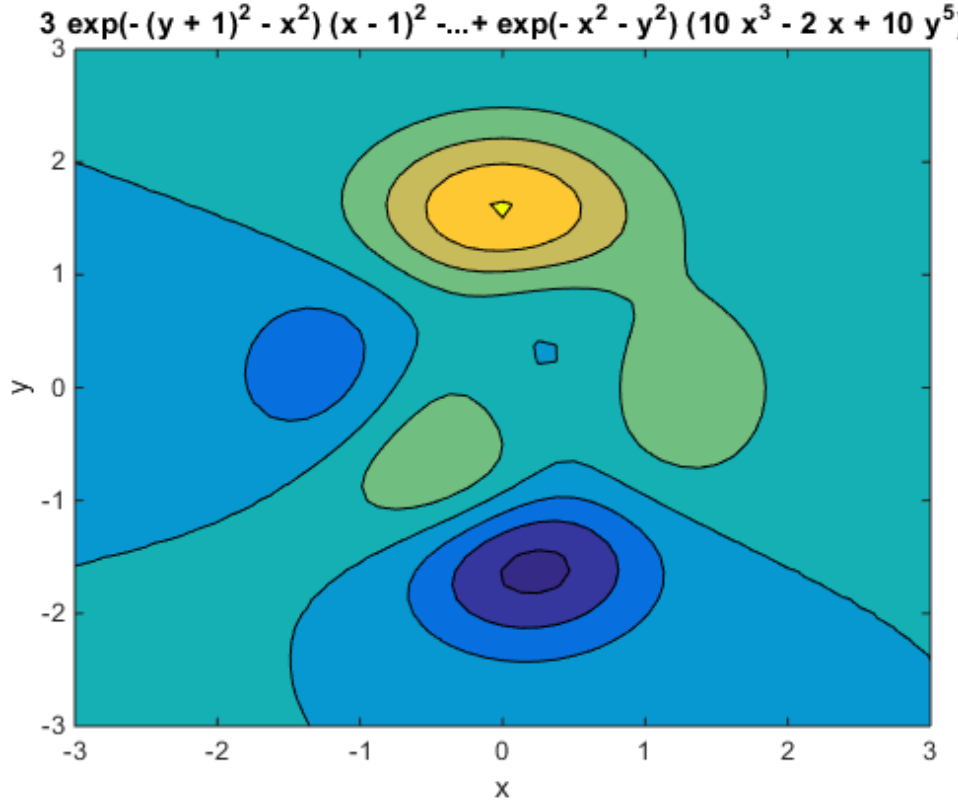
`ezcontourf` requires a `sym` argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the symbolic expression

```
syms x y
f = 3*(1-x)^2*exp(-(x^2)-(y+1)^2)...
   - 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2)...
   - 1/3*exp(-(x+1)^2 - y^2);
```

For convenience, this expression is written on three lines.

Pass the `sym f` to `ezcontourf` along with a domain ranging from `-3` to `3` and specify a grid of 49-by-49.

```
ezcontourf(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB abbreviates the string.

See Also

`contourf` | `ezcontour` | `ezmesh` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc`

ezmesh

3-D mesh plotter

Syntax

```
ezmesh(f)
ezmesh(f, domain)
ezmesh(x,y,z)
ezmesh(x,y,z,[smin,smax,tmin,tmax])
ezmesh(x,y,z,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
```

Description

`ezmesh(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmesh(f, domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezmesh(u^2 - v^3, [0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezmesh(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmesh(x,y,z,[smin,smax,tmin,tmax])` or `ezmesh(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots *f* over a disk centered on the domain.

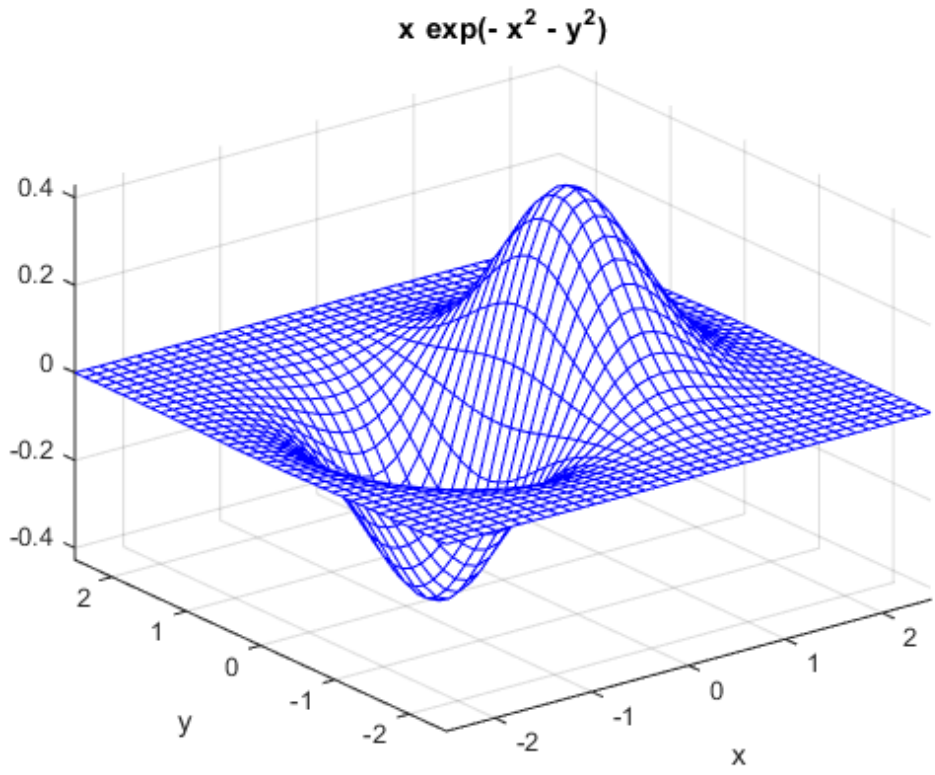
Examples

This example visualizes the function,

$$f(x,y) = xe^{-x^2-y^2},$$

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color.

```
syms x y
ezmesh(x*exp(-x^2-y^2),[-2.5,2.5],40)
colormap([0 0 1])
```



See Also

`ezcontour` | `ezcontourf` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `mesh`

ezmeshc

Combined mesh and contour plotter

Syntax

```
ezmeshc(f)
ezmeshc(f,domain)
ezmeshc(x,y,z)
ezmeshc(x,y,z,[smin,smax,tmin,tmax])
ezmeshc(x,y,z,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
```

Description

`ezmeshc(f)` creates a graph of $f(x,y)$, where f is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezmeshc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector [*xmin, xmax, ymin, ymax*] or a 2-by-1 vector [*min, max*] (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezmeshc(u^2 - v^3, [0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezmeshc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezmeshc(x,y,z,[smin,smax,tmin,tmax])` or `ezmeshc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots *f* over a disk centered on the domain.

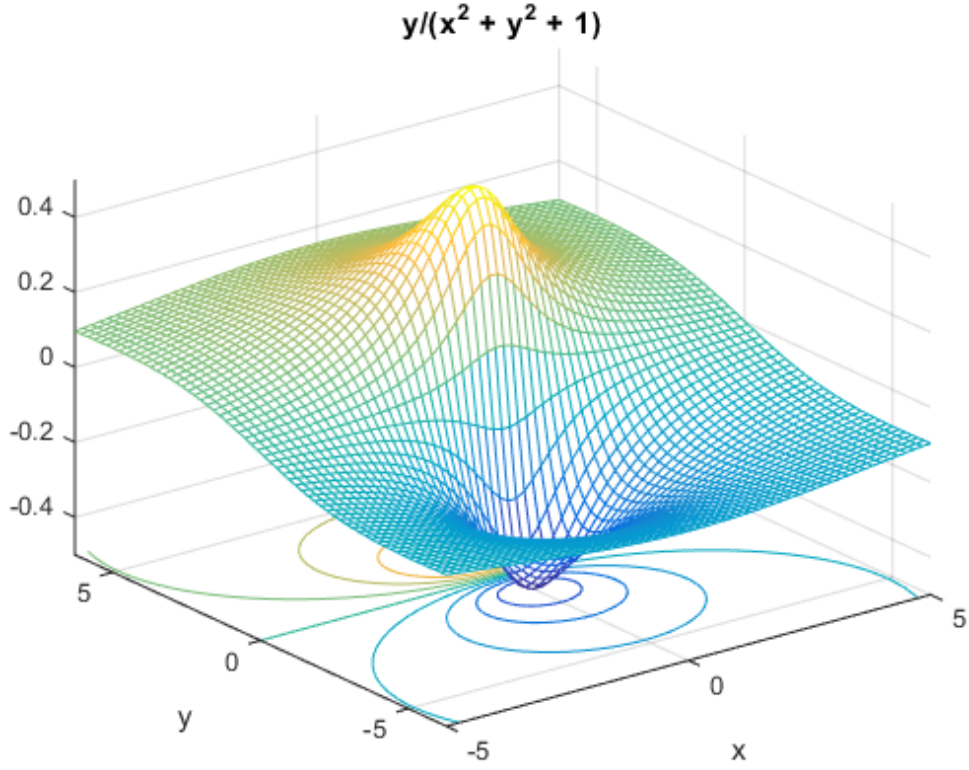
Examples

Create a mesh/contour graph of the expression,

$$f(x,y) = \frac{y}{1 + x^2 + y^2},$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$. 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

`ezcontour` | `ezcontourf` | `ezmesh` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `meshc`

ezplot

Plot symbolic expression, equation, or function

Syntax

```
ezplot(f)
ezplot(f,[min,max])
ezplot(f,[xmin,xmax,ymin,ymax])
ezplot(f,fign)
ezplot(x,y)
ezplot(x,y,[tmin,tmax])
ezplot(f,figure_handle)
```

Description

`ezplot(f)` plots a symbolic expression, equation, or function f. By default, `ezplot` plots a univariate expression or function over the range [–2π 2π] or over a subinterval of this range. If f is an equation or function of two variables, the default range for both variables is [–2π 2π] 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(f,fign)` displays the plot in the plot window with the number fign. The title of each plot window contains the word Figure and the number, for example, **Figure 1**, **Figure 2**, and so on. If the plot window with the number fign is already opened, `ezplot` overwrites the content of that window with the new plot.

`ezplot(x,y)` plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default range $0 <= t <= 2\pi$ or over a subinterval of this range.

`ezplot(x,y,[tmin,tmax])` plots $x = x(t)$ and $y = y(t)$ over the specified range $tmin <= t <= tmax$.

`ezplot(f,figure_handle)` plots f in the plot window identified by the handle figure_handle.

Input Arguments

`f`

Symbolic expression, equation, or function.

`[min,max]`

Numbers specifying the plotting range. For a univariate expression or function, the plotting range applies to that variable. For an equation or function of two variables, the plotting range applies to both variables. In this case, the range is the same for the abscissa and the ordinate.

Default: `[-2*pi,2*pi]` or its subinterval.

`[xmin,xmax,ymin,ymax]`

Numbers specifying the plotting range along the abscissa (first two numbers) and the ordinate (last two numbers).

Default: `[-2*pi,2*pi,-2*pi,2*pi]` or its subinterval.

`fign`

Number of the figure window where you want to display a plot.

Default: If no plot windows are open, then 1. If one plot window is open, then the number in the title of that window. If more than one plot window is open, then the highest number in the titles of open windows.

`x,y`

Symbolic expressions or functions defining a parametric curve $x = x(t)$ and $y = y(t)$.

`[tmin,tmax]`

Numbers specifying the plotting range for a parametric curve.

Default: `[0,2*pi]` or its subinterval.

`figure_handle`

Figure handle specifying the plot window in which you create or modify a plot.

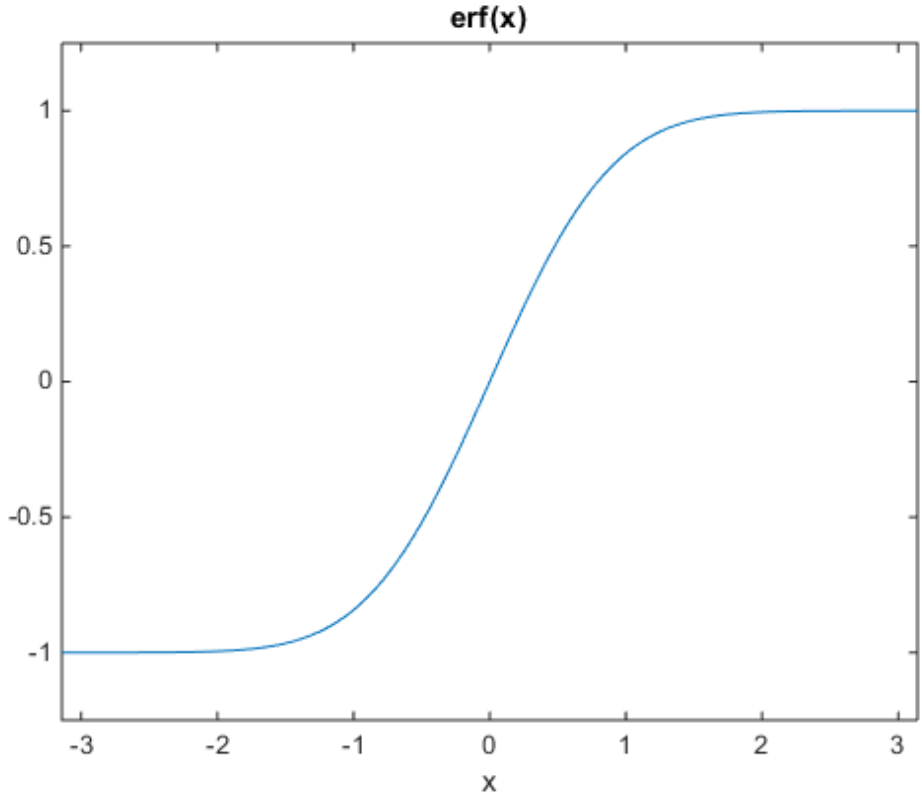
Default: Current figure handle returned by `gcf`.

Examples

Plot Over a Particular Range

Plot the expression `erf(x)*sin(x)` over the range $[-\pi, \pi]$:

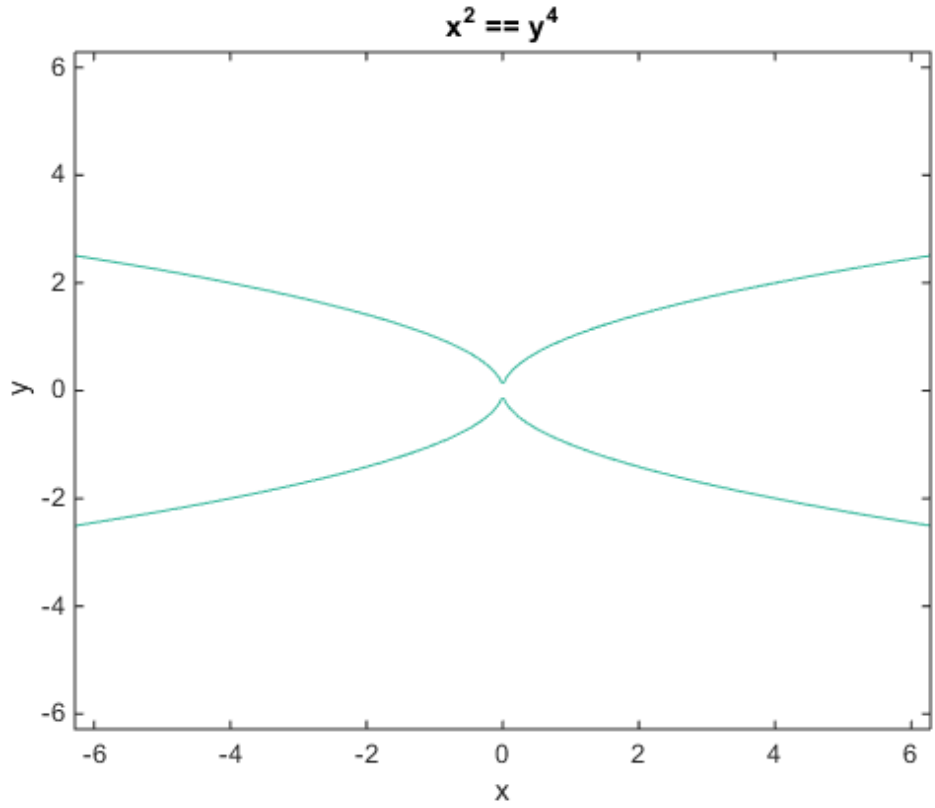
```
syms x
ezplot(erf(x), [-pi, pi])
```



Plot Over the Default Range

Plot this equation over the default range.

```
syms x y
ezplot(x^2 == y^4)
```



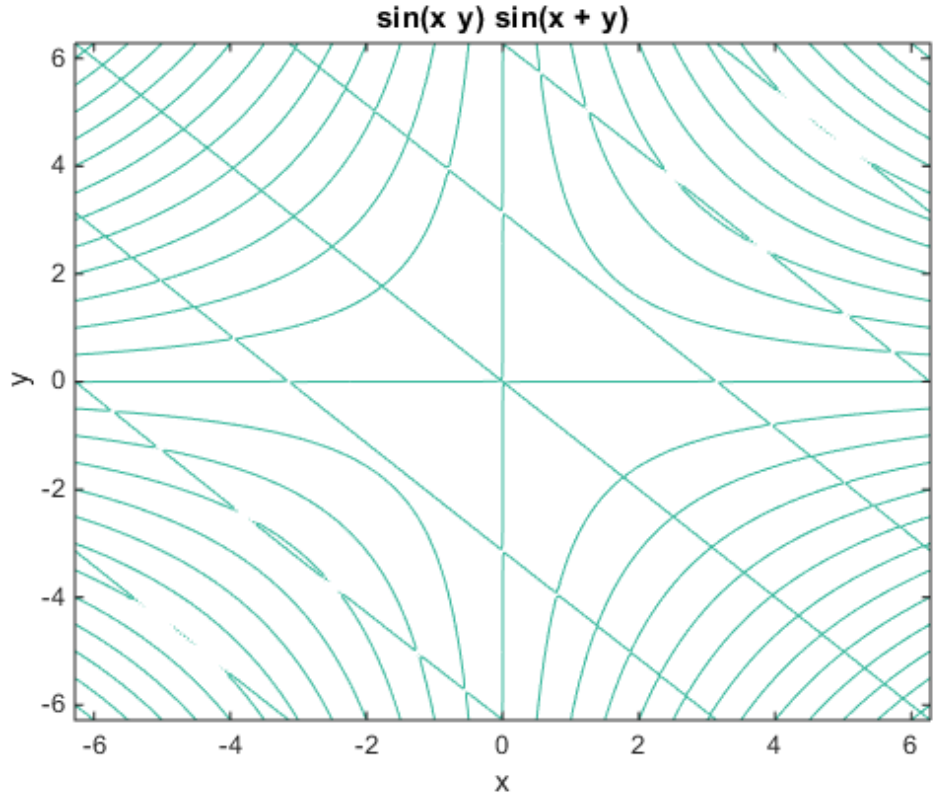
Plot a Symbolic Function

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:

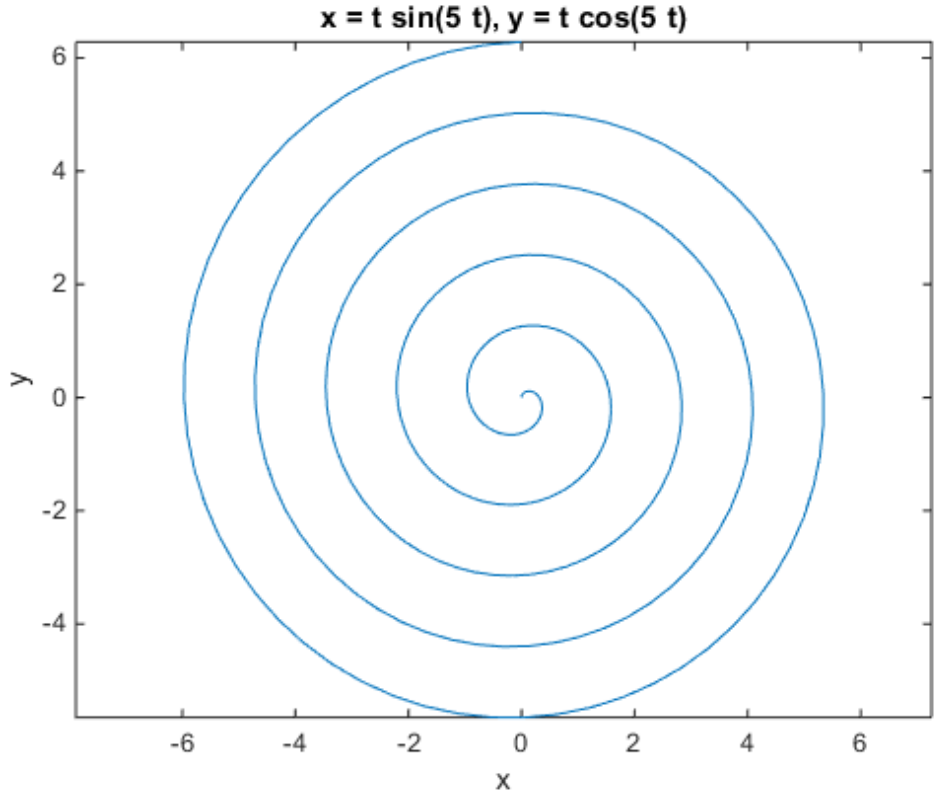
```
ezplot(f)
```



Plot a Parametric Curve

Plot this parametric curve:

```
syms t
x = t*sin(5*t);
y = t*cos(5*t);
ezplot(x, y)
```



More About

Tips

- If you do not specify a plot range, `ezplot` uses the interval [–2π 2π] as a starting point. Then it can choose to display a part of the plot over a subinterval of [–2π 2π] where the plot has significant variation. Also, when selecting the plotting range, `ezplot` omits extreme values associated with singularities.
- `ezplot` open a plot window and displays a plot there. If any plot windows are already open, `ezplot` does not create a new window. Instead, it displays the new plot in the currently active window. (Typically, it is the window with the highest number.) To

display the new plot in a new plot window or in an existing window other than that with highest number, use fign.

- If f is an equation or function of two variables, then the alphabetically first variable defines the abscissa (horizontal axis) and the other variable defines the ordinate (vertical axis). Thus, `ezplot(x^2 == a^2,[-3,3,-2,2])` creates the plot of the equation $x^2 = a^2$ with $-3 <= a <= 3$ along the horizontal axis, and $-2 <= x <= 2$ along the vertical axis.

- "Create Plots"

See Also

`ezcontour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `plot`

ezplot3

3-D parametric curve plotter

Syntax

```
ezplot3(x,y,z)
ezplot3(x,y,z,[tmin,tmax])
ezplot3(...,'animate')
```

Description

`ezplot3(x,y,z)` plots the spatial curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the default domain $0 < t < 2\pi$.

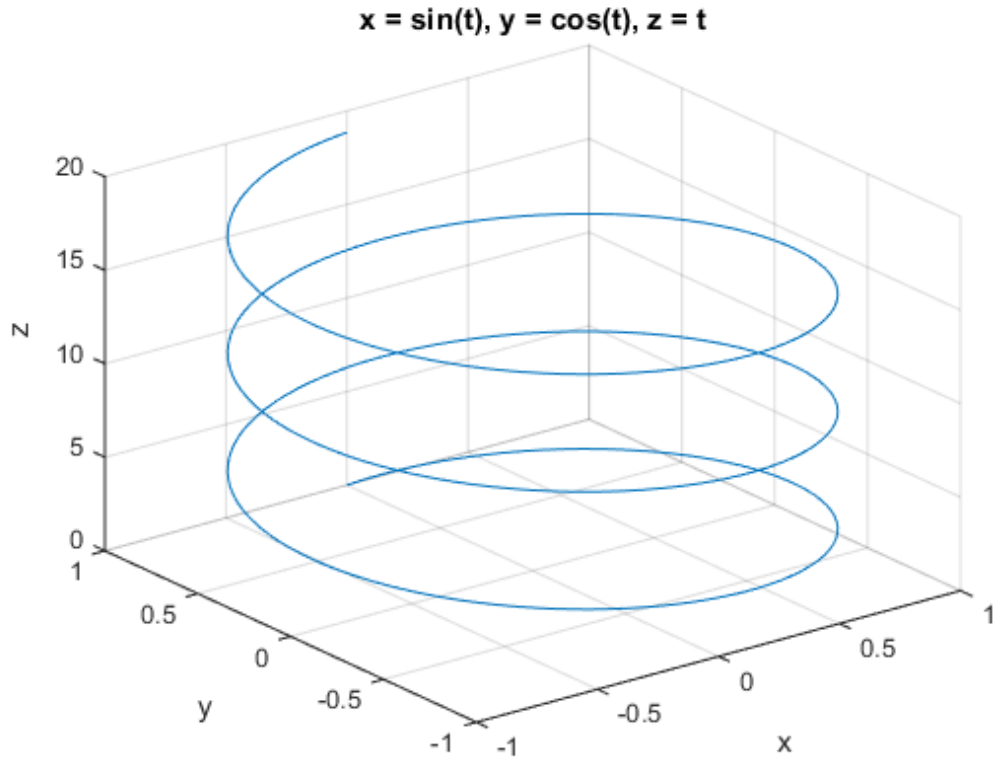
`ezplot3(x,y,z,[tmin,tmax])` plots the curve $x = x(t)$, $y = y(t)$, and $z = z(t)$ over the domain $tmin < t < tmax$.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

Examples

Plot the parametric curve $x = sin(t)$, $y = cos(t)$, $z = t$ over the domain $[0, 6\pi]$.

```
syms t
ezplot3(sin(t), cos(t), t,[0,6*pi])
```



See Also

`ezcontour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezpolar` | `ezsurf` | `ezsurfc` | `plot3`

ezpolar

Polar coordinate plotter

Syntax

```
ezpolar(f)
ezpolar(f, [a, b])
```

Description

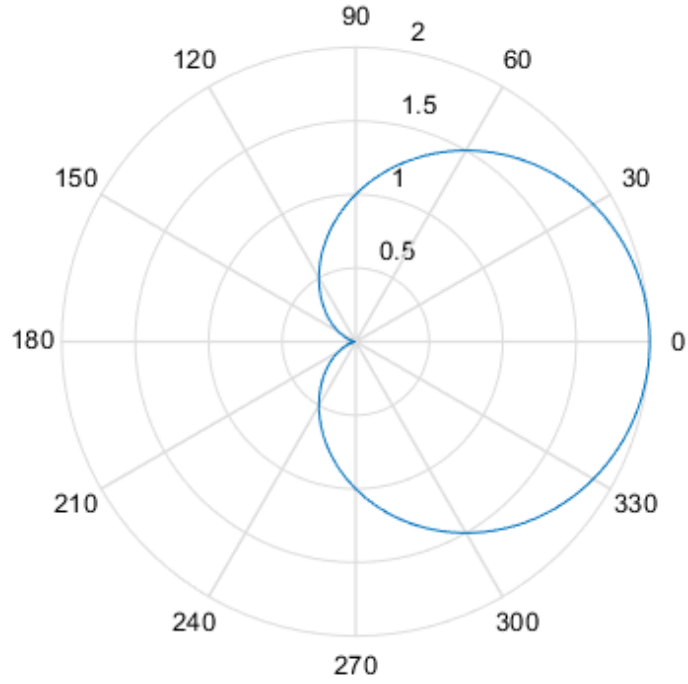
`ezpolar(f)` plots the polar curve $r = f(\theta)$ over the default domain $0 < \theta < 2\pi$.

`ezpolar(f, [a, b])` plots f for $a < \theta < b$.

Examples

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

ezsurf

Plot 3-D surface

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 a Function Over the Default Range

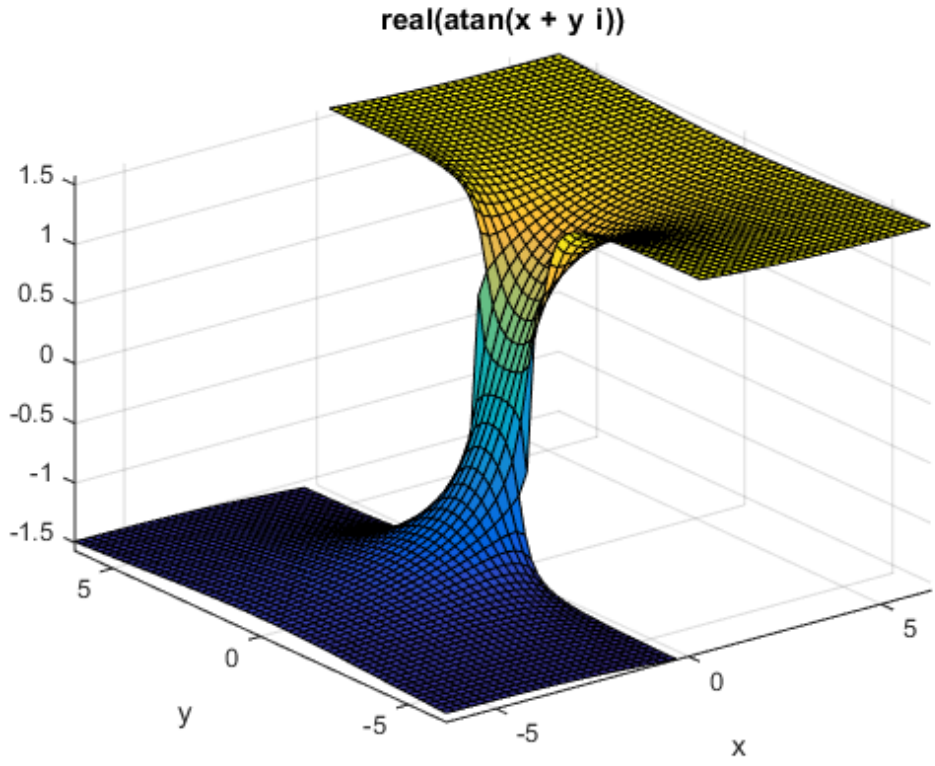
Plot the symbolic function `f(x,y) = real(atan(x + i*y))` over the default range `-2*pi < x < 2*pi`, `-2*pi < y < 2*pi`.

Create the symbolic function.

```
syms f(x,y)
f(x,y) = real(atan(x + i*y));
```

Plot this function using `ezsurf`.

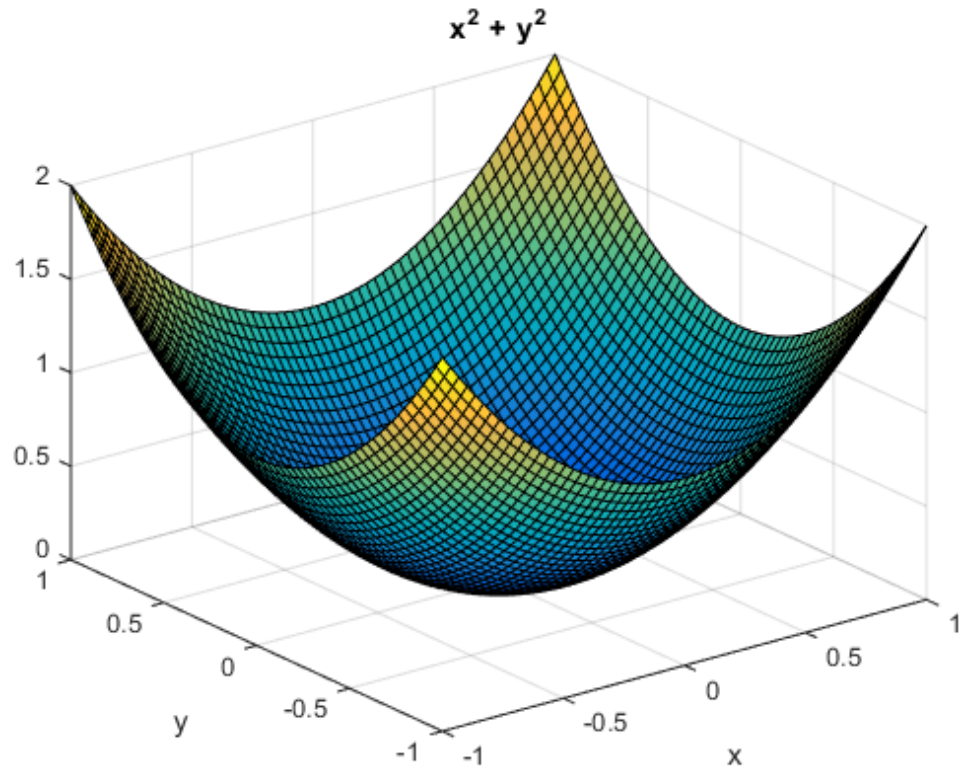
```
ezsurf(f)
```



Specify Plotting Ranges

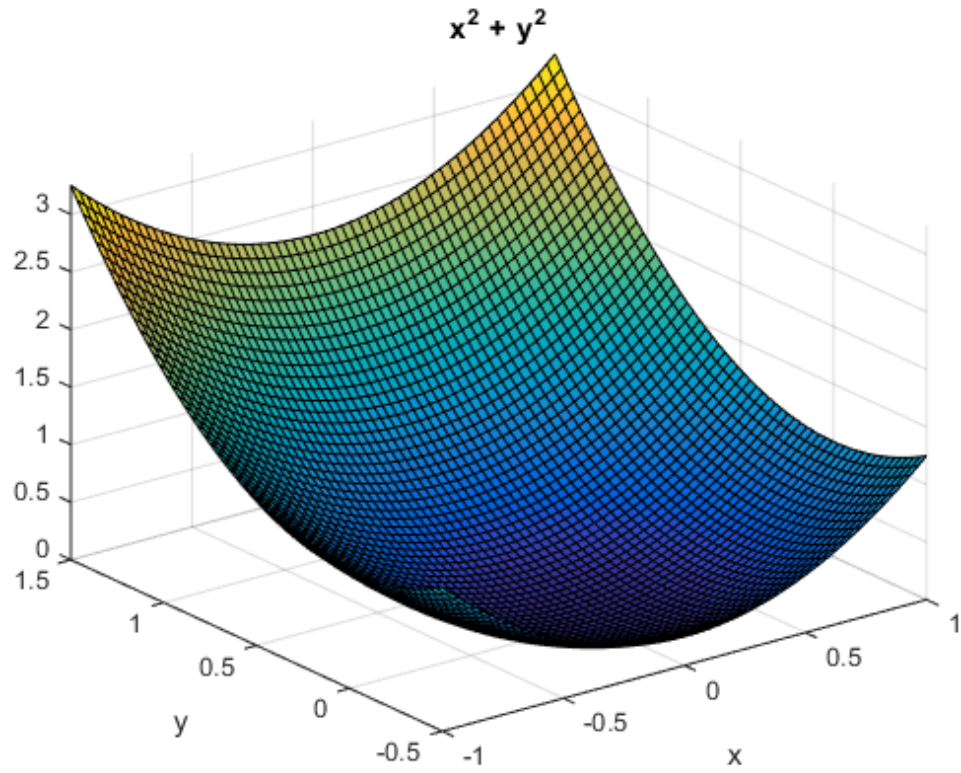
Plot the symbolic expression `x^2 + y^2` over the range `-1 < x < 1`. Because you do not specify the range for the `y`-axis, `ezsurf` chooses it automatically.

```
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])
```



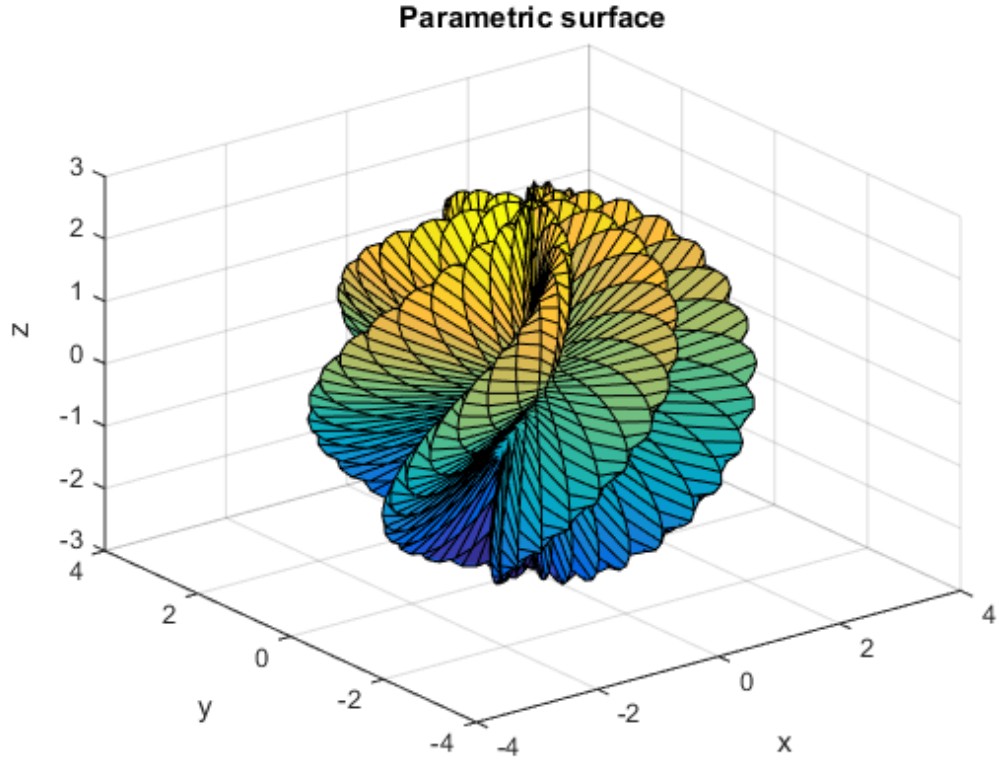
Plot a Parameterized Surface

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);
```

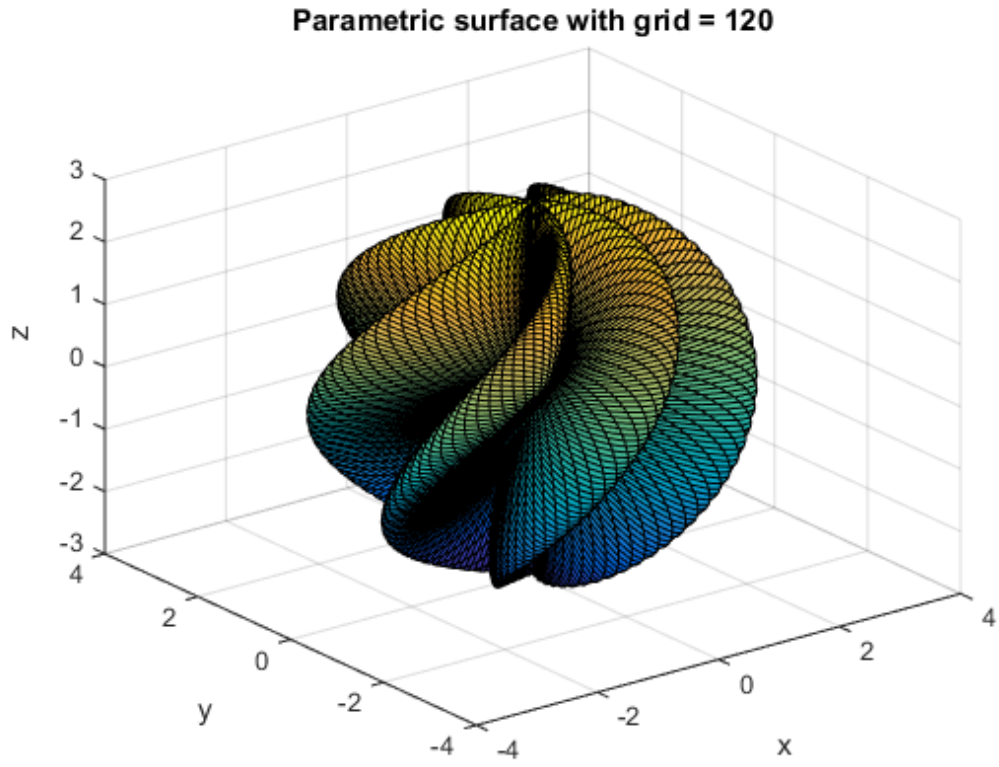
Plot the function using `ezsurf`.

```
ezsurf(x, y, z, [0, 2*pi, 0, pi])
title('Parametric surface')
```



To create a smoother plot, increase the number of mesh points.

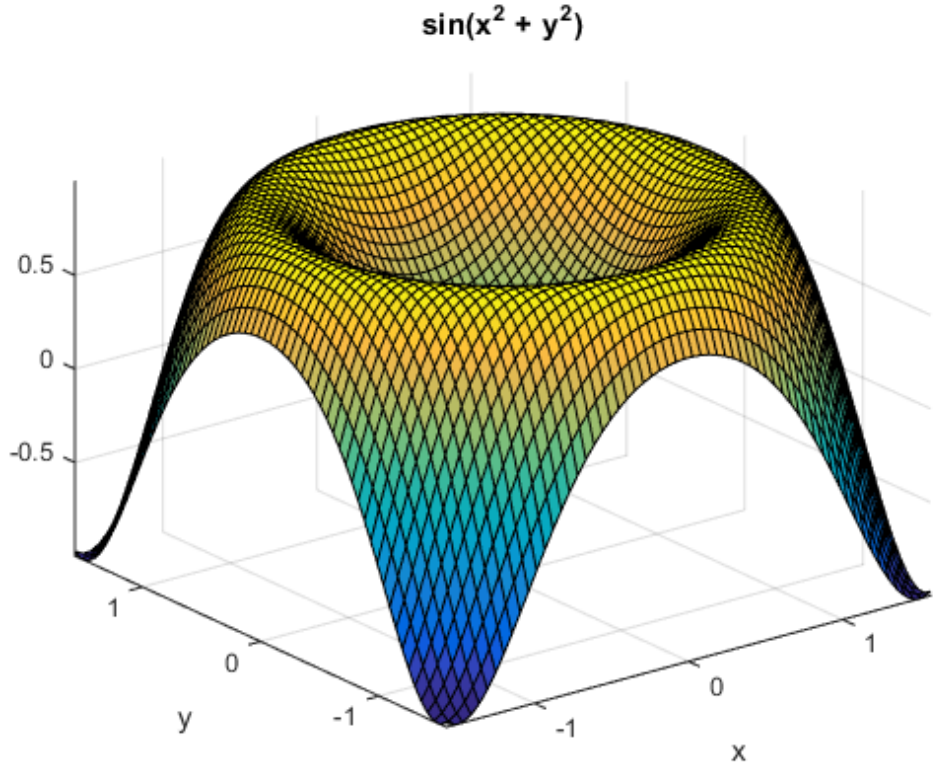
```
ezsurf(x, y, z, [0, 2*pi, 0, pi], 120)
title('Parametric surface with grid = 120')
```



Specify a Disk Plotting Range

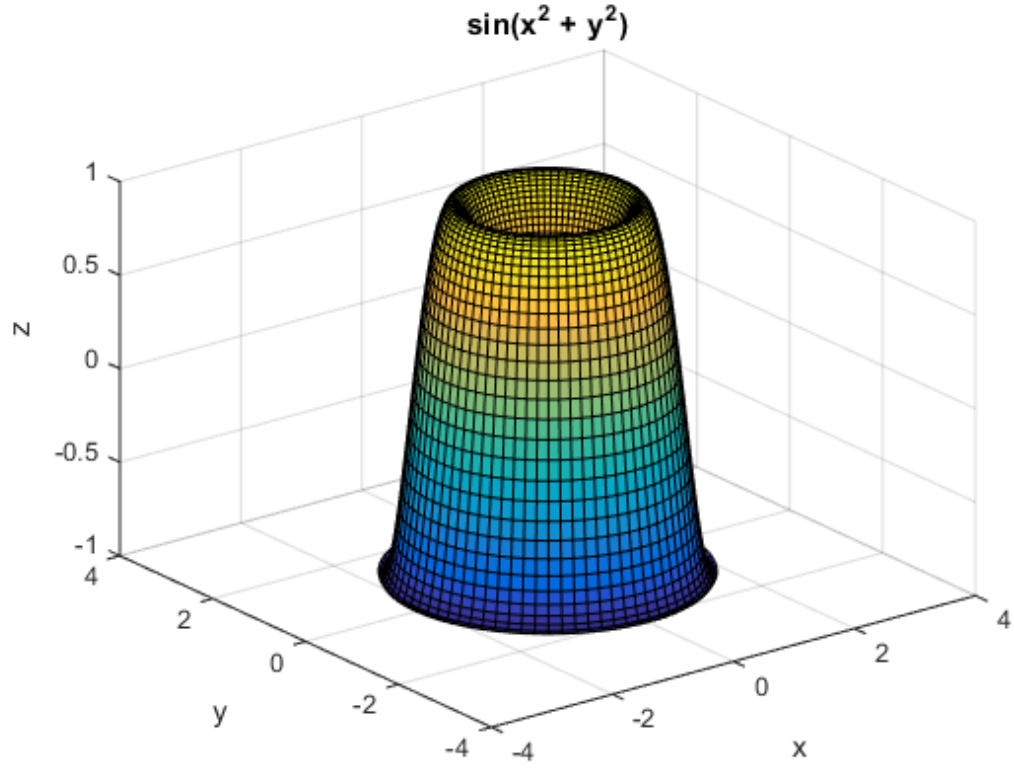
First, plot the expression `sin(x^2 + y^2)` over the square range `-pi/2 < x < pi/2`, `-pi/2 < y < pi/2`.

```
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')
```



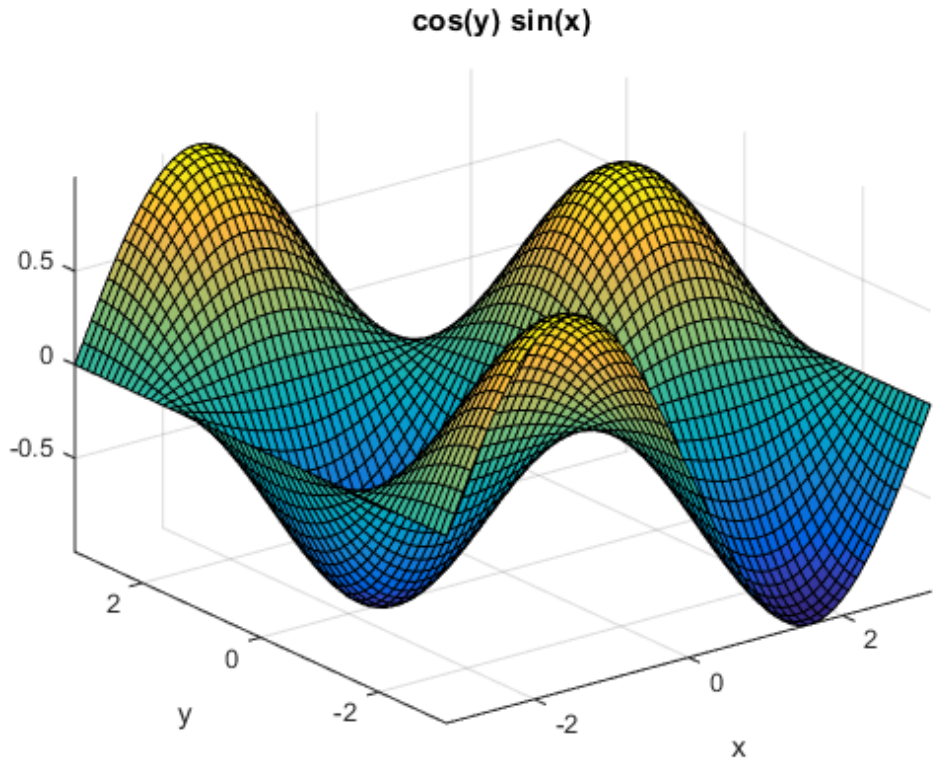
Use a Handle to the Surface Plot

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'
```

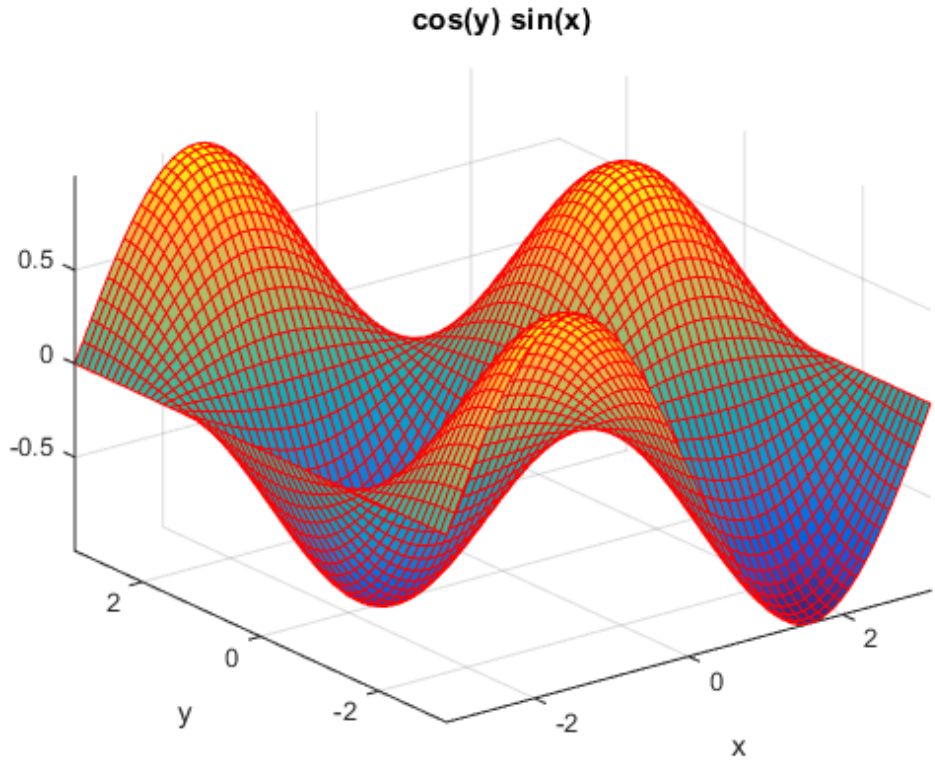
```
    FaceLighting: 'flat'
       FaceAlpha: 1
           XData: [60x60 double]
           YData: [60x60 double]
           ZData: [60x60 double]
           CData: [60x60 double]

  Use GET to 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]

  Use GET to show all properties
```



- “Create Plots”

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.

More About

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

`ezcontour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezpolar` | `ezsurfc` | `surf`

ezsurfc

Combined surface and contour plotter

Syntax

```
ezsurfc(f)
ezsurfc(f,domain)
ezsurfc(x,y,z)
ezsurfc(x,y,z,[smin,smax,tmin,tmax])
ezsurfc(x,y,z,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
```

Description

`ezsurfc(f)` creates a graph of $f(x,y)$, where `f` is a symbolic expression that represents a mathematical function of two variables, such as x and y.

The function f is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$. MATLAB software chooses the computational grid according to the amount of variation that occurs; if the function f is not defined (singular) for points on the grid, then these points are not plotted.

`ezsurfc(f,domain)` plots f over the specified `domain`. `domain` can be either a 4-by-1 vector *[xmin, xmax, ymin, ymax]* or a 2-by-1 vector *[min, max]* (where, $min < x < max$, $min < y < max$).

If f is a function of the variables u and v (rather than x and y), then the domain endpoints *umin*, *umax*, *vmin*, and *vmax* are sorted alphabetically. Thus, `ezsurfc(u^2 - v^3,[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

`ezsurfc(x,y,z)` plots the parametric surface $x = x(s,t)$, $y = y(s,t)$, and $z = z(s,t)$ over the square $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurfc(x,y,z,[smin,smax,tmin,tmax])` or `ezsurfc(x,y,z,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots *f* over a disk centered on the domain.

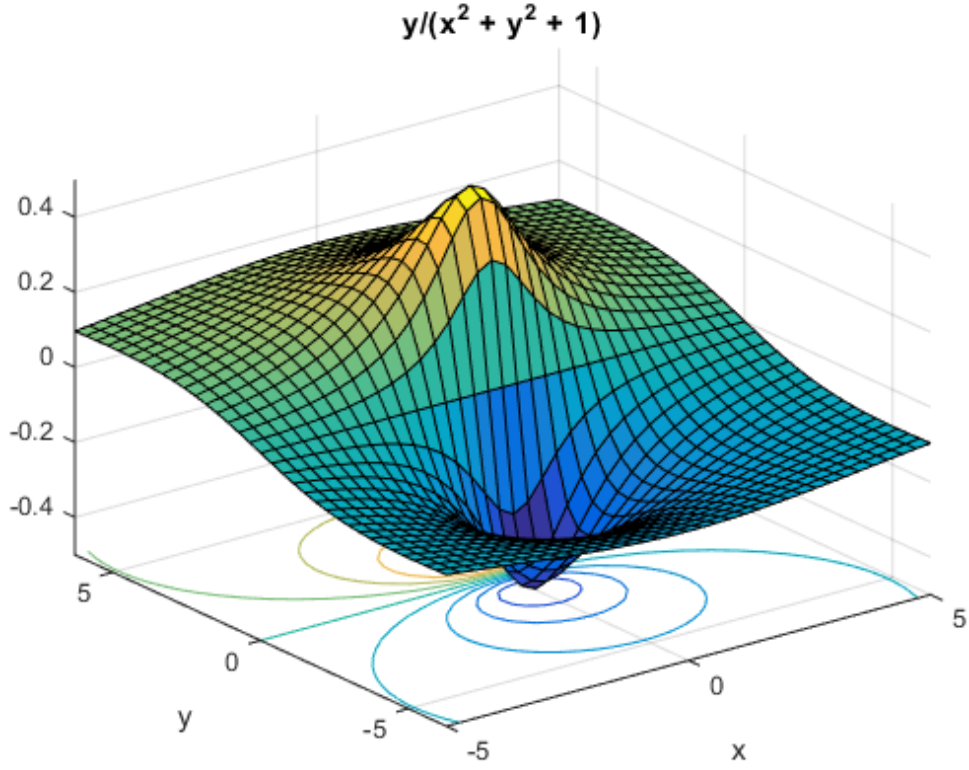
Examples

Create a surface/contour plot of the expression,

$$f(x,y) = \frac{y}{1 + x^2 + y^2},$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$, with a computational grid of size 35-by-35. 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

ezcontour | ezcontourf | ezmesh | ezmeshc | ezplot | ezpolar | ezsurf | surfc

factor

Factorization

Syntax

```
F = factor(x)
F = factor(x,vars)
```

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.

Examples

Factor an Integer

```
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 a Large Number

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 a Symbolic Fraction

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 a Polynomial

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`.

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.

Output Arguments

F — Factors of input

symbolic vector

Factors of input, returned as a symbolic vector.

More About

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`

factorial

Factorial function

Syntax

```
factorial(n)
factorial(A)
```

Description

`factorial(n)` returns the factorial of n.

`factorial(A)` returns the factorials of each element of A.

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]
```

More About

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`

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, with result a symbolic object. If F with the arguments x1,...,xn throws an error in MuPAD, then this syntax throws an error in MATLAB.

`[result,status] = feval(symengine,F,x1,...,xn)` lets you catch errors thrown by MuPAD. This syntax returns the error status in status, and the error message in result if `status` is nonzero. If status is 0, result is a symbolic object. Otherwise, result is a string.

Input Arguments

`F`

MuPAD function name or symbolic object.

`x1,...,xn`

Arguments of F.

Output Arguments

`result`

Symbolic object or string containing a MuPAD error message.

status

Integer indicating the error status. If F with the arguments x1,...,xn executes without errors, the error status is 0.

Examples

```
syms a b c x
p = a*x^2+b*x+c;
feval(symengine,'polylib::discrim', p, x)

ans =
b^2 - 4*a*c
```

Alternatively, the same calculation based on variables not defined in the MATLAB workspace is:

```
feval(symengine,'polylib::discrim', 'a*x^2 + b*x + c', 'x')

ans =
b^2 - 4*a*c
```

Try using `polylib::discrim` to compute the discriminant of the following nonpolynomial expression:

```
[result, status] = feval(symengine,'polylib::discrim',...
                                   'a*x^2 + b*x + c*ln(x)', 'x')

result =
Error: An arithmetical expression is expected. [normal]

status =
     2
```

Alternatives

`evalin` lets you evaluate MuPAD expressions without explicitly specifying their arguments.

More About

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.

- "Evaluations in Symbolic Computations"
- "Level of Evaluation"

See Also

`evalin` | `read` | `symengine`

Related Examples

- "Call Built-In MuPAD Functions from MATLAB"

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 =
    [1x2 double]    [2]    [4]

varsBlocks =
    [1x2 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 =
    [1x3 double]    [1x2 double]    [4]

varsBlocks =
    [1x3 double]    [1x2 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})`.

More About

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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies` | `tril` | `triu`

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)
```

More About

Tips

- `finverse` does not issue a warning when the inverse is not unique.

See Also

`compose` | `syms`

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

`round` | `ceil` | `floor` | `frac`

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

`round` | `ceil` | `fix` | `frac`

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

fortran

Fortran representation of symbolic expression

Syntax

```
fortran(S)
fortran(S,'file',fileName)
```

Description

`fortran(S)` returns the Fortran code equivalent to the expression `S`.

`fortran(S,'file',fileName)` writes an "optimized" Fortran code fragment that evaluates the symbolic expression `S` to the file named `fileName`. "Optimized" means intermediate variables are automatically generated in order to simplify the code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
fortran(f)
```

return

```
ans =
      t0 = x-x**2*(1.0D0/2.0D0)+x**3*(1.0D0/3.0D0)-x**4*(1.0D0/4.0D0)+x*
     &*5*(1.0D0/5.0D0)
```

The statements

```
H = sym(hilb(3));
fortran(H)
```

return

```
ans =
      H(1,1) = 1.0D0
      H(1,2) = 1.0D0/2.0D0
      H(1,3) = 1.0D0/3.0D0
      H(2,1) = 1.0D0/2.0D0
      H(2,2) = 1.0D0/3.0D0
      H(2,3) = 1.0D0/4.0D0
      H(3,1) = 1.0D0/3.0D0
      H(3,2) = 1.0D0/4.0D0
      H(3,3) = 1.0D0/5.0D0
```

The statements

```
syms x
z = exp(-exp(-x));
fortran(diff(z,3),'file','fortrantest')
```

return a file named `fortrantest` containing the following:

```
      t7 = exp(-x)
      t8 = exp(-t7)
      t0 = t8*exp(x*(-2))*(-3)+t8*exp(x*(-3))+t7*t8
```

See Also

`ccode` | `latex` | `matlabFunction` | `pretty`

fourier

Fourier transform

Syntax

```
fourier(f,trans_var,eval_point)
```

Description

`fourier(f,trans_var,eval_point)` computes the Fourier transform of f with respect to the transformation variable trans_var at the point eval_point.

Input Arguments

`f`

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

`trans_var`

Symbolic variable representing the transformation variable. This variable is often called the “time variable” or the “space variable”.

Default: The variable `x`. If f does not contain `x`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable, expression, vector or matrix representing the evaluation point. This variable is often called the “frequency variable”.

Default: The variable `w`. If `w` is the transformation variable of f, then the default evaluation point is the variable `v`.

Examples

Compute the Fourier transform of this expression with respect to the variable `x` at the evaluation point `y`:

```
syms x y
f = exp(-x^2);
fourier(f, x, y)

ans =
pi^(1/2)*exp(-y^2/4)
```

Compute the Fourier transform of this expression calling the `fourier` function with one argument. If you do not specify the transformation variable, then `fourier` uses the variable `x`.

```
syms x t y
f = exp(-x^2)*exp(-t^2);
fourier(f, y)

ans =
pi^(1/2)*exp(-t^2)*exp(-y^2/4)
```

If you also do not specify the evaluation point, `fourier` uses the variable `w`:

```
fourier(f)

ans =
pi^(1/2)*exp(-t^2)*exp(-w^2/4)
```

Compute the following Fourier transforms that involve the Dirac and Heaviside functions:

```
syms t w
fourier(t^3, t, w)

ans =
-pi*dirac(3, w)*2*i

syms t0
fourier(heaviside(t - t0), t, w)

ans =
exp(-t0*w*i)*(pi*dirac(w) - i/w)
```

If `fourier` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(t) w
F = fourier(f, t, w)
```

```
F =
fourier(f(t), t, w)
```

`ifourier` returns the original expression:

```
ifourier(F, w, t)
```

```
ans =
f(t)
```

For further computations, remove the assumption on variable `x`:

```
syms x clear
```

The Fourier transform of a function is related to the Fourier transform of its derivative:

```
syms f(t) w
fourier(diff(f(t), t), t, w)
```

```
ans =
w*fourier(f(t), t, w)*i
```

Find the fourier transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
fourier([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])
```

```
ans =
[               2*pi*exp(x)*dirac(a),     2*pi*dirac(b)]
[ -pi*(dirac(c - 1) - dirac(c + 1))*i, -2*pi*dirac(1, d)]
```

When the input arguments are nonscalars, `fourier` acts on them element-wise. If `fourier` is called with both scalar and nonscalar arguments, then `fourier` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
fourier(x,[x, w; y, z],[a, b; c, d])
```

```
ans =
[ pi*dirac(1, a)*2*i, 2*pi*x*dirac(b)]
[    2*pi*x*dirac(c), 2*pi*x*dirac(d)]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, the second argument must be a scalar.

```
syms f1(x) f2(x) a b
f1(x) = exp(x);
f2(x) = x;
fourier([f1, f2],x,[a, b])
```

```
ans =
[ fourier(exp(x), x, a), pi*dirac(1, b)*2*i]
```

More About

Fourier Transform

The Fourier transform of the expression $f = f(x)$ with respect to the variable x at the point w is defined as follows:

$$F(w) = c\int_{-\infty}^{\infty} f(x)\, e^{iswx}\, dx.$$

Here, c and s are parameters of the Fourier transform. The `fourier` function uses $c = 1$, $s = -1$.

Tips

- If you call `fourier` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If f is a matrix, `fourier` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `fourier` acts element-wise on all components of the matrix.
- To compute the inverse Fourier transform, use `ifourier`.

- "Compute Fourier and Inverse Fourier Transforms" on page 2-183

References

Oberhettinger F., “Tables of Fourier Transforms and Fourier Transforms of Distributions”, Springer, 1990.

See Also

`ifourier` | `ilaplace` | `iztrans` | `laplace` | `ztrans`

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

`round` | `ceil` | `floor` | `fix`

fresnelc

Fresnel cosine integral function

Syntax

```
fresnelc(z)
```

Description

`fresnelc(z)` returns the Fresnel cosine integral 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 + (31*i)/100)]
```

Use `vpa` to approximate results:

```
vpa(y)
```

```
ans =
[ -0.48825340607534075450022350335726, 0.000999999999999975325988997279422003,...
 0.86166573430841730950055370401908 - 0.25236540291386150167658349493972*i]
```

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 a Symbolic Function

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)
```

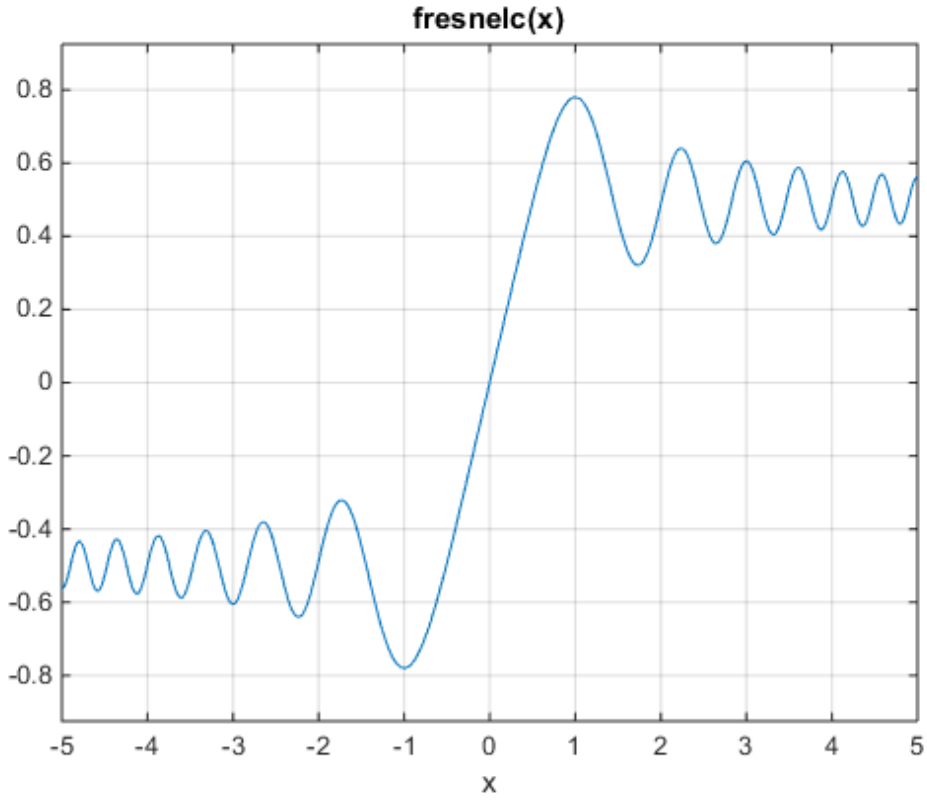
```
ans =
[ fresnelc(sin(x)), fresnelc(2*i), -fresnelc(7)]
ans =
[           0,      fresnelc(2)]
[ fresnelc(i), fresnelc(exp(x))]
```

Plot the Fresnel Cosine Integral Function

Plot the Fresnel cosine integral function from $x = -5$ to $x = 5$:

```
syms x
ezplot(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) - pi^2*x^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 - (pi^2*x^5)/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 | N-D 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 N-D array, or a symbolic variable, expression, vector, matrix, or function.

More About

Fresnel Cosine Integral

The Fresnel cosine integral of z is

$$\text{fresnelc}(z) = \int_0^z \cos\left(\frac{\pi t^2}{2}\right) dt.$$

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`

fresnels

Fresnel sine integral function

Syntax

```
fresnels(z)
```

Description

`fresnels(z)` returns the Fresnel sine integral 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 + (31*i)/100)]
```

Use `vpa` to approximate the results:

```
vpa(y)
```

```
ans =
[ -0.34341567836369824219530081595807, 0.00000000052359877559820659249174920261227,...
 0.76969209233306959998384249252902 + 0.29449530344285433030167256417637*i]
```

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 a Symbolic Function

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)
```

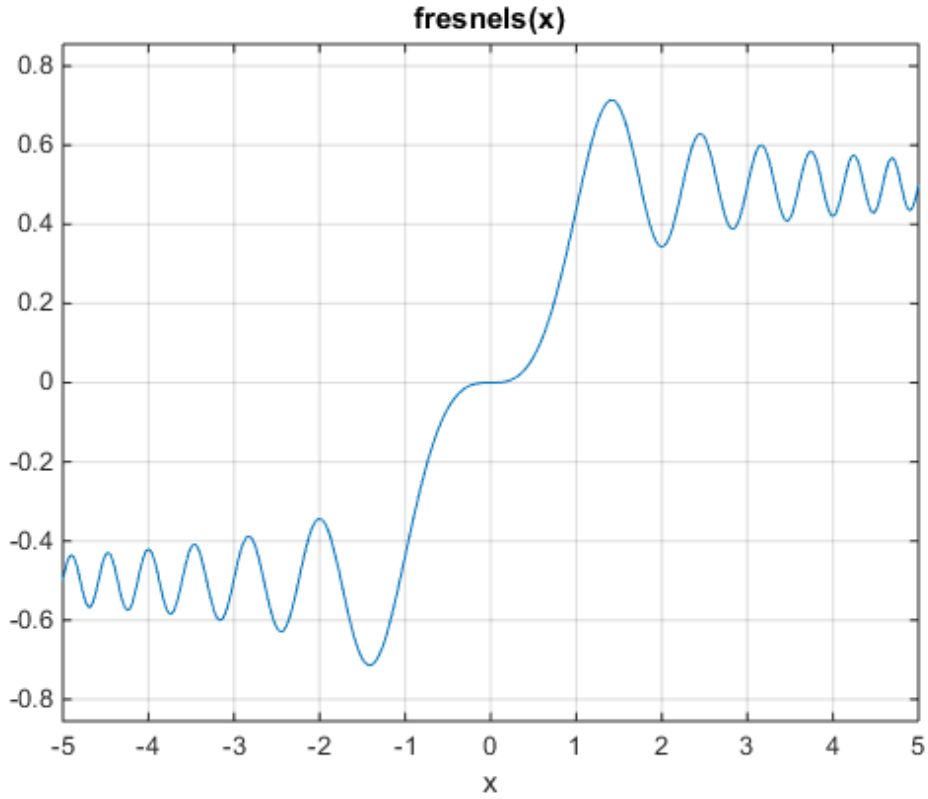
```
ans =
[ fresnels(sin(x)), fresnels(2*i), -fresnels(7)]
ans =
[           0,      fresnels(2)]
[ fresnels(i), fresnels(exp(x))]
```

Fresnel Sine Integral Function

Plot the Fresnel sine integral function from $x = -5$ to $x = 5$.

```
syms x
ezplot(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*pi^2*x*sin((pi*x^2)/2) - pi^3*x^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 | N-D 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 N-D array or as a symbolic variable, expression, vector, matrix, or function.

More About

Fresnel Sine Integral

The Fresnel sine integral of z is

$$\text{fresnels}(z) = \int_0^z \sin\left(\frac{\pi t^2}{2}\right) dt$$

.

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 $z = 0$, $z = \pm\infty$, and $z = \pm i\infty$ which are 0, ±5, and #0.5i. `fresnels(z)` returns symbolic function calls for all other symbolic values of `z`.

See Also

`erf` | `fresnelc`

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 4-512.

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))
```

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 + 0.053465*i, 0.11432 + 0.47579*i, 0.36208 - 0.52925*i]
[ 0.21343 + 0.073771*i,  1.3849 + 0.65649*i, 0.41164 - 0.73026*i]
[  0.26298 - 0.12724*i,  0.51074 - 1.1323*i,   1.2362 + 1.2595*i]
```

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 - 2.2737e-13*i, 1.0 - 9.0949e-13*i, 6.0 + 6.8212e-13*i]
[ 3.0 - 3.4106e-13*i, 5.0 + 4.5475e-13*i, 7.0 - 1.1369e-13*i]
[ 4.0 + 4.5475e-13*i, 9.0 - 4.5475e-13*i, 2.0 + 4.5475e-13*i]
```

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 - 4.7154*i, 0.54204 + 0.5947*i, 0.13764 - 0.80906*i]
[ 0.38028 + 0.033194*i, 0.65189 - 3.8732*i, 0.056763 - 1.0898*i]
[   0.2994 - 0.24756*i, - 0.105 - 1.6513*i,  0.89453 - 3.0309*i]
```

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 + 5.6417e-12*i, 1.0 - 1.5064e-12*i, 6.0 + 8.2423e-13*i]
[ 3.0 - 3.4106e-13*i,  5.0 - 2.558e-13*i, 7.0 - 7.3896e-13*i]
[ 4.0 + 5.6843e-14*i, 9.0 - 9.3081e-13*i, 2.0 - 1.1369e-13*i]
```

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.

More About

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:

$$\mathrm{f}\left(\begin{pmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & \cdots & 0 & \lambda \end{pmatrix}\right) = \begin{pmatrix} \frac{\mathrm{f}(\lambda)}{0!} & \frac{\mathrm{f}'(\lambda)}{1!} & \frac{\mathrm{f}''(\lambda)}{2!} & \cdots & \frac{\mathrm{f}^{(n-1)}(\lambda)}{(n-1)!} \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \frac{\mathrm{f}''(\lambda)}{2!} \\ \vdots & \ddots & \ddots & \ddots & \frac{\mathrm{f}'(\lambda)}{1!} \\ 0 & \cdots & \cdots & 0 & \frac{\mathrm{f}(\lambda)}{0!} \end{pmatrix}$$

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`

funtool

Function calculator

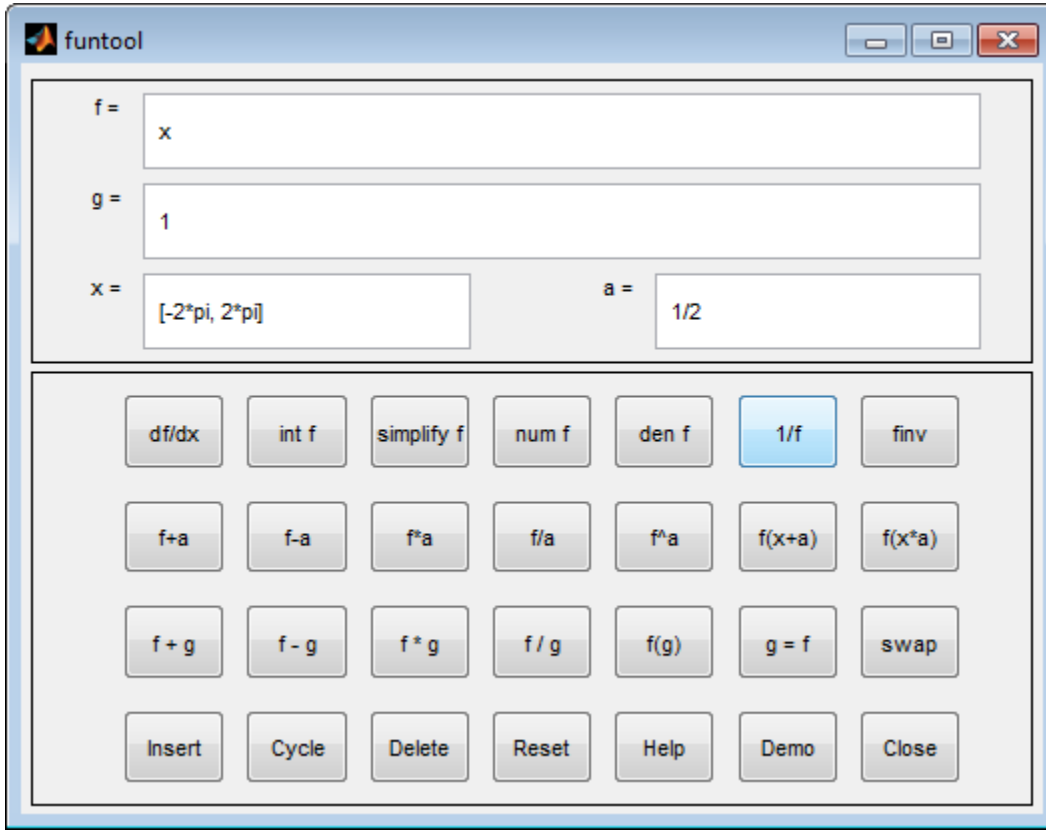
Syntax

```
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

`ezplot | syms`

gamma

Gamma function

Syntax

```
gamma(X)
```

Description

`gamma(X)` returns the gamma function 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]
```

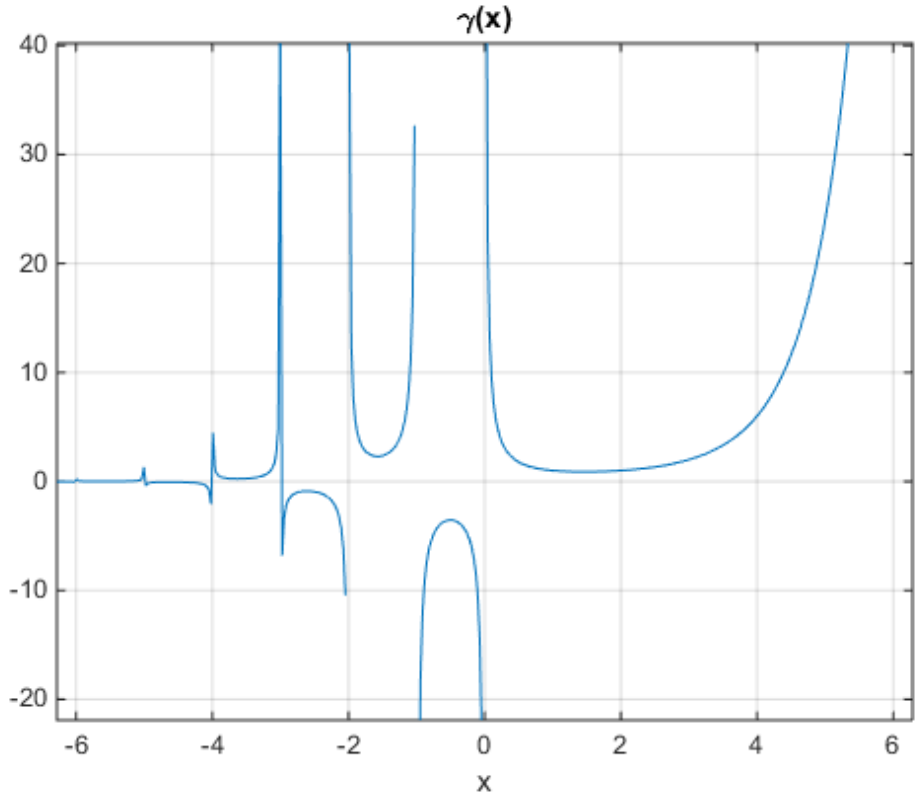
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.24658411512650858900694446388517,...
2.65927187288003053998988105057 38,...
-3.54490770181103205459633496668 23,...
2.67893853470774763365569294097 47,...
1.0, 6.0]
```

Plot the Gamma Function

Plot the gamma function and add grid lines.

```
syms x
ezplot(gamma(x)); grid on
```



Handle Expressions Containing the 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.

More About

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` | `nchoosek` | `pochhammer` | `psi`

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 =
[ 1.524063822430784524881056493926 3,...
0.572364942924700087071713675676 53,...
0.303150275147523568675862817372 01,...
-0.066740877459477468649396334098109,...
0.693147180559945309417232121458 18]
```

Definition of the Logarithmic Gamma Function on the 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(i), gammaln(- 1 + 2*i), gammaln(-2/3), gammaln(-10/3)]
```

Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(complex)
```

```
ans =
[ - 0.650923199301856338885216831503 95 - 1.87243664726242981711885334943 66*i,...
- 3.373944923207924837947607366472 5 - 3.475593946280811043293192158355 8*i,...
1.390885755035931451165187152442 3 - 3.141592653589793238462643383279 5*i,...
- 0.937190173349287273700964675981 78 - 12.566370614359172953850573533118*i]
```

Compute the logarithmic gamma function of negative infinity:

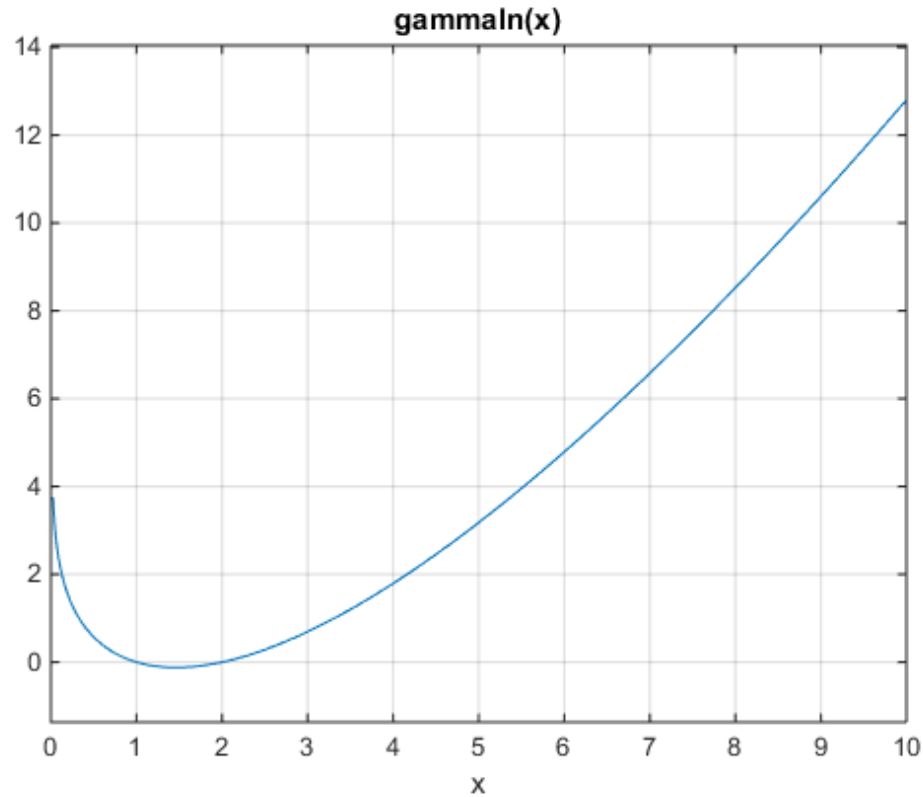
```
gammaln(sym(-Inf))
```

```
ans =
NaN
```

Plot the Logarithmic Gamma Function

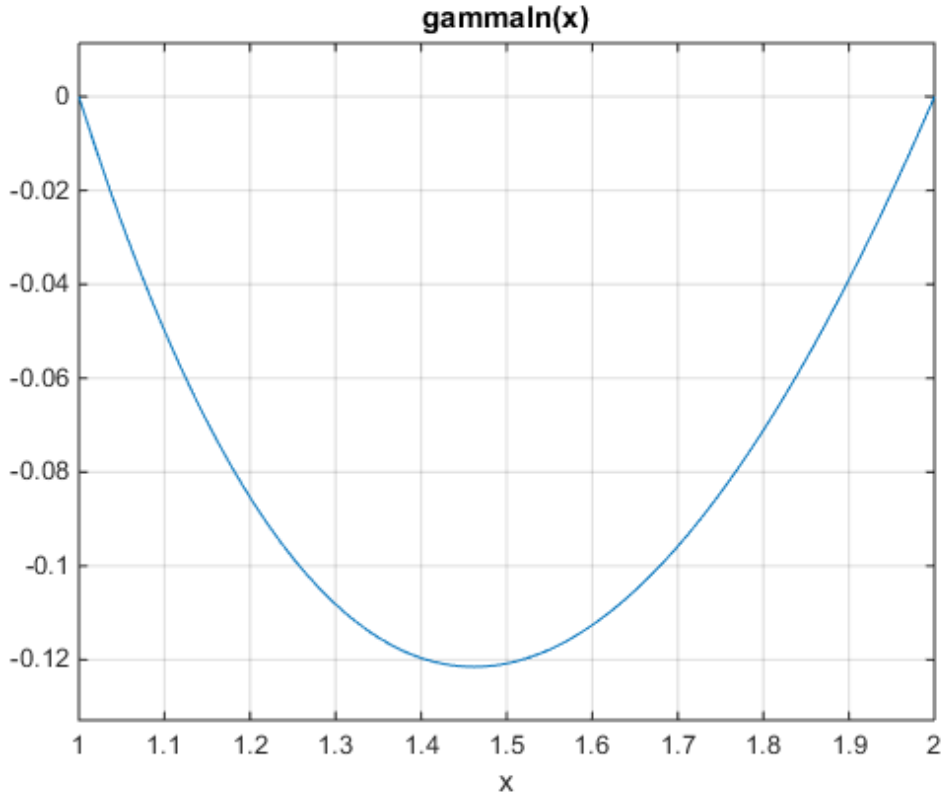
Plot the logarithmic gamma function on the interval from 0 to 10.

```
syms x
ezplot(gammaln(x), 0, 10)
grid on
```



To see the negative values better, plot the same function on the interval from 1 to 2.

```
ezplot(gammaln(x), 1, 2)
grid on
```



Handle Expressions Containing the 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 =
pi*i + log(2)
```

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.

See Also

`beta` | `gamma` | `log` | `nchoosek` | `psi`

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 $a_1, a_2, \ldots$ is a number g, such that $g/a_1, g/a_2, \ldots$ 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 + 10*i
```

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.

```
(30*C*4) + (56*D*4) == G*4

ans =
     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.

More About

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`

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
```

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)`.

More About

Tips

- If A and B are both numbers, then `A >= B` compares A and B and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A >= B` returns a symbolic greater than or equal to relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.

- If both A and B are arrays, then these arrays must have the same dimensions. `A >= B` returns an array of relations `A(i,j,...)>=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if A is a variable (for example, `x`), and B is an *m*-by-*n* matrix, then A is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x >= i` becomes `x >= 0`, and `x >= 3 + 2*i` becomes `x >= 3`.

- "Set Assumptions" on page 1-32

See Also

`eq` | `gt` | `isAlways` | `le` | `logical` | `lt` | `ne`

gegenbauerC

Gegenbauer polynomials

Syntax

```
gegenbauerC(n,a,x)
```

Description

`gegenbauerC(n,a,x)` represents the `n`th-degree Gegenbauer (ultraspherical) polynomial 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 =
-8.017872638023574152120885203729 1e35
```

Approximate the polynomial coefficients by using `vpa`, and then substitute `x = sym(1/3)` into the result. This approach is also numerically unstable.

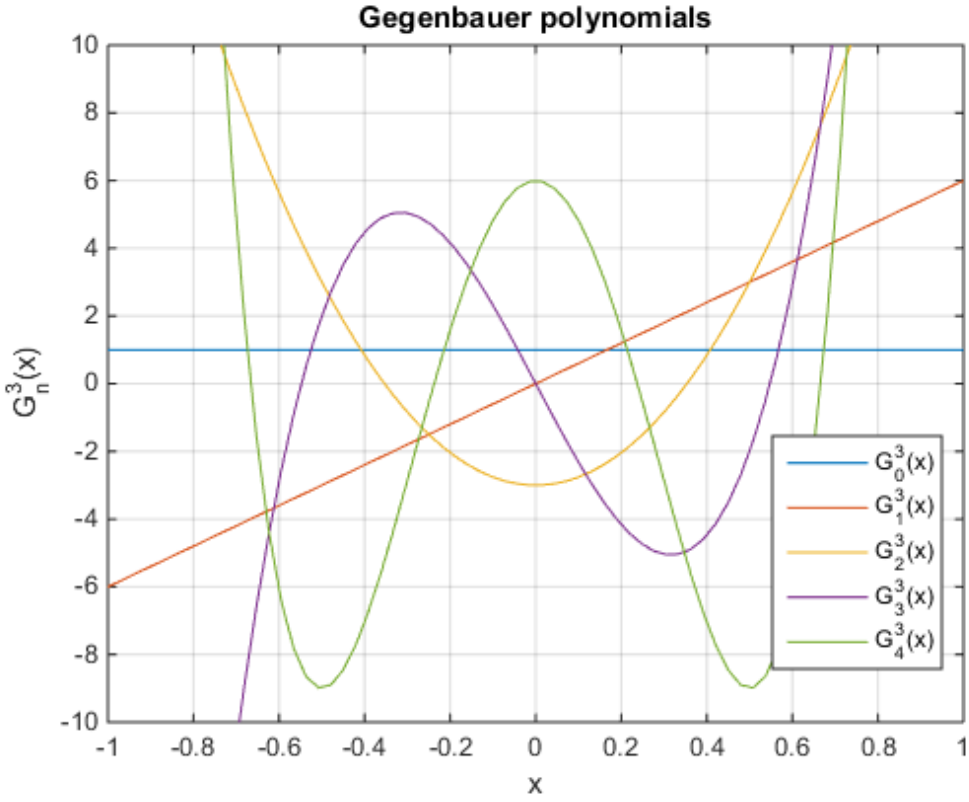
```
subs(vpa(C500), x, sym(1/3))
```

```
ans =
-8.112541240585847024688721392316 7e36
```

Plot Gegenbauer Polynomials

Plot the first five Gegenbauer polynomials for the parameter `a = 3`.

```
syms x y
for n = [0, 1, 2, 3, 4]
  ezplot(gegenbauerC(n,3,x))
  hold on
end

hold off

axis([-1, 1, -10, 10])
grid on
ylabel('G_n^3(x)')

legend('G_0^3(x)', 'G_1^3(x)', 'G_2^3(x)', 'G_3^3(x)', 'G_4^3(x)',...
                                                'Location', 'Best')
title('Gegenbauer 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.

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.

More About

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`

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 a 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 3-24

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. 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`

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 the Gradient of a 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 the Gradient of a Function

Find the gradient of a function `f(x, y)`, and plot it as a quiver (velocity) plot.

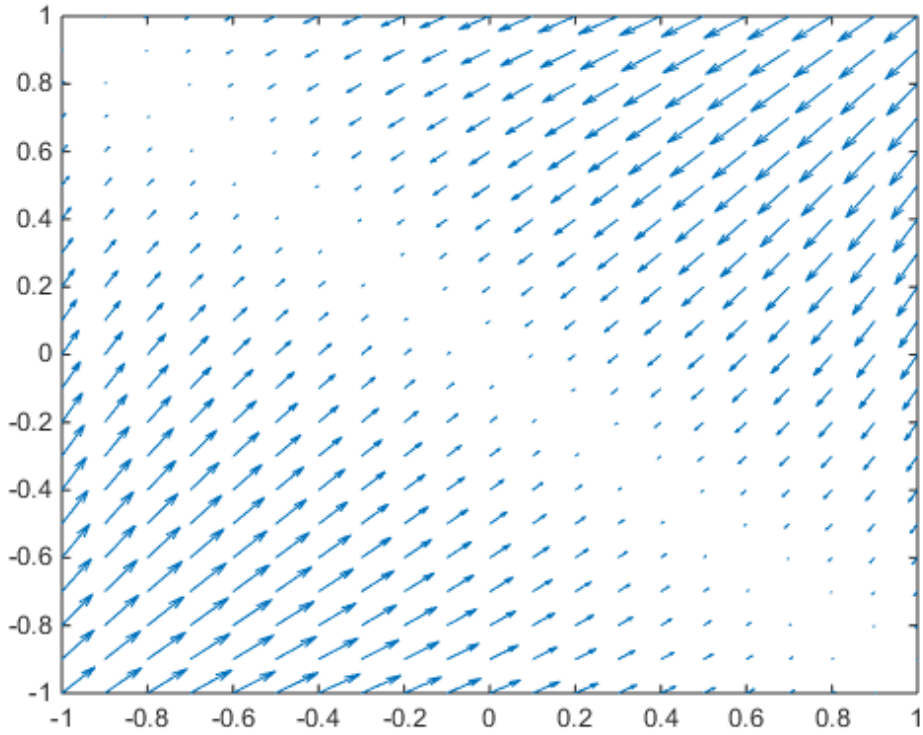
Find the gradient vector of `f(x, y)` with respect to vector `[x, y]`. The gradient is vector `g` with these components.

```
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.

More About

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`

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
```

More About

Tips

- If A and B are both numbers, then A > B compares A and B and returns logical 1 (true) or logical 0 (false). Otherwise, A > B returns a symbolic greater than relation. You can use that relation as an argument for such functions as assume, assumeAlso, and subs.
- If both A and B are arrays, then these arrays must have the same dimensions. A > B returns an array of relations A(i,j,...)>B(i,j,...)
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if A is a variable (for example, x), and B is an m-by-n matrix, then A is expanded into m-by-n matrix of elements, each set to x.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, x > i becomes x > 0, and x > 3 + 2*i becomes x > 3.

- "Set Assumptions" on page 1-32

See Also

`eq` | `ge` | `isAlways` | `le` | `logical` | `lt` | `ne`

harmonic

Harmonic function (harmonic number)

Syntax

```
harmonic(x)
```

Description

`harmonic(x)` returns the harmonic function 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(i), 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, 0.671865985524009837878390572804 31...
 + 1.076674047468581174134050794 75*i,...
 2.154522544221385878269433675 1358]
```

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 =
7.486469861549345911657517205 3329
```

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 the Symbolic Function

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

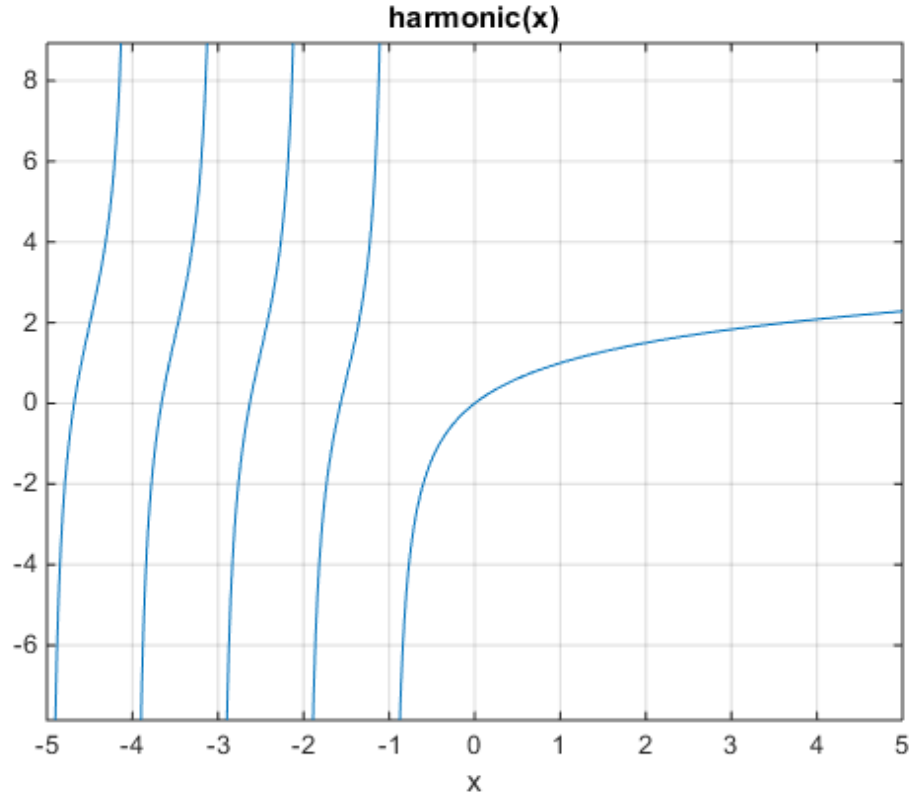
Find the harmonic function for elements of vector `V` and matrix `M`.

```
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(3*i)]
ans =
[ harmonic(exp(x*i)), 3/2]
[                Inf, Inf]
```

Plot Harmonic Function

Plot the harmonic function from `x` = -5 to `x` = 5.

```
syms x
ezplot(harmonic(x),[-5,5]), grid on
```



Differentiate and Find the Limit of the 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 the 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 the 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 | N-D array | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix | symbolic N-D array

Input, specified as number, vector, matrix, or as an N-D array or symbolic variable, expression, function, vector, matrix, or N-D array.

More About

Harmonic Function

The harmonic function for x is defined as

$$\text{harmonic}(x) = \sum_{k=1}^{x} \frac{1}{k}$$

It is also defined as

$$\text{harmonic}(x) = \Psi(x) + \gamma$$

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 $\text{harmonic}(x+1) = \text{harmonic}(x) + \frac{1}{x}$ to obtain a result with an argument x from the interval [0, 1].

`expand` expands `harmonic` using the equations $\text{harmonic}(x+1) = \text{harmonic}(x) + \frac{1}{x}$, $\text{harmonic}(-x) = \text{harmonic}(x) - \frac{1}{x} + \pi\cot(\pi x)$, and the Gauss multiplication formula for `harmonic(kx)`, where k is an integer.

`harmonic` implements the following explicit formulae:

$$\text{harmonic}\left(\frac{-1}{2}\right) = -2\ln(2)$$

$$\text{harmonic}\left(-\frac{2}{3}\right) = -\frac{3}{2}\ln(3) - \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(-\frac{1}{3}\right) = -\frac{3}{2}\ln(3) + \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{-3}{4}\right) = -3\ln(2) - \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{-1}{4}\right) = -3\ln(2) + \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{-5}{6}\right) = -2\ln(2) - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}\left(\frac{-1}{6}\right) = -2\ln(2) - \frac{3}{2}\ln(3) + \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}(0) = 0$$

$$\text{harmonic}\left(\frac{1}{2}\right) = 2 - 2\ln(2)$$

$$\text{harmonic}\left(\frac{1}{3}\right) = 3 - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{2}{3}\right) = \frac{3}{2} - \frac{3}{2}\ln(3) + \frac{\sqrt{3}}{6}\pi$$

$$\text{harmonic}\left(\frac{1}{4}\right) = 4 - 3\ln(2) - \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{3}{4}\right) = \frac{4}{3} - 3\ln(2) + \frac{\pi}{2}$$

$$\text{harmonic}\left(\frac{1}{6}\right) = 6 - 2\ln(2) - \frac{3}{2}\ln(3) - \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}\left(\frac{5}{6}\right) = \frac{6}{5} - 2\ln(2) - \frac{3}{2}\ln(3) + \frac{\sqrt{3}}{2}\pi$$

$$\text{harmonic}(1) = 1$$

$$\text{harmonic}(\infty) = \infty$$

$$\text{harmonic}(-\infty) = NaN$$

See Also

`beta` | `factorial` | `gamma` | `gammaln` | `nchoosek` | `zeta`

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 the 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
```

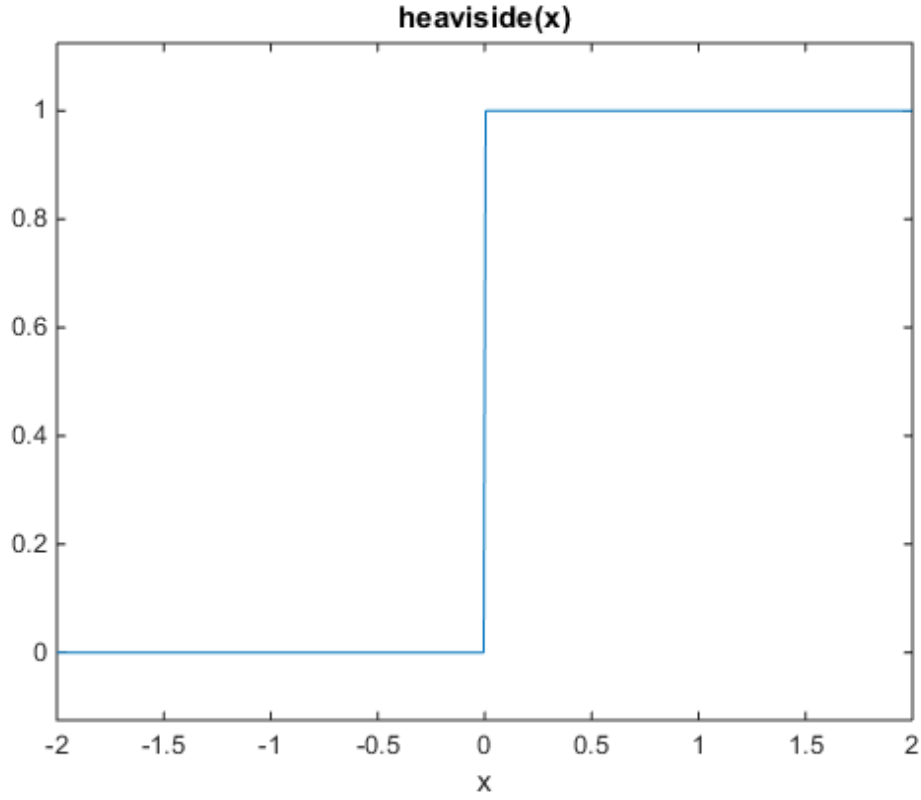
For further computations, clear the assumptions:

```
syms x clear
```

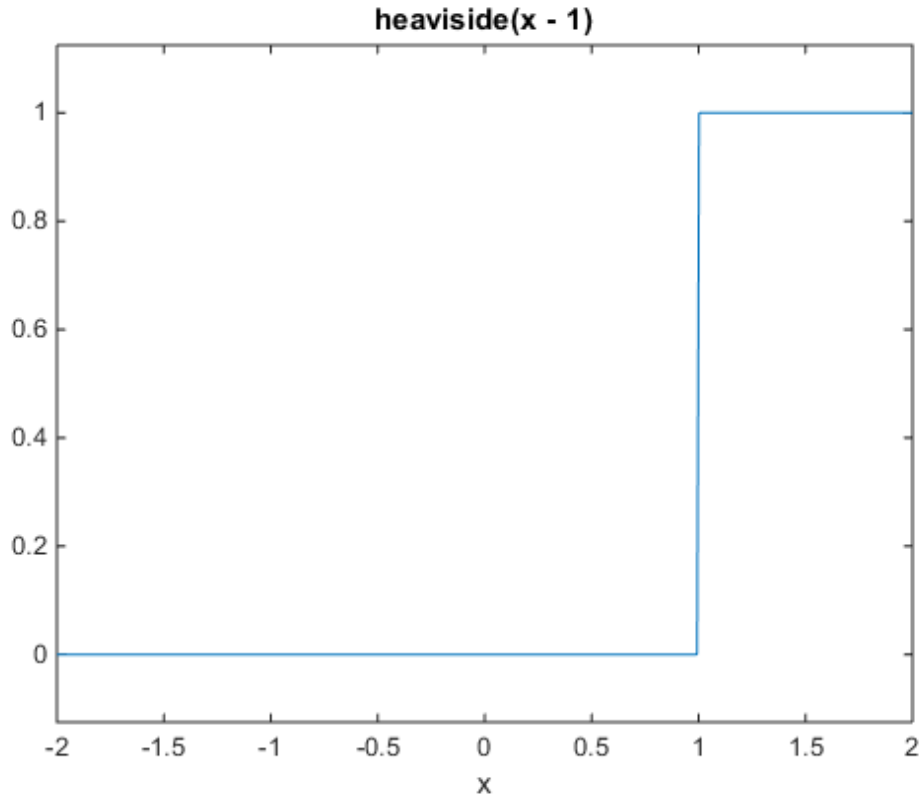
Plot the Heaviside Function

Plot the Heaviside step function for `x` and `x - 1`.

```
syms x
ezplot(heaviside(x), [-2, 2])
```




```
ezplot(heaviside(x - 1), [-2, 2])
```



Evaluate the Heaviside Function for a 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 the 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
```

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`

hermiteH

Hermite polynomials

Syntax

```
hermiteH(n,x)
```

Description

`hermiteH(n,x)` represents the nth-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.

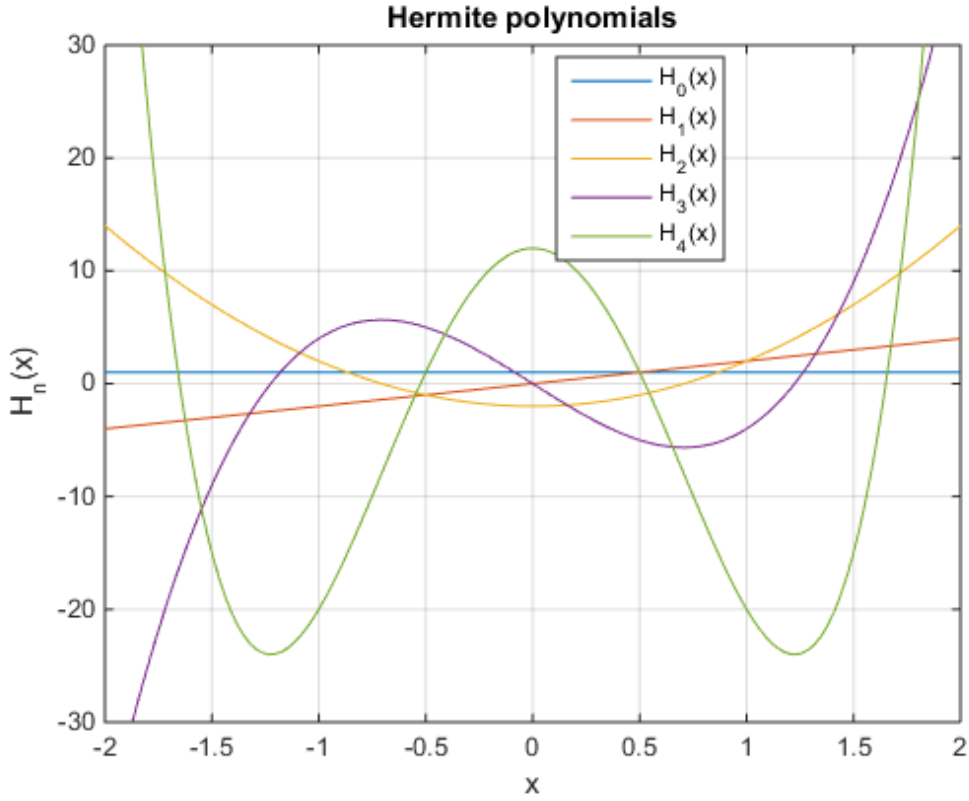
```
hermiteH(5, sym([1/6, 1/3, 1/2, 2/3, 3/4]))
```

```
ans =
[ 4681/243, 8312/243, 41, 8944/243, 963/32]
```

Plot Hermite Polynomials

Plot the first five Hermite polynomials.

```
syms x y
for n = [0, 1, 2, 3, 4]
  ezplot(hermiteH(n,x))
  hold on
end

hold off

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.

More About

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`

hessian

Hessian matrix of scalar function

Syntax

```
hessian(f,v)
```

Description

`hessian(f,v)` finds the Hessian matrix 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 a 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.

More About

Hessian Matrix

The Hessian matrix of $f(x)$ is the square matrix of the second partial derivatives of $f(x)$.

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

See Also

curl | diff | divergence | gradient | jacobian | laplacian | potential | vectorPotential

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`

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`

hypergeom

Hypergeometric function

Syntax

```
hypergeom(a,b,z)
```

Description

`hypergeom(a,b,z)` represents the generalized hypergeometric function.

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], 3*i)
```

Use vpa to approximate symbolic results with the required number of digits:

```
vpa(symA,10)

ans =
   - 1.21741893 - 0.8330405509*i
                     1.209063189
 - 0.2027516975 + 0.2405013423*i
```

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 canceling identical parameters, the list of upper parameters contains 0, the resulting hypergeometric function is constant with the value 1:

```
hypergeom([0, 0, 2, 3], [a, 0, 4], x)

ans =
1
```

If, after canceling identical parameters, 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.

More About

Generalized Hypergeometric Function

The generalized hypergeometric function of order p, q is defined as follows:

$$ {}_pF_q\left(a;b;z\right) = \sum_{k=0}^{\infty}\left(\frac{\left(a_1\right)_k\left(a_2\right)_k\ldots\left(a_p\right)_k}{\left(b_1\right)_k\left(b_2\right)_k\ldots\left(b_q\right)_k}\right)\left(\frac{z^k}{k!}\right) $$

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:

$$ {}_0F_q\left(;b;z\right) = \sum_{k=0}^{\infty} \frac{1}{\left(b_1\right)_k \left(b_2\right)_k \ldots \left(b_q\right)_k} \left(\frac{z^k}{k!}\right) $$

$$ {}_pF_0\left(a;;z\right) = \sum_{k=0}^{\infty} \left(a_1\right)_k \left(a_2\right)_k \ldots \left(a_p\right)_k \left(\frac{z^k}{k!}\right) $$

$$ {}_0F_0\left(;;z\right) = \sum_{k=0}^{\infty} \left(\frac{z^k}{k!}\right) = e^z $$

Pochhammer Symbol

The Pochhammer symbol is defined as follows:

$$ \left(x\right)_n = \frac{\Gamma\left(x+n\right)}{\Gamma\left(x\right)} $$

If n is a positive integer, then $(x)_n = x(x + 1)...(x + n - 1)$.

Tips

- For most exact arguments, the hypergeometric function returns a symbolic function call. If an upper parameter coincides with a lower parameter, these values cancel and are removed from the parameter lists.
- If, after cancellation of identical parameters, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, then ${}_pF_q(a;b;z)$ is a polynomial in z.
- The following special values are implemented:
  - ${}_pF_p(a;a;z) = {}_0F_0(;;z) = e^z$.
  - ${}_pF_q(a;b;z) = 1$ if the list of upper parameters a contains more `0`s than the list of lower parameters b.
  - ${}_pF_q(a;b;0) = 1$.

Algorithms

The series

$$ {}_pF_q\left(a;b;z\right) = \sum_{k=0}^{\infty}\left(\frac{\left(a_1\right)_k\left(a_2\right)_k\ldots\left(a_p\right)_k}{\left(b_1\right)_k\left(b_2\right)_k\ldots\left(b_q\right)_k}\right)\left(\frac{z^k}{k!}\right) $$

- Converges for any $|z| < \infty$ if $p \le q$.
- Converges for $|z| < 1$ if $p = q + 1$. For $|z| >= 1$, the series diverges, and ${}_pF_q$ is defined by analytic continuation.
- Diverges for any $z <> 0$ if $p > q + 1$. The series defines an asymptotic expansion of ${}_pF_q(a;b;z)$ around $z = 0$. The positive real axis is the branch cut.

If one of the parameters in a is equal to **0** or a negative integer, then the series terminates, turning into what is called a hypergeometric polynomial.

${}_pF_q(a;b;z)$ is symmetric with respect to the parameters, that is, it does not depend on the order of the arrangement a_1, a_2, ... in a or b_1, b_2, ... in b.

If at least one upper parameter equals $n = \mathbf{0}, \mathbf{-1}, \mathbf{-2}, \ldots$, the function turns into a polynomial of degree n. If the previous condition for the lower parameters b is relaxed, and there is some lower parameter equal to $m = \mathbf{0}, \mathbf{-1}, \mathbf{-2}, \ldots$, then the function ${}_pF_q(a;b;z)$ also reduces to a polynomial in z provided $n > m$. It is undefined if $m > n$ or if no upper parameter is a nonpositive integer (resulting in division by **0** in one of the series coefficients). The case $m = n$ is handled by the following rule.

. If for r values of the upper parameters, there are r values of the lower parameters equal to them (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 the function ${}_pF_q(a;b;z)$ 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 of the c_i is zero or a negative integer. For details, see Luke, Y.L. "The Special Functions and Their Approximations", vol. 1, p. 42.

$U(z) = {}_pF_q\left(a;b;z\right)$ satisfies a differential equation in z:

$$ \left[\delta\left(\delta+b-1\right)-z(\delta+a)\right]U\left(z\right)=0, \quad \delta = z\frac{\partial}{\partial z}, $$

where $(\delta + a)$ and $(\delta + b)$ stand for

$$\prod_{i=1}^{p}(\delta + a_i)$$

and

$$\prod_{j=1}^{q}(\delta + b_j),$$

respectively. 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, thus explaining the convergence properties of the hypergeometric series.

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` | `whittakerM` | `whittakerW`

ifourier

Inverse Fourier transform

Syntax

```
ifourier(F,trans_var,eval_point)
```

Description

`ifourier(F,trans_var,eval_point)` computes the inverse Fourier transform of F with respect to the transformation variable trans_var at the point eval_point.

Input Arguments

`F`

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

`trans_var`

Symbolic variable representing the transformation variable. This variable is often called the “frequency variable”.

Default: The variable `w`. If F does not contain `w`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable or expression representing the evaluation point. This variable is often called the “time variable” or the “space variable”.

Default: The variable `x`. If `x` is the transformation variable of F, then the default evaluation point is the variable `t`.

Examples

Compute the inverse Fourier transform of this expression with respect to the variable `y` at the evaluation point `x`:

```
syms x y
F = sqrt(sym(pi))*exp(-y^2/4);
ifourier(F, y, x)

ans =
exp(-x^2)
```

Compute the inverse Fourier transform of this expression calling the `ifourier` function with one argument. If you do not specify the transformation variable, `ifourier` uses the variable `w`:

```
syms a w t real
F = exp(-w^2/(4*a^2));
ifourier(F, t)

ans =
exp(-a^2*t^2)/(2*pi^(1/2)*(1/(4*a^2))^(1/2))
```

If you also do not specify the evaluation point, `ifourier` uses the variable `x`:

```
ifourier(F)

ans =
exp(-a^2*x^2)/(2*pi^(1/2)*(1/(4*a^2))^(1/2))
```

For further computations, remove the assumptions:

```
syms a w t clear
```

Compute the following inverse Fourier transforms that involve the Dirac and Heaviside functions:

```
syms t w
ifourier(dirac(w), w, t)

ans =
1/(2*pi)

ifourier(2*exp(-abs(w)) - 1, w, t)
```

```
ans =
-(2*pi*dirac(t) - 4/(t^2 + 1))/(2*pi)

ifourier(1/(w^2 + 1), w, t)

ans =
(pi*exp(-t)*heaviside(t) + pi*heaviside(-t)*exp(t))/(2*pi)
```

If `ifourier` cannot find an explicit representation of the transform, it returns results in terms of the direct Fourier transform:

```
syms F(w) t
f = ifourier(F, w, t)

f =
fourier(F(w), w, -t)/(2*pi)
```

Find the inverse fourier transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
ifourier([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])

ans =
[                         exp(x)*dirac(a),    dirac(b)]
[ (dirac(c - 1)*i)/2 - (dirac(c + 1)*i)/2, dirac(1, d)]
```

When the input arguments are nonscalars, `ifourier` acts on them element-wise. If `ifourier` is called with both scalar and nonscalar arguments, then `ifourier` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
ifourier(x,[x, w; y, z],[a, b; c, d])

ans =
[ -dirac(1, a)*i, x*dirac(b)]
[     x*dirac(c), x*dirac(d)]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, the second argument must be a scalar.

```
syms f1(x) f2(x) a b
```

```
f1(x) = exp(x);
f2(x) = x;
ifourier([f1, f2],x,[a, b])

ans =
[ fourier(exp(x), x, -a)/(2*pi), -dirac(1, b)*i]
```

More About

Inverse Fourier Transform

The inverse Fourier transform of the expression $F = F(w)$ with respect to the variable w at the point x is defined as follows:

$$f(x) = \frac{|s|}{2\pi c}\int_{-\infty}^{\infty} F(w)e^{-iswx}dw.$$

Here, c and s are parameters of the inverse Fourier transform. The `ifourier` function uses $c = 1$, $s = -1$.

Tips

- If you call `ifourier` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If F is a matrix, `ifourier` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `ifourier` acts element-wise on all components of the matrix.
- The toolbox computes the inverse Fourier transform via the direct Fourier transform:

  $$ifourier(F,w,t) = \frac{1}{2\pi}fourier(F,w,-t)$$

  If `ifourier` cannot find an explicit representation of the inverse Fourier transform, it returns results in terms of the direct Fourier transform.
- To compute the direct Fourier transform, use `fourier`.
- "Compute Fourier and Inverse Fourier Transforms" on page 2-183

References

Oberhettinger, F. “Tables of Fourier Transforms and Fourier Transforms of Distributions”, Springer, 1990.

See Also

`fourier` | `ilaplace` | `iztrans` | `laplace` | `ztrans`

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. See "Upper Incomplete Gamma Function" on page 4-592. The MATLAB `gammainc` function uses the definition of the lower incomplete gamma function, `gammainc(z, nu) = 1 - igamma(nu, z)/gamma(nu)`. The order of input arguments differs between these functions.

Examples

Compute the 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 the 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.

More About

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`

ilaplace

Inverse Laplace transform

Syntax

```
ilaplace(F,trans_var,eval_point)
```

Description

`ilaplace(F,trans_var,eval_point)` computes the inverse Laplace transform of F with respect to the transformation variable trans_var at the point eval_point.

Input Arguments

`F`

Symbolic expression or function, vector or matrix of symbolic expressions or functions.

`trans_var`

Symbolic variable representing the transformation variable. This variable is often called the “complex frequency variable”.

Default: The variable `s`. If F does not contain `s`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable or expression representing the evaluation point. This variable is often called the “time variable”.

Default: The variable `t`. If `t` is the transformation variable of F, then the default evaluation point is the variable `x`.

Examples

Compute the inverse Laplace transform of this expression with respect to the variable `y` at the evaluation point `x`:

```
syms x y
F = 1/y^2;
ilaplace(F, y, x)

ans =
x
```

Compute the inverse Laplace transform of this expression calling the `ilaplace` function with one argument. If you do not specify the transformation variable, `ilaplace` uses the variable `s`.

```
syms a s x
F = 1/(s - a)^2;
ilaplace(F, x)

ans =
x*exp(a*x)
```

If you also do not specify the evaluation point, `ilaplace` uses the variable `t`:

```
ilaplace(F)

ans =
t*exp(a*t)
```

Compute the following inverse Laplace transforms that involve the Dirac and Heaviside functions:

```
syms s t
ilaplace(1, s, t)

ans =
dirac(t)

ilaplace(exp(-2*s)/(s^2 + 1) + s/(s^3 + 1), s, t)

ans =
heaviside(t - 2)*sin(t - 2) - exp(-t)/3 +...
(exp(t/2)*(cos((3^(1/2)*t)/2) + 3^(1/2)*sin((3^(1/2)*t)/2)))/3
```

If `ilaplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms F(s) t
f = ilaplace(F, s, t)
```

```
f =
ilaplace(F(s), s, t)
```

`laplace` returns the original expression:

```
laplace(f, t, s)
```

```
ans =
F(s)
```

Find the inverse Laplace transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
ilaplace([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])
```

```
ans =
[        exp(x)*dirac(a),      dirac(b)]
[ ilaplace(sin(y), y, c), dirac(1, d)*i]
```

When the input arguments are nonscalars, `ilaplace` acts on them element-wise. If `ilaplace` is called with both scalar and nonscalar arguments, then `ilaplace` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
ilaplace(x,[x, w; y, z],[a, b; c, d])
```

```
ans =
[ dirac(1, a), x*dirac(b)]
[  x*dirac(c), x*dirac(d)]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, 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)]
```

More About

Inverse Laplace Transform

The inverse Laplace transform is defined by a contour integral in the complex plane:

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} F(s) e^{st} ds.$$

Here, c is a suitable complex number.

Tips

- If you call `ilaplace` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If F is a matrix, `ilaplace` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `ilaplace` acts element-wise on all components of the matrix.
- To compute the direct Laplace transform, use `laplace`.
- "Compute Laplace and Inverse Laplace Transforms" on page 2-190

See Also

`fourier` | `ifourier` | `iztrans` | `laplace` | `ztrans`

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(sym('2*exp(1 + i)'))

ans =
2*exp(1)*sin(1)
```

In general, `imag` cannot extract the entire imaginary parts from symbolic expressions containing variables. However, `imag` can rewrite and sometimes simplify the input expression:

```
syms a x y
imag(a + 2)
imag(x + y*i)

ans =
imag(a)

ans =
imag(x) + real(y)
```

If you assign numeric values to these variables or if you specify that these variables are real, `imag` can extract the imaginary part of the expression:

```
syms a
 a = 5 + 3*i;
imag(a + 2)

ans =
     3

syms x y real
imag(x + y*i)

ans =
y
```

Clear the assumption that `x` and `y` are real:

```
syms x y clear
```

Find the imaginary parts of the elements of matrix A:

```
A = sym('[-1 + i, sinh(x); exp(10 + 7*i), exp(pi*i)]');
imag(A)
```

```
ans =
[              1, imag(sinh(x))]
[ exp(10)*sin(7),             0]
```

Alternatives

You can compute the imaginary part of `z` via the conjugate: `imag(z)= (z - conj(z))/2i`.

More About

Tips

- Calling `imag` for a number that is not a symbolic object invokes the MATLAB `imag` function.

See Also

`conj` | `in` | `real` | `sign` | `signIm`

in

Numeric type of symbolic input

Syntax

```
in(x,type)
```

Description

`in(x,type)` returns logical `1` (`true`) if the symbolic input x is of the specified type, and logical `0` (`false`) if it is not of the specified type. When `in(x,type)` appears in output, it expresses the logical condition that x is of the specified type.

Examples

Test if Numeric Input Is an Integer, Real, or Rational

Test if `5` is an integer, real, or rational. Since `in` only accepts symbolic inputs, convert `5` to a symbolic object using `sym`.

```
in(sym(5),'integer')
in(sym(5),'real')
in(sym(5),'rational')
```

```
ans =
     1
ans =
     1
ans =
     1
```

`in` returns logical `1` (`true`) for all types, which means `5` is an integer, real, and rational.

Test if `1i` is integer, real, or rational.

```
in(sym(1i),'integer')
```

```
in(sym(1i),'real')
in(sym(1i),'rational')
```

```
ans =
     0
ans =
     0
ans =
     0
```

`in` returns logical `0` (`false`) for all types, which means `1i` is not an integer, real, or rational.

Test if the symbolic variable `x` is an integer, real, or rational.

```
syms x
in(x,'integer')
in(x,'real')
in(x,'rational')
```

```
ans =
in(x, 'integer')
ans =
in(x, 'real')
ans =
in(x, 'rational')
```

`in` cannot decide if x is of the specified types, and returns the function call.

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: Cannot prove 'in(x, 'real')'.
  In sym.isAlways at 38
ans =
     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 =
     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.

```
[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.

```
assume(conds)
solve(solx>0, solx<2*pi, params)
```

```
ans =
1
```

Under the assumption, `solve` returns only integer values of `k`.

Clear the assumption on `k` to use it in further computations.

```
sym(params, 'clear')
```

Test if the Floating-Point Number is an Integer

Test if the floating-point numbers `-1.0`, `5.3`, and `Inf` are integers. Since `in` only accepts symbolic inputs, convert the numbers to symbolic objects using `sym`.

```
in(sym(-1.0),'integer')
in(sym(5.3),'integer')
in(sym(Inf),'integer')
```

```
ans =
     1
ans =
     0
ans =
in(Inf, 'integer')
```

`in` returns logical `1` (`true`) meaning `-1.0` is an integer and logical `0` (`false`) meaning `5.3` is not an integer. The `in` function cannot decide if `Inf` is an integer and therefore returns the function call. To return a logical output, use `isAlways`.

```
isAlways(in(sym(Inf),'integer'))
```

```
ans =
     0
```

Test if the Symbolic Expression Is an Integer, Real, or Rational

Test if the expression `expr1` is integer or real.

```
expr1 = sym(5 + 8i);
in(expr1,'integer')
in(expr1,'real')
```

```
ans =
     0
ans =
     0
```

`in` returns logical `0` (`false`) for both types, which means `expr1` is not real or rational.

Test if expression `expr2` is rational.

```
syms x y
expr2 = y*sin(x);
in(expr2,'rational')
```

```
ans =
in(y*sin(x), 'rational')
```

`in` returns the function call because it cannot decide if `expr2` is rational.

Test If the Symbolic Function Is Real

Test if the symbolic function `f` is real.

```
syms f(x)
f(x) = 50*x;
y = in(f,'real')
```

```
y(x) =
in(50*x, 'real')
```

`in` returns the function call because it cannot decide if `f` is real.

Test if Elements of the 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,       i,    x]
[ sin(y),     3*x,    0]
[    Inf, 3^(1/2), 22/7]
```

Test if the elements of `M` are rational. The `in` function acts on `M` element by element.

```
in(M,'rational')
```

```
ans =
[  in(61/50, 'rational'),       in(i, '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')]
```

Though `in` can decide some elements of `M`, `in` returns the function call because the matrix contains other elements that cannot be decided. The `in` function cannot partially evaluate a matrix. To force a logical evaluation of all elements and return a logical output, use `isalways`. Note that `isalways` returns logical `0` (`false`) for statements that cannot be decided and issues a warning for those statements.

```
isAlways(in(M,'rational'))
```

```
Warning: Cannot prove 'in(sin(y), 'rational')'.
Warning: Cannot prove 'in(3*x, 'rational')'.
Warning: Cannot prove 'in(x, 'rational')'.
ans =
     1     0     0
     0     0     1
     0     0     1
```

Input Arguments

x — Symbolic input

symbolic number | symbolic vector | symbolic matrix | symbolic multidimensional array | symbolic expression | symbolic function

Symbolic input, specified as a symbolic number, vector, matrix, multidimensional array, expression, or function.

type — Type of number

'integer' | 'rational' | 'real'

Type of number, specified as 'integer', 'rational', or 'real'.

More About

Tips

- `in` does not consider assumptions on symbolic variables.
- `in` cannot partially evaluate a vector, matrix, or multidimensional array. If such an input contains undecidable elements, then `in` does not evaluate any element of the input. To force an evaluation, use `isAlways` on the function call to `in`, for example,

`isAlways(in(x,'real'))`. See “Test if Elements of the Symbolic Matrix Are Rational” on page 4-605.

See Also

`assume` | `assumeAlso` | `false` | `imag` | `isalways` | `isequaln` | `isfinite` | `isinf` | `logical` | `real` | `true`

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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies` | `spy`

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 a 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 a 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*sign(cos(x)))/2
```

If you want a simple practical solution, try `IgnoreAnalyticConstraints`:

```
int(acos(sin(x)), x, 'IgnoreAnalyticConstraints', true)
```

```
ans =
(x*(pi - x))/2
```

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 the 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.

More About

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:
  - $\log(e^x) = x$

- $\text{asin}(\sin(x)) = x$, $\text{acos}(\cos(x)) = x$, $\text{atan}(\tan(x)) = x$
- $\text{asinh}(\sinh(x)) = x$, $\text{acosh}(\cosh(x)) = x$, $\text{atanh}(\tanh(x)) = x$
- $W_k(x \cdot e^x) = x$ for all values of k

- "Integration" on page 2-12

See Also

`diff` | `symprod` | `symsum` | `symvar`

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`
`int64`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	8	`int64`

See Also

sym | vpa | single | uint8 | double | uint16 | uint32 | uint64

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

eig | det | rank

isAlways

Check whether equation or inequality holds for all values of its variables

Compatibility

`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" on page 4-623.

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 a Condition

Check if this inequality is valid for all values of `x`.

```
syms x
isAlways(abs(x) >= 0)
```

```
ans =
     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 =
     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 =
     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 =
     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: Cannot prove 'x <= 2*x'.
ans =
     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 =
     0
```

Instead of `false`, you can also specify `error` to return an error, and `true` to return logical `1` (`true`).

Test a Condition with Assumptions

Check this inequality under the assumption that `x` is positive. 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 =
     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.

More About

- "Assumptions on Symbolic Objects"
- "Clear Assumptions and Reset the Symbolic Engine"

See Also

`assume` | `assumeAlso` | `assumptions` | `in` | `isequaln` | `isfinite` | `isinf` | `isnan` | `logical` | `sym` | `syms`

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 =
     0
```

For positive `x`, these expressions are identical:

```
assume(x > 0)
isequaln(abs(x), x)
```

```
ans =
     1
```

For further computations, remove the assumption:

```
syms x clear
```

Compare Two Matrices

Use `isequaln` to compare these two matrices:

```
A = hilb(3);
B = sym(A);
isequaln(A, B)
```

```
ans =
     1
```

Compare Vectors Containing `NaN` Values

Use `isequaln` to compare these vectors:

```
syms x
A1 = [x NaN NaN];
A2 = [x NaN NaN];
A3 = [x NaN NaN];
isequaln(A1, A2, A3)
```

```
ans =
     1
```

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.

More About

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` | `isequaln` | `isfinite` | `isinf` | `isnan` | `logical`

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 a 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 =
     1     0     0
     1     0     0
```

Determine If the 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 =
     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     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.

More About

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` | `isequaln` | `isinf` | `isnan` | `logical`

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 a 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 =
     0     0     1
     0     1     0
```

Determine If the 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     1     1     1
```

Nevertheless, the cotangents of the approximated values are not infinite due to the round-off errors:

```
isinf(cot(V_approx))

ans =
     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.

More About

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` | `isequaln` | `isfinite` | `isnan` | `logical`

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 a 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 =
     1
```

Reduce Differential Index of a 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 =
     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 =
     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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

isnan

Check whether symbolic array elements are NaNs

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 a Symbolic Array Are NaNs

Using `isnan`, determine which elements of this symbolic matrix are `NaN`s:

```
isnan(sym([pi NaN Inf; 1 + i Inf + i NaN + i]))
```

```
ans =
     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.

More About

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` | `isequaln` | `isfinite` | `isinf` | `logical`

iztrans

Inverse Z-transform

Syntax

```
iztrans(F,trans_index,eval_point)
```

Description

`iztrans(F,trans_index,eval_point)` computes the inverse Z-transform of F with respect to the transformation index trans_index at the point eval_point.

Input Arguments

`F`

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

`trans_index`

Symbolic variable representing the transformation index. This variable is often called the “complex frequency variable”.

Default: The variable `z`. If F does not contain `z`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable or expression representing the evaluation point. This variable is often called the “discrete time variable”.

Default: The variable `n`. If `n` is the transformation index of F, then the default evaluation point is the variable `k`.

Examples

Compute the inverse Z-transform of this expression with respect to the transformation index `x` at the evaluation point `k`:

```
syms k x
F =  2*x/(x - 2)^2;
iztrans(F, x, k)

ans =
2^k + 2^k*(k - 1)
```

Compute the inverse Z-transform of this expression calling the `iztrans` function with one argument. If you do not specify the transformation index, `iztrans` uses the variable `z`.

```
syms z a k
F = exp(a/z);
iztrans(F, k)

ans =
a^k/factorial(k)
```

If you also do not specify the evaluation point, `iztrans` uses the variable `n`:

```
iztrans(F)

ans =
a^n/factorial(n)
```

Compute the inverse Z-transforms of these expressions. The results involve the Kronecker's delta function.

```
syms n z
iztrans(1/z, z, n)

ans =
kroneckerDelta(n - 1, 0)

iztrans((z^3 + 3*z^2 + 6*z + 5)/z^5, z, n)

ans =
kroneckerDelta(n - 2, 0) + 3*kroneckerDelta(n - 3, 0) +...
6*kroneckerDelta(n - 4, 0) + 5*kroneckerDelta(n - 5, 0)
```

If `iztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms F(z) n
f = iztrans(F, z, n)
```

```
f =
iztrans(F(z), z, n)
```

`ztrans` returns the original expression:

```
ztrans(f, n, z)
```

```
ans =
F(z)
```

Find the inverse Z-transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
iztrans([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])
```

```
ans =
[ exp(x)*kroneckerDelta(a, 0), kroneckerDelta(b, 0)]
[       iztrans(sin(y), y, c),   iztrans(z, z, d)*i]
```

When the input arguments are nonscalars, `iztrans` acts on them element-wise. If `iztrans` is called with both scalar and nonscalar arguments, then `iztrans` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
iztrans(x,[x, w; y, z],[a, b; c, d])
```

```
ans =
[       iztrans(x, x, a), x*kroneckerDelta(b, 0)]
[ x*kroneckerDelta(c, 0), x*kroneckerDelta(d, 0)]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, 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)]
```

More About

Inverse Z-Transform

If R is a positive number, such that the function $F(z)$ is analytic on and outside the circle $|z| = R$, then the inverse Z-transform is defined as follows:

$$f(n) = \frac{1}{2\pi i} \oint_{|z|=R} F(z) z^{n-1} dz, \quad n = 0, 1, 2...$$

Tips

- If you call `iztrans` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If F is a matrix, `iztrans` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `iztrans` acts element-wise on all components of the matrix.
- To compute the direct Z-transform, use `ztrans`.
- "Compute Z-Transforms and Inverse Z-Transforms" on page 2-197

See Also

`fourier` | `ifourier` | `ilaplace` | `kroneckerDelta` | `laplace` | `ztrans`

jacobian

Jacobian matrix

Syntax

```
jacobian(f,v)
```

Description

`jacobian(f,v)` computes the Jacobian matrix of f with respect to v. The (i,j) element of the result is $\frac{\partial f(i)}{\partial \mathrm{v}(j)}$.

Examples

Jacobian of a 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 a 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 a 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.

More About

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:

$$J(x_1,\dots x_n) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `laplacian` | `potential` | `vectorPotential`

jacobiP

Jacobi polynomials

Syntax

```
jacobiP(n,a,b,x)
```

Description

`jacobiP(n,a,b,x)` returns the nth degree "Jacobi Polynomial" on page 4-650 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 =
10157367338124939405 0.64541318209
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 the 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)
```

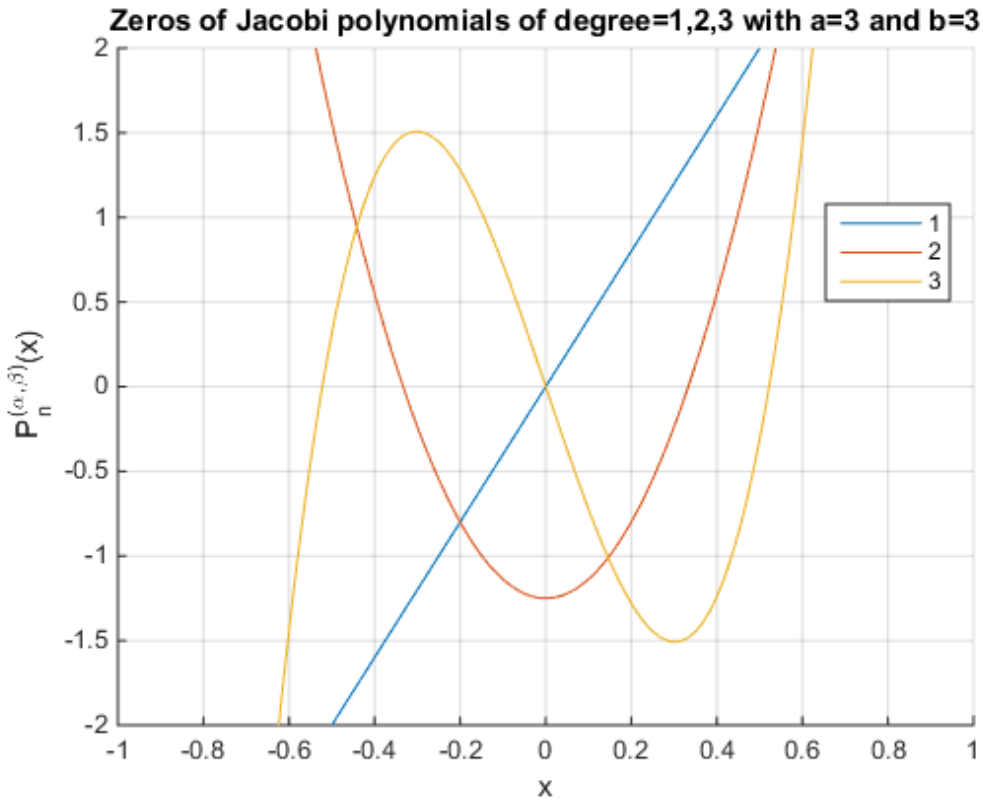
```
J =
[ (5*x)/2 - 1/2,     3*x]
[       3*x + 1, 3*x - 1]
```

`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 `y`-axis limits to `-2<y<2` using `ylim`.

```
syms x
hold on
grid on
for n = 1:3
    ezplot(jacobiP(n,3,3,x),[-1 1])
end
ylim([-2 2]);
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 the 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.

More About

Jacobi Polynomial

The Jacobi polynomials are given by the recursion formula

$$2nc_n c_{2n-2} P(n,a,b,x) = c_{2n-1}\left(c_{2n-2}c_{2n}x + a^2 - b^2\right)P(n-1,a,b,x) - 2(n-1+a)(n-1+b)c_{2n}P(n-2,a,b,x),$$

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

$$G\left(n,a,x\right)=\frac{\Gamma\left(a+\frac{1}{2}\right)\Gamma\left(n+2a\right)}{\Gamma\left(2a\right)\Gamma\left(n+a+\frac{1}{2}\right)}P\left(n,a-\frac{1}{2},a-\frac{1}{2},x\right).$$

See Also

`chebyshevT` | `chebyshevU` | `gegenbauerC` | `hermiteH` | `hypergeom` | `laguerreL` | `legendreP`

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` | `inv`

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

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 =
     0
```

Compare the 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 a Vector of Numbers with a 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)
```

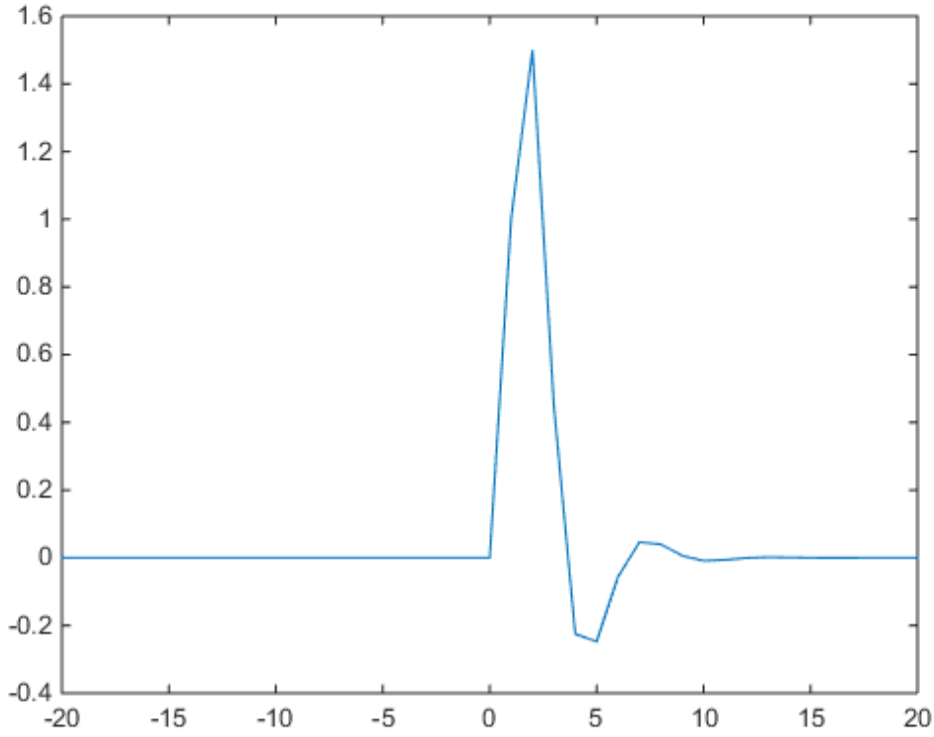
Use this output as input to `ztrans` to return the initial input expression.

```
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.

More About

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`

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 =
(C3*hypergeom([i/2], [1 + i], t/(2*z^2)))/z^i + (C2*kummerU(i/2, 1 + i, t/(2*z^2)))/z^
```

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, 3*i)
```

Use `vpa` to approximate symbolic results with the required number of digits.

```
vpa(symA,10)
```

```
ans =
                  0.8233667846
                  0.7284037305
 0.4434362538 - 0.3204327531*i
```

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.

More About

Confluent Hypergeometric Function (Kummer U Function)

The confluent hypergeometric function (Kummer U function) is one of the solutions of the differential equation

$$z\frac{\partial^2}{\partial z^2}y+(b-z)\frac{\partial}{\partial z}y-ay=0$$

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:

$$W_{a,b}(z)=e^{-z/2}\,z^{b+1/2}\,U\left(b-a+\frac{1}{2},2b+1,z\right)$$

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`

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 4-670.

`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 the 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 the 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 the 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 mupadmex
Error in MuPAD command: The function 'laguerreL' is not
```

```
defined for parameter values '-5/2' and '-3/2'.
[lasterror]
  Evaluating: trapfcn
```

Find the 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 the 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 the Taylor Series Expansion of a Laguerre Polynomial

Use `taylor` to find the Taylor series expansion of the generalized Laguerre polynomial of degree `2` at `x = 0`.

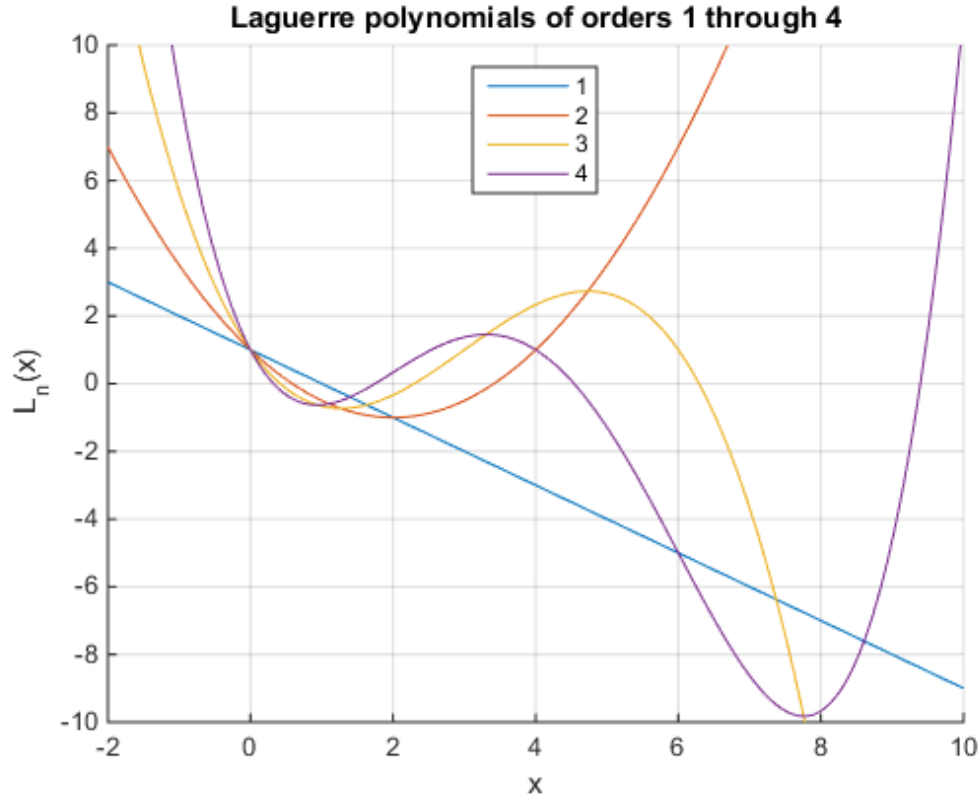
```
syms a x
expr = laguerreL(2,a,x);
taylor(expr,x)
```

```
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` for `-2<x<10`. To better view the plot, use `ylim` to set the `y`-axis limits.

```
syms x
hold on
grid on
for n = 1:4
 ezplot(laguerreL(n,x),[-2  10])
end
ylim([-10, 10])
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.

More About

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,

$$\left\langle \text{laguerreL}(n,a,x), \text{laguerreL}(m,a,x)\right\rangle = \begin{cases} 0 & \text{if } n \neq m \\ \dfrac{\Gamma(a+n+1)}{n!} & \text{if } n = m. \end{cases}$$

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 the Generalized Laguerre Function" on page 4-666.

- 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 $\mathrm{laguerreL}(n,a,0) = \binom{n+a}{a}$ 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

$$\mathrm{laguerreL}(n,a,x) = (-1)^a\, e^x\, \mathrm{laguerreL}(-n-a-1,a,-x)$$

- 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`

lambertw

Lambert W function

Syntax

```
lambertw(x)
lambertw(k,x)
```

Description

`lambertw(x)` is the Lambert W function of x. This syntax returns the principal branch of the Lambert W function, and, therefore, it is equivalent to `lambertw(0,x)`.

`lambertw(k,x)` is the kth branch of the Lambert W function.

Examples

Equation Returning the 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
simplify(x - subs(W*exp(W), W, lambertw(k,x))) == 0

k =
    -2    -1     0     1     2

ans =
     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, i)]

Wmin1 =
[             -Inf,              -1]
[ lambertw(-1, pi), lambertw(-1, i)]
```

Use vpa to approximate symbolic results with the required number of digits:

```
vpa(W0, 10)
vpa(Wmin1, 5)
```

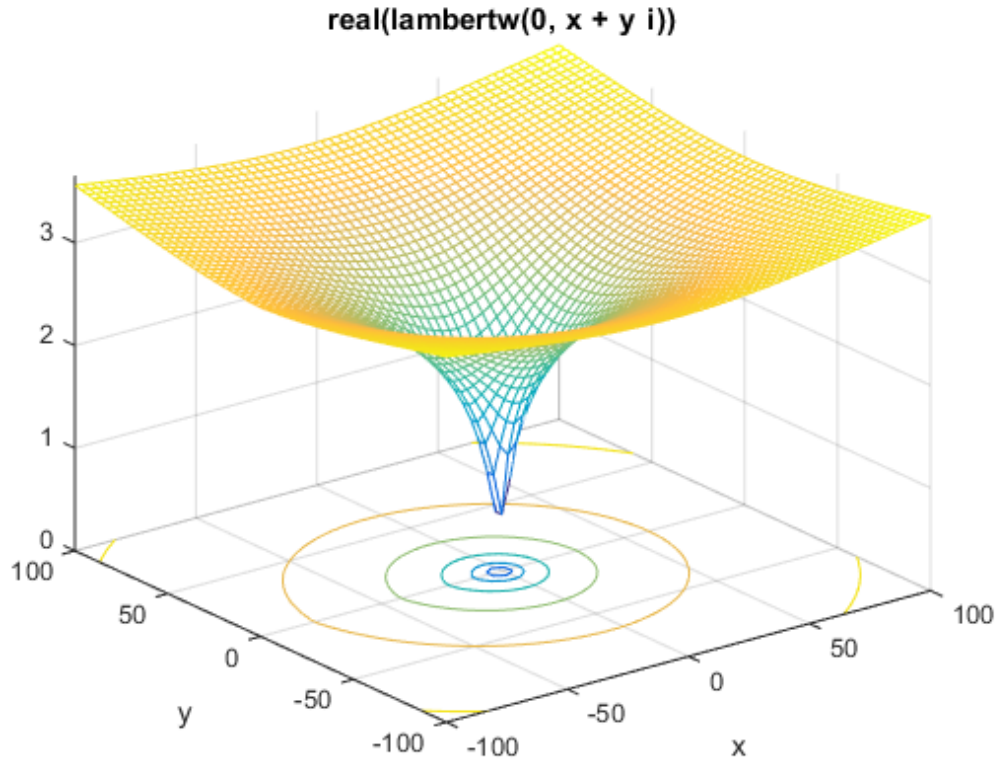
```
ans =
[           0,                          -1.0]
[ 1.073658195, 0.3746990207 + 0.576412723*i]

ans =
[                 -Inf,                 -1.0]
[ - 0.39097 - 4.6281*i, - 1.0896 - 2.7664*i]
```

Lambert W Function Plot on the Complex Plane

Plot the principal branch of the Lambert W function on the complex plane.

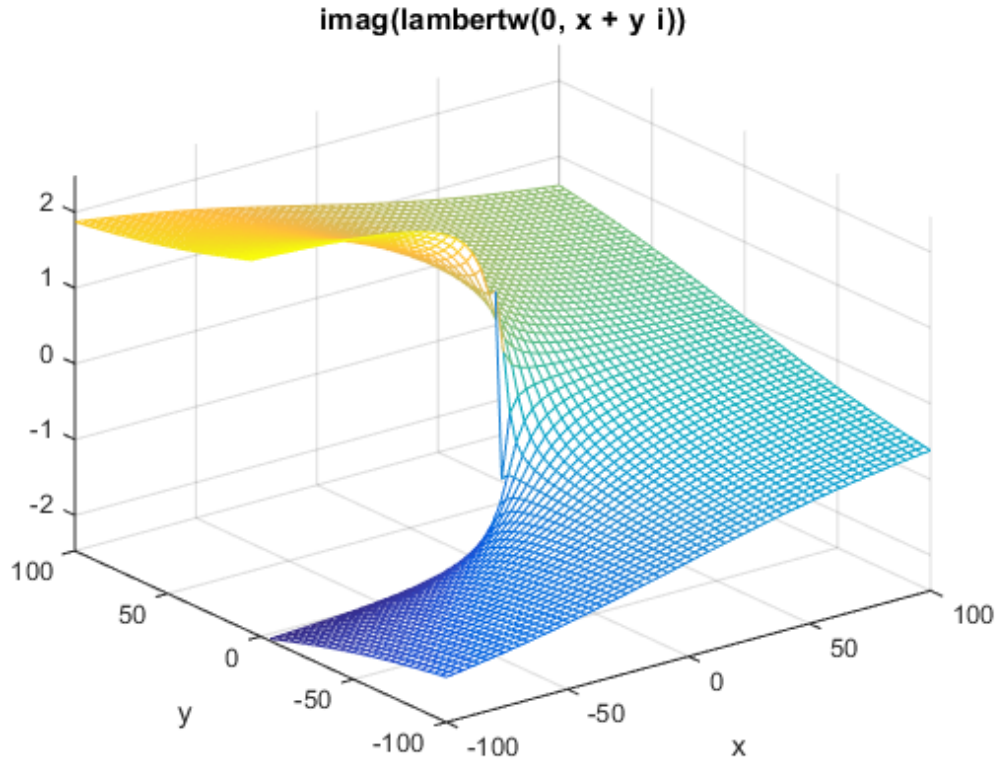
Create the combined mesh and contour plot of the real value of the Lambert W function on the complex plane.

```
syms x y real
ezmeshc(real(lambertw(x + i*y)), [-100, 100, -100, 100])
```

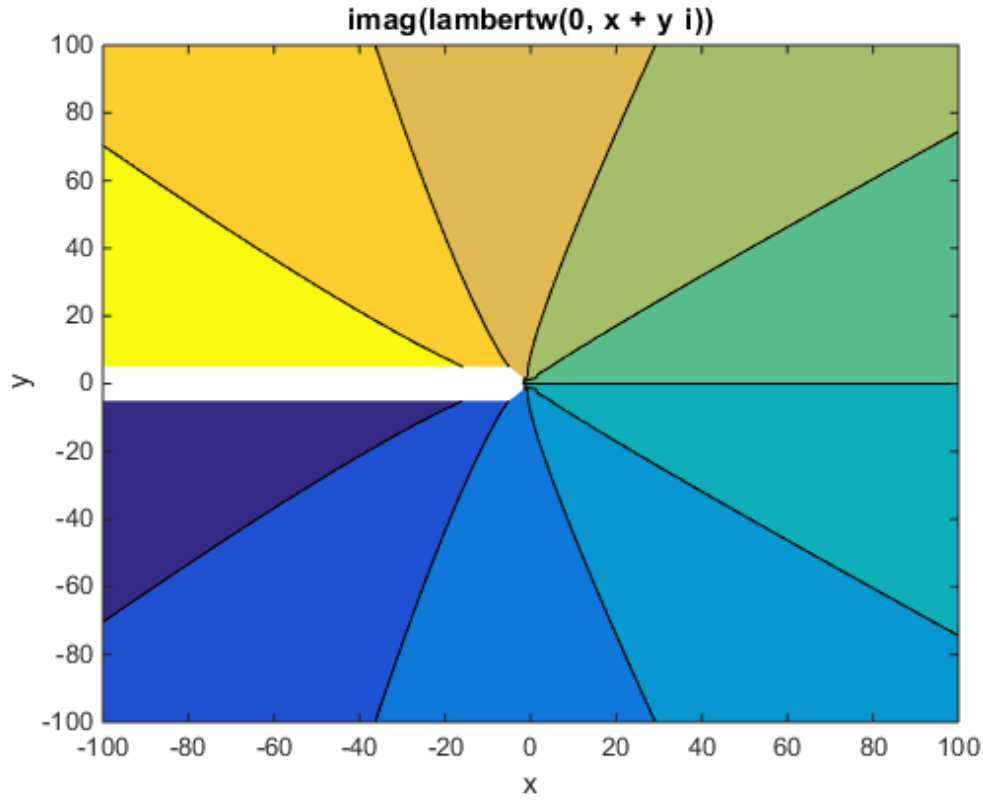


Now, plot the imaginary value of the Lambert W function on the complex plane. This function has a branch cut along the negative real axis. For better perspective, create the mesh and contour plots separately.

```
ezmesh(imag(lambertw(x + i*y)), [-100, 100, -100, 100])
```

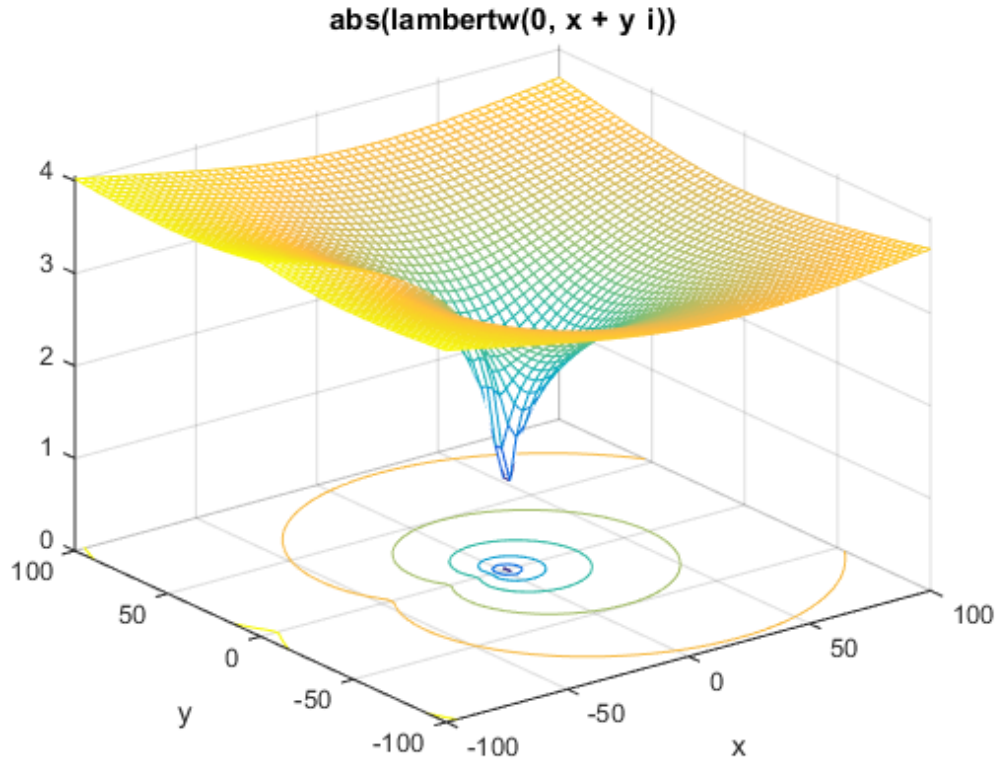



```
ezcontourf(imag(lambertw(x + i*y)), [-100, 100, -100, 100])
```



Plot the absolute value of the Lambert W function on the complex plane.

```
ezmeshc(abs(lambertw(x + i*y)), [-100, 100, -100, 100])
```



For further computations, clear the assumptions on `x` and `y`:

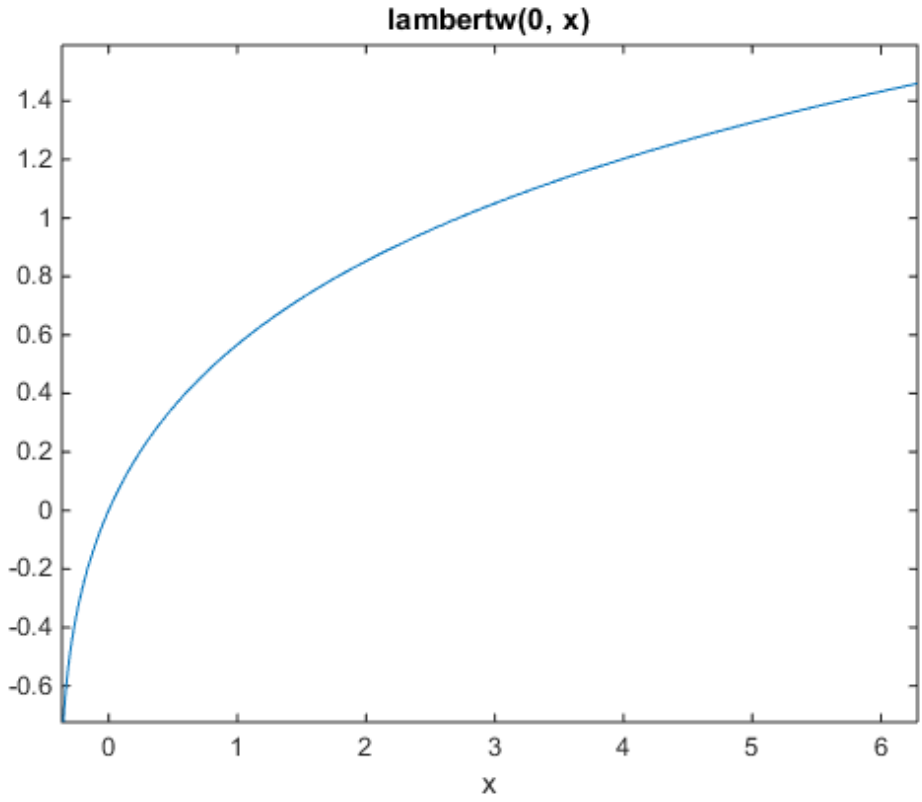
```
syms x y clear
```

Plot of the Two Main Branches

Plot the two main branches, $W_0(x)$ and $W_{-1}(x)$, of the Lambert W function.

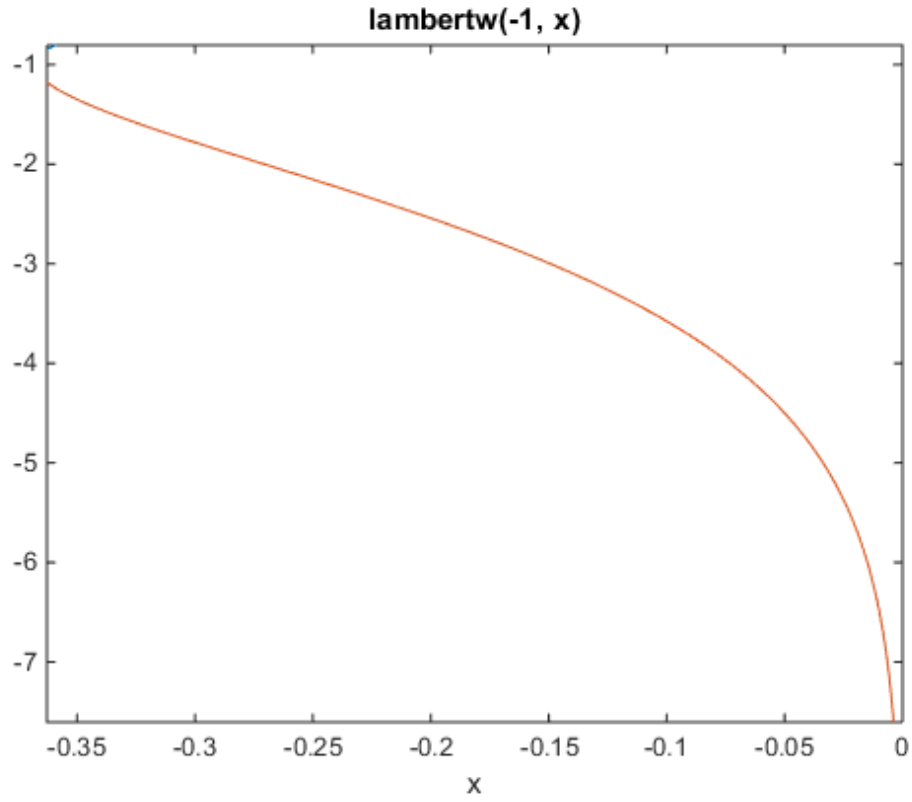
Plot the principal branch $W_0(x)$:

```
syms x
ezplot(lambertw(x))
```



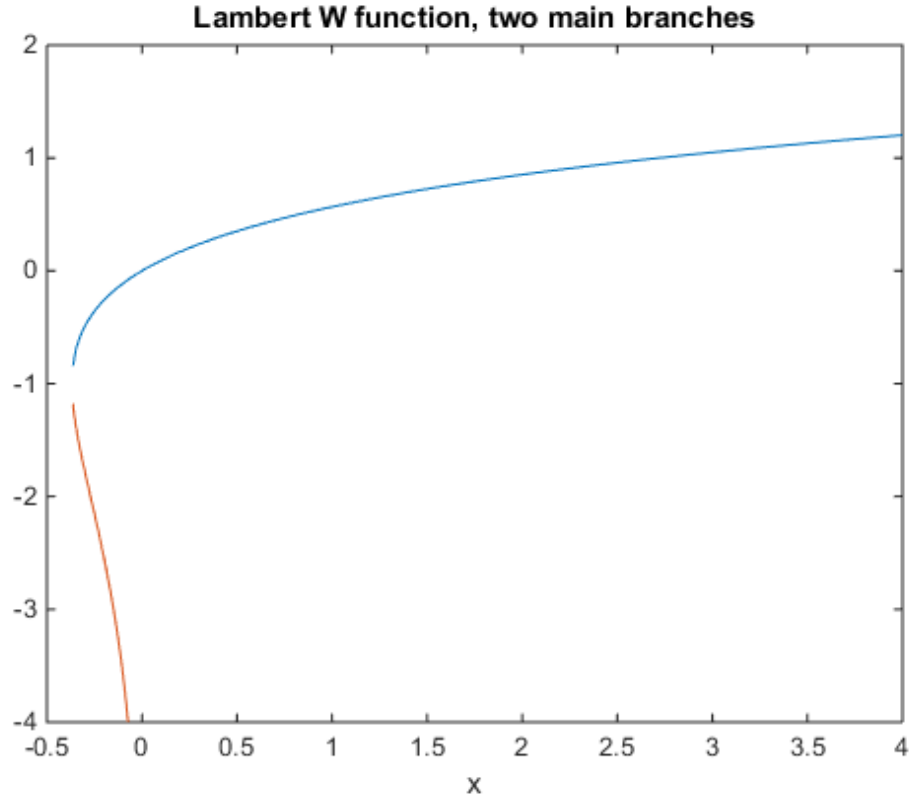
Add the branch $W_{-1}(x)$:

```
hold on
ezplot(lambertw(-1, x))
```



Adjust the axes limits and add the title:

```
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`.

More About

Lambert W Function

The Lambert W function `W(x)` represents the solutions of the equation $x = W(x)e^{W(x)}$ for any complex number `x`.

Tips

- The equation $x = w(x)e^{w(x)}$ 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 $x = w(x)e^{w(x)}$ 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 `w = -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`. For `k <> {0, -1}`, `lambertw(k,x)` is always complex.
- 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.

[2] Corless, R.M, G.H. Gonnet, D.E.G. Hare, and D.J. Jeffrey “Lambert’s W Function in Maple” *The Maple Technical Newsletter (MapleTech)*, vol.9, pp. 12–22, 1993.

See Also

Functions

`wrightOmega`

MuPAD Functions

`lambertW`

laplace

Laplace transform

Syntax

```
laplace(f,trans_var,eval_point)
```

Description

`laplace(f,trans_var,eval_point)` computes the Laplace transform of f with respect to the transformation variable trans_var at the point eval_point.

Input Arguments

`f`

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

`trans_var`

Symbolic variable representing the transformation variable. This variable is often called the “time variable”.

Default: The variable `t`. If f does not contain `t`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable or expression representing the evaluation point. This variable is often called the “complex frequency variable”.

Default: The variable `s`. If `s` is the transformation variable of f, then the default evaluation point is the variable `z`.

Examples

Compute the Laplace transform of this expression with respect to the variable `x` at the evaluation point `y`:

```
syms x y
f = 1/sqrt(x);
laplace(f, x, y)
```

```
ans =
pi^(1/2)/y^(1/2)
```

Compute the Laplace transform of this expression calling the `laplace` function with one argument. If you do not specify the transformation variable, `laplace` uses the variable `t`.

```
syms a t y
f = exp(-a*t);
laplace(f, y)
```

```
ans =
1/(a + y)
```

If you also do not specify the evaluation point, `laplace` uses the variable `s`:

```
laplace(f)
```

```
ans =
1/(a + s)
```

Compute the following Laplace transforms that involve the Dirac and Heaviside functions:

```
syms t s
laplace(dirac(t - 3), t, s)
```

```
ans =
exp(-3*s)
```

```
laplace(heaviside(t - pi), t, s)
```

```
ans =
exp(-pi*s)/s
```

If `laplace` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(t) s
F = laplace(f, t, s)
```

```
F =
laplace(f(t), t, s)
```

`ilaplace` returns the original expression:

```
ilaplace(F, s, t)
```

```
ans =
f(t)
```

The Laplace transform of a function is related to the Laplace transform of its derivative:

```
syms f(t) s
laplace(diff(f(t), t), t, s)
```

```
ans =
s*laplace(f(t), t, s) - f(0)
```

Find the Laplace transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
laplace([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])
```

```
ans =
[    exp(x)/a,   1/b]
[ 1/(c^2 + 1), i/d^2]
```

When the input arguments are nonscalars, `laplace` acts on them element-wise. If `laplace` is called with both scalar and nonscalar arguments, then `laplace` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
laplace(x,[x, w; y, z],[a, b; c, d])
```

```
ans =
[ 1/a^2, x/b]
[   x/c, x/d]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, 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]
```

More About

Laplace Transform

The Laplace transform is defined as follows:

$$F(s) = \int_0^\infty f(t)\, e^{-st} dt.$$

Tips

- If you call `laplace` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If f is a matrix, `laplace` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `laplace` acts element-wise on all components of the matrix.
- To compute the inverse Laplace transform, use `ilaplace`.
- "Compute Laplace and Inverse Laplace Transforms" on page 2-190

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `ztrans`

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
```

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])
```

More About

Laplacian of a Scalar Function

The Laplacian of the scalar function or functional expression f with respect to the vector $X = (X_1,...,X_n)$ is the sum of the second derivatives of f with respect to $X_1,...,X_n$:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f_i}{\partial x_i^2}$$

Tips

- If x is a scalar, `gradient(f, x) = diff(f, 2, x)`.

See Also

`curl` | `diff` | `divergence` | `gradient` | `hessian` | `jacobian` | `potential` | `vectorPotential`

latex

LaTeX representation of symbolic expression

Syntax

```
latex(S)
```

Description

`latex(S)` returns the LaTeX representation of the symbolic expression `S`.

Examples

The statements

```
syms x
f = taylor(log(1+x));
latex(f)
```

return

```
ans =
\frac{x^5}{5} - \frac{x^4}{4} + \frac{x^3}{3} - \frac{x^2}{2} + x
```

The statements

```
H = sym(hilb(3));
latex(H)
```

return

```
ans =
\left(\begin{array}{ccc} 1 & \frac{1}{2} & \frac{1}{3}\\...
\frac{1}{2} & \frac{1}{3} & \frac{1}{4}\\...
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{array}\right)
```

The statements

```
syms t
```

```
alpha = sym('alpha');
A = [alpha t alpha*t];
latex(A)
```

return

```
ans =
\left(\begin{array}{ccc} \mathrm{alpha} & t & \mathrm{alpha}\, t...
\end{array}\right)
```

You can use the `latex` command to annotate graphs:

```
syms x
f = taylor(log(1+x));
ezplot(f)
hold on
title(['$' latex(f) '$'],'interpreter','latex')
hold off
```

See Also

`ccode` | `fortran` | `pretty` | `texlabel`

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 + 30*i
```

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.

More About

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`

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
```

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)`.

More About

Tips

- If A and B are both numbers, then `A <= B` compares A and B and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A <= B` returns a symbolic less than or equal

to relation. You can use that relation as an argument for such functions as `assume`, `assumeAlso`, and `subs`.

- If both A and B are arrays, then these arrays must have the same dimensions. `A <= B` returns an array of relations `A(i,j,...)<=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if A is a variable (for example, `x`), and B is an *m*-by-*n* matrix, then A is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x <= i` becomes `x <= 0`, and `x <= 3 + 2*i` becomes `x <= 3`.

- "Set Assumptions" on page 1-32

See Also

`eq` | `ge` | `gt` | `isAlways` | `logical` | `lt` | `ne`

legendreP

Legendre polynomials

Syntax

```
legendreP(n,x)
```

Description

`legendreP(n,x)` returns the nth degree "Legendre Polynomial" on page 4-704 at x.

Examples

Find the 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 the 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 the 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 the Taylor Series Expansion of a Legendre Polynomial

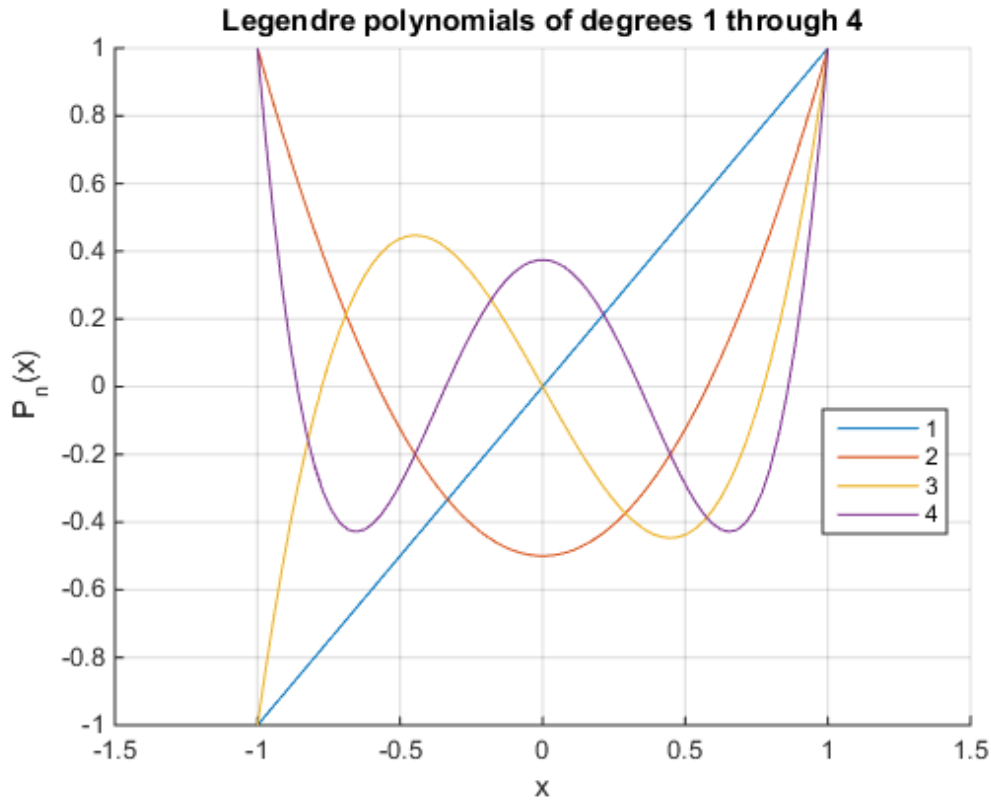
Use `taylor` to find the Taylor series expansion of the Legendre polynomial of degree `2` at `x = 0`.

```
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`. To better view the plot, better set the axes limits using `axis`.

```
syms x y
hold on
for n=1:4
    ezplot(legendreP(n,x))
end
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 a 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.

More About

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`

limit

Compute limit of symbolic expression

Syntax

```
limit(expr,x,a)
limit(expr,a)
limit(expr)
limit(expr,x,a,'left')
limit(expr,x,a,'right')
```

Description

`limit(expr,x,a)` computes bidirectional limit of the symbolic expression `expr` when `x` approaches *`a`*.

`limit(expr,a)` computes bidirectional limit of the symbolic expression `expr` when the default variable approaches *`a`*.

`limit(expr)` computes bidirectional limit of the symbolic expression `expr` when the default variable approaches `0`.

`limit(expr,x,a,'left')` computes the limit of the symbolic expression `expr` when `x` approaches *`a`* from the left.

`limit(expr,x,a,'right')` computes the limit of the symbolic expression `expr` when `x` approaches *`a`* from the right.

Examples

Compute bidirectional limits for the following expressions:

```
syms x h
limit(sin(x)/x)
limit((sin(x + h) - sin(x))/h, h, 0)
```

```
ans =
1

ans =
cos(x)
```

Compute the limits from the left and right for the following expressions:

```
syms x
limit(1/x, x, 0, 'right')
limit(1/x, x, 0, 'left')

ans =
Inf

ans =
-Inf
```

Compute the limit for the functions presented as elements of a vector:

```
syms x a
v = [(1 + a/x)^x, exp(-x)];
limit(v, x, inf)

ans =
[ exp(a), 0]
```

See Also

`diff` | `taylor`

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: The system is rank-deficient. Solution is not
unique.
  In sym.linsolve at 67
```

```
X =
                x/a
 -(x - a*y)/(a*b)
                 0
R =
2
```

More About

Matrix Representation of a System of Linear Equations

A system of linear equations

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
&\ldots \\
&a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m
\end{aligned}
$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix:

$$
A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}
$$

and $\vec{b}$ is the vector containing the right sides of equations:

$$
\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}
$$

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`

Related Examples

- "Solve a System of Algebraic Equations"

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

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`

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.

Input Arguments

`cond`

Equation, inequality, or vector or matrix of equations or inequalities. You also can combine several conditions by using the logical operators `and`, `or`, `xor`, `not`, or their shortcuts.

Examples

Use `logical` to check whether `1` if less than `2`:

```
logical(1 < 2)
```

```
ans =
     1
```

Check if the following two conditions are both valid. To check if several conditions are valid at the same time, combine these conditions by using the logical operator `and` or its shortcut `&`.

```
syms x
logical(1 < 2 & x == x)
```

```
ans =
     1
```

Check this inequality. Note that `logical` evaluates the left side of the inequality.

```
logical(4 - 1 > 2)
```

```
ans =
     1
```

`logical` also evaluates more complicated symbolic expressions on both sides of equations and inequalities. For example, it evaluates the integral on the left side of this equation:

```
syms x
logical(int(x, x, 0, 2) - 1 == 1)
```

```
ans =
     1
```

Check the validity of this equation using `logical`. Without an additional assumption that `x` is nonnegative, this equation is invalid.

```
syms x
logical(x == sqrt(x^2))
```

```
ans =
     0
```

Use `assume` to set an assumption that `x` is nonnegative. Now the expression `sqrt(x^2)` evaluates to `x`, and `logical` returns `1`:

```
assume(x >= 0)
logical(x == sqrt(x^2))
```

```
ans =
     1
```

Note that `logical` typically ignores assumptions on variables:

```
syms x
assume(x == 5)
logical(x == 5)
```

```
ans =
```

```
     0
```

To compare expressions taking into account assumptions on their variables, use `isAlways`:

```
isAlways(x == 5)
```

```
ans =
     1
```

For further computations, clear the assumption on `x`:

```
syms x clear
```

Do not use `logical` to check equations and inequalities that require simplification or mathematical transformations. For such equations and inequalities, `logical` might return unexpected results. For example, `logical` does not recognize mathematical equivalence of these expressions:

```
syms x
logical(sin(x)/cos(x) == tan(x))
```

```
ans =
     0
```

`logical` also does not realize that this inequality is invalid:

```
logical(sin(x)/cos(x) ~= tan(x))
```

```
ans =
     1
```

To test the validity of equations and inequalities that require simplification or mathematical transformations, use `isAlways`:

```
isAlways(sin(x)/cos(x) == tan(x))
```

```
ans =
     1
```

```
isAlways(sin(x)/cos(x) ~= tan(x))
```

```
Warning: Cannot prove 'sin(x)/cos(x) ~= tan(x)'.
ans =
     0
```

More About

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.

- "Assumptions on Symbolic Objects" on page 1-32
- "Clear Assumptions and Reset the Symbolic Engine" on page 3-43

See Also

`assume` | `assumeAlso` | `assumptions` | `in` | `isAlways` | `isequaln` | `isfinite` | `isinf` | `isnan` | `sym` | `syms`

logint

Logarithmic integral function

Syntax

```
logint(X)
```

Description

`logint(X)` represents the logarithmic integral function (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)]
```

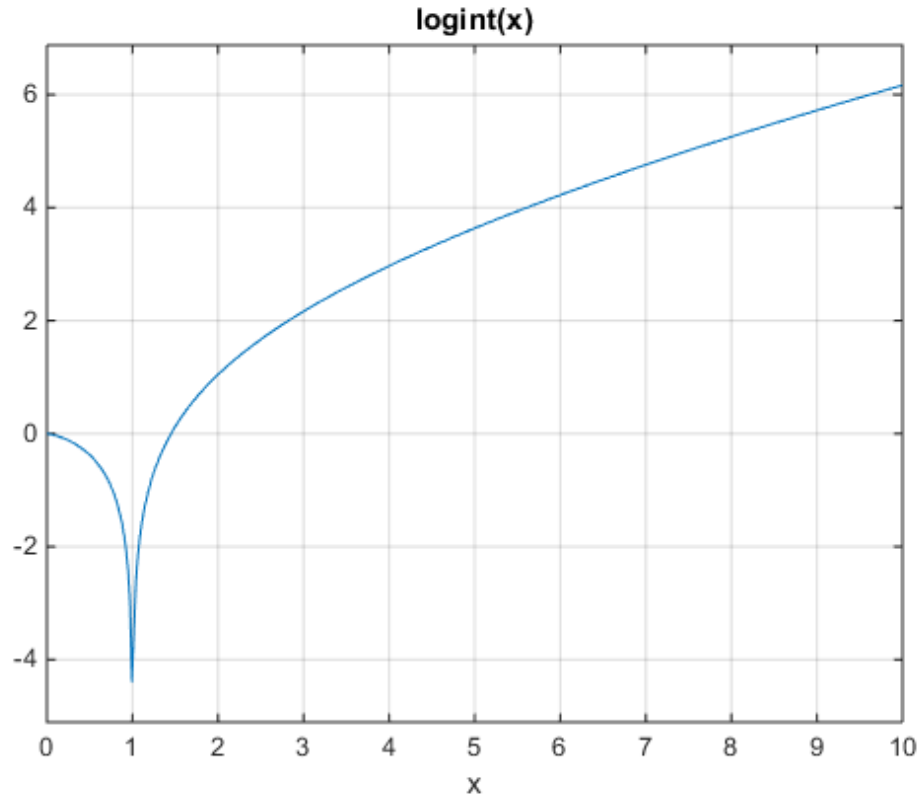
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.073667912046425485990100965230 15...
 + 3.422733378777362789592375061797 7*i,...
0,...
-0.118662056447123105305095706472 04,...
-0.378671043061087976727207184636 56,...
-Inf,...
1.045163780117492784844588889194 6,...
6.165599504787297937522981752669 5]
```

Plot the Integral Logarithm

Plot the integral logarithm function on the interval from 0 to 10.

```
syms x
ezplot(logint(x), [0, 10]);
grid on
```



Handle Expressions Containing the 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.

More About

Logarithmic Integral Function

The logarithmic integral function, also called the integral logarithm, is defined as follows:

$$\mathrm{logint}(x) = \mathrm{Li}(x) = \int_0^x \frac{1}{\ln(t)}\,dt$$

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`

logm

Matrix logarithm

Syntax

```
R = expm(A)
```

Description

`R = expm(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`

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
```

More About

Tips

- If A and B are both numbers, then A < B compares A and B and returns logical 1 (true) or logical 0 (false). Otherwise, A < B returns a symbolic less than relation. You can use that relation as an argument for such functions as assume, assumeAlso, and subs.
- If both A and B are arrays, then these arrays must have the same dimensions. A < B returns an array of relations A(i,j,...)<B(i,j,...)
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if A is a variable (for example, x), and B is an m-by-n matrix, then A is expanded into m-by-n matrix of elements, each set to x.

- The field of complex numbers is not an ordered field. MATLAB projects complex numbers in relations to a real axis. For example, `x < i` becomes `x < 0`, and `x < 3 + 2*i` becomes `x < 3`.

- "Set Assumptions" on page 1-32

See Also

`eq` | `ge` | `gt` | `isAlways` | `le` | `logical` | `ne`

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`.

`[L,U,p] = lu(A,'vector')` returns the permutation information as a vector p, such that `A(p,:) = L*U`.

`[L,U,p,q] = lu(A,'vector')` returns the permutation information as two row vectors p and q, such that `A(p,q) = L*U`.

`[L,U,P,Q,R] = lu(A)` returns an upper triangular matrix U, a lower triangular matrix L, permutation matrices P and Q, and a scaling matrix R, such that `P*(R\A)*Q = L*U`.

`[L,U,p,q,R] = lu(A,'vector')` returns the permutation information in two row vectors p and q, such that `R(:,p)\A(:,q) = L*U`.

`lu(A)` returns the matrix that contains the strictly lower triangular matrix L (the matrix without its unit diagonal) and the upper triangular matrix U as submatrices. Thus,

`lu(A)` returns the matrix `U + L - eye(size(A))`, where L and U are defined as `[L,U,P] = lu(A)`. The matrix A must be square.

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 =
     1     1     1
     1     1     1
     1     1     1
```

Restore the permutation matrix `P` from the vector `p`:

```
P = zeros(3, 3);
for i = 1:3
```

```
    P(i, p(i)) = 1;
end
P

P =
     0     1     0
     0     0     1
     1     0     0
```

Compute the LU factorization of this matrix returning the permutation information in the form of two vectors `p` and `q`:

```
syms a
A = [a, 2, 3*a; 2*a, 3, 4*a; 4*a, 5, 6*a];
[L, U, p, q] = lu(A, 'vector')

L =
[ 1, 0, 0]
[ 2, 1, 0]
[ 4, 3, 1]

U =
[ a,  2,  3*a]
[ 0, -1, -2*a]
[ 0,  0,    0]

p =
[ 1, 2, 3]

q =
[ 1, 2, 3]
```

Use `isAlways` to check that `A(p, q) = L*U`:

```
isAlways(A(p, q) == L*U)

ans =
     1     1     1
     1     1     1
     1     1     1
```

Compute the LU factorization of this matrix returning the lower and upper triangular matrices, permutation matrices, and the scaling matrix:

```
syms a
```

```
A = [0, a; 1/a, 0; 0, 1/5; 0,-1];
[L, U, P, Q, R] = lu(A)

L =
[ 1,       0, 0, 0]
[ 0,       1, 0, 0]
[ 0, 1/(5*a), 1, 0]
[ 0,    -1/a, 0, 1]

U =
[ 1/a, 0]
[   0, a]
[   0, 0]
[   0, 0]

P =
[ 0, 1, 0, 0]
[ 1, 0, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]

Q =
[ 1, 0]
[ 0, 1]

R =
[ 1, 0, 0, 0]
[ 0, 1, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]
```

Use `isAlways` to check that `P*(R\A)*Q = L*U`:

```
isAlways(P*(R\A)*Q == L*U)

ans =
     1     1
     1     1
     1     1
     1     1
```

Compute the LU factorization of this matrix using the `'vector'` flag to return the permutation information as vectors `p` and `q`. Also compute the scaling matrix `R`:

```
syms a
```

```
A = [0, a; 1/a, 0; 0, 1/5; 0,-1];
[L, U, p, q, R] = lu(A, 'vector')

L =
[ 1,       0, 0, 0]
[ 0,       1, 0, 0]
[ 0, 1/(5*a), 1, 0]
[ 0,    -1/a, 0, 1]

U =
[ 1/a, 0]
[   0, a]
[   0, 0]
[   0, 0]

p =
[ 2, 1, 3, 4]

q =
[ 1, 2]

R =
[ 1, 0, 0, 0]
[ 0, 1, 0, 0]
[ 0, 0, 1, 0]
[ 0, 0, 0, 1]
```

Use `isAlways` to check that `R(:,p)\A(:,q) = L*U`:

```
isAlways(R(:,p)\A(:,q) == L*U)

ans =
     1     1
     1     1
     1     1
     1     1
```

Call the `lu` function for this matrix:

```
syms a
A = [0 0 a; a 2 3; 0 a 2];
lu(A)

ans =
[ a, 2, 3]
```

```
[ 0, a, 2]
[ 0, 0, a]
```

Verify that the resulting matrix is equal to `U + L - eye(size(A))`, where `L` and `U` are defined as `[L,U,P] = lu(A)`:

```
[L,U,P] = lu(A);
U + L - eye(size(A))
```

```
ans =
[ a, 2, 3]
[ 0, a, 2]
[ 0, 0, a]
```

More About

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` | `linalg::factorLU` | `lu` | `qr` | `svd` | `vpa`

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 the form

$$M\left(t,x(t)\right)\dot{x}\left(t\right)=F\left(t,x(t)\right)$$

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`.

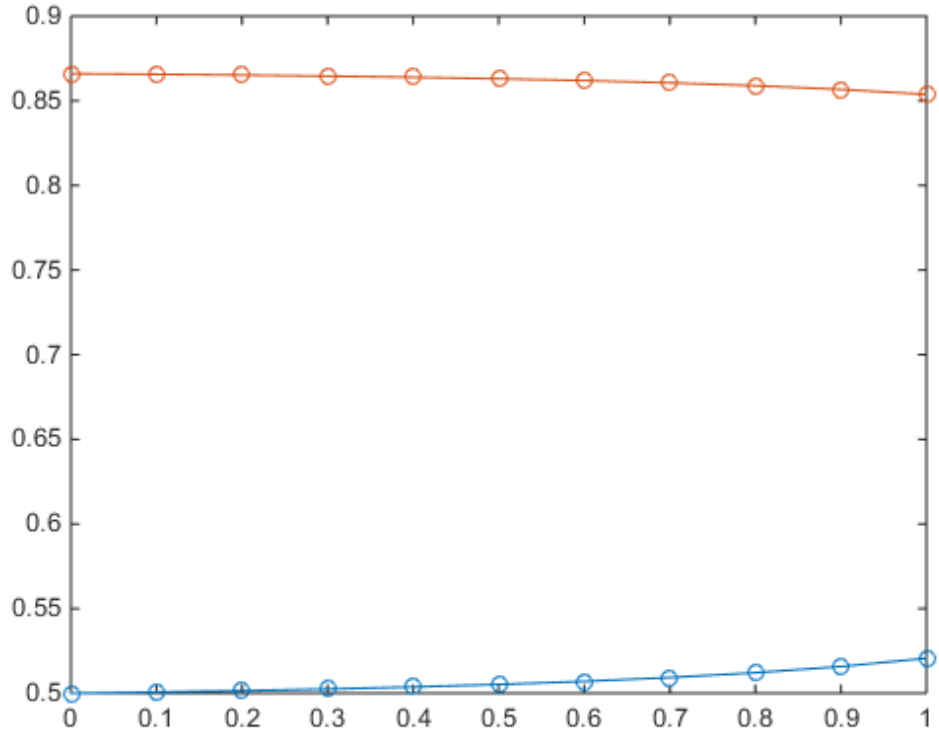
```
syms Y1 Y2;
M = subs(M, [vars, m, r, f], [Y1, Y2, 100, 1, @(t,x1,x2) t + x1*x2]);
F = subs(F, [vars, m, r, f], [Y1, Y2, 100, 1, @(t,x1,x2) t + x1*x2]);
```

Create the following function handles `MM` and `FF`. You can use these function handles as input arguments for `odeset` and `ode15s`. Note that 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]});
```

Use `ode15s` to solve the system.

```
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 coincides with the number of equations eqs.

See Also

`daeFunction` | `decic` | `findDecoupledBlocks` | `incidenceMatrix` | `isLowIndexDAE` | `matlabFunction` | `ode15s` | `odeset` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

matlabFunction

Convert symbolic expression to function handle or file

Syntax

```
g = matlabFunction(f)
g = matlabFunction(f1,...,fN)
g = matlabFunction(f,Name,Value)
g = matlabFunction(f1,...,fN,Name,Value)
```

Description

`g = matlabFunction(f)` converts the symbolic expression or function f to a MATLAB function with the handle g.

`g = matlabFunction(f1,...,fN)` converts a vector of the symbolic expressions or functions f1,...,fN to a MATLAB function with multiple outputs. The function handle is g.

`g = matlabFunction(f,Name,Value)` converts the symbolic expression or function f to a MATLAB function using additional options specified by one or more Name,Value pair arguments.

`g = matlabFunction(f1,...,fN,Name,Value)` converts a vector of the symbolic expressions or functions f1,...,fN to a MATLAB function with multiple outputs using additional options specified by one or more Name,Value pair arguments.

Input Arguments

`f`

Symbolic expression or function.

`f1,...,fN`

Vector of symbolic expressions or functions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'file'`

Generate a file with *optimized* code. The generated file can accept double or matrix arguments and evaluate the symbolic expression applied to the arguments. Optimized means intermediate variables are automatically generated to simplify or speed up the code. MATLAB generates intermediate variables as a lowercase letter `t` followed by an automatically generated number, for example `t32`. The value of this parameter must be a string representing the path to the file.

Default: If the value string is empty, `matlabFunction` generates an anonymous function. If the string does not end in `.m`, the function appends `.m`.

`'outputs'`

Specify the names of output variables. The value must be a cell array of strings.

Default: The names of output variables coincide with the names you use calling `matlabFunction`. If you call `matlabFunction` using an expression instead of individual variables, the default names of output variables consist of the word `out` followed by a number, for example, `out3`.

`'vars'`

Specify the order of the input variables or symbolic vectors in the resulting function handle or the file. The value of this parameter must be either a cell array of strings or symbolic arrays, or a vector of symbolic variables. The number of value entries must equal or exceed the number of free variables in f.

Default: When converting symbolic expressions, the order is alphabetical. When converting symbolic functions, the input arguments appear in front of other variables. Other variables are sorted alphabetically.

Output Arguments

g

MATLAB function handle.

Examples

Convert this symbolic expression to a MATLAB function with the handle `ht`:

```
syms x y
r = sqrt(x^2 + y^2);
ht = matlabFunction(sin(r)/r)
```

```
ht =
   @(x,y)sin(sqrt(x.^2+y.^2)).*1.0./sqrt(x.^2+y.^2)
```

Create this symbolic function:

```
syms x y
f(x, y) = x^3 + y^3;
```

Convert `f` to a MATLAB function:

```
ht = matlabFunction(f)
```

```
ht =
    @(x,y)x.^3+y.^3
```

Convert this expression to a MATLAB function generating the file `myfile` that contains the function:

```
syms x y z
r = x^2 + y^2 + z^2;
f = matlabFunction(log(r)+r^(-1/2),'file','myfile');
```

If the file `myfile.m` already exists in the current folder, `matlabFunction` replaces the existing function with the converted symbolic expression. You can open and edit the resulting file:

```
function out1 = myfile(x,y,z)
```

```
%MYFILE
%    OUT1 = MYFILE(X,Y,Z)

t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
t5 = t2 + t3 + t4;
out1 = log(t5) + 1.0./sqrt(t5);
```

Convert this expression to a MATLAB function using an empty string to represent a path to the file. An empty string causes `matlabFunction` to generate an anonymous function:

```
syms x y z
r = x^2 + y^2 + z^2;
f = matlabFunction(log(r)+r^(-1/2),'file','')

f =
    @(x,y,z)log(x.^2+y.^2+z.^2)+1.0./sqrt(x.^2+y.^2+z.^2)
```

When converting this expression to a MATLAB function, specify the order of the input variables:

```
syms x y z
r = x^2 + y^2 + z^2;
matlabFunction(r, 'file', 'my_function',...
'vars', [y z x]);
```

The created `my_function` accepts variables in the required order:

```
function r = my_function(y,z,x)
%MY_FUNCTION
%    R = MY_FUNCTION(Y,Z,X)

r = x.^2 + y.^2 + z.^2;
```

When converting this expression to a MATLAB function, specify its second input argument as a vector:

```
syms x y z t
r = (x^2 + y^2 + z^2)*exp(-t);
matlabFunction(r, 'file', 'my_function',...
'vars', {t, [x y z]});
```

The resulting function operates on vectors:

```
function r = my_function(t,in2)
%MY_FUNCTION
%    R = MY_FUNCTION(T,IN2)

x = in2(:,1);
y = in2(:,2);
z = in2(:,3);
r = exp(-t).*(x.^2+y.^2+z.^2);
```

When converting this expression to a MATLAB function, specify the names of the output variables:

```
syms x y z
r = x^2 + y^2 + z^2;
q = x^2 - y^2 - z^2;
f = matlabFunction(r, q, 'file', 'my_function',...
'outputs', {'name1','name2'});
```

The generated function returns `name1` and `name2`:

```
function [name1,name2] = my_function(x,y,z)
%MY_FUNCTION
%    [NAME1,NAME2] = MY_FUNCTION(X,Y,Z)

t2 = x.^2;
t3 = y.^2;
t4 = z.^2;
name1 = t2+t3+t4;
if nargout > 1
    name2 = t2-t3-t4;
end
```

Convert this MuPAD expression to a MATLAB function:

```
syms x y
f = evalin(symengine, 'arcsin(x) + arccos(y)');
matlabFunction(f, 'file', 'my_function');
```

The generated file contains the same expressions written in the MATLAB language:

```
function f = my_function(x,y)
%MY_FUNCTION
%    F = MY_FUNCTION(X,Y)

f = asin(x) + acos(y);
```

More About

Tips

- To convert a MuPAD expression or function to a MATLAB function, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine,'MuPAD_Function',x1,...,xn)`. `matlabFunction` cannot correctly convert some MuPAD expressions to MATLAB functions. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the conversion results. To verify the results, execute the resulting function.
- When you use the file argument, use `rehash` to make the generated function available immediately. `rehash` updates the MATLAB list of known files for directories on the search path.

- "Generate MATLAB Functions" on page 2-220
- "Create MATLAB Functions from MuPAD Expressions" on page 3-47

See Also

`ccode` | `evalin` | `feval` | `fortran` | `matlabFunctionBlock` | `rehash` | `simscapeEquation` | `subs` | `sym2poly`

matlabFunctionBlock

Convert symbolic expression to MATLAB Function block

Note: `emlBlock` has been removed. Use `matlabFunctionBlock` instead.

Syntax

```
matlabFunctionBlock(block,f)
matlabFunctionBlock(block,f1,...,fN)
matlabFunctionBlock(block,f,Name,Value)
matlabFunctionBlock(block,f1,...,fN,Name,Value)
```

Description

`matlabFunctionBlock(block,f)` converts the symbolic expression or function f to a MATLAB Function block that you can use in Simulink models. Here, block specifies the name of the block that you create or modify.

`matlabFunctionBlock(block,f1,...,fN)` converts a vector of the symbolic expressions or functions f1,...,fN to a MATLAB Function block with multiple outputs.

`matlabFunctionBlock(block,f,Name,Value)` converts the symbolic expression or function f to a MATLAB Function block using additional options specified by one or more Name,Value pair arguments.

`matlabFunctionBlock(block,f1,...,fN,Name,Value)` converts a vector of the symbolic expressions or functions f to a MATLAB Function block with multiple outputs using additional options specified by one or more Name,Value pair arguments.

Input Arguments

`f`

Symbolic expression or function.

`f1,...,fN`

Vector of symbolic expressions or functions.

`block`

String specifying the block name that you create or modify.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'functionName'`

Specify the name of the function. The value must be a string.

Default: The value coincides with the block name.

`'outputs'`

Specify the names of output ports. The value must be a cell array of strings. The number of entries must equal or exceed the number of free variables in f.

Default: The name of an output port consists of the word `out` followed by the output port number, for example, `out3`.

`'vars'`

Specify the order of the variables and the corresponding input ports of a block. The value must be either a cell array of strings or symbolic arrays, or a vector of symbolic variables. The number of entries must equal or exceed the number of free variables in f.

Default: When converting symbolic expressions, the order is alphabetical. When converting symbolic functions, the input arguments appear in front of other variables. Other variables are sorted alphabetically.

Examples

Before you can convert a symbolic expression to a MATLAB Function block, create an empty model or open an existing one:

```
new_system('my_system')
open_system('my_system')
```

Use `matlabFunctionBlock` to create the block `my_block` containing the symbolic expression:

```
syms x y z
f = x^2 + y^2 + z^2;
matlabFunctionBlock('my_system/my_block',f)
```

If you use the name of an existing block, `matlabFunctionBlock` replaces the definition of an existing block with the converted symbolic expression.

You can open and edit the resulting block. To open a block, double-click it:

```
function f = my_block(x,y,z)
%#codegen

f = x.^2 + y.^2 + z.^2;
```

Save and close `my_system`:

```
save_system('my_system')
close_system('my_system')
```

Create this symbolic function:

```
syms x y z
f(x, y, z) = x^2 + y^2 + z^2;
```

Convert `f` to a MATLAB Function block:

```
new_system('my_system')
open_system('my_system')
matlabFunctionBlock('my_system/my_block',f)
```

Generate a block and set the function name to `my_function`:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system')
open_system('my_system')
matlabFunctionBlock('my_system/my_block', f,...
```

```
'functionName', 'my_function')
```

When generating a block, specify the order of the input variables:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system')
open_system('my_system')
matlabFunctionBlock('my_system/my_block', f,...
'vars', [y z x])
```

When generating a block, rename the output variables and the corresponding ports:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system')
open_system('my_system')
matlabFunctionBlock('my_system/my_block', f, f + 1, f + 2,...
'outputs', {'name1','name2','name3'})
```

Call `matlabFunctionBlock` using several options simultaneously:

```
syms x y z
f = x^2 + y^2 + z^2;
new_system('my_system')
open_system('my_system')
matlabFunctionBlock('my_system/my_block', f, f + 1, f + 2,...
'functionName', 'my_function','vars', [y z x],...
'outputs',{'name1','name2','name3'})
```

Convert this MuPAD expression to a MATLAB Function block:

```
syms x y
new_system('my_system')
open_system('my_system')
f = evalin(symengine, 'arcsin(x) + arccos(y)');
matlabFunctionBlock('my_system/my_block', f)
```

The resulting block contains the same expressions written in the MATLAB language:

```
function f = my_block(x,y)
%#codegen

f = asin(x) + acos(y);
```

More About

Tips

- To convert a MuPAD expression or function to a MATLAB Function block, use `f = evalin(symengine,'MuPAD_Expression')` or `f = feval(symengine,'MuPAD_Function',x1,...,xn)`. `matlabFunctionBlock` cannot correctly convert some MuPAD expressions to a block. These expressions do not trigger an error message. When converting a MuPAD expression or function that is not on the MATLAB vs. MuPAD Expressions list, always check the conversion results. To verify the results, you can run the simulation containing the resulting block.

- "Generate MATLAB Function Blocks" on page 2-225
- "Create MATLAB Function Blocks from MuPAD Expressions" on page 3-50

See Also

`ccode` | `evalin` | `feval` | `fortran` | `matlabFunction` | `simscapeEquation` | `subs` | `sym2poly`

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 a 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 a 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 a 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 - i
```

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 + i
```

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.

More About

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`

mfun

Numeric evaluation of special mathematical function

Compatibility

`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`.

Examples

Evaluate the Fresnel cosine integral.

```
mfun('FresnelC',0:4)
```

```
ans =
         0    0.7799    0.4883    0.6057    0.4984
```

Evaluate the hyperbolic cosine integral.

```
mfun('Chi',[3*i 0])

ans =
0.1196 + 1.5708i   NaN
```

See Also

mfunlist

mfunlist

List special functions for use with `mfun`

Compatibility

`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:	`bernoulli(n)` `bernoulli(n,t)`	`bernoulli(n)` `bernoulli(n,t)`	$n \geq 0$

Function Name	Definition	mfun Name	Special Function Syntax	Arguments
	$\frac{e^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \cdot \frac{t^{n-1}}{n!}$			$0 < \lvert t \rvert < 2\pi$

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, $0 < k < 1$.

Complete elliptic integrals with complementar 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	`EllipticCK(k)`
		`EllipticCE(k)`
		`EllipticCPi(a,`
			`ellipticCK(k)`
			`ellipticCE(k)`
			`ellipticCPi(a,`
				`a` is real, $-\infty < a < \infty$.
				`k` is real, $0 < k < 1$.

Function Name	Definition	mfun Name	Special Function Syntax	Arguments
	elliptic integrals use the parameter $m = k^2 = \sin^2 \alpha$.			
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)`	
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)`	$n \geq 0$ $\lvert t \rvert < \frac{\pi}{2}$

Function Name	Definition	mfun Name	Special Function Syntax	Arguments

Exponential integrals	$Ei(n,z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt$
	
	$Ei(x) = PV\left(-\int_{-\infty}^{x} \frac{e^t}{t} \right)$
		`Ei(n,z)`
		
		`Ei(x)`
			`expint(n,x)`
			
			`ei(x)`
				$n \geq 0$
				
				$\mathrm{Real}(z) > 0$

Fresnel sine and cosine integrals	$C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt$
	
	$S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt$
		`FresnelC(x)`
		
		`FresnelS(x)`
			`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)`	$n > 0$

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)`	

Function Name	Definition	mfun Name	Special Function Syntax	Arguments

(Generalized) hypergeometri 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`
		
		where
		
		`n = [n1,n2,...]`
		
		`d = [d1,d2,...]`
			`hypergeom(n,d,`
			
			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`
			`ellipticF(x,k)`
			
			`ellipticF(x,k)`
			
			`ellipticPi(x,a`
				$0<x\le\infty$.
				
				`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	$\mathrm{lnGAMMA}(z)=\ln(\Gamma(z))$	`lnGAMMA(z)`	`gammaln(z)`	
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)`	$n\ge 0$

Function Name	Definition	mfun Name	Special Function Syntax	Arguments
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` 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,`	`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,`	`a` is a nonrational algebraic expression or a rational number greater than `-1`.
Legendre	`P(n,x)`	`legendreP(n,x)`	

Examples

```
mfun('H',5,10)
```

```
ans =
```

```
     3041200

mfun('dawson',3.2)

ans =
    0.1655
```

Limitations

In general, the accuracy of a function will be lower near its roots and when its arguments are relatively large.

Running time depends on the specific function and its parameters. In general, calculations are slower than standard MATLAB calculations.

References

[1] Abramowitz, M. and I.A., Stegun, *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*. New York: Dover, 1972.

See Also

`mfun`

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 a 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 a 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 a 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 + i
```

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 - i/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.

More About

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`

minpoly

Minimal polynomial of matrix

Syntax

```
minpoly(A)
minpoly(A,var)
```

Description

`minpoly(A)` returns a vector of the coefficients of the minimal polynomial of A. If A is a symbolic matrix, `minpoly` returns a symbolic vector. Otherwise, it returns a vector with elements of type `double`.

`minpoly(A,var)` returns the minimal polynomial of A in terms of var.

Input Arguments

`A`

Matrix.

`var`

Free symbolic variable.

Default: If you do not specify `var`, `minpoly` returns a vector of coefficients of the minimal polynomial instead of returning the polynomial itself.

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
```

More About

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`

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 a Scalar from an 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.

More About

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

`plus`

mod

Symbolic modulus after division

Syntax

```
mod(a,b)
```

Description

`mod(a,b)` finds the modulus 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.

More About

Modulus

The modulus of a and b is

$$\mathrm{mod}(a,b) = a - b * \mathrm{floor}\left(\frac{a}{b}\right),$$

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 $\mathrm{mod}(a,0) = 0$.

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`

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”.

More About

- “Create MuPAD Notebooks”
- “Open MuPAD Notebooks”

See Also

`getVar` | `mupadwelcome` | `openmn` | `openmu` | `setVar`

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 strings.

```
mupadNotebookTitle(allNBs)

ans =
    'myFile1'
    'myFile2'
```

```
'Notebook1
```

Return a Single Notebook Title as a String

`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 string by using `char`.

Create a new notebook with the handle `nb`:

```
nb = mupad;
```

Find the title of that notebook and convert it to a string:

```
titleAsStr = char(mupadNotebookTitle(nb));
```

Use the title the same way as any string:

```
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 3-24
- “Close MuPAD Notebooks from MATLAB” on page 3-16

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`

mupadwelcome

Start MuPAD interfaces

Syntax

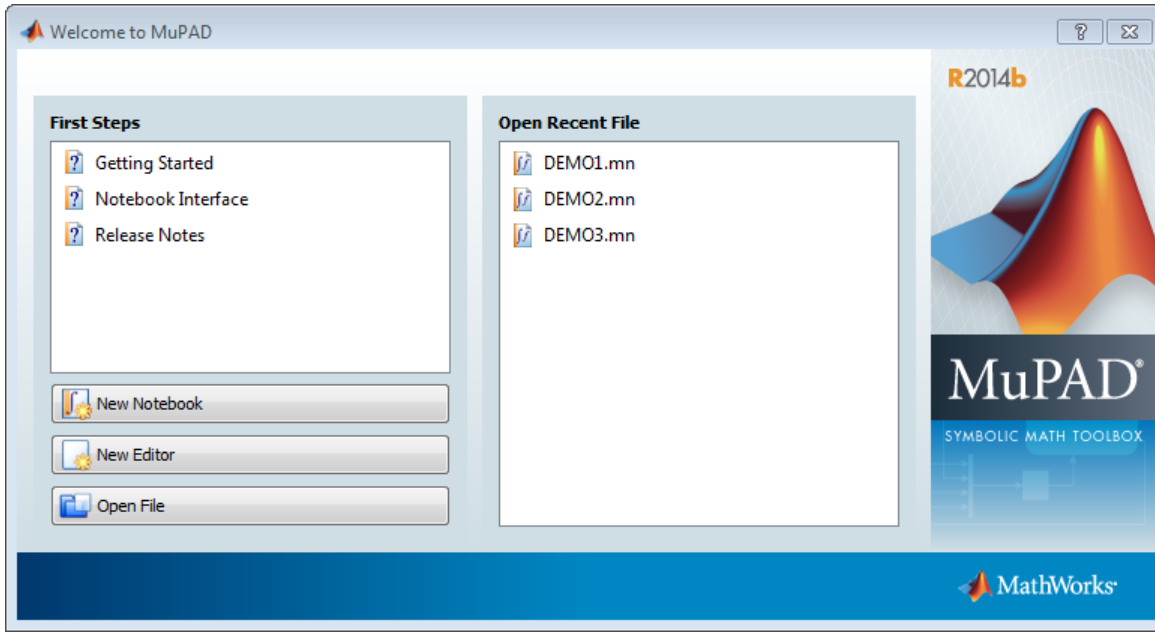
```
mupadwelcome
```

Description

`mupadwelcome` opens a window that enables you to start various interfaces:

- MuPAD Notebook app, 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.



More About

- “Create MuPAD Notebooks”
- “Open MuPAD Notebooks”

See Also

`mupad`

nchoosek

Binomial coefficient

Syntax

```
nchoosek(n,k)
```

Description

`nchoosek(n,k)` returns the binomial coefficient of n and k.

Input Arguments

n

Symbolic number, variable or expression.

k

Symbolic number, variable or expression.

Examples

Compute the binomial coefficients for these expressions:

```
syms n
[nchoosek(n, n), nchoosek(n, n + 1), nchoosek(n, n - 1)]
```

```
ans =
[ 1, 0, n]
```

If one or both parameters are negative numbers, convert these numbers to symbolic objects:

```
[nchoosek(sym(-1), 3), nchoosek(sym(-7), 2), nchoosek(sym(-5), -5)]
```

```
ans =
[ -1, 28, 1]
```

If one or both parameters are complex numbers, convert these numbers to symbolic objects:

```
[nchoosek(sym(i), 3), nchoosek(sym(i), i), nchoosek(sym(i), i + 1)]
```

```
ans =
[ 1/2 + i/6, 1, 0]
```

Differentiate the binomial coefficient:

```
syms n
diff(nchoosek(n, 2))
```

```
ans =
-(psi(n - 1) - psi(n + 1))*nchoosek(n, 2)
```

Expand the binomial coefficient:

```
syms n k
expand(nchoosek(n, k))
```

```
ans =
-(n*gamma(n))/(k^2*gamma(k)*gamma(n - k) - k*n*gamma(k)*gamma(n - k))
```

More About

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` | `gamma` | `factorial` | `psi`

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
```

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)`.

More About

Tips

- If A and B are both numbers, then `A ~= B` compares A and B and returns logical `1` (`true`) or logical `0` (`false`). Otherwise, `A ~= B` returns a symbolic inequality. You can use that inequality as an argument for such functions as `assume`, `assumeAlso`, and `subs`.
- If both A and B are arrays, then these arrays must have the same dimensions. `A ~= B` returns an array of inequalities `A(i,j,...)~=B(i,j,...)`
- If one input is scalar and the other an array, then the scalar input is expanded into an array of the same dimensions as the other array. In other words, if A is a variable (for example, `x`), and B is an *m*-by-*n* matrix, then A is expanded into *m*-by-*n* matrix of elements, each set to `x`.
- "Set Assumptions" on page 1-32

See Also

`eq` | `ge` | `gt` | `isAlways` | `le` | `logical` | `lt`

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 0s and 1s 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`

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 a 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`

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.

`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 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 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)
```

More About

1-norm of a Matrix

The 1-norm of an m-by-n matrix A is defined as follows:

$$\|A\|_1 = \max_j \left(\sum_{i=1}^{m} |A_{ij}| \right), \text{ where } j = 1 \ldots n$$

2-norm of a Matrix

The 2-norm of an m-by-n matrix A is defined as follows:

$$\|A\|_2 = \sqrt{\max \text{ eigenvalue of } A^H A}$$

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:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\left(\sum_{j=1}^{n}\left|A_{ij}\right|^2\right)}$$

Infinity Norm of a Matrix

The infinity norm of an m-by-n matrix A is defined as follows:

$$\|A\|_\infty = \max\left(\sum_{j=1}^{n}\left|A_{1j}\right|, \sum_{j=1}^{n}\left|A_{2j}\right|, \ldots, \sum_{j=1}^{n}\left|A_{mj}\right|\right)$$

P-norm of a Vector

The P-norm of a 1-by-n or n-by-1 vector V is defined as follows:

$$\|V\|_P = \left(\sum_{i=1}^{n}|V_i|^P\right)^{1/P}$$

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:

$$\|V\|_\infty = \max\left(|V_i|\right), \text{ where } i = 1 \ldots n$$

The negative infinity norm of a 1-by-n or n-by-1 vector V is defined as follows:

$$\|V\|_{-\infty} = \min\left(\left|V_i\right|\right), \text{ where } i = 1 \ldots n$$

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`

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 =
     0

ans =
     1
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)
```

```
s1 =
FALSE

s2 =
TRUE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
```

```
logical(s2)

ans =
     0

ans =
     1
```

More About

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` | `logical` | `or` | `xor`

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

numden

Numerator and denominator

Syntax

```
[N,D] = numden(A)
```

Description

`[N,D] = numden(A)` converts each element of `A` to a rational form where the numerator and denominator are relatively prime polynomials with integer coefficients. `A` is a symbolic or a numeric matrix. `N` is the symbolic matrix of numerators, and `D` is the symbolic matrix of denominators.

Examples

Find the numerator and denominator of the symbolic number:

```
[n, d] = numden(sym(4/5))
```

```
n =
4

d =
5
```

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
```

The statements

```
syms a b
A = [a, 1/b]
[n,d] = numden(A)
```

return

```
A =
[a, 1/b]

n =
[a, 1]

d =
[1, b]
```

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`

odeToVectorField

Convert higher-order differential equations to systems of first-order differential equations

Syntax

```
V = odeToVectorField(eqn1,...,eqnN)
[V,Y] = odeToVectorField(eqn1,...,eqnN)
```

Description

`V = odeToVectorField(eqn1,...,eqnN)` converts higher-order differential equations eqn1,...,eqnN to a system of first-order differential equations. This syntax returns a symbolic vector representing the resulting system of first-order differential equations.

`[V,Y] = odeToVectorField(eqn1,...,eqnN)` converts higher-order differential equations eqn1,...,eqnN to a system of first-order differential equations. This syntax returns two symbolic vectors. The first vector represents the resulting system of first-order differential equations. The second vector shows the substitutions made during conversion.

Input Arguments

`eqn1,...,eqnN`

Symbolic equations, strings separated by commas and representing a system of ordinary differential equations, or array of symbolic equations or strings. Each equation or string represents an ordinary differential equation.

When representing `eqn` as a symbolic equation, you must create a symbolic function, for example `y(x)`. Here `x` is an independent variable for which you solve an ordinary differential equation. Use the `==` operator to create an equation. Use the `diff` function to indicate differentiation. For example, to convert $d^2y(x)/dt^2 = x*y(x)$, use:

```
syms y(x)
V = odeToVectorField(diff(y, 2) == x*y)
```

When representing `eqn` as a string, use the letter `D` to indicate differentiation. By default, `odeToVectorField` assumes that the independent variable is `t`. Thus, `Dy` means `dy/dt`. You can specify the independent variable. The letter `D` followed by a digit indicates repeated differentiation. Any character immediately following a differentiation operator is a dependent variable. For example, to convert $d^2y(x)/dt^2 = x*y(x)$, enter:

```
V = odeToVectorField('D2y = x*y','x')
```

or

```
V = odeToVectorField('D2y == x*y','x')
```

Output Arguments

V

Symbolic vector representing the system of first-order differential equations. Each element of this vector is the right side of the first-order differential equation $Y[i]' = V[i]$.

Y

Symbolic vector representing the substitutions made when converting the input equations eqn1,...,eqnN to the elements of V.

Examples

Convert this fifth-order differential equation to a system of first-order differential equations:

```
syms y(t)
V = odeToVectorField(t^3*diff(y, 5) + 2*t*diff(y, 4) + diff(y, 2) + y^2 == -3*t)
```

```
V =
                                  Y[2]
                                  Y[3]
                                  Y[4]
                                  Y[5]
 -(3*t + Y[1]^2 + 2*t*Y[5] + Y[3])/t^3
```

Convert this system of first- and second-order differential equations to a system of first-order differential equations. To see the substitutions that `odeToVectorField` makes for this conversion, use two output arguments:

```
syms f(t) g(t)
[V,Y] = odeToVectorField(diff(f, 2) == f + g, diff(g) == -f + g)
```

```
V =
 Y[1] - Y[2]
        Y[3]
 Y[1] + Y[2]

Y =
  g
  f
 Df
```

Convert this second-order differential equation to a system of first-order differential equations:

```
syms y(t)
V = odeToVectorField(diff(y, 2) == (1 - y^2)*diff(y) - y)
```

```
V =
                        Y[2]
 - (Y[1]^2 - 1)*Y[2] - Y[1]
```

Generate a MATLAB function from this system of first-order differential equations using `matlabFunction` with V as an input:

```
M = matlabFunction(V,'vars', {'t','Y'})
```

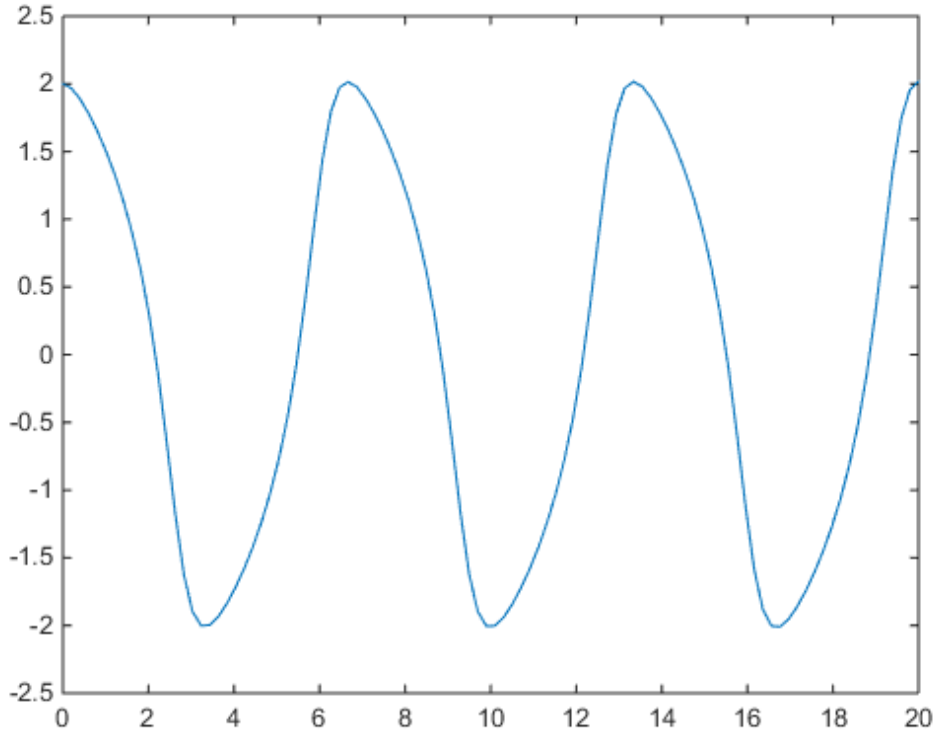
```
M =
    @(t,Y)[Y(2);-(Y(1).^2-1.0).*Y(2)-Y(1)]
```

To solve this system, call the MATLAB `ode45` numerical solver using the generated MATLAB function as an input:

```
sol = ode45(M,[0 20],[2 0]);
```

Plot the solution using `linspace` to generate 100 points in the interval [0,20] and `deval` to evaluate the solution for each point:

```
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y)
```



Convert the second-order differential equation $y''(x) = x$ with the initial condition $y(0) = t$ to a system. Specify the differential equation and initial condition as strings. Also specify that `x` is an independent variable:

```
V = odeToVectorField('D2y = x', 'y(0) = t', 'x')
```

```
V =
 Y[2]
    x
```

If you define equations by strings and do not specify the independent variable, `odeToVectorField` assumes that the independent variable is `t`. This assumption makes the equation $y''(t) = x$ inconsistent with the initial condition $y(0) = t$. In this case, $y''(t) = d^2t/dt^2 = 0$, and `odeToVectorField` errors.

More About

Tips

- The names of symbolic variables used in differential equations should not contain the letter `D` because `odeToVectorField` assumes that `D` is a differential operator and any character immediately following `D` is a dependent variable.
- To generate a MATLAB function for the resulting system of first-order differential equations, use `matlabFunction` with V as an input. Then, you can use the generated MATLAB function as an input for the MATLAB numerical solvers `ode23` and `ode45`.
- The highest-order derivatives must appear in eqn1,...,eqnN linearly. For example, `odeToVectorField` can convert these equations:

  - $y''(t) = -t^2$
  - $y*y''(t) = -t^2$. `odeToVectorField` can convert this equation because it can be rewritten as $y''(t) = -t^2/y$.

  However, it cannot convert these equations:

  - $y''(t)^2 = -t^2$
  - $\sin(y''(t)) = -t^2$

Algorithms

To convert an nth-order differential equation

$$a_n(t)y^{(n)} + a_{n-1}(t)y^{(n-1)} + a_{n-2}(t)y^{(n-2)} + \ldots + a_2(t)y'' + a_1(t)y' + a_0(t)y + r(t) = 0$$

into a system of first-order differential equations, make these substitutions:

$$\begin{aligned}
&Y_1 = y \\
&Y_2 = y' \\
&Y_3 = y'' \\
&\ldots \\
&Y_{n-1} = y^{(n-2)} \\
&Y_n = y^{(n-1)}
\end{aligned}$$

Using the new variables, you can rewrite the equation as a system of n first-order differential equations:

$$
\begin{aligned}
&Y_1' = y' = Y_2 \\
&Y_2' = y'' = Y_3 \\
&\dots \\
&Y_{n-1}' = y^{(n-1)} = Y_n \\
&Y_n' = -\frac{a_{n-1}(t)}{a_n(t)}Y_n - \frac{a_{n-2}(t)}{a_n(t)}Y_{n-1} - \dots - \frac{a_1(t)}{a_n(t)}Y_2 - \frac{a_0(t)}{a_n(t)}Y_1 + \frac{r(t)}{a_n(t)}
\end{aligned}
$$

`odeToVectorField` returns the right sides of these equations as the elements of vector `V`.

When you convert a system of higher-order differential equations to a system of first-order differential equations, it can be helpful to see the substitutions that `odeToVectorField` made during the conversion. These substitutions are listed as elements of vector `Y`.

See Also

`dsolve` | `matlabFunction` | `ode23` | `ode45` | `syms`

openmn

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');
```

More About

- “Create MuPAD Notebooks”
- “Open MuPAD Notebooks”

See Also

`mupad` | `open` | `openmu` | `openxvc` | `openxvz`

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.

More About

- "Open MuPAD Notebooks"

See Also

`mupad` | `open` | `openmn` | `openxvc` | `openxvz`

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')
```

More About

- "Open MuPAD Notebooks"

See Also

`mupad` | `open` | `openmn` | `openmu` | `openxvz`

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')
```

More About

- "Open MuPAD Notebooks"

See Also

`mupad` | `open` | `openmn` | `openmu` | `openxvc`

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 `|`:

```
syms x
range = x < -1 | x > 1;
```

Replace variable `x` with these numeric values. If you replace `x` with 10, one inequality is valid. If you replace `x` with 0, both inequalities are invalid. Note that `subs` does not evaluate these inequalities to logical `1` or `0`.

```
x1 = subs(range, x, 10)
x2 = subs(range, x, 0)
```

```
x1 =
1 < 10 | 10 < -1
x2 =
0 < -1 | 1 < 0
```

To evaluate these inequalities to logical `1` or `0`, use `logical` or `isAlways`:

```
logical(x1)
isAlways(x2)
```

```
ans =
     1

ans =
     0
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)
```

```
s1 =
TRUE
```

```
s2 =
FALSE
```

Convert symbolic TRUE or FALSE to logical values using `logical`:

```
logical(s1)
logical(s2)
```

```
ans =
     1

ans =
     0
```

More About

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values TRUE or FALSE. These values are not the same as logical 1 (`true`) and logical 0 (`false`). To convert symbolic TRUE or FALSE to logical values, use `logical`.

See Also

`all` | `and` | `any` | `isAlways` | `logical` | `not` | `xor`

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 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)]
```

More About

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

`linalg::normalize` | `linalg::orthog` | `norm` | `null` | `orth` | `rank` | `svd`

pade

Padé approximant

Syntax

```
pade(f)
pade(f,x)
pade(f,x,a)
pade(___,Name,Value)
```

Description

`pade(f)` returns the third-order Padé approximant of the expression f at `x = 0`, where `symvar` determines x. For details, see "Padé Approximant" on page 4-841.

`pade(f,x)` returns the third-order Padé approximant of the expression f at `x = 0`.

`pade(f,x,a)` returns the third-order Padé approximant of expression `f` at `x = a`.

`pade(___,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Examples

Find the 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 the 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 the Value of a Function at a 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 =
-1.215851878956908644724488132684 2
```

Increase the Accuracy of the 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 =
0.00000733280596978061865556894483177 99
```

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 4-841. 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
```

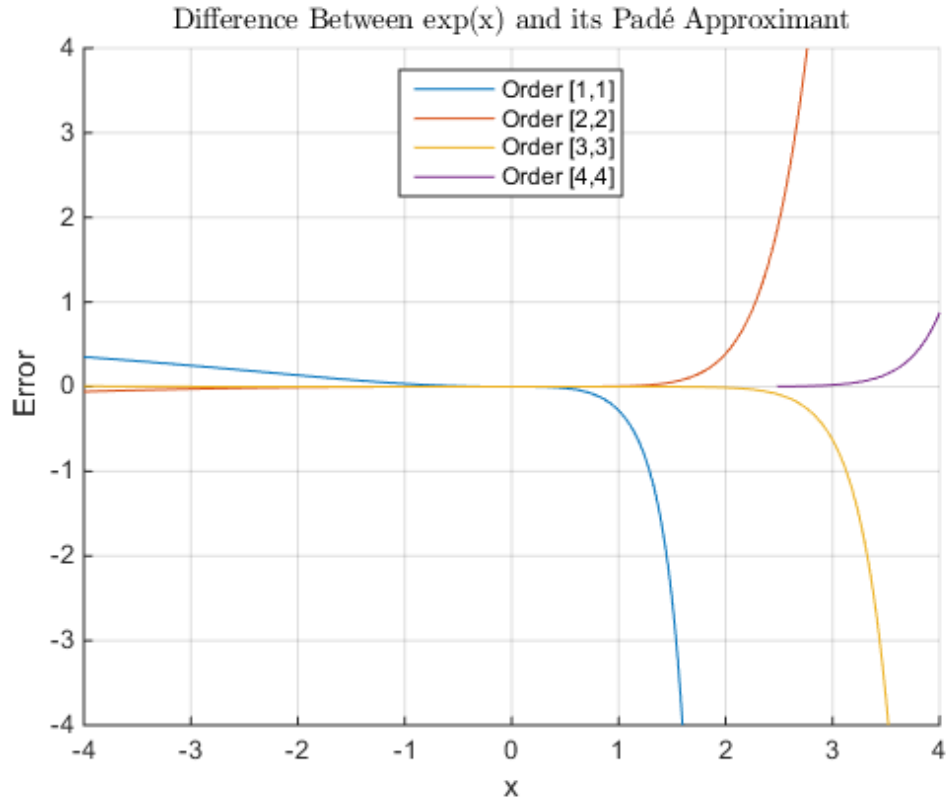
```
p22Rel =
(x*(x^2 - 15))/(3*(2*x^2 - 5))
error =
0.00000000840840148061133117137653177259 98
```

The accuracy increases if the expansion point is a pole or zero and `OrderMode` is set to `Relative`.

Plot the Accuracy of the Padé Approximant

Plot the difference between `exp(x)` and its Padé 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 Padé approximant.

```
syms x
expr = exp(x);

hold on
grid on

for i = 1:4
    ezplot(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 Pad\''{e} Approximant',...
                                               'interpreter','latex')
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.

`x` — Expansion variable

symbolic variable

Expansion variable, specified as a symbolic variable.

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 can also specify the expansion point as a Name,Value pair argument. If the expansion point is specified in 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 the expansion point is specified in 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' — Alternative form of Padé approximant

absolute (default) | relative

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 `(x-a)^p` where `p` is the multiplicity of the zero or pole at the expansion point. For details, see "Padé Approximant" on page 4-841.

More About

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 specify the expansion point both as the input argument a and as a Name,Value pair argument, the Name,Value pair argument is used to for the expansion point.

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 $\pm\infty$ and $\pm i\infty$ 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

`taylor`

Related Examples

- "Padé Approximant"

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 + i/32,     1/24 - i/32]
[ - 7/32 - (5*i)/96, 5/16 + (7*i)/96]
[       7/24 + i/16, 1/96 - (3*i)/32]
```

Check that `A*X*A = A` and `X*A*X = X`:

```
logical(A*X*A == A)
```

```
ans =
     1     1     1
     1     1     1
```

```
logical(X*A*X == X)
```

```
ans =
     1     1
     1     1
     1     1
```

Now, verify that `A*X` and `X*A` are Hermitian matrices:

```
logical(A*X == (A*X)')
```

```
ans =
     1     1
     1     1
```

```
logical(X*A == (X*A)')
```

```
ans =
     1     1     1
     1     1     1
     1     1     1
```

Compute the pseudoinverse of this matrix:

```
syms a
A = [1 a; -a 1];
```

```
X = pinv(A)
```

```
X =
[ (a*conj(a) + 1)/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a - conj(a)))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1),
- (a - conj(a))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a*conj(a) + 1))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1)]
[ (a - conj(a))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) +...
(conj(a)*(a*conj(a) + 1))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1),
(a*conj(a) + 1)/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1) -...
(conj(a)*(a - conj(a)))/(a^2*conj(a)^2 + a^2 + conj(a)^2 + 1)]
```

Now, compute the pseudoinverse of `A` assuming that `a` is real:

```
assume(a,'real')
A = [1 a; -a 1];
X = pinv(A)
```

```
X =
[ 1/(a^2 + 1), -a/(a^2 + 1)]
[ a/(a^2 + 1),  1/(a^2 + 1)]
```

For further computations, remove the assumption:

```
syms a clear
```

More About

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`

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 a Scalar to an 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 an Expression to a 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.

More About

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

`minus`

pochhammer

Pochhammer symbol

Syntax

```
pochhammer(x,n)
```

Description

`pochhammer(x,n)` returns the "Pochhammer Symbol" on page 4-853 $(x)_n$.

Examples

Find the 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 the Output 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 the 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 the Pochhammer Symbol

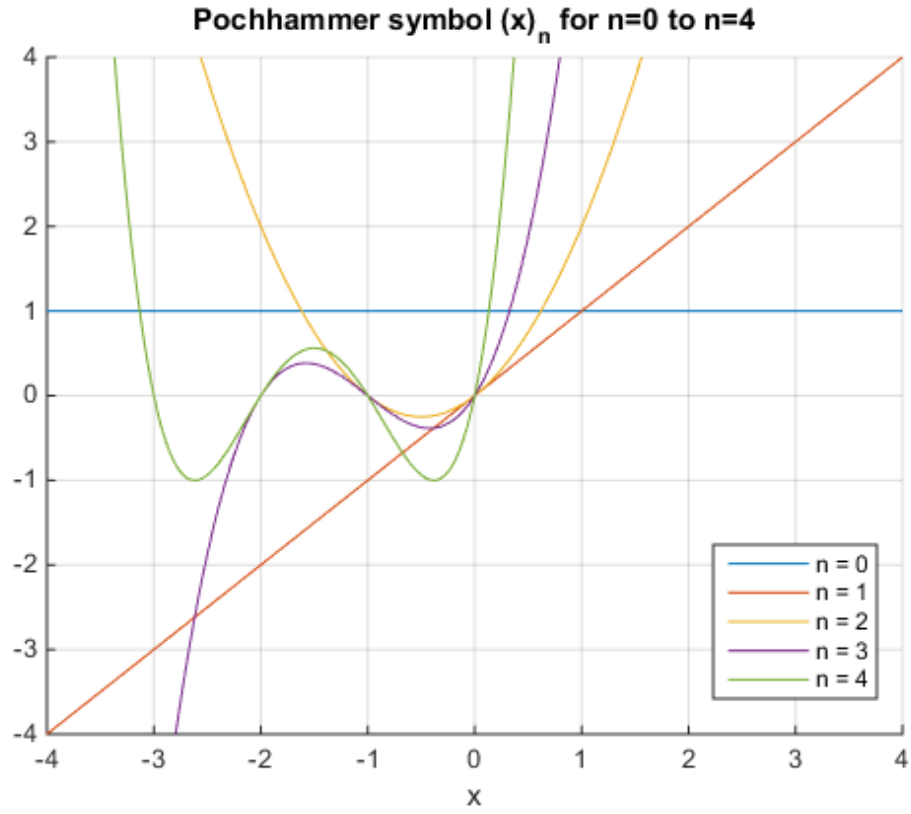
Use `taylor` to find the Taylor series expansion of `pochhammer` with `n = 3` around the expansion point `x = 2`.

```
syms x
taylor(pochhammer(x,3),x,2)

ans =
26*x + 9*(x - 2)^2 + (x - 2)^3 - 28
```

Plot the Pochhammer Symbol

Plot the Pochhammer symbol from `n = 0` to `n = 4` for `x`. Use `axis` to display the region of interest.

```
syms x
hold on
for n = 0:4
    ezplot(pochhammer(x,n))
end
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.

More About

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`

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 =
i

ans =
1
```

Find the poles of this expression. If you do not specify a variable, `poles` uses the default variable determined by `symvar`:

```
syms x a
poles(1/((x - 1)*(a - 2)))

ans =
1
```

To find the poles of this expression as a function of variable `a`, specify `a` as the second argument:

```
syms x a
poles(1/((x - 1)*(a - 2)), a)

ans =
2
```

Find the poles of the tangent function in the interval `(-pi, pi)`:

```
syms x
poles(tan(x), x, -pi, pi)

ans =
 -pi/2
  pi/2
```

The tangent function has an infinite number of poles. If you do not specify the interval, `poles` cannot find all of them. It issues a warning and returns an empty symbolic object:

```
syms x
poles(tan(x))

Warning: Cannot determine the poles.
```

```
ans =
[ empty sym ]
```

If `poles` can prove that the expression or function does not have any poles in the specified interval, it returns an empty symbolic object without issuing a warning:

```
syms x
poles(tan(x), x, -1, 1)
```

```
ans =
Empty sym: 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
```

More About

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`

poly2sym

Polynomial coefficient vector to symbolic polynomial

Syntax

```
r = poly2sym(c)
r = poly2sym(c,v)
```

Description

`r = poly2sym(c)` returns a symbolic representation of the polynomial whose coefficients form the numeric vector `c`. The default symbolic variable is `x`. The variable `v` can be specified as a second input argument. If `c = [c1 c2 ... cn]`, `r = poly2sym(c)` has the form

$$c_1 x^{n-1} + c_2 x^{n-2} + \ldots + c_n$$

`poly2sym` uses `sym`'s default (rational) conversion mode to convert the numeric coefficients to symbolic constants. This mode expresses the symbolic coefficient approximately as a ratio of integers, if `sym` can find a simple ratio that approximates the numeric value, otherwise as an integer multiplied by a power of 2.

`r = poly2sym(c,v)` is a polynomial in the symbolic variable `v` with coefficients from the vector `c`. If `v` has a numeric value and `sym` expresses the elements of `c` exactly, `eval(poly2sym(c))` returns the same value as `polyval(c, v)`.

Examples

The command

```
poly2sym([1 3 2])
```

returns

```
ans =
```

```
x^2 + 3*x + 2
```

The command

```
poly2sym([.694228, .333, 6.2832])
```

returns

```
ans =
(6253049924220329*x^2)/9007199254740992 +...
(333*x)/1000 + 3927/625
```

The command

```
poly2sym([1 0 1 -1 2], y)
```

returns

```
ans =
y^4 + y^2 - y + 2
```

See Also

sym | sym2poly | polyval

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))]
```

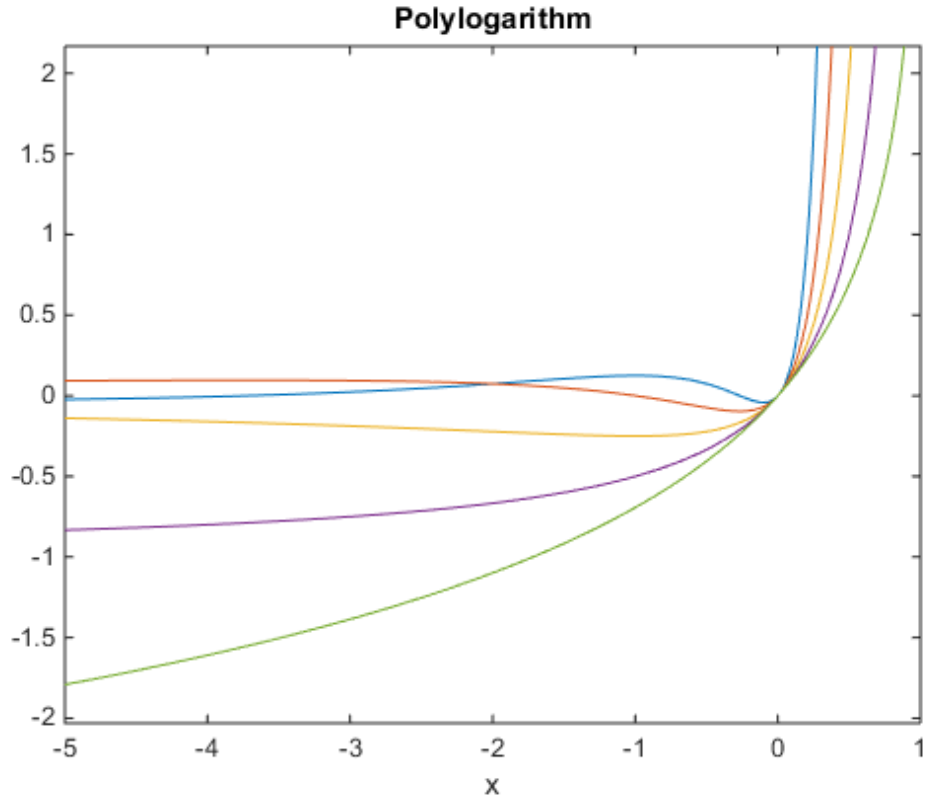
```
ans =
[ pi^4/90, -(15*zeta(5))/16, catalan*i - pi^2/48]
```

Plot the Polylogarithm

Plot the polylogarithms of the orders from -3 to 1.

```
syms x
for n = -3:1
  ezplot(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.

More About

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 $[1, \infty)$ for $n \geq 1$.

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`

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
```

More About

Scalar Potential of a Gradient Vector Field

The potential of a gradient vector field $V(X) = [v_1(x_1,x_2,...),v_2(x_1,x_2,...),...]$ is the scalar $P(X)$ such that $V(X) = \nabla P(X)$.

The vector field is gradient if and only if the corresponding Jacobian is symmetrical:

$$\left(\frac{\partial v_i}{\partial x_j}\right) = \left(\frac{\partial v_j}{\partial x_i}\right)$$

The `potential` function represents the potential in its integral form:

$$P(X) = \int_0^1 (X - Y) \cdot V\left(Y + \lambda (X - Y)\right) d\lambda$$

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`

pretty

Prettyprint symbolic expressions

Syntax

```
pretty(X)
```

Description

`pretty(X)` prints symbolic output of `X` in a format that resembles typeset mathematics.

Examples

The following statements:

```
A = sym(pascal(2))
B = eig(A)
pretty(B)
```

return:

```
A =
[ 1, 1]
[ 1, 2]

B =

 3/2 - 5^(1/2)/2
 5^(1/2)/2 + 3/2

/ 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);
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 | i
                   \ 9 #2      /
   #1 == -----------------------
                      2

          / sqrt(11) sqrt(27)   17 \1/3
   #2 == | ----------------- - -- |
          \         27          27 /
```

psi

Digamma function

Syntax

```
psi(x)
psi(k,x)
```

Description

`psi(x)` computes the digamma function of x.

`psi(k,x)` computes the polygamma function of x, which is the kth 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]
```

More About

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` | `gamma` | `nchoosek` | `factorial`

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 `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 the 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 the 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:

```
A == Q*R
```

```
ans =
     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 a 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`:

```
A*X == b
```

```
ans =
     1
     1
     1
     1
     1
```

Use QR Decomposition with Permutation Information to Solve a 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

symbolic matrix

Permutation information, returned as a symbolic matrix, such that `A*P = Q*R`.

p — Permutation information

symbolic vector

Permutation information, returned as a symbolic vector, 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`.

More About

QR Factorization of a 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`

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(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`

rank

Compute rank of symbolic matrix

Syntax

```
rank(A)
```

Description

`rank(A)` computes the rank of the symbolic matrix `A`.

Examples

Compute the rank of the following numeric matrix:

```
B = magic(4);
rank(B)
```

```
ans =
     3
```

Compute the rank of the following symbolic matrix:

```
syms a b c d
A = [a b;c d];
rank(A)
```

```
ans =
2
```

See Also

`eig` | `null` | `rref` | `size`

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.

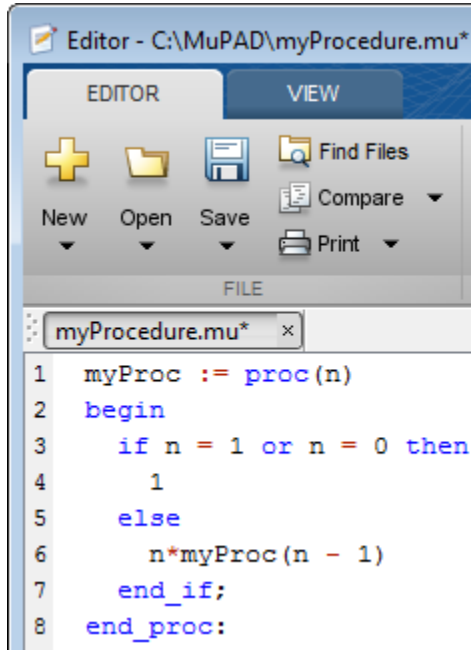
Input Arguments

`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`.



Before you can call this procedure at the MATLAB Command Window, you must read the file `myProcedure.mu` into the symbolic engine. To read a program file into the symbolic engine, use `read`:

```
read(symengine, 'myProcedure.mu')
```

If the file is not on the MATLAB path, specify the full path to this file. For example, if `myProcedure.mu` is in the `MuPAD` folder on disk `C`, enter:

```
read(symengine, 'C:/MuPAD/myProcedure.mu')
```

Now you can access the procedure `myProc` using `evalin` or `feval`. For example, compute the factorial of 10:

```
feval(symengine, 'myProc', 10)

ans =
3628800
```

Alternatives

You also can use `feval` to call the MuPAD `read` function. The `read` function available from the MATLAB Command Window is equivalent to calling the MuPAD `read` function with the `Plain` option. It ignores any MuPAD aliases defined in the program file:

```
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" ')
```

More About

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 4-895.

- “Use Your Own MuPAD Procedures” on page 3-38
- “Conflicts Caused by Syntax Conversions” on page 3-29

See Also

`evalin` | `feval` | `symengine`

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(sym('2*exp(1 + i)'))

ans =
2*cos(1)*exp(1)
```

In general, `real` cannot extract the entire real parts from symbolic expressions containing variables. However, `real` can rewrite and sometimes simplify the input expression:

```
syms a x y
real(a + 2)
real(x + y*i)

ans =
real(a) + 2

ans =
real(x) - imag(y)
```

If you assign numeric values to these variables or specify that these variables are real, `real` can extract the real part of the expression:

```
syms a
a = 5 + 3*i;
real(a + 2)

ans =
     7

syms x y real
real(x + y*i)

ans =
x
```

Clear the assumption that `x` and `y` are real:

```
syms x y clear
```

Find the real parts of the elements of matrix A:

```
A = sym('[-1 + i, sinh(x); exp(10 + 7*i), exp(pi*i)]');
real(A)
```

```
ans =
[             -1, real(sinh(x))]
[ cos(7)*exp(10),            -1]
```

Alternatives

You can compute the real part of z via the conjugate: `real(z)= (z + conj(z))/2`.

More About

Tips

- Calling `real` for a number that is not a symbolic object invokes the MATLAB `real` function.

See Also

`conj` | `imag` | `in` | `sign` | `signIm`

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

Compute the rectangular pulse function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[rectangularPulse(-1, 1, -2)
rectangularPulse(-1, 1, -1)
rectangularPulse(-1, 1, 0)
rectangularPulse(-1, 1, 1)
rectangularPulse(-1, 1, 2)]
```

```
ans =
         0
    0.5000
    1.0000
    0.5000
         0
```

Compute the rectangular pulse function for the numbers converted to symbolic objects:

```
[rectangularPulse(sym(-1), 1, -2)
rectangularPulse(-1, sym(1), -1)
rectangularPulse(-1, 1, sym(0))
rectangularPulse(sym(-1), 1, 1)
rectangularPulse(sym(-1), 1, 2)]
```

```
ans =
   0
 1/2
   1
 1/2
   0
```

If `a < b`, the rectangular pulse function for `x = a` and `x = b` equals `1/2`:

```
syms a b x
assume(a < b)
rectangularPulse(a, b, a)
rectangularPulse(a, b, b)
```

```
ans =
1/2

ans =
```

```
1/2
```

For further computations, remove the assumption:

```
syms a b clear
```

For `a = b`, the rectangular pulse function returns `0`:

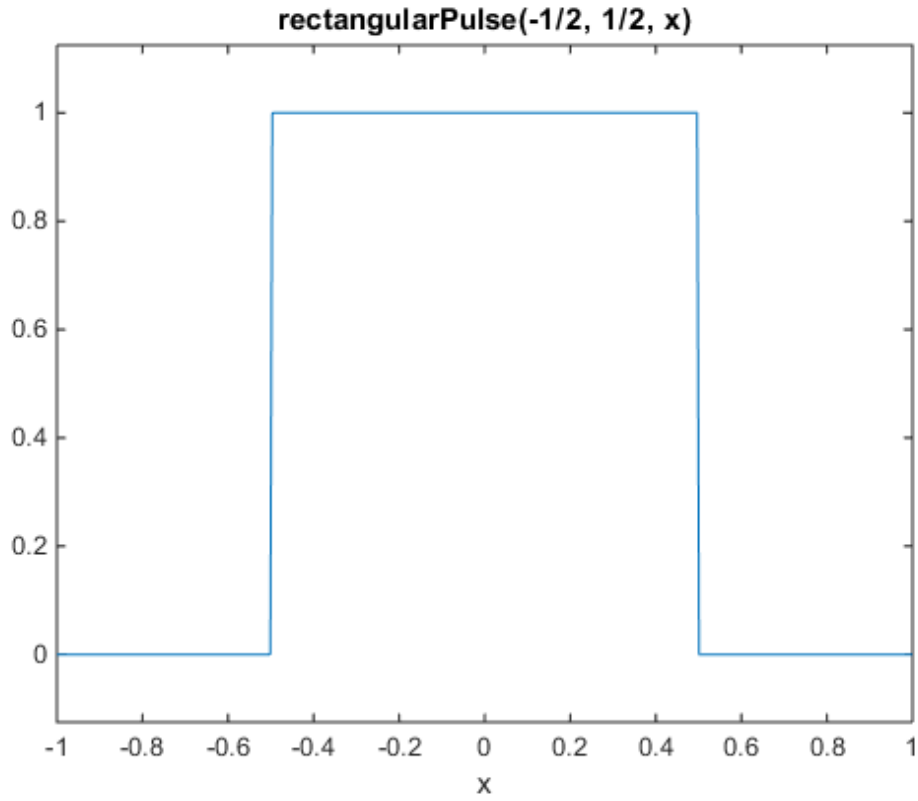
```
syms a x
rectangularPulse(a, a, x)

ans =
0
```

Use `rectangularPulse` with one input argument as a shortcut for computing `rectangularPulse(-1/2, 1/2, x)`:

```
syms x
rectangularPulse(x)

ans =
rectangularPulse(-1/2, 1/2, x)

[rectangularPulse(sym(-1))
rectangularPulse(sym(-1/2))
rectangularPulse(sym(0))
rectangularPulse(sym(1/2))
rectangularPulse(sym(1))]

ans =
   0
 1/2
   1
 1/2
   0
```

Plot the rectangular pulse function:

```
syms x
ezplot(rectangularPulse(x), [-1, 1])
```



Call `rectangularPulse` with infinities as its rising and falling edges:

```
syms x
rectangularPulse(-inf, 0, x)
rectangularPulse(0, inf, x)
rectangularPulse(-inf, inf, x)

ans =
heaviside(-x)

ans =
heaviside(x)

ans =
1
```

More About

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`

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 a 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 =
     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 =
     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*Dxt(t)^2 + 2*Dyt(t)^2 + 2*y(t)*diff(Dyt(t), t)
                                             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`.

More About

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` | `reduceDAEToODE` | `reduceDifferentialOrder` | `reduceRedundancies`

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 the 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 =
     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: The index of the reduced DAEs is larger...
than 1. [daetools::reduceDAEIndex]

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 =
 D(q2)(t) - D(D(q3))(t) - q1(t) + x1(t)
                x3(t) - q3(t) + t*x2(t)
              D(q3)(t) - q2(t) + x2(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 =
```

```
 D(q2)(t) - D(D(q3))(t) - q1(t) + x1(t)
                x3(t) - q3(t) + t*x2(t)
              D(q3)(t) - q2(t) + x2(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.

More About

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` | `reduceDAEIndex` | `reduceDifferentialOrder` | `reduceRedundancies`

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 a 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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceRedundancies`

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 =
      solvedEquations: [2x1 sym]
    constantVariables: [1x2 sym]
    replacedVariables: [1x2 sym]
       otherEquations: [1x1 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` | `reduceDAEIndex` | `reduceDAEToODE` | `reduceDifferentialOrder`

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.

More About

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`

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`

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 the Symbolic Row Vector into a 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 the 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 the Dimension of the 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 a 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 the 3-D Array into a 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`

rewrite

Rewrite expression in new terms

Syntax

```
rewrite(expr,target)
rewrite(A,target)
```

Description

`rewrite(expr,target)` rewrites the symbolic expression expr in terms of target. The returned expression is mathematically equivalent to the original expression.

`rewrite(A,target)` rewrites each element of A in terms of target.

Input Arguments

`expr`

Symbolic expression.

`A`

Vector or matrix of symbolic expressions.

`target`

One of these strings: `exp`, `log`, `sincos`, `sin`, `cos`, `tan`, `sqrt`, or `heaviside`.

Examples

Rewrite these trigonometric functions in terms of the exponential function:

```
syms x
rewrite(sin(x), 'exp')
rewrite(cos(x), 'exp')
```

```
rewrite(tan(x), 'exp')

ans =
(exp(-x*i)*i)/2 - (exp(x*i)*i)/2

ans =
exp(-x*i)/2 + exp(x*i)/2

ans =
-(exp(x*2*i)*i - i)/(exp(x*2*i) + 1)
```

Rewrite the tangent function in terms of the sine function:

```
syms x
rewrite(tan(x), 'sin')

ans =
-sin(x)/(2*sin(x/2)^2 - 1)
```

Rewrite the hyperbolic tangent function in terms of the sine function:

```
syms x
rewrite(tanh(x), 'sin')

ans =
(sin(x*i)*i)/(2*sin((x*i)/2)^2 - 1)
```

Rewrite these inverse trigonometric functions in terms of the natural logarithm:

```
syms x
rewrite(acos(x), 'log')
rewrite(acot(x), 'log')

ans =
-log(x + (1 - x^2)^(1/2)*i)*i

ans =
(log(1 - i/x)*i)/2 - (log(i/x + 1)*i)/2
```

Rewrite the rectangular pulse function in terms of the Heaviside step function:

```
syms a b x
rewrite(rectangularPulse(a, b, x), 'heaviside')

ans =
heaviside(x - a) - heaviside(x - b)
```

Rewrite the triangular pulse function in terms of the Heaviside step function:

```
syms a b c x
rewrite(triangularPulse(a, b, c, x), 'heaviside')
```

```
ans =
(heaviside(x - a)*(a - x))/(a - b) - (heaviside(x - b)*(a - x))/(a - b)...
 - (heaviside(x - b)*(c - x))/(b - c) + (heaviside(x - c)*(c - x))/(b - c)
```

Call `rewrite` to rewrite each element of this matrix of symbolic expressions in terms of the exponential function:

```
syms x
A = [sin(x) cos(x); sinh(x) cosh(x)];
rewrite(A, 'exp')
```

```
ans =
[ (exp(-x*i)*i)/2 - (exp(x*i)*i)/2, exp(-x*i)/2 + exp(x*i)/2]
[             exp(x)/2 - exp(-x)/2,     exp(-x)/2 + exp(x)/2]
```

Rewrite the cosine function in terms of sine function. Here `rewrite` replaces the cosine function using the identity `cos(2*x) = 1 – 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 the sine function with either $-\sqrt{1-\cos^2(x)}$ or $\sqrt{1-\cos^2(x)}$ because these expressions are only valid for `x` within particular intervals:

```
syms x
rewrite(sin(x),'cos')
```

```
ans =
sin(x)
```

More About

Tips

- `rewrite` replaces symbolic function calls in expr with the target function only if such replacement is mathematically valid. Otherwise, it keeps the original function calls.

- “Simplifications” on page 2-40

See Also

`collect` | `combine` | `expand` | `factor` | `horner` | `numden` | `simplify` | `simplifyFraction`

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

`floor` | `ceil` | `fix` | `frac`

rref

Compute reduced row echelon form of matrix

Syntax

```
rref(A)
```

Description

`rref(A)` computes the reduced row echelon form of the symbolic matrix `A`. If the elements of a matrix contain free symbolic variables, `rref` regards the matrix as nonzero.

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` | `rank` | `size` | `linsolve`

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 string or a symbolic expression. The height of each rectangle is determined by the value of the function in the middle of each interval.

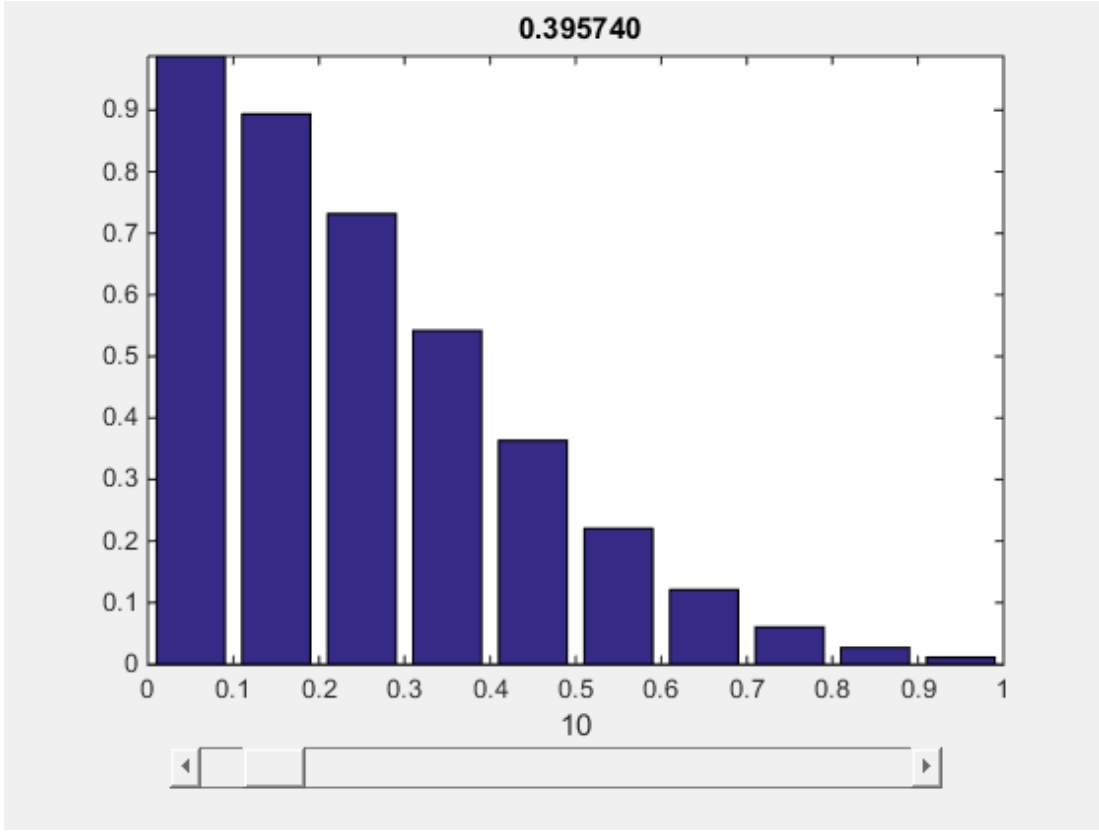
`rsums(f,a,b)` and `rsums(f,[a,b])` approximates the integral for *x* from `a` to `b`.

Examples

Visualize Riemann Sums

Use `rsums('exp(-5*x^2)')` or `rsums exp(-5*x^2)` to create the following plot.

```
rsums exp(-5*x^2)
```


0.395740
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
10

sec

Symbolic secant function

Syntax

```
sec(X)
```

Description

`sec(X)` returns the secant function 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)]
```

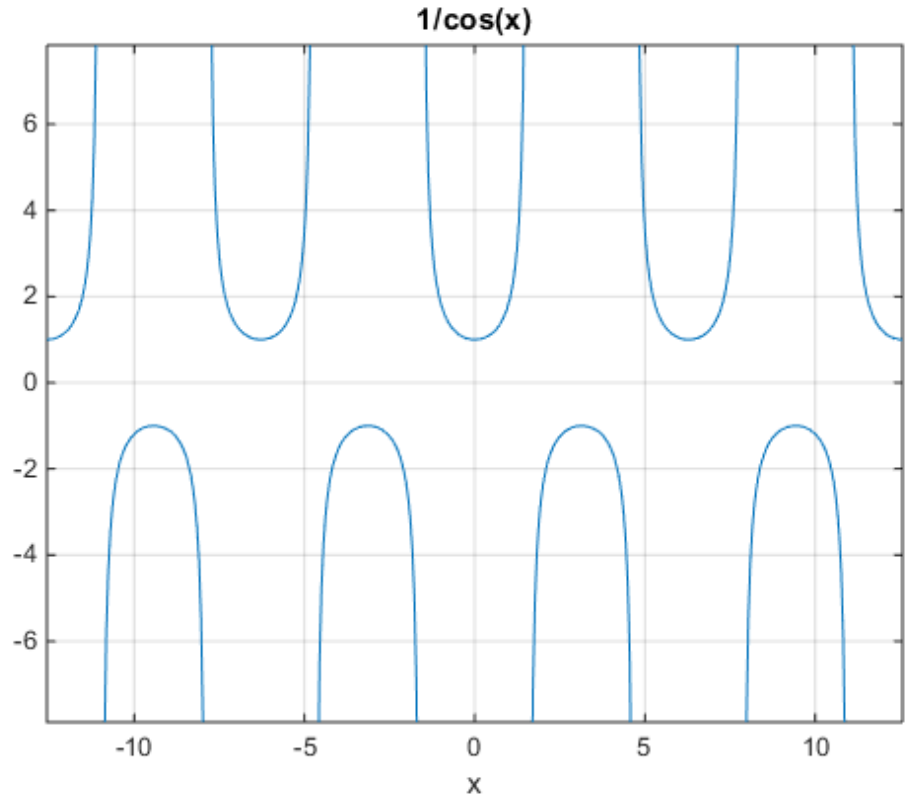
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -2.402997961722380989754600401420 1,...
-1.0,...
1.154700538379251529018297561003 9,...
-1.603875471609676504944409278029 8,...
225.953059314024932690375427035 57]
```

Plot the Secant Function

Plot the secant function on the interval from -4π to 4π.

```
syms x
ezplot(sec(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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*i)/2 + exp(x*i)/2)
```

Input Arguments

X — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

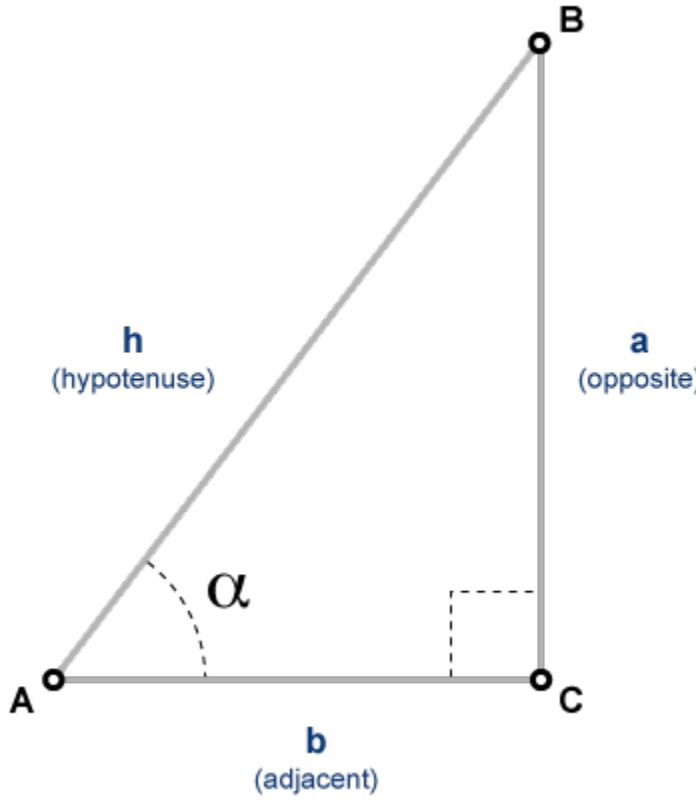
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

More About

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

$$\text{secant}(\alpha) = \frac{2}{e^{i\alpha} + e^{-i\alpha}}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sin` | `tan`

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*2*i)/7), 1/cosh(1)]
```

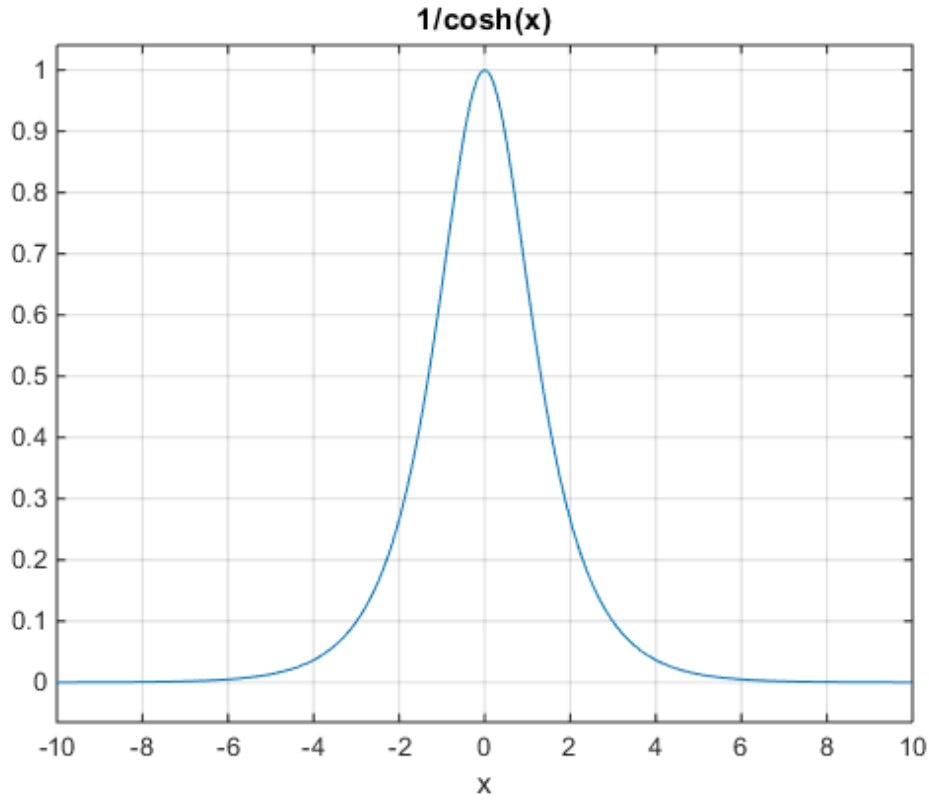
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 0.26580222883407969212086273981989,...
-1.0,...
1.1547005383792515290182975610039,...
1.0,...
2.0,...
-1.6038754716096765049444092780298,...
0.64805427366388539957497735322615]
```

Plot the Hyperbolic Secant Function

Plot the hyperbolic secant function on the interval from -10 to 10.

```
syms x
ezplot(sech(x), [-10, 10])
grid on
```



Handle Expressions Containing the 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`

setVar

Assign variable in MuPAD notebook

Compatibility

`setvar(nb,MATLABvar)` will be removed in a future release. Use the three argument version `setvar(nb,'MuPADvar',MATLABexpr)` instead.

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 a 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 a MATLAB Symbolic Expression to a 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 3-24

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`

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 + (2*i)/5),                              -1]
[   2^(1/2)*(- 1/2 + i/2), 5^(1/2)*18^(1/2)*(1/30 - i/10)]
```

Sign of Symbolic Expression

Find the sign of this expression assuming that the value `x` is negative:

```
syms x
assume(x < 0)
sign(5*x^3)

ans =
-1
```

For further computations, clear the assumption:

```
syms x clear
```

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.

More About

Sign Function

The sign function of any number z is defined via the absolute value of z:

$$\mathrm{sign}(z) = \frac{z}{|z|}$$

Thus, the sign function of a real number z can be defined as follows:

$$\mathrm{sign}(z) = \begin{cases} -1 \text{ if } x < 0 \\ 0 \text{ if } x = 0 \\ 1 \text{ if } x > 0 \end{cases}$$

Tips

- Calling `sign` for a number that is not a symbolic object invokes the MATLAB `sign` function.

See Also

`abs` | `angle` | `imag` | `real` | `signIm`

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(i/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*i), signIm(x - 3*i)]
```

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.

More About

Tips

- `signIm(NaN)` returns `NaN`.

See Also

`conj` | `imag` | `real` | `sign`

simple

Search for simplest form of symbolic expression

Note: `simple` will be removed in a future release. Use `simplify(S)` instead of `simple(S)`. There is no replacement for `[r, how] = simple(S)`.

Syntax

```
simple(S)
simple(S,Name,Value)
r = simple(S)
r = simple(S,Name,Value)
[r,how] = simple(S)
[r,how] = simple(S,Name,Value)
```

Description

`simple(S)` applies different algebraic simplification functions and displays all resulting forms of `S`, and then returns the shortest form.

`simple(S,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`r = simple(S)` tries different algebraic simplification functions without displaying the results, and then returns the shortest form of `S`.

`r = simple(S,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`[r,how] = simple(S)` tries different algebraic simplification functions without displaying the results, and then returns the shortest form of `S` and a string describing the corresponding simplification method.

`[r,how] = simple(S,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Input Arguments

`S`

Symbolic expression or symbolic matrix.

Default: false

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'IgnoreAnalyticConstraints'`

If the value is `true`, apply purely algebraic simplifications to an expression. With `IgnoreAnalyticConstraints`, `simple` can return simpler results for expressions for which it would return more complicated results otherwise. Using `IgnoreAnalyticConstraints` also can lead to results that are not equivalent to the initial expression.

Default: `false`

Output Arguments

`r`

A symbolic object representing the shortest form of S

`how`

A string describing the simplification method that gives the shortest form of S

More About

Tips

- Simplification of mathematical expression is not a clearly defined subject. There is no universal idea as to which form of an expression is simplest. The form of a mathematical expression that is simplest for one problem might turn out to be complicated or even unsuitable for another problem.
- If S is a matrix, the result represents the shortest representation of the entire matrix, which is not necessarily the shortest representation of each individual element.

Algorithms

When you use `IgnoreAnalyticConstraints`, `simple` applies these rules:

- $\log(a) + \log(b) = \log(a \cdot b)$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$(a \cdot b)^c = a^c \cdot b^c.$$

- $\log(a^b) = b \cdot \log(a)$ for all values of a and b. In particular, the following equality is valid for all values of a, b, and c:

  $$(a^b)^c = a^{b \cdot c}.$$

- If f and g are standard mathematical functions and $f(g(x)) = x$ for all small positive numbers, $f(g(x)) = x$ is assumed to be valid for all complex x. In particular:

  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

- "Simplifications" on page 2-40

See Also

`collect` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify`

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 Elements of a Symbolic Matrix

Call `simplify` for this symbolic matrix. When the input argument is a vector or matrix, `simplify` tries to find a simpler form of each element of the vector or matrix.

```
syms x
simplify([(x^2 + 5*x + 6)/(x + 2),...
sin(x)*sin(2*x) + cos(x)*cos(2*x);
(exp(-x*i)*i)/2 - (exp(x*i)*i)/2, sqrt(16)])
```

```
ans =
[  x + 3, cos(x)]
[ sin(x),      4]
```

Get Simpler Results Using IgnoreAnalyticConstraints

Try to simplify this expression. By default, `simplify` does not combine powers and logarithms because combining them is not valid for generic complex values.

```
syms x
s = (log(x^2 + 2*x + 1) - log(x + 1))*sqrt(x^2);
simplify(s)
```

```
ans =
-(log(x + 1) - log((x + 1)^2))*(x^2)^(1/2)
```

To apply the simplification rules that let the `simplify` function combine powers and logarithms, set `IgnoreAnalyticConstraints` to `true`:

```
simplify(s, 'IgnoreAnalyticConstraints', true)
```

```
ans =
x*log(x + 1)
```

Get Simpler Results Using Steps

Simplify this expression:

```
syms x
f = ((exp(-x*i)*i)/2 - (exp(x*i)*i)/2)/(exp(-x*i)/2 + ...
                                        exp(x*i)/2);
simplify(f)
```

```
ans =
-(exp(x*2*i)*i - i)/(exp(x*2*i) + 1)
```

By default, `simplify` uses one internal simplification step. You can get different, often shorter, simplification results by increasing the number of simplification steps:

```
simplify(f, 'Steps', 10)
simplify(f, 'Steps', 30)
```

```
simplify(f, 'Steps', 50)
```

```
ans =
(2*i)/(exp(x*2*i) + 1) - i

ans =
((cos(x) - sin(x)*i)*i)/cos(x) - i

ans =
tan(x)
```

Simplify Favoring Real Numbers

To force `simplify` favor real values over complex values, set the value of `Criterion` to `preferReal`:

```
syms x
f = (exp(x + exp(-x*i)/2 - exp(x*i)/2)*i)/2 -...
    (exp(- x - exp(-x*i)/2 + exp(x*i)/2)*i)/2;
simplify(f, 'Criterion','preferReal', 'Steps', 100)
```

```
ans =
cos(sin(x))*sinh(x)*i + sin(sin(x))*cosh(x)
```

If `x` is a real value, then this form of expression explicitly shows the real and imaginary parts.

Although the result returned by `simplify` with the default setting for `Criterion` is shorter, here the complex value is a parameter of the sine function:

```
simplify(f, 'Steps', 100)
```

```
ans =
sin(x*i + sin(x))
```

When you set `Criterion` to `preferReal`, the simplifier disfavors expression forms where complex values appear inside subexpressions. In case of nested subexpressions, the deeper the complex value appears inside an expression, the least preference this form of an expression gets.

Simplify Expressions with Complex Arguments in Exponents

Setting `Criterion` to `preferReal` helps you avoid complex arguments in exponents.

Simplify these symbolic expressions:

```
simplify(sym(i)^i, 'Steps', 100)
simplify(sym(i)^(i+1), 'Steps', 100)

ans =
exp(-pi/2)

ans =
(-1)^(1/2 + i/2)
```

Now, simplify the second expression with the `Criterion` set to `preferReal`:

```
simplify(sym(i)^(i+1), 'Criterion', 'preferReal', 'Steps', 100)

ans =
exp(-pi/2)*i
```

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 strings.

`'default'`	Use the default (internal) simplification criteria.
`'preferReal'`	Favor the forms of S containing real values over the forms containing complex values. If any form of S contains complex

	values, the simplifier disfavors the forms where complex values appear inside subexpressions. In case of nested subexpressions, the deeper the complex value appears inside an expression, the least preference this form of an expression gets.

`'IgnoreAnalyticConstraints'` — Simplification rules
`false` (default) | `true`

Simplification rules, specified as the comma-separated pair consisting of `'IgnoreAnalyticConstraints'` and one of these values.

`false`	Use strict simplification rules. `simplify` always returns results equivalent to the initial expression.
`true`	Apply purely algebraic simplifications to an expression. `simplify` can return simpler results for expressions for which it would return more complicated results otherwise. Setting `IgnoreAnalyticConstraints` to `true` can lead to results that are not equivalent to the initial expression.

`'Seconds'` — Time limit for the simplification process
`Inf` (default) | `positive number`

Time limit for the simplification process, specified as the comma-separated pair consisting of `'Seconds'` and a positive value that denotes the maximal time in seconds.

`'Steps'` — Number of simplification steps
`1` (default) | `positive number`

Number of simplification steps, specified as the comma-separated pair consisting of `'Steps'` and a positive value that denotes the maximal number of internal simplification steps. Note that increasing the number of simplification steps can slow down your computations.

`simplify(S,'Steps',n)` is equivalent to `simplify(S,n)`, where `n` is the number of simplification steps.

Alternative Functionality

Besides the general simplification function (`simplify`), the toolbox provides a set of functions for transforming mathematical expressions to particular forms. For example,

you can use particular functions to expand or factor expressions, collect terms with the same powers, find a nested (Horner) representation of an expression, or quickly simplify fractions. If the problem that you want to solve requires a particular form of an expression, the best approach is to choose the appropriate simplification function. These simplification functions are often faster than `simplify`.

More About

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`, `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:

  - $\log(e^x) = x$
  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k

See Also

`collect` | `combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplifyFraction`

Related Examples

- “Simplifications” on page 2-40

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)
```

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`.

More About

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.
- “Simplifications”

See Also

`collect` | `combine` | `expand` | `factor` | `horner` | `numden` | `rewrite` | `simplify`

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 == x.der+sin(time)+y*5.0;
ans =
y.der == x.der+sin(time)+y*5.0;
```

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 == -x1+x1.der+sin(time);
```

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 == -x1+x2+sin(time)+x3.der;
```

More About

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 app and want to convert the results to Simscape equations, use the `generate::Simscape` function in MuPAD.

- "Generate Simscape Equations" on page 2-227

See Also

`matlabFunctionBlock` | `matlabFunction` | `ccode` | `fortran`

sin

Symbolic sine function

Syntax

```
sin(X)
```

Description

`sin(X)` returns the sine function 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)]
```

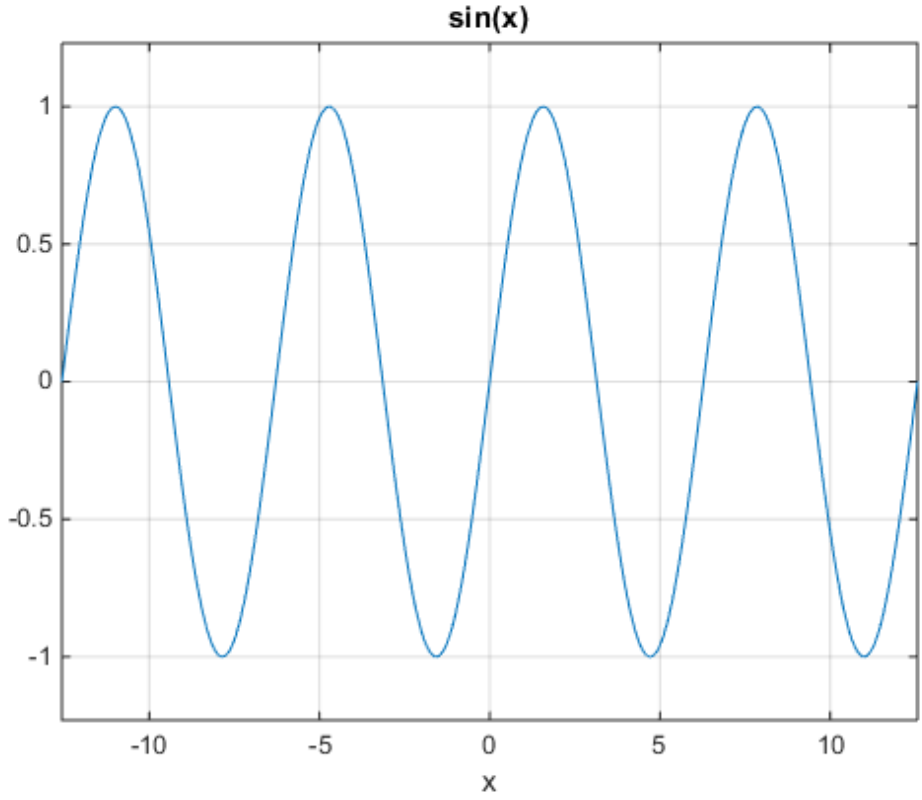
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.90929742682568169539601986591174,...
0,...
0.5,...
0.78183148246802980870844452667406,...
-0.99999020655070345705156489902552]
```

Plot the Sine Function

Plot the sine function on the interval from -4π to 4π.

```
syms x
ezplot(sin(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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*i)*i)/2 - (exp(x*i)*i)/2
```

Input Arguments

X — Input

symbolic number | symbolic variable | symbolic expression | symbolic function | symbolic vector | symbolic matrix

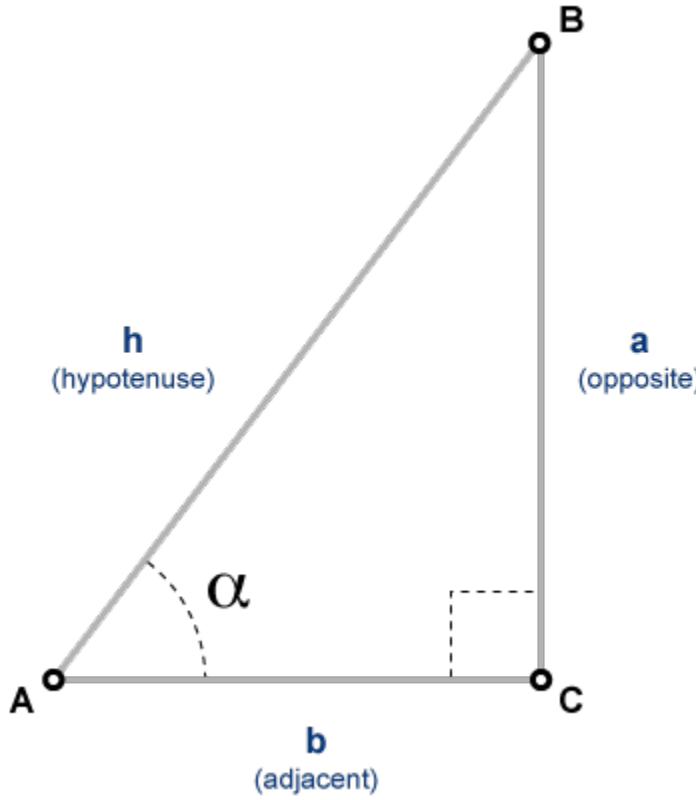
Input, specified as a symbolic number, variable, expression, or function, or as a vector or matrix of symbolic numbers, variables, expressions, or functions.

More About

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

$$\text{sine}(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{2i}.$$

See Also

`acos` | `acot` | `acsc` | `asec` | `asin` | `atan` | `cos` | `cot` | `csc` | `sec` | `tan`

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

`sym` | `vpa` | `double`

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, i/2, sinh((pi*2*i)/7), -i]
```

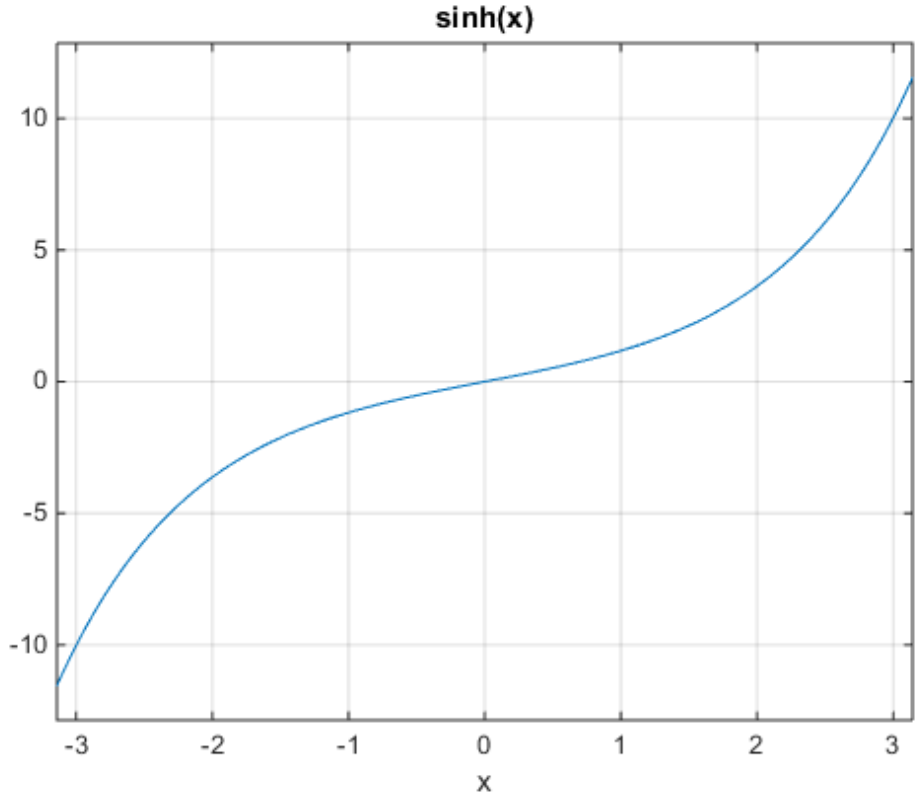
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -3.626860407847018767668213982801 3,...
0,...
0.5*i,...
0.781831482468029808708444526674 06*i,...
-1.0*i]
```

Plot the Hyperbolic Sine Function

Plot the hyperbolic sine function on the interval from $-\pi$ to π.

```
syms x
ezplot(sinh(x), [-pi, pi])
grid on
```



Handle Expressions Containing the 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`

sinhint

Hyperbolic sine integral function

Syntax

```
sinhint(X)
```

Description

`sinhint(X)` returns the hyperbolic sine integral function 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)]
```

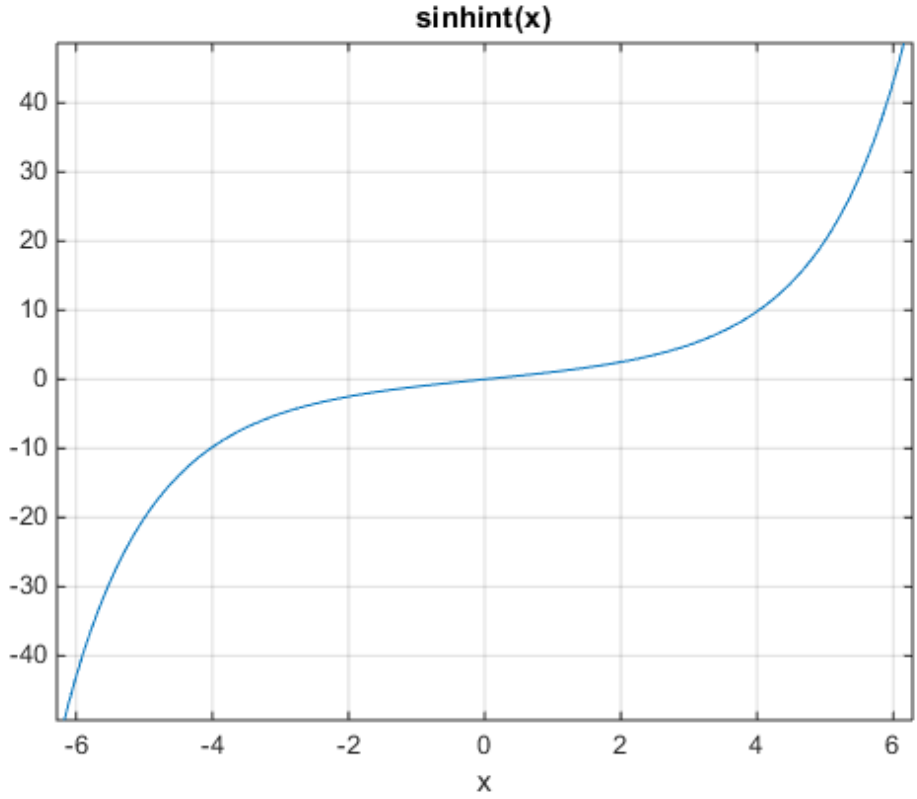
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -5.469640345115342150636958009127 7,...
-1.057250875375728514571842354895 9,...
0,...
1.802743198288293882089794577617,...
53.736750620859153990408011863262]
```

Plot the Hyperbolic Sine Integral Function

Plot the hyperbolic sine integral function on the interval from `-2*pi` to `2*pi` .

```
syms x
ezplot(sinhint(x), [-2*pi, 2*pi])
grid on
```



Handle Expressions Containing the 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.

More About

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

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)]
```

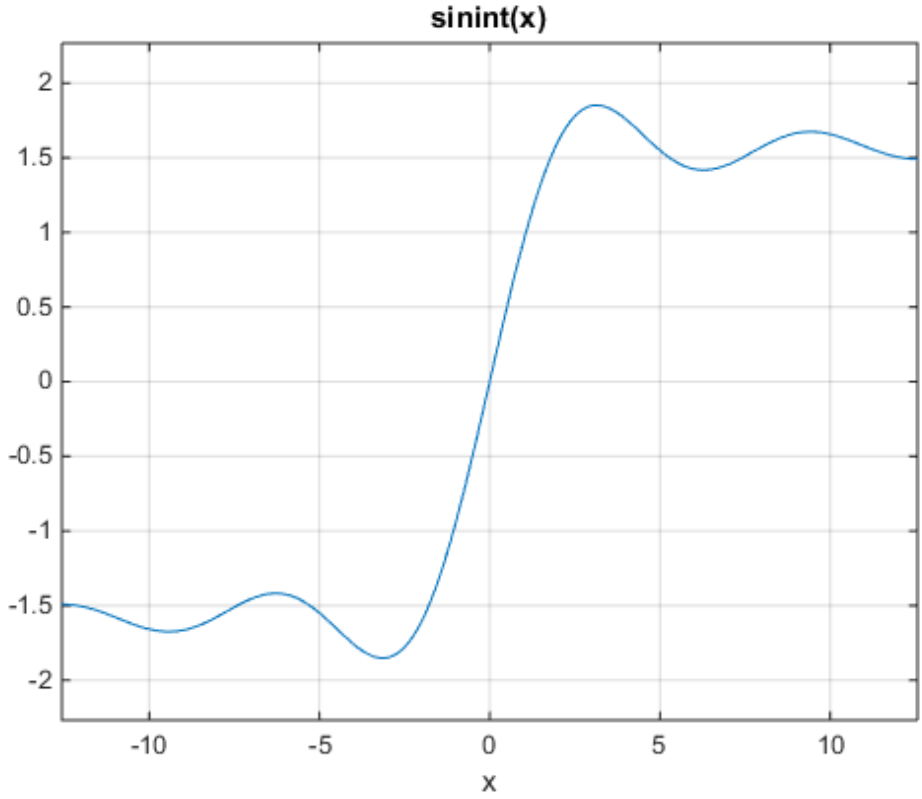
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -1.851937051982466170361053370158,...
0,...
1.370762168154488480069678288381 6,...
1.851937051982466170361053370158,...
0.946083070367183014941353313823 18]
```

Plot the Sine Integral Function

Plot the sine integral function on the interval from `-4*pi` to `4*pi` .

```
syms x
ezplot(sinint(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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.

More About

Sine Integral Function

The sine integral function is defined as follows:

$$\mathrm{Si}(x) = \int_0^x \frac{\sin(t)}{t}\,dt$$

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

`coshint` | `cosint` | `eulergamma` | `int` | `sin` | `sinhint` | `ssinint`

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`

solve

Equations and systems solver

Compatibility

`solve` does not accept string inputs containing multiple input arguments. In future releases, string inputs will be deprecated. In place of string inputs, first declare the variables using `syms` and pass them as a comma-separated list or vector.

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 variable to solve for is determined by `symvar`. 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 `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 `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 Univariate Equations

Use the == operator to specify the equation `sin(x) == 1` and solve it.

```
syms x
eqn = sin(x) == 1;
solve(eqn, x)
```

```
ans =
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 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, this means solutions might exist but `solve` did not find any solutions.

```
solve(3*x+2, 3*x+1, x)
```

```
ans =
Empty sym: 0-by-1
```

Solve Multivariate Equations

To avoid ambiguities when solving equations with symbolic parameters, specify the variable for which you want to solve an equation. If you do not specify the variable for which you want to solve the equation, `solve` chooses a variable using `symvar`. Here, `solve` chooses the variable *x*.

```
syms a b c x
sola = solve(a*x^2 + b*x + c == 0, a)
sol = solve(a*x^2 + b*x + c == 0)
```

```
sola =
-(c + b*x)/x^2
sol =
 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)
 -(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

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 `b` and `a` by specifying the variables explicitly.

```
syms a b
[b, a] = solve(a + b == 1, 2*a - b == 4, b, a)
```

```
b =
-2/3

a =
5/3
```

Work with Parameters and Conditions Returned by solve

To return the complete solution of an equation with parameters and conditions of the solution, specify ReturnConditions as `true`.

Solve the equation sin(*x*) = 0. The `solve` function returns a structure with three fields. The `S.x` field contains the solution, `S.parameters` contains the parameters in the solution, and `S.conditions` contains the conditions on the solution.

```
syms x
S = solve(sin(x) == 0, x, 'ReturnConditions', true);
S.x
S.parameters
S.conditions
```

```
ans =
pi*k
ans =
k
ans =
in(k, 'integer')
```

The solution `pi*k` contains the parameter `k` and is valid under the condition `in(k, 'integer')`. This means the parameter `k` must be an integer.

To find a numeric solution for `x`, substitute for `k` with a value using `subs`. Check if the value satisfies the condition on `k` using `isAlways`.

Check if `k = 4` satisfies `in(k, 'integer')`.

```
isAlways(subs(S.conditions, S.parameters, 4))
```

```
ans =
     1
```

`isAlways` returns logical `1` (`true`) meaning `4` is a valid value for `k`. Substitute `4` for `k` to obtain a solution for `x`. Use `vpa` to obtain a numeric approximation.

```
solx = subs(S.x, S.parameters, 4)
vpa(solx)
```

```
solx =
4*pi
ans =
12.566370614359172953850573533118
```

To find a valid value of `k` for `0<x<2*pi`, assume the condition `S.conditions` and use `solve` to solve these conditions for `k`. Substitute the value of `k` into the solution for `x`.

```
assume(S.conditions)
```

```
solk = solve(S.x>0, S.x<2*pi, S.parameters)
solx = subs(S.x, S.parameters, solk)
```

```
solk =
1
solx =
pi
```

A valid value of `k` for `0<x<2*pi` is `1`. This produces the solution `x = pi`.

Solve a System of Equations Assigning Solutions to Variables

When solving a system of equations, use multiple output arguments to assign the solutions directly to output variables. The solver returns a symbolic array of solutions for each independent variable.

```
syms a u v
[sola, solu, solv] = solve(a*u^2 + v^2 == 0, u - v == 1, a^2 + 6 == 5*a, a, u, v)
```

```
sola =
 2
 2
 3
 3
solu =
 1/3 - (2^(1/2)*i)/3
 (2^(1/2)*i)/3 + 1/3
 1/4 - (3^(1/2)*i)/4
 (3^(1/2)*i)/4 + 1/4
solv =
 - (2^(1/2)*i)/3 - 2/3
   (2^(1/2)*i)/3 - 2/3
 - (3^(1/2)*i)/4 - 3/4
   (3^(1/2)*i)/4 - 3/4
```

Entries with the same index form the solutions of a system.

```
solutions = [sola, solu, solv]
```

```
solutions =
[ 2, 1/3 - (2^(1/2)*i)/3, - (2^(1/2)*i)/3 - 2/3]
[ 2, (2^(1/2)*i)/3 + 1/3,   (2^(1/2)*i)/3 - 2/3]
[ 3, 1/4 - (3^(1/2)*i)/4, - (3^(1/2)*i)/4 - 3/4]
[ 3, (3^(1/2)*i)/4 + 1/4,   (3^(1/2)*i)/4 - 3/4]
```

A solution of the system is `a = 2`, `u = 1/3 - (2^(1/2)*i)/3`, and `v = -(2^(1/2)*i)/3 - 2/3`.

Return Complete Solution of System of Equations with Parameters and Conditions

To return the complete solution of a system of equations with parameters and conditions of the solution, specify ReturnConditions as `true`. Provide two additional output variables for output arguments parameters and conditions.

```
syms x y
[solx, soly, params, conditions] = solve(sin(x) == cos(2*y), x^2 == y,...
 [x, y], 'ReturnConditions', true)
```

```
solx =
   1/4 - (16*pi*k - 4*pi + 1)^(1/2)/4
   (16*pi*k - 4*pi + 1)^(1/2)/4 + 1/4
   (4*pi + 16*pi*k + 1)^(1/2)/4 - 1/4
 - (4*pi + 16*pi*k + 1)^(1/2)/4 - 1/4
soly =
 ((16*pi*k - 4*pi + 1)^(1/2)/4 - 1/4)^2
 ((16*pi*k - 4*pi + 1)^(1/2)/4 + 1/4)^2
 ((4*pi + 16*pi*k + 1)^(1/2)/4 - 1/4)^2
 ((4*pi + 16*pi*k + 1)^(1/2)/4 + 1/4)^2
params =
k
conditions =
 in(k, 'integer')
 in(k, 'integer')
 in(k, 'integer')
 in(k, 'integer')
```

A solution is formed by the elements of the same index in `solx`, `soly`, and `conditions`. Any element of `params` can appear in any solution. For example, a solution is `x = (4*pi + 16*pi*k + 1)^(1/2)/4 - 1/4`, and `y = ((4*pi + 16*pi*k + 1)^(1/2)/4 - 1/4)^2`, with the parameter `k` under the condition `in(k, 'integer')`. This condition means `k` must be an integer for the solution to be valid.

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

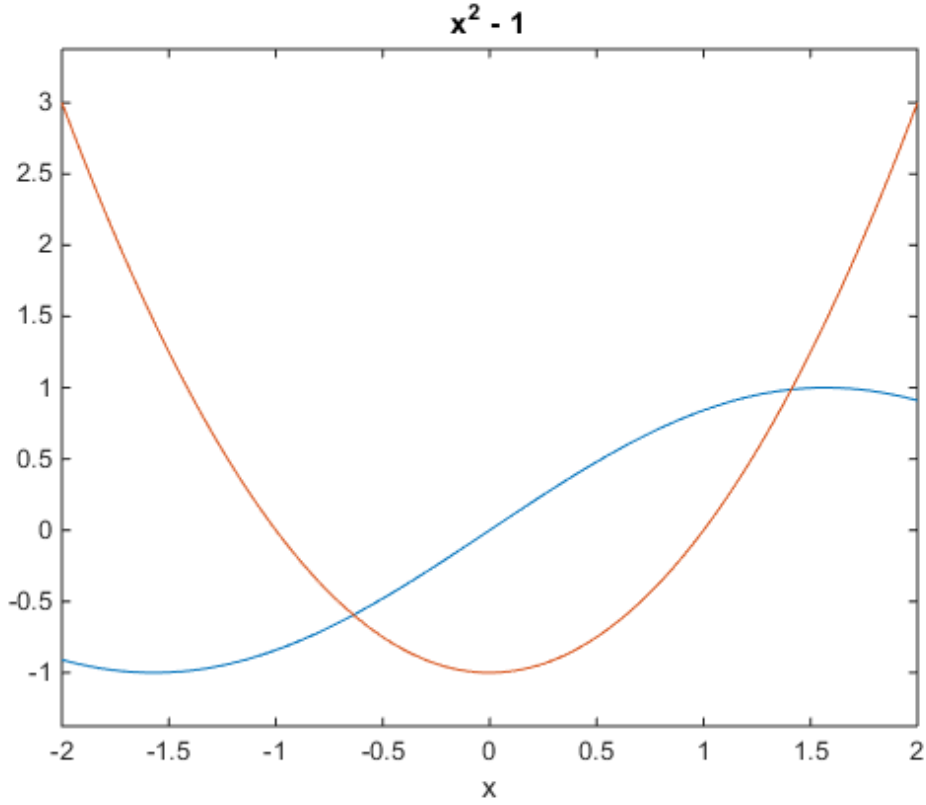
solver. Because the equation is not polynomial, an attempt to find all possible solutions can take a long time. The numeric solver does not try to find all numeric solutions for this equation. Instead, it returns only the first solution it finds.

```
syms x
solve(sin(x) == x^2 - 1, x)
```

```
Warning: Cannot solve symbolically. Returning a numeric
approximation instead.
> In solve at 301
ans =
-0.63673265080528201088799090383828
```

Plotting the left and the right sides of the equation in one graph shows that the equation also has a positive solution.

```
ezplot(sin(x), -2, 2)
hold on
ezplot(x^2 - 1, -2, 2)
hold off
```



You can find this solution by calling the numeric solver `vpasolve` directly and specifying the interval where this solution can be found.

```
vpasolve(sin(x) == x^2 - 1, x, [0 2])
```

```
ans =
1.4096240040025962492355939705895
```

Solve Inequalities

`solve` can solve inequalities to find a solution that satisfies the conditions. Solve the following conditions. 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
S = solve(x^2+y^2+x*y<1, x>0, y>0, [x, y], 'ReturnConditions', true);
solx = S.x
soly = S.y
params = S.parameters
conditions = S.conditions
```

```
solx =
(- 3*v^2 + u)^(1/2)/2 - v/2
soly =
v
params =
[ u, v]
conditions =
4*v^2 < u & u < 4 & 0 < v
```

Check if the values `u = 7/2` and `v = 1/2` satisfy the condition using `subs` and `isAlways`.

```
isAlways(subs(S.conditions, S.parameters, [7/2,1/2]))
```

```
ans =
     1
```

`isAlways` returns logical `1` (`true`) indicating that these values satisfy the condition. Substitute these parameter values into `S.x` and `S.y` to find a solution for `x` and `y`.

```
solx = subs(S.x, S.parameters, [7/2,1/2])
soly = subs(S.y, S.parameters, [7/2,1/2])
```

```
solx =
11^(1/2)/4 - 1/4
soly =
1/2
```

To convert the solution into numeric form, use `vpa`.

```
vpa(solx)
vpa(soly)
```

```
ans =
0.57915619758884996227873318416767
ans =
0.5
```

Return Real Solutions

Solve this equation. It has five solutions.

```
syms x
solve(x^5 == 3125, x)
```

```
ans =
                                                                  5
 - (2^(1/2)*(5 - 5^(1/2))^(1/2)*5*i)/4 - (5*5^(1/2))/4 - 5/4
   (2^(1/2)*(5 - 5^(1/2))^(1/2)*5*i)/4 - (5*5^(1/2))/4 - 5/4
   (5*5^(1/2))/4 - (2^(1/2)*(5^(1/2) + 5)^(1/2)*5*i)/4 - 5/4
   (5*5^(1/2))/4 + (2^(1/2)*(5^(1/2) + 5)^(1/2)*5*i)/4 - 5/4
```

If you need a solution in real numbers, set argument `Real` to `true`. The only real solution of this equation is `5`.

```
solve(x^5 == 3125, 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
solve(sin(x) + cos(2*x) == 1, x)
```

```
ans =
        0
     pi/6
 (5*pi)/6
```

To pick only one solution, use `PrincipalValue`.

```
solve(sin(x) + cos(2*x) == 1, x, 'PrincipalValue', true)
```

```
ans =
0
```

Apply Simplification Rules That Shorten the Result

By default, `solve` does not apply simplifications that are not always mathematically correct. As a result, `solve` cannot solve this equation symbolically.

```
syms x
solve(exp(log(x)*log(3*x))==4, x)
```

```
Warning: Cannot solve symbolically. Returning a numeric
approximation instead.
> In solve at 301
ans =
- 14.009379055223370038369334703094 - 2.925531005211111903666871798876 9*i
```

Set IgnoreAnalyticConstraints to `true` to apply simplifications that might allow `solve` to find a result. For details, see “Algorithms” on page 4-1017.

```
S = solve(exp(log(x)*log(3*x))==4, 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 should be verified.

Ignore Assumptions on Variables

The `sym` and `syms` functions let you set assumptions for symbolic variables. For example, declare that the variable *x* can have only positive values.

```
syms x positive
```

When you solve an equation or a system of equations with respect to such a variable, the solver only returns solutions consistent with the assumptions.

```
solve(x^2 + 5*x - 6 == 0, x)
```

```
ans =
1
```

To allow solutions that do not satisfy the assumptions, set `IgnoreProperties` to `true`.

```
solve(x^2 + 5*x - 6 == 0, 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 RootOf

When solving polynomials, `solve` might return solutions containing `RootOf`. To numerically approximate these solutions, use `vpa`. Consider the following equation and solution.

```
syms x
s = solve(x^4 + x^3 + 1 == 0, x)

s =
 RootOf(z^4 + z^3 + 1, z)[1]
 RootOf(z^4 + z^3 + 1, z)[2]
 RootOf(z^4 + z^3 + 1, z)[3]
 RootOf(z^4 + z^3 + 1, z)[4]
```

Because there are no parameters in this solution, you can use `vpa` to approximate it numerically.

```
vpa(s)

ans =
 - 1.0189127943851558447865795886366 + 0.60256541999859902604398442197193*i
 - 1.0189127943851558447865795886366 - 0.60256541999859902604398442197193*i
     0.5189127943851558447865795886366 + 0.66660984493201857915375880073 3*i
     0.5189127943851558447865795886366 - 0.66660984493201857915375880073 3*i
```

Solve Polynomial Equations of High Degree

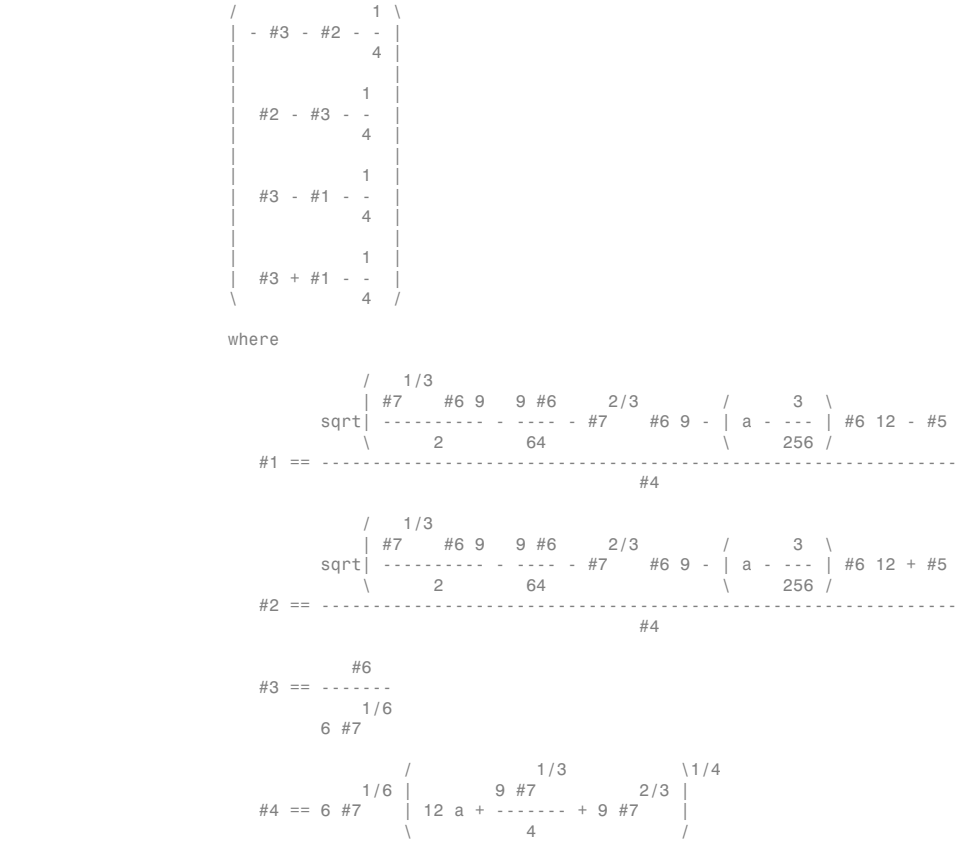
When you solve a higher-order polynomial equation, the solver might use `RootOf` to return the results.

```
syms x a
solve(x^4 + x^3 + a == 0, x)
```

```
ans =
 RootOf(z^4 + z^3 + a, z)[1]
 RootOf(z^4 + z^3 + a, z)[2]
 RootOf(z^4 + z^3 + a, z)[3]
 RootOf(z^4 + z^3 + a, z)[4]
```

To get an explicit solution for such equations, try calling the solver with `MaxDegree`. The option specifies the maximum degree of polynomials for which the solver tries to return explicit solutions. The default value is 3. Increasing this value, you can get explicit solutions for higher-order polynomials. For example, increase the value of `MaxDegree` to 4 and get explicit solutions instead of `RootOf` for the fourth-order polynomial.

```
s = solve(x^4 + x^3 + a == 0, x, 'MaxDegree', 4);
pretty(s)
```



```
      sqrt(3) sqrt(6) sqrt(9 a + #8) 3
#5 == --------------------------------
                      8

           /             1/3           \
           |         9 #7           2/3 |
#6 == sqrt| 12 a + ------- + 9 #7     |
           \             4             /

      a   #8
#7 == - + --
      2   18

                  / 135 a      /      3  \2       /      3  \3    189  \
#8 == sqrt(3) sqrt| ----- + 18 | a - --- |  - 256 | a - --- |  - ----- |
                  \  256       \     256 /        \     256 /    65536 /
```

Input Arguments

`eqn` — Equation to solve

symbolic expression | symbolic equation

Equation to solve, specified as a symbolic expression or symbolic equation. Symbolic equations are defined by the relation operator `==`. If `eqn` is a symbolic expression (without the right side), the solver assumes that the right side is 0, and solves the equation `eqn == 0`.

`var` — Variable for which you solve 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 in solution and conditions on solution

`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 means 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. Note that 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 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

3 (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 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 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 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 symbolic variables

Parameters in solution, returned as a vector of symbolic variables. 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.

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.

More About

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`.
- 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:
  - $\log(e^x) = x$

  - $\mathrm{asin}(\sin(x)) = x$, $\mathrm{acos}(\cos(x)) = x$, $\mathrm{atan}(\tan(x)) = x$
  - $\mathrm{asinh}(\sinh(x)) = x$, $\mathrm{acosh}(\cosh(x)) = x$, $\mathrm{atanh}(\tanh(x)) = x$
  - $W_k(x \cdot e^x) = x$ for all values of k
- 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` | `linsolve` | `subs` | `symvar` | `vpasolve`

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 the Elements of a Vector

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 the Elements of a Sorted Matrix Had in the 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 a 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.

More About

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` relies on the internal order that MuPAD uses to store these objects.

See Also

`max` | `min`

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 =
     1     1     1
     1     1     1
     1     1     1
```

Use the syntax with two output arguments to return the square root of a matrix and the residual:

```
A = vpa(sym([0 0; 0 5/3]), 100);
[X,resnorm] = sqrtm(A)

X =
[ 0,                                   0]
[ 0, 1.2909944487358056283930884665941]

resnorm =
2.9387358770557187699218413430556e-40
```

More About

Square Root of a Matrix

The square root of a matrix A is a matrix X, such that $X^2 = A$ and the eigenvalues of X are the square roots of the eigenvalues of A.

Tips

- Calling `sqrtm` for a matrix that is not a symbolic object invokes the MATLAB `sqrtm` function.
- If A has an eigenvalue 0 of algebraic multiplicity larger than its geometric multiplicity, the square root of A does not exist.

See Also

`cond` | `eig` | `expm` | `funm` | `jordan` | `logm` | `norm`

ssinint

Shifted sine integral function

Syntax

```
ssinint(X)
```

Description

`ssinint(X)` returns the shifted sine integral function `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)]
```

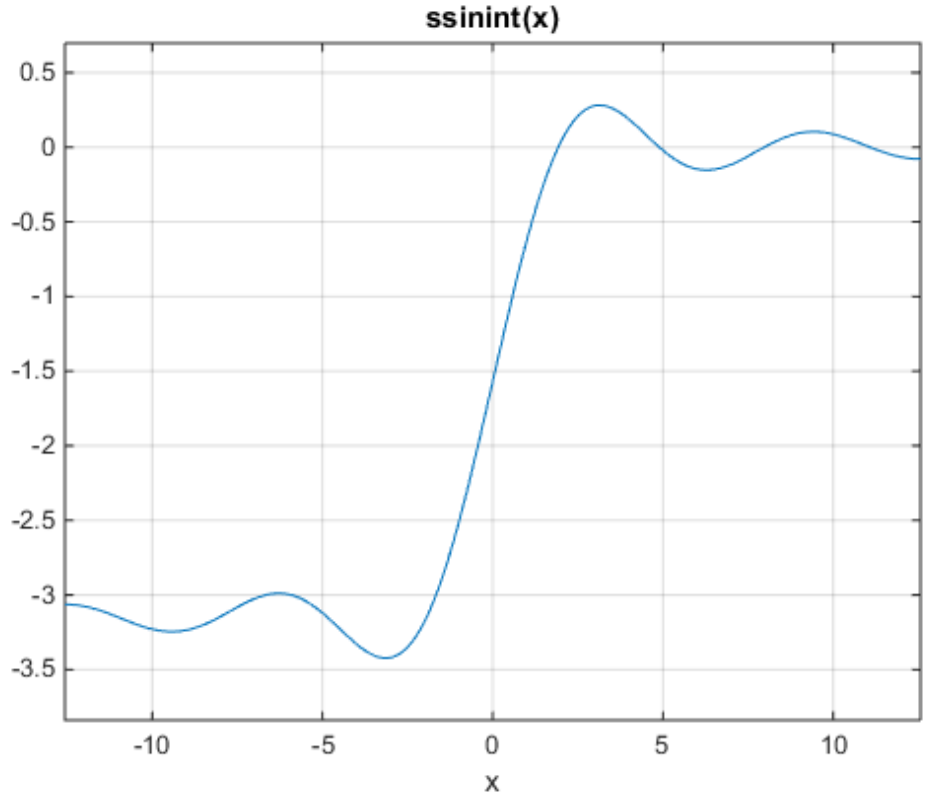
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -3.422733378777362789592375061797 7,...
-1.570796326794896619231321691639 8,...
-0.200034158640408139161643403258 18,...
0.281140725187569551129731678518 24,...
-0.624713256427713604289968377816 57]
```

Plot the Shifted Sine Integral Function

Plot the shifted sine integral function on the interval from `-4*pi` to `4*pi` .

```
syms x
ezplot(ssinint(x), [-4*pi, 4*pi])
grid on
```



Handle Expressions Containing the 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.

More About

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 Ssi(x) = Si(x) - π/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`

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`.

`[r,var] = subexpr(expr,var)` substitutes the common subexpression by the symbolic variable var.

`[r,var] = subexpr(expr,'var')` is equivalent to `[r,var] = subexpr(expr,var)`.

Examples

Rewrite an Expression Using Abbreviations

`subexpr(expr)` finds a common subexpression in the expression `expr`, and replaces it with the symbolic variable `sigma`.

Solve this 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);
```

These long expressions have common subexpressions. Abbreviating these common subexpressions shortens the expressions. To abbreviate subexpressions, use `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))*i)/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))*i)/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

`subexpr(expr,var)` lets you specify the variable name to use for abbreviations.

Solve this 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);
```

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 =
                                                                    s^(1/3)...
 - b/(3*a) - (- b^2/(9*a^2) + c/(3*a))/s^(1/3)
 (- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 - (3^(1/2)*(s^(1/3)...
 + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)
 (- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 + (3^(1/2)*(s^(1/3)...
 + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)
s =
((d/(2*a) + b^3/(27*a^3) - (b*c)/(6*a^2))^2 + (- b^2/(9*a^2) +...
 c/(3*a))^3)^(1/2) - b^3/(27*a^3) - d/(2*a) + (b*c)/(6*a^2)
```

Alternatively, use the string `s` to specify the abbreviation variable.

```
[abbrSolutions, s] = subexpr(solutions, 's')
```

```
abbrSolutions =
                                                                    s^(1/3)...
 - b/(3*a) - (- b^2/(9*a^2) + c/(3*a))/s^(1/3)
 (- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 - (3^(1/2)*(s^(1/3)...
 + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)
```

```
 (- b^2/(9*a^2) + c/(3*a))/(2*s^(1/3)) - s^(1/3)/2 + (3^(1/2)*(s^(1/3)...
 + (- b^2/(9*a^2) + c/(3*a))/s^(1/3))*i)/2 - b/(3*a)
s =
((d/(2*a) + b^3/(27*a^3) - (b*c)/(6*a^2))^2 + (- b^2/(9*a^2) + c/(3*a))^3)^(1/2)...
 - b^3/(27*a^3) - d/(2*a) + (b*c)/(6*a^2)
```

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

string | symbolic variable

Variable to use for substituting common subexpressions, specified as a string or symbolic variable.

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

`pretty` | `subs`

Related Examples

- “Substitute with subexpr” on page 2-46

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 evaluating s.

`subs(s,new)` returns a copy of s replacing all occurrences of the default variable in s with new, and then evaluating s. The default variable is defined by `symvar`.

`subs(s)` returns a copy of s replacing symbolic variables in s with their values obtained from the calling function and the MATLAB workspace, and then evaluating s. Variables with no assigned values remain as variables.

Examples

Single Substitution

Replace `a` with `4` in this expression.

```
syms a b
subs(a + b, a, 4)
```

```
ans =
b + 4
```

Replace `a*b` with `5` in this expression.

```
subs(a*b^2, a*b, 5)
```

```
ans =
5*b
```

Value That Gets Substituted by Default

Substitute the default value in this expression with `a`. If you do not specify which variable or expression that you want to replace, `subs` uses `symvar` to find the default variable. For `x + y`, the default variable is `x`.

```
syms x y a
symvar(x + y, 1)
```

```
ans =
x
```

Therefore, subs replaces `x` with `a`.

```
subs(x + y, a)
```

```
ans =
a + y
```

Single Input

Solve this ordinary differential equation.

```
syms a y(t)
y = dsolve(diff(y) == -a*y)
```

```
y =
C2*exp(-a*t)
```

Now, specify the values of the symbolic parameters `a` and `C2`.

```
a = 980;
C2 = 3;
```

Although the values `a` and `C2` are now in the MATLAB workspace, `y` is not evaluated with the account of these values.

```
y
```

```
y =
C2*exp(-a*t)
```

To evaluate `y` taking into account the new values of `a` and `C2`, use `subs`.

```
subs(y)
```

```
ans =
3*exp(-980*t)
```

Multiple Substitutions

Make multiple substitutions by specifying the old and new values as vectors.

```
syms a b
subs(cos(a) + sin(b), [a, b], [sym('alpha'), 2])
```

```
ans =
sin(2) + cos(alpha)
```

You also can use cell arrays for that purpose.

```
subs(cos(a) + sin(b), {a, b}, {sym('alpha'), 2})
```

```
ans =
sin(2) + cos(alpha)
```

Scalar and Matrix Expansion

Replace variable `a` in this expression with the 3-by-3 magic square matrix. Note that the constant `1` expands to the 3-by-3 matrix with all its elements equal to `1`.

```
syms a t
subs(exp(a*t) + 1, a, -magic(3))
```

```
ans =
[ exp(-8*t) + 1,   exp(-t) + 1, exp(-6*t) + 1]
[ exp(-3*t) + 1, exp(-5*t) + 1, exp(-7*t) + 1]
[ exp(-4*t) + 1, exp(-9*t) + 1, exp(-2*t) + 1]
```

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.

Multiple Scalar Expansion

Replace variables `x` and `y` with these 2-by-2 matrices. When you make multiple substitutions involving vectors or matrices, use cell arrays to specify the old and new values.

```
syms x y
subs(x*y, {x, y}, {[0 1; -1 0], [1 -1; -2 1]})
```

```
ans =
[ 0, -1]
[ 2,  0]
```

Note that these substitutions are elementwise.

```
[0 1; -1 0].*[1 -1; -2 1]
```

```
ans =
     0    -1
     2     0
```

Substitutions in Equations

Replace `sin(x + 1)` with `a` in this equation.

```
syms x a
subs(sin(x + 1) + 1 == x, sin(x + 1), a)

ans =
a + 1 == x
```

Substitutions in Functions

Replace `x` with `a` in this symbolic function.

```
syms x y a
syms f(x, y)
f(x, y) = x + y;
f = subs(f, x, a)

f(x, y) =
a + y
```

`subs` replaces the values in the symbolic function formula, but does not replace input arguments of the function.

```
formula(f)
argnames(f)

ans =
a + y

ans =
[ x, y]
```

You can replace the arguments of a symbolic function explicitly.

```
syms x y
f(x, y) = x + y;
f(a, y) = subs(f, x, a);
f

f(a, y) =
a + y
```

Original Expression

Assign the expression `x + y` to `s`.

```
syms x y
s = x + y;
```

Replace `y` in this expression with the value `1`. Here, `s` itself does not change.

```
subs(s, y, 1);
s
```

```
s =
x + y
```

To replace the value of `s` with the new expression, assign the result returned by `subs` to `s`.

```
s = subs(s, y, 1);
s
```

```
s =
x + 1
```

Structure Array

Suppose you want to verify the solutions of this system of equations.

```
syms x y
eqs = [x^2 + y^2 == 1, x == y];
S = solve(eqs, x, y);
S.x
S.y
```

```
ans =
 -2^(1/2)/2
  2^(1/2)/2
ans =
 -2^(1/2)/2
  2^(1/2)/2
```

To verify the correctness of the returned solutions, substitute the solutions into the original system.

```
logical(subs(eqs, S))
```

```
ans =
     1     1
     1     1
```

Input Arguments

s — Input

symbolic variable | symbolic expression | symbolic equation | symbolic function | symbolic array | symbolic vector | symbolic matrix

Input specified as a symbolic variable, expression, equation, function, array, vector, or matrix.

old — Existing element that needs to be replaced

symbolic variable | symbolic expression | string representing variable or expression | symbolic array | symbolic vector | symbolic matrix | array of strings | vector of strings | matrix of strings

Existing element that needs to be replaced specified as a symbolic variable, expression, string, array, vector, or matrix.

new — New element

number | symbolic variable | symbolic expression | string representing variable or expression | symbolic array | symbolic vector | symbolic matrix | array of strings | vector of strings | matrix of strings | structure array

New element specified as a number, variable, expression, string, array, vector, matrix, or structure array.

More About

Tips

- `subs(s,old,new)` does not modify s. To modify s, use `s = subs(s,old,new)`.
- If old and new are both vectors or cell arrays of the same size, `subs` replaces each element of old by the corresponding element of new.
- If old is a scalar, and new is a vector or matrix, then `subs(s,old,new)` replaces all instances of old in s with new, performing all operations elementwise. All constant terms in s are replaced with the constant times a vector or matrix of all 1s.
- If `s` is a univariate polynomial and new is a numeric matrix, use `polyvalm(sym2poly(s), new)` to evaluate `s` in the matrix sense. All constant terms are replaced with the constant times an identity matrix.

See Also

`double` | `eval` | `evalAt` | `simplify` | `subexpr` | `subs` | `subset` | `subsex` | `subsop` | `vpa`

Related Examples

- “Substitutions in Symbolic Expressions” on page 1-16
- “Substitute with subs” on page 2-48
- “Combine subs and double for Numeric Evaluations” on page 2-52

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)` produces the "economy size" decomposition. If X is an `m`-by-`n` matrix with `m > n`, then `svd` computes only the first `n` columns of `U`. In this case, S is an `n`-by-`n` matrix. For `m <= n`, this syntax is equivalent to `svd(X)`.

`[U,S,V] = svd(X,'econ')` also produces the "economy size" decomposition. If X is an `m`-by-`n` matrix with `m >= n`, then this syntax is equivalent to `svd(X,0)`. For `m < n`, `svd` computes only the first `m` columns of V. In this case, S is an `m`-by-`m` matrix.

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
```

```
                        17.888543819998317571273389349 85
                       4.4721359549995793928183473374626
 0.00000000000000000000042127245515076439434819165724023*i
```

Singular Values and Singular Vectors

Compute the singular values and singular vectors of the 4-by-4 magic square:

```
old = digits(10);
A = sym(magic(4))
[U, S, V] = svd(A)
digits(old)
```

```
A =
[ 16,  2,  3, 13]
[  5, 11, 10,  8]
[  9,  7,  6, 12]
[  4, 14, 15,  1]

U =
[ 0.5,  0.6708203932,  0.5, -0.2236067977]
[ 0.5, -0.2236067977, -0.5, -0.6708203932]
[ 0.5,  0.2236067977, -0.5,  0.6708203932]
[ 0.5, -0.6708203932,  0.5,  0.2236067977]

S =
[ 34.0,           0,           0,               0]
[    0, 17.88854382,           0,               0]
[    0,           0, 4.472135955,               0]
[    0,           0,           0, 1.108401846e-15]

V =
[ 0.5,  0.5,  0.6708203932,  0.2236067977]
[ 0.5, -0.5, -0.2236067977,  0.6708203932]
[ 0.5, -0.5,  0.2236067977, -0.6708203932]
[ 0.5,  0.5, -0.6708203932, -0.2236067977]
```

Compute the product of U, S, and the Hermitian transpose of V with the 10-digit accuracy. The result is the original matrix A with all its elements converted to floating-point numbers:

```
vpa(U*S*V',10)
```

```
ans =
```

```
[ 16.0,  2.0,  3.0, 13.0]
[  5.0, 11.0, 10.0,  8.0]
[  9.0,  7.0,  6.0, 12.0]
[  4.0, 14.0, 15.0,  1.0]
```

"Economy Size" Decomposition

Use the second input argument 0 to compute the "economy size" decomposition of this 2-by-3 matrix:

```
old = digits(10);
A = sym([1 1;2 2; 2 2]);
[U, S, V] = svd(A, 0)
```

```
U =
[ 0.3333333333, -0.6666666667]
[ 0.6666666667,  0.6666666667]
[ 0.6666666667, -0.3333333333]

S =
[ 4.242640687, 0]
[           0, 0]

V =
[ 0.7071067812,  0.7071067812]
[ 0.7071067812, -0.7071067812]
```

Now, use the second input argument `'econ'` to compute the "economy size" decomposition of matrix B. Here, the 3-by-2 matrix B is the transpose of A.

```
B = A';
[U, S, V] = svd(B, 'econ')
digits(old)
```

```
U =
[ 0.7071067812, -0.7071067812]
[ 0.7071067812,  0.7071067812]

S =
[ 4.242640687, 0]
[           0, 0]

V =
[ 0.3333333333,  0.6666666667]
```

```
[ 0.6666666667, -0.6666666667]
[ 0.6666666667,  0.3333333333]
```

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.

More About

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` | `numeric::singularvalues` | `numeric::singularvectors` | `numeric::svd` | `qr` | `svd` | `vpa`

Related Examples

- “Singular Value Decomposition” on page 2-78

sym

Create symbolic objects

Syntax

```
var = sym('var')
var = sym('var',set)
sym('var','clear')
Num = sym(Num)
Num = sym(Num,flag)
A = sym('A',dim)
A = sym(A,set)
sym(A,'clear')
f(arg1,...,argN) = sym('f(arg1,...,argN)')
fsym = sym(fh)
```

Description

`var = sym('var')` creates the symbolic variable var.

`var = sym('var',set)` creates the symbolic variable var and states that var belongs to set.

`sym('var','clear')` clears assumptions previously set on the symbolic variable var.

`Num = sym(Num)` converts a number or a numeric matrix Num to symbolic form.

`Num = sym(Num,flag)` converts a number or a numeric matrix Num to symbolic form. The second argument specifies the technique for converting floating-point numbers.

`A = sym('A',dim)` creates a vector or a matrix of symbolic variables.

`A = sym(A,set)`, where A is an *existing* symbolic vector or matrix, sets an assumption that each element of A belongs to set. To check assumptions set on A, use the `assumptions` function. This syntax does not create A. To create a symbolic vector or a symbolic matrix A, use `A = sym('A',[m n])` or `A = sym('A',n)`.

`sym(A,'clear')`, where A is an *existing* symbolic vector or matrix, clears assumptions previously set on elements of A. This syntax does not create A. To create a symbolic vector or a symbolic matrix `A`, use `A = sym('A',[m n])` or `A = sym('A',n)`.

`f(arg1,...,argN) = sym('f(arg1,...,argN)')` creates the symbolic function f and specifies that arg1,...,argN are the input arguments of f. This syntax does not create symbolic variables arg1,...,argN. The arguments arg1,...,argN must be *existing* symbolic variables.

`fsym = sym(fh)` uses the symbolic form of the body of the function handle fh to create the symbolic expression, vector, or matrix `fsym`.

Input Arguments

`var`

String that represents the variable name. It must begin with a letter and can contain only alphanumeric characters.

`set`

Either `real` or `positive`.

`Num`

Number, vector, or matrix of numbers.

`flag`

One of these strings: `r`, `d`, `e`, or `f`.

- `r` stands for "rational." Floating-point numbers obtained by evaluating expressions of the form `p/q`, `p*pi/q`, `sqrt(p)`, `2^q`, and `10^q` for modest sized integers `p` and `q` are converted to the corresponding symbolic form. This effectively compensates for the round-off error involved in the original evaluation, but might not represent the floating-point value precisely. If no simple rational approximation can be found, an expression of the form `p*2^q` with large integers `p` and `q` reproduces the floating-point value exactly. For example, `sym(4/3,'r')` is `'4/3'`, but `sym(1+sqrt(5),'r')` is `7286977268806824*2^(-51)`.
- `d` stands for "decimal." The number of digits is taken from the current setting of `digits` used by `vpa`. Fewer than 16 digits loses some accuracy, while

more than 16 digits might not be warranted. For example, with `digits(10)`, `sym(4/3,'d')` is `1.333333333`, while with digits `digits(20)`, `sym(4/3,'d')` is `1.3333333333333332593`, which does not end in a string of `3`s, but is an accurate decimal representation of the floating-point number nearest to `4/3`.

- `e` stands for "estimate error." The `'r'` form is supplemented by a term involving the variable `'eps'`, which estimates the difference between the theoretical rational expression and its actual floating-point value. For example, `sym(3*pi/4,'e')` is `3*pi/4*(1+3143276*eps/65)`.
- `f` stands for "floating-point." All values are represented in the form `N*2^e` or `-N*2^e`, where `N` and `e` are integers, `N >= 0`. For example, `sym(1/10,'f')` is `3602879701896397/36028797018963968` .

Default: `r`

`A`

String that represents the base for generated names of vector or matrix elements. It must be a valid variable name. (To verify if the name is a valid variable name, use `isvarname`.)

Default: The generated names of elements of a vector use the form `Ak`, and the generated names of elements of a matrix use the form `Ai_j`. The values of `k`, `i`, and `j` range from 1 to `m` or 1 to `n`. To specify another form for generated names of matrix elements, use `'%d'` in the first input. For example, `A = sym('A%d%d', [3 3])` generates the 3-by-3 symbolic matrix `A` with the elements `A11, A12, ..., A33`.

`dim`

Integer or vector of two integers specifying dimensions of A. For example, if `dim` is a vector `[m n]`, then the syntax `A = sym('A',[m n])` creates an `m`-by-`n` matrix of symbolic variables. If `dim` is an integer `n`, then the syntax `A = sym('A',n)` creates a square `n`-by-`n` matrix of symbolic variables.

`f`

Name of a symbolic function. It must begin with a letter and contain only alphanumeric characters.

`arg1,...,argN`

Arguments of a symbolic function. Each argument must be an *existing* symbolic variable.

fh

Function handle.

Output Arguments

var

Symbolic variable.

Num

Symbolic number or vector or matrix of symbolic numbers.

A

Vector or matrix of automatically generated symbolic variables.

f

Symbolic function.

fsym

Symbolic number, variable, expression; vector or matrix of symbolic numbers, variables, expressions.

Examples

Create the symbolic variables `x` and `y`:

```
x = sym('x');
y = sym('y');
```

Create the symbolic variables `x` and `y` assuming that `x` is real and `y` is positive:

```
x = sym('x','real');
y = sym('y','positive');
```

Check the assumptions on `x` and `y` using `assumptions`:

```
assumptions
```

```
ans =
[ in(x, 'real'), 0 < y]
```

For further computations, clear the assumptions:

```
sym('x','clear');
sym('y','clear');
assumptions
```

```
ans =
Empty sym: 1-by-0
```

The `sym` function lets you choose the conversion technique by specifying the optional second argument, which can be `'r'`, `'f'`, `'d'`, or `'e'`. The default is `'r'`. For example, convert the number 1/3 to a symbolic object:

```
r = sym(1/3)
f = sym(1/3, 'f')
d = sym(1/3, 'd')
e = sym(1/3, 'e')
```

```
r =
1/3

f =
6004799503160661/18014398509481984

d =
0.33333333333333331482961625624739099293950

e =
1/3 - eps/12
```

Create the 3-by-4 symbolic matrix `A` with the auto-generated elements `A1_1, ..., A3_4` :

```
A = sym('A', [3 4])
```

```
A =
[ A1_1, A1_2, A1_3, A1_4]
[ A2_1, A2_2, A2_3, A2_4]
[ A3_1, A3_2, A3_3, A3_4]
```

Now create the 4-by-4 matrix `B` with the elements `x_1_1, ..., x_4_4`:

```
B = sym('x_%d_%d', [4 4])
```

```
B =
[ x_1_1, x_1_2, x_1_3, x_1_4]
[ x_2_1, x_2_2, x_2_3, x_2_4]
[ x_3_1, x_3_2, x_3_3, x_3_4]
[ x_4_1, x_4_2, x_4_3, x_4_4]
```

This syntax does not define elements of a symbolic matrix as separate symbolic objects. To access an element of a matrix, use parentheses:

```
A(2, 3)
B (4, 2)
```

```
ans =
A2_3

ans =
x_4_2
```

You can use symbolic matrices and vectors generated by the `sym` function to define other matrices:

```
A = diag(sym('A',[1 4]))
```

```
A =
[ A1,  0,  0,  0]
[  0, A2,  0,  0]
[  0,  0, A3,  0]
[  0,  0,  0, A4]
```

Perform operations on symbolic matrices by using the operators that you use for numeric matrices. For example, find the determinant and the trace of the matrix `A`:

```
det(A)
```

```
ans =
A1*A2*A3*A4
```

```
trace(A)
```

```
ans =
A1 + A2 + A3 + A4
```

Use the `sym` function to set assumptions on each element of a symbolic matrix. You cannot create a symbolic matrix and set an assumption on all its elements in one `sym`

function call. Use two separate `sym` function calls. The first call creates a matrix, and the second call specifies an assumption:

```
A = sym('A%d%d', [2 2]);
A = sym(A, 'positive')
```

```
A =
[ A11, A12]
[ A21, A22]
```

Now, MATLAB assumes that all elements of `A` are positive:

```
solve(A(1, 1)^2 - 1, A(1, 1))
```

```
ans =
1
```

To check the assumptions set on the elements of `A`, use `assumptions`:

```
assumptions(A)
```

```
ans =
[ 0 < A21, 0 < A11, 0 < A22, 0 < A12]
```

To clear all previously set assumptions on elements of a symbolic matrix, also use the `sym` function:

```
A = sym(A, 'clear');
assumptions(A)
```

```
ans =
Empty sym: 1-by-0
```

Create the symbolic function `f` whose input arguments are symbolic variables `x` and `y`:

```
x = sym('x');
y = sym('y');
f(x, y) = sym('f(x, y)')
```

```
f(x, y) =
f(x, y)
```

Alternatively, you can use the assignment operation to create the symbolic function `f`:

```
f(x, y) = x + y
```

```
f(x, y) =
x + y
```

Alternatives

- To create several symbolic variables in one function call, use `syms`. When using `syms`, do not enclose variables in quotes and do not use commas between variable names:

  ```
  syms var1 var2 var3
  ```

  `syms` also lets you create real variables or positive variables. It also lets you clear assumptions set on a variable.
- `assume` and `assumeAlso` provide more flexibility for setting assumptions on variable.
- When creating a symbolic function, use `sym` to create arg1,...,argN as symbolic variables. Then use the assignment operation to create the symbolic function `f`, for example:

  ```
  x = sym('x');
  y = sym('y');
  f(x, y) = x + y
  ```
- `syms f(x, y)` is equivalent to these commands:

  ```
  x = sym('x');
  y = sym('y');
  f(x, y) = sym('f(x, y)')
  ```

More About

Tips

- For compatibility with previous versions, `sym('var','unreal')` is equivalent to `sym('var','clear')`.
- Statements like `pi = sym('pi')` and `delta = sym('1/10')` create symbolic numbers that avoid the floating-point approximations inherent in the values of `pi` and `1/10`. The `pi` created in this way temporarily replaces the built-in numeric function with the same name.
- `clear x` does *not* clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`. To remove assumptions, use one of these options:

- `sym('x','clear')` removes assumptions from `x` without affecting any other symbolic variables.
- `reset(symengine)` resets the symbolic engine and therefore removes assumptions on all variables. The variables themselves remain in the MATLAB workspace.
- `clear all` clears all objects in the MATLAB workspace and resets the symbolic engine.

See Also

`assume` | `assumeAlso` | `assumptions` | `clear` | `clear all` | `digits` | `double` | `eps` | `reset` | `symfun` | `syms` | `symvar`

Related Examples

- “Create Symbolic Variables and Expressions” on page 1-7
- “Create Symbolic Functions” on page 1-9
- “Assumptions on Symbolic Objects” on page 1-32
- “Estimate Precision of Numeric to Symbolic Conversions” on page 1-19

sym2poly

Symbolic-to-numeric polynomial conversion

Syntax

```
c = sym2poly(s)
```

Description

`c = sym2poly(s)` returns a row vector containing the numeric coefficients of a symbolic polynomial. The coefficients are ordered in descending powers of the polynomial's independent variable. In other words, the vector's first entry contains the coefficient of the polynomial's highest term; the second entry, the coefficient of the second highest term; and so on.

Examples

The command

```
syms x u v
sym2poly(x^3 - 2*x - 5)
```

returns

```
ans =
 1     0    -2    -5
```

The command

```
sym2poly(u^4 - 3 + 5*u^2)
```

returns

```
ans =
 1     0     5     0    -3
```

and the command

```
sym2poly(sin(pi/6)*v + exp(1)*v^2)
```

returns

```
ans =
2.7183    0.5000         0
```

See Also

poly2sym | subs | sym | polyval

symengine

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`

symfun

Create symbolic functions

Syntax

```
f = symfun(formula,inputs)
```

Description

`f = symfun(formula,inputs)` creates the symbolic function f and symbolic variables inputs representing its input arguments. The symbolic expression formula defines the body of the function f.

Input Arguments

`formula`

Symbolic expression or vector or matrix of symbolic expressions. This argument represents the body of f. If it contains other symbolic variables besides inputs, those variables must already exist in the MATLAB workspace.

`inputs`

Array that contains input arguments of f. For each argument, `symfun` creates a symbolic variable. Argument names must begin with a letter and can contain only alphanumeric characters.

Output Arguments

`f`

Symbolic function. The name of a symbolic function must begin with a letter and contain only alphanumeric characters.

Examples

Create the symbolic variables `x` and `y`. Then use `symfun` to create the symbolic function `f(x, y) = x + y`:

```
syms x y
f = symfun(x + y, [x y])
```

```
f(x, y) =
x + y
```

Create the symbolic variables `x` and `y`. Then use `symfun` to create an arbitrary symbolic function `f(x, y)`. An arbitrary symbolic function does not have a mathematical expression assigned to it.

```
syms x y
f = symfun(sym('f(x, y)'), [x y])
```

```
f(x, y) =
f(x, y)
```

Alternatives

Use the assignment operation to simultaneously create a symbolic function and define its body. The arguments `x` and `y` must be symbolic variables in the MATLAB workspace.

```
syms x y
f(x, y) = x + y
```

Use `syms` to create an arbitrary symbolic function `f(x, y)`. The following command creates the symbolic function `f` and the symbolic variables `x` and `y`.

```
syms f(x, y)
```

Use `sym` to create an arbitrary symbolic function `f(x, y)`. The arguments `x` and `y` must be symbolic variables in the MATLAB workspace.

```
syms x y
f(x, y) = sym('f(x, y)')
```

More About

- “Create Symbolic Functions” on page 1-9

See Also

argnames | dsolve | formula | matlabFunction | odeToVectorField | sym | syms | symvar

symprod

Product of series

Syntax

```
symprod(expr,var)
symprod(expr,var,a,b)
```

Description

`symprod(expr,var)` evaluates the product of a series, where expression expr defines the terms of a series, with respect to the symbolic variable var. The value of the variable var changes from 1 to `var`. If you do not specify the variable, `symprod` uses the default variable determined by `symvar`. If expr is a constant, then the default variable is `x`.

`symprod(expr,var,a,b)` evaluates the product of a series, where expression expr defines the terms of a series, with respect to the symbolic variable var. The value of the variable var changes from a to b. If you do not specify the variable, `symprod` uses the default variable determined by `symvar`. If expr is a constant, then the default variable is `x`.

`symprod(expr,var,[a,b])`, `symprod(expr,var,[a b])`, and `symprod(expr,var,[a;b])` are equivalent to `symprod(expr,var,a,b)`.

Examples

Evaluate the Product of a Series

Evaluate the product of a series for the symbolic expressions `k` and `k^2`:

```
syms k
symprod(k)
symprod((2*k - 1)/k^2)
```

```
ans =
factorial(k)
```

```
ans =
(1/2^(2*k)*2^(k + 1)*factorial(2*k))/(2*factorial(k)^3)
```

Evaluate the Product of a Series Specifying Bounds

Evaluate the product of a series for these expressions specifying the bounds:

```
syms k
symprod(1 - 1/k^2, k, 2, Inf)
symprod(k^2/(k^2 - 1), k, 2, Inf)
```

```
ans =
1/2

ans =
2
```

You also can specify bounds as a row or column vector:

```
syms k
symprod(1 - 1/k^2, k, [2, Inf])
symprod(k^2/(k^2 - 1), k, [2; Inf])
```

```
ans =
1/2

ans =
2
```

Evaluate the Product of a Series Specifying the Product Index and Bounds

Evaluate the product of a series for this multivariable expression with respect to `k`:

```
syms k x
symprod(exp(k*x)/x, k, 1, 10000)
```

```
ans =
exp(50005000*x)/x^10000
```

Input Arguments

`expr` — 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 or function, a constant, or a vector or matrix of symbolic expressions, functions, or constants.

`var` — Product index

symbolic variable

Product index, specified as a symbolic variable. If you do not specify this variable, `symprod` uses the default variable determined by `symvar(expr,1)`. If expr 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).

More About

Definite Product

The definite product of a series is defined as

$$\prod_{i=a}^{b} x_i = x_a \cdot x_{a+1} \cdot \ldots \cdot x_b$$

Indefinite Product

$$f(i) = \prod_i x_i$$

is called the indefinite product of x_i over i, if the following identity holds for all values of i:

$$\frac{f(i+1)}{f(i)} = x_i$$

Note: `symprod` does not compute indefinite products.

See Also

`cumprod` | `cumsum` | `int` | `syms` | `symsum` | `symvar`

syms

Shortcut for creating symbolic variables and functions

Syntax

```
syms var1 ... varN
syms var1 ... varN set
syms var1 ... varN clear
syms f(arg1,...,argN)
```

Description

`syms var1 ... varN` creates symbolic variables var1 ... varN.

`syms var1 ... varN set` creates symbolic variables var1 ... varN and states that these variables belong to set.

`syms var1 ... varN clear` removes assumptions previously set on symbolic variables var1 ... varN.

`syms f(arg1,...,argN)` creates the symbolic function f and symbolic variables arg1,...,argN representing the input arguments of f.

Input Arguments

`var1 ... varN`

Names of symbolic variables. Each name must begin with a letter and contain only alphanumeric characters.

`set`

Either `real` or `positive`.

`f`

Name of a symbolic function. It must begin with a letter and contain only alphanumeric characters.

`arg1,...,argN`

Arguments of a symbolic function. For each argument, `syms` creates a symbolic variable. Argument names must begin with a letter and contain only alphanumeric characters.

Examples

Create symbolic variables `x` and `y` using `syms`:

```
syms x y
```

Create symbolic variables `x` and `y`, and assume that they are real:

```
syms x y real
```

To see assumptions set on `x` and `y`, use `assumptions`:

```
assumptions(x)
assumptions(y)
```

```
ans =
in(x, 'real')
ans =
in(y, 'real')
```

Clear the assumptions that `x` and `y` are real:

```
syms x y clear
assumptions
```

```
ans =
Empty sym: 1-by-0
```

Create a symbolic function `f` that accepts two arguments, `x` and `y`:

```
syms f(x, y)
```

Specify the formula for this function:

```
f(x, y) = x + 2*y
```

```
f(x, y) =
x + 2*y
```

Compute the function value at the point `x = 1` and `y = 2`:

```
f(1, 2)
```

```
ans =
5
```

Create symbolic function `f` and specify its formula by this symbolic matrix:

```
syms x
f(x) = [x x^2; x^3 x^4];
```

Compute the function value at the point `x = 2`:

```
f(2)
```

```
ans =
[ 2,  4]
[ 8, 16]
```

Now compute the value of this function for `x = [1 2; 3 4]`. The result is a cell array of symbolic matrices:

```
y = f([1 2; 3 4])
```

```
y =
    [2x2 sym]    [2x2 sym]
    [2x2 sym]    [2x2 sym]
```

To access the contents of each cell in a cell array, use braces:

```
y{1}
```

```
ans =
[ 1, 2]
[ 3, 4]
```

```
y{2}
```

```
ans =
[  1,  8]
[ 27, 64]
```

```
y{3}
```

```
ans =
```

```
[ 1,  4]
[ 9, 16]

y{4}

ans =
[  1,  16]
[ 81, 256]
```

Alternatives

- `syms` is a shortcut for `sym`. This shortcut lets you create several symbolic variables in one function call. Alternatively, you can use `sym` and create each variable separately:

  ```
  var1 = sym('var1');
  ...
  varN = sym('varN');
  ```

  `sym` also lets you create real variables or positive variables. It also lets you clear assumptions set on a variable.
- `assume` and `assumeAlso` provide more flexibility for setting assumptions on variable.
- When creating a symbolic function, use `syms` to create arg1,...,argN as symbolic variables. Then use the assignment operation to create the symbolic function f, for example:

  ```
  syms x y
  f(x, y) = x + y
  ```

More About

Tips

- For compatibility with previous versions, `syms var1 ... varN unreal` is equivalent to `syms var1 ... varN clear`.
- In functions and scripts, do not use `syms` to create symbolic variables with the same names as MATLAB functions. For these names MATLAB does not create symbolic variables, but keeps the names assigned to the functions. If you want to create a symbolic variable with the same name as some MATLAB function inside a function or a script, use `sym`. For example:

```
alpha = sym('alpha')
```

- `clear x` does *not* clear the symbolic object of its assumptions, such as real, positive, or any assumptions set by `assume`. To remove assumptions, use one of these options:
  - `syms x clear` removes assumptions from `x` without affecting any other symbolic variables.
  - `reset(symengine)` resets the symbolic engine and therefore removes assumptions on all variables. The variables themselves remain in the MATLAB workspace.
  - `clear all` removes all objects in the MATLAB workspace and resets the symbolic engine.

- "Create Symbolic Variables and Expressions" on page 1-7
- "Create Symbolic Functions" on page 1-9
- "Assumptions on Symbolic Objects" on page 1-32

See Also

`assume` | `assumeAlso` | `assumptions` | `clear all` | `reset` | `sym` | `symfun` | `symvar`

symsum

Sum of series

Syntax

```
symsum(expr,var)
symsum(expr,var,a,b)
```

Description

`symsum(expr,var)` finds the indefinite sum of a series, where expression expr defines the terms of a series, with respect to the symbolic variable var. This is an expression `s`, such that `expr(var) = s(var + 1) - s(var)`. If you do not specify the variable, `symsum` uses the default variable determined by `symvar`. If expr is a constant, then the default variable is `x`.

`symsum(expr,var,a,b)` evaluates the sum of a series, where expression expr defines the terms of a series, with respect to the symbolic variable var. The value of the variable var changes from a to b. If you do not specify the variable, `symsum` uses the default variable determined by `symvar`. If expr is a constant, then the default variable is `x`.

`symsum(expr,var,[a,b])`, `symsum(expr,var,[a b])`, and `symsum(expr,var,[a;b])` are equivalent to `symsum(expr,var,a,b)`.

Examples

Evaluate the Sum of a Series

Evaluate the sum of a series for the symbolic expressions `k` and `k^2`:

```
syms k
symsum(k)
symsum(1/k^2)
```

```
ans =
k^2/2 - k/2
```

```
ans =
-psi(1, k)
```

Evaluate the Sum of a Series Specifying Bounds

Evaluate the sum of a series for these expressions specifying the bounds:

```
syms k
symsum(k^2, 0, 10)
symsum(1/k^2,1,Inf)
```

```
ans =
385

ans =
pi^2/6
```

You also can specify bounds as a row or column vector:

```
syms k
symsum(k^2, [0, 10])
symsum(1/k^2, [1; Inf])
```

```
ans =
385

ans =
pi^2/6
```

Evaluate the Sum of a Series Specifying the Summation Index and Bounds

Evaluate the sum of a series for this multivariable expression with respect to `k`:

```
syms k x
symsum(x^k/sym('k!'), k, 0, Inf)
```

```
ans =
exp(x)
```

Input Arguments

`expr` — 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 or function, a constant, or a vector or matrix of symbolic expressions, functions, or constants.

`var` — 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 expr 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).

More About

Definite Sum

The definite sum of series is defined as

$$\sum_{i=a}^{b} x_i = x_a + x_{a+1} + \ldots + x_b$$

Indefinite Sum

$$f(i) = \sum_{i} x_i$$

is called the indefinite sum of x_i over i, if the following identity is true for all values of i.

$$f(i+1) - f(i) = x_i$$

See Also

`cumprod` | `cumsum` | `int` | `symprod` | `syms` | `symvar`

Related Examples

- "Symbolic Summation" on page 2-19

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)` returns a vector containing n symbolic variables in s alphabetically closest to `x`. If s is a symbolic function, `symvar(s,n)` returns the input arguments of s in front of other free variables in s.

Input Arguments

`s`

Symbolic expression, matrix, or function.

`n`

Integer.

Examples

Find all symbolic variables in the sum:

```
syms wa wb wx yx ya yb
symvar(wa + wb + wx + ya + yb + yx)
```

```
ans =
[ wa, wb, wx, ya, yb, yx]
```

Find all symbolic variables in this function:

```
syms x y a b
f(a, b) = a*x^2/(sin(3*y - b));
symvar(f)

ans =
[ a, b, x, y]
```

Now find the first three symbolic variables in `f`. For a symbolic function, `symvar` with two arguments returns the function inputs in front of other variables:

```
symvar(f, 3)

ans =
[ a, b, x]
```

For a symbolic expression or matrix, `symvar` with two arguments returns variables sorted by their proximity to `x`:

```
symvar(a*x^2/(sin(3*y - b)), 3)

ans =
[ x, y, b]
```

Find the default symbolic variable of these expressions:

```
syms v z
g = v + z;
symvar(g, 1)

ans =
z

syms aaa aab
g = aaa + aab;
symvar(g, 1)

ans =
aaa

syms X1 x2 xa xb
g = X1 + x2 + xa + xb;
```

```
symvar(g, 1)

ans =
x2
```

More About

Tips

- `symvar(s)` can return variables in a different order than `symvar(s,n)`.
- `symvar` does treat the constants `pi`, `i`, and `j` as variables.
- If there are no symbolic variables in `s`, `symvar` returns the empty vector.
- When performing differentiation, integration, substitution or solving equations, MATLAB uses the variable returned by `symvar(s,1)` as a default variable. For a symbolic expression or matrix, `symvar(s,1)` returns the variable closest to `x`. For a function, `symvar(s,1)` returns the first input argument of s.

Algorithms

When sorting the symbolic variables by their proximity to `x`, `symvar` uses this algorithm:

1. The variables are sorted by the first letter in their names. The ordering is x y w z v u ... a X Y W Z V U ... A. The name of a symbolic variable cannot begin with a number.
2. For all subsequent letters, the ordering is alphabetical, with all uppercase letters having precedence over lowercase: 0 1 ... 9 A B ... Z a b ... z.

- "Find a Default Symbolic Variable" on page 1-14

See Also

`sym` | `symfun` | `syms`

tan

Symbolic tangent function

Syntax

```
tan(X)
```

Description

`tan(X)` returns the tangent function 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)]
```

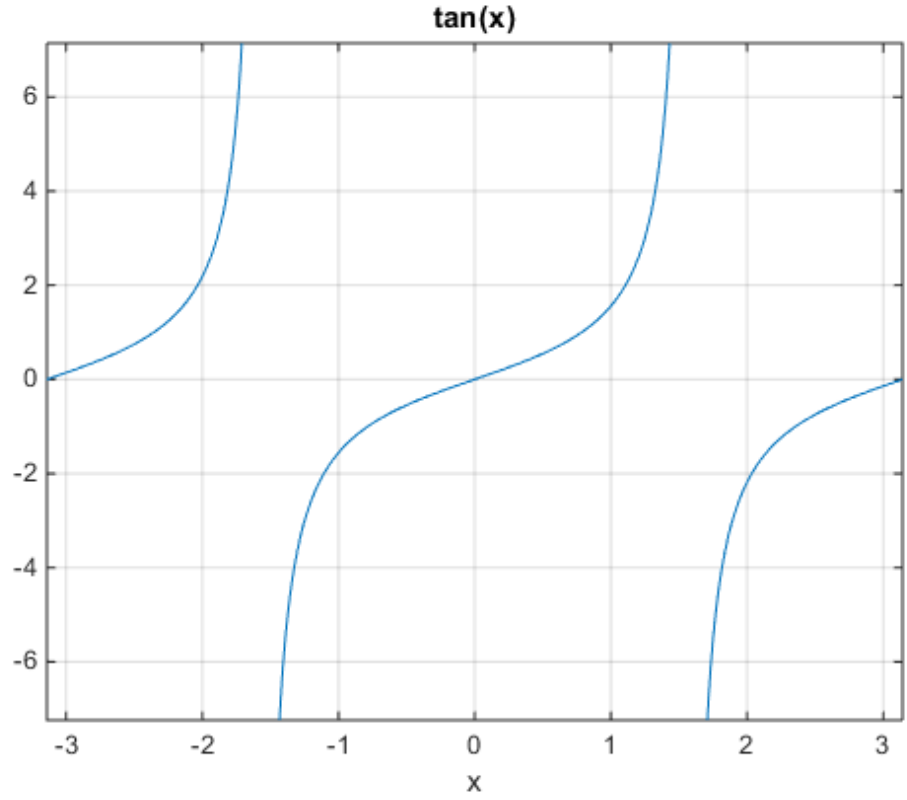
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ 2.185039863261518991643306102313 7,...
0,...
0.577350269189625764509148780501 96,...
-1.253960337662703837570910978336 5,...
-225.950846454195142025795483203 45]
```

Plot the Tangent Function

Plot the tangent function on the interval from $-\pi$ to π.

```
syms x
ezplot(tan(x), [-pi, pi])
grid on
```



Handle Expressions Containing the 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*2*i)*i - i)/(exp(x*2*i) + 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.

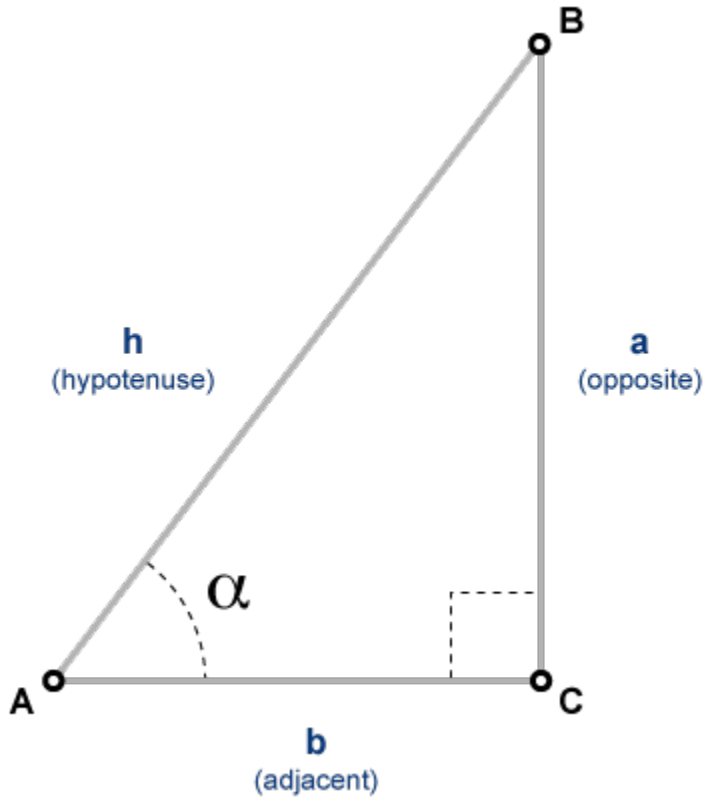
More About

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

$$\text{tangent}(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{i\left(e^{i\alpha} + e^{-i\alpha}\right)}.$$

.

See Also

acos | acot | acsc | asec | asin | atan | cos | cot | csc | sec | sin

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)*i)/3, 3^(1/2)*i, -tanh((pi*2*i)/7)]
```

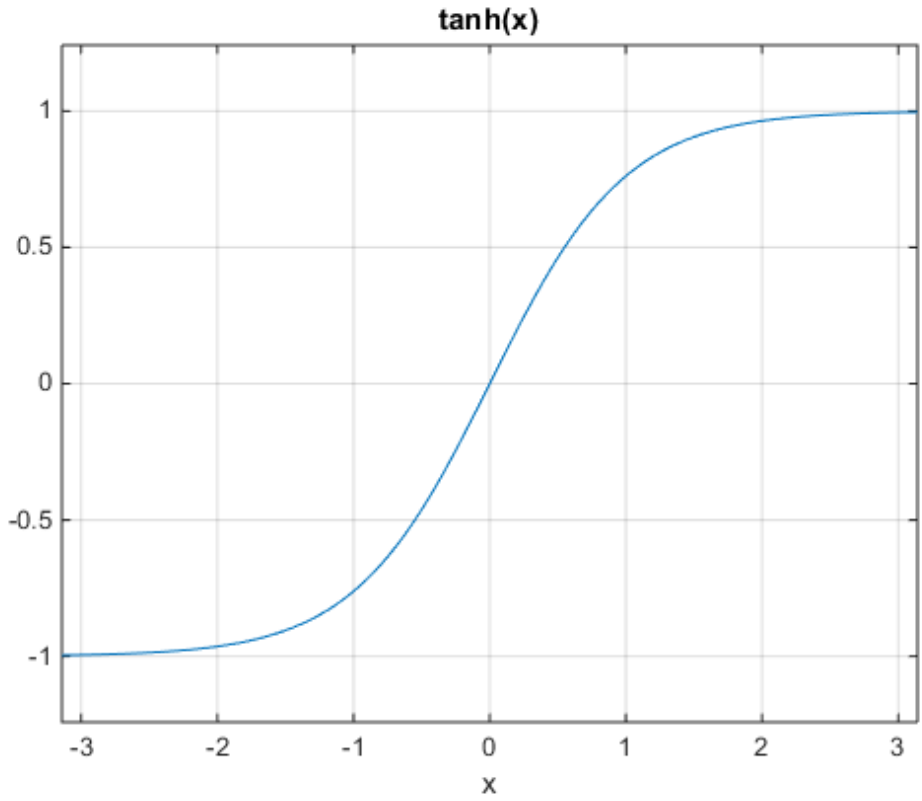
Use `vpa` to approximate symbolic results with floating-point numbers:

```
vpa(symA)

ans =
[ -0.964027580075816883946413724100 92,...
0,...
0.577350269189625764509148780501 96*i,...
1.732050807568877293527446341505 9*i,...
-1.253960337662703837570910978336 5*i]
```

Plot the Hyperbolic Tangent Function

Plot the hyperbolic tangent function on the interval from $-\pi$ to π.

```
syms x
ezplot(tanh(x), [-pi, pi])
grid on
```



Handle Expressions Containing the 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`

taylor

Taylor series expansion

Syntax

```
taylor(f)
taylor(f,Name,Value)
taylor(f,v)
taylor(f,v,Name,Value)
taylor(f,v,a)
taylor(f,v,a,Name,Value)
```

Description

`taylor(f)` computes the Taylor series expansion of f up to the fifth order. The expansion point is 0.

`taylor(f,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`taylor(f,v)` computes the Taylor series expansion of f with respect to v.

`taylor(f,v,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

`taylor(f,v,a)` computes the Taylor series expansion of f with respect to v around the expansion point a.

`taylor(f,v,a,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Input Arguments

`f`

Symbolic expression.

`v`

Symbolic variable or vector of symbolic variables with respect to which you want to compute the Taylor series expansion.

Default: Symbolic variable or vector of symbolic variables of f determined by `symvar`.

`a`

Real number (including infinities and symbolic numbers) specifying the expansion point. For multivariate Taylor series expansions, use a vector of numbers.

Default: 0

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'ExpansionPoint'`

Specify the expansion point `a`. The value `a` is a scalar or a vector.

Default: If you specify the expansion point as a third argument a of `taylor`, then the value of that argument. Otherwise, 0.

`'Order'`

Specify the truncation order `n`, where `n` is a positive integer. `taylor` computes the Taylor polynomial approximation with the order `n-1`. The truncation order `n` is the exponent in the *O*-term: $O(v^n)$.

Default: 6

`'OrderMode'`

Specify whether you want to use absolute or relative order when computing the Taylor polynomial approximation. The value must be one of these strings: `Absolute` or `Relative`. *Absolute order* is the truncation order of the computed series. *Relative order*

`n` means that the exponents of v in the computed series range from the leading order `m` to the highest exponent `m + n - 1`. Here `m + n` is the exponent of v in the *O*-term: $O(v^{m+n})$.

Default: `Absolute`

Examples

Compute the Maclaurin series expansions of these functions:

```
syms x
taylor(exp(x))
taylor(sin(x))
taylor(cos(x))
```

```
ans =
x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1

ans =
x^5/120 - x^3/6 + x

ans =
x^4/24 - x^2/2 + 1
```

Compute the Taylor series expansions around $x = 1$ for these functions. The default expansion point is 0. To specify a different expansion point, use `ExpansionPoint`:

```
syms x
taylor(log(x), x, 'ExpansionPoint', 1)
```

```
ans =
x - (x - 1)^2/2 + (x - 1)^3/3 - (x - 1)^4/4 + (x - 1)^5/5 - 1
```

Alternatively, specify the expansion point as the third argument of `taylor`:

```
taylor(acot(x), x, 1)
```

```
ans =
pi/4 - x/2 + (x - 1)^2/4 - (x - 1)^3/12 + (x - 1)^5/40 + 1/2
```

Compute the Maclaurin series expansion for this function. The default truncation order is 6. Taylor series approximation of this function does not have a fifth-degree term, so `taylor` approximates this function with the fourth-degree polynomial:

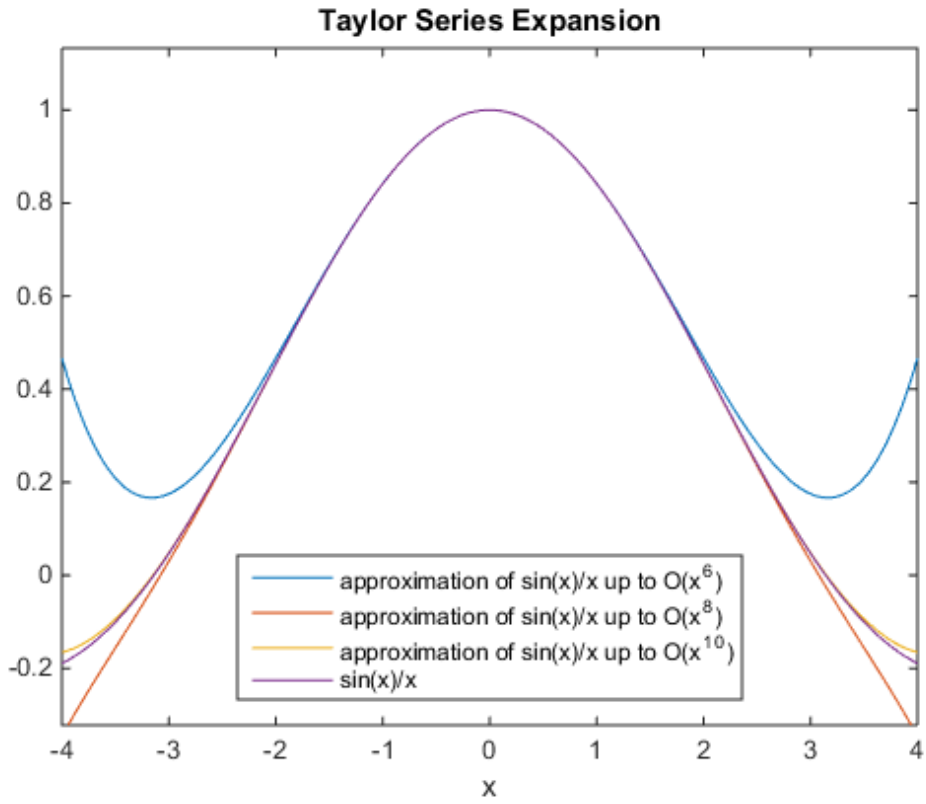
```
syms x
f = sin(x)/x;
t6 = taylor(f)

t6 =
x^4/120 - x^2/6 + 1
```

Use `Order` to control the truncation order. For example, approximate the function up to the orders 8 and 10:

```
t8 = taylor(f, 'Order', 8)
t10 = taylor(f, 'Order', 10)

t8 =
- x^6/5040 + x^4/120 - x^2/6 + 1

t10 =
x^8/362880 - x^6/5040 + x^4/120 - x^2/6 + 1
```

Plot the original function `f` and its approximations `t6`, `t8`, and `t10`. Note how the accuracy of the approximation depends on the truncation order.

```
ezplot(t6, [-4, 4])
hold on
ezplot(t8, [-4, 4])
ezplot(t10, [-4, 4])
ezplot(f, [-4, 4])

legend('approximation of sin(x)/x up to O(x^6)',...
'approximation of sin(x)/x up to O(x^8)',...
'approximation of sin(x)/x up to O(x^1^0)',...
'sin(x)/x',...
'Location', 'South')

title('Taylor Series Expansion')
hold off
```



Compute the Taylor series expansion of this expression. By default, `taylor` uses an absolute order, which is the truncation order of the computed series.

```
taylor(1/(exp(x)) - exp(x) + 2*x, x, 'Order', 5)
```

```
ans =
-x^3/3
```

To compute the Taylor series expansion with a relative truncation order, use `OrderMode`. For some expressions, a relative truncation order provides more accurate approximations.

```
taylor(1/(exp(x)) - exp(x) + 2*x, x, 'Order', 5, 'OrderMode', 'Relative')
```

```
ans =
```

```
- x^7/2520 - x^5/60 - x^3/3
```

Compute the Maclaurin series expansion of this multivariate function. If you do not specify the vector of variables, `taylor` treats `f` as a function of one independent variable.

```
syms x y z
f = sin(x) + cos(y) + exp(z);
taylor(f)
```

```
ans =
x^5/120 - x^3/6 + x + cos(y) + exp(z)
```

Compute the multivariate Maclaurin expansion by specifying the vector of variables:

```
syms x y z
f = sin(x) + cos(y) + exp(z);
taylor(f, [x, y, z])
```

```
ans =
x^5/120 - x^3/6 + x + y^4/24 - y^2/2 + z^5/120 + z^4/24 + z^3/6 + z^2/2 + z + 2
```

Compute the multivariate Taylor expansion by specifying both the vector of variables and the vector of values defining the expansion point:

```
syms x y
f = y*exp(x - 1) - x*log(y);
taylor(f, [x, y], [1, 1], 'Order', 3)
```

```
ans =
x + (x - 1)^2/2 + (y - 1)^2/2
```

If you specify the expansion point as a scalar `a`, `taylor` transforms that scalar into a vector of the same length as the vector of variables. All elements of the expansion vector equal `a`:

```
taylor(f, [x, y], 1, 'Order', 3)
```

```
ans =
x + (x - 1)^2/2 + (y - 1)^2/2
```

More About

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 v is a vector, then the expansion point `a` must be a scalar or a vector of the same length as v. If v is a vector and `a` is a scalar, then `a` is expanded into a vector of the same length as v with all elements equal to `a`.

- "Taylor Series" on page 2-20

See Also

`pade` | `symvar` | `taylortool`

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.

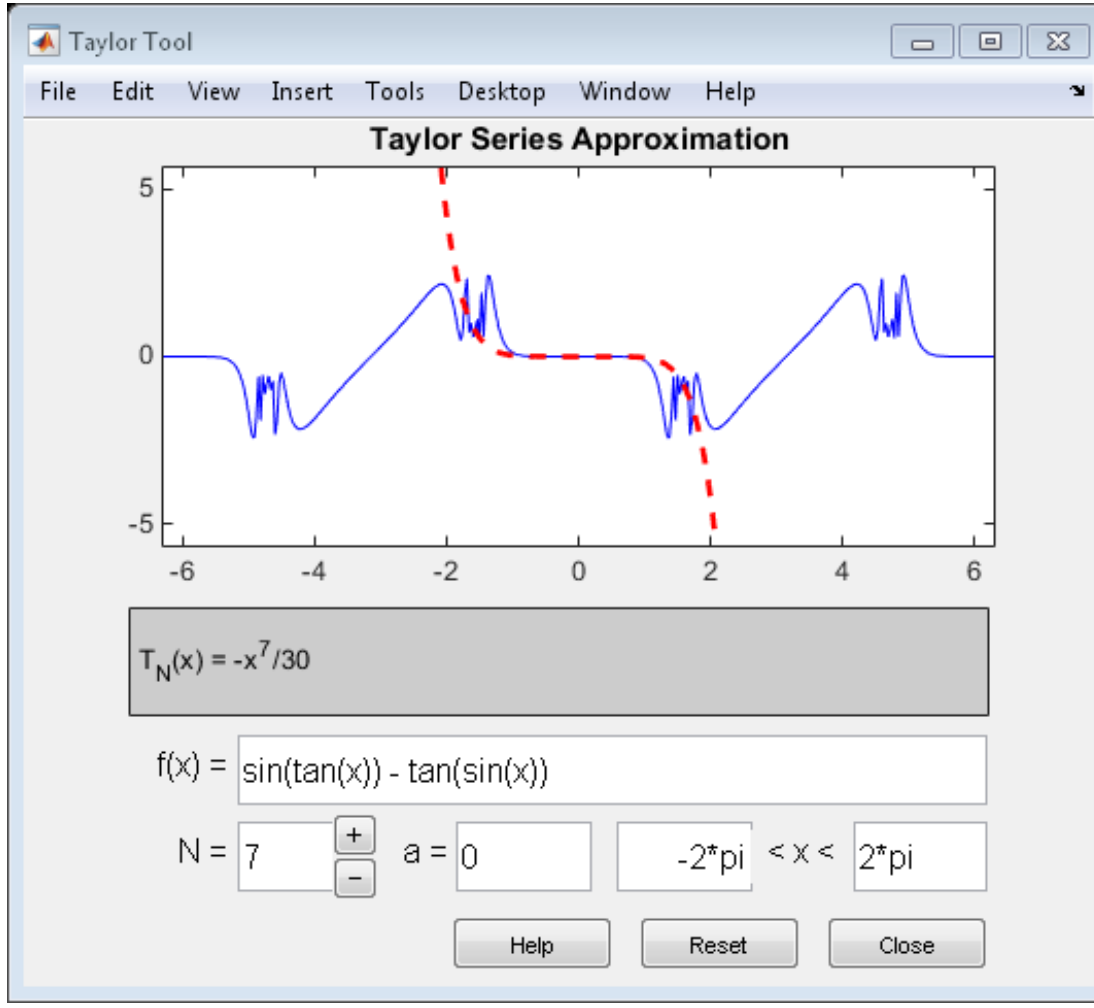
`taylortool('f')` initiates the GUI for the given expression `f`.

Examples

Open the Taylor Series Calculator For a Particular Expression

Open the Taylor series calculator for `sin(tan(x)) - tan(sin(x))`:

```
taylortool('sin(tan(x)) - tan(sin(x))')
```



More About

- "Taylor Series" on page 2-20

See Also

`funtool` | `rsums`

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 text strings. `texlabel` converts Greek variable names, such as delta, into Greek letters. Annotation functions, such as `title`, `xlabel`, and `text` can use the TeX string as input. To obtain the LaTeX representation, use `latex`.

`texlabel(expr,'literal')` interprets Greek variable names literally.

Examples

Generate TeX String

Use `texlabel` to generate TeX strings 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} - {\pi} {\delta}^{{2}/{3}}
```

To make `texlabel` interpret Greek variable names literally, include the argument `'literal'`.

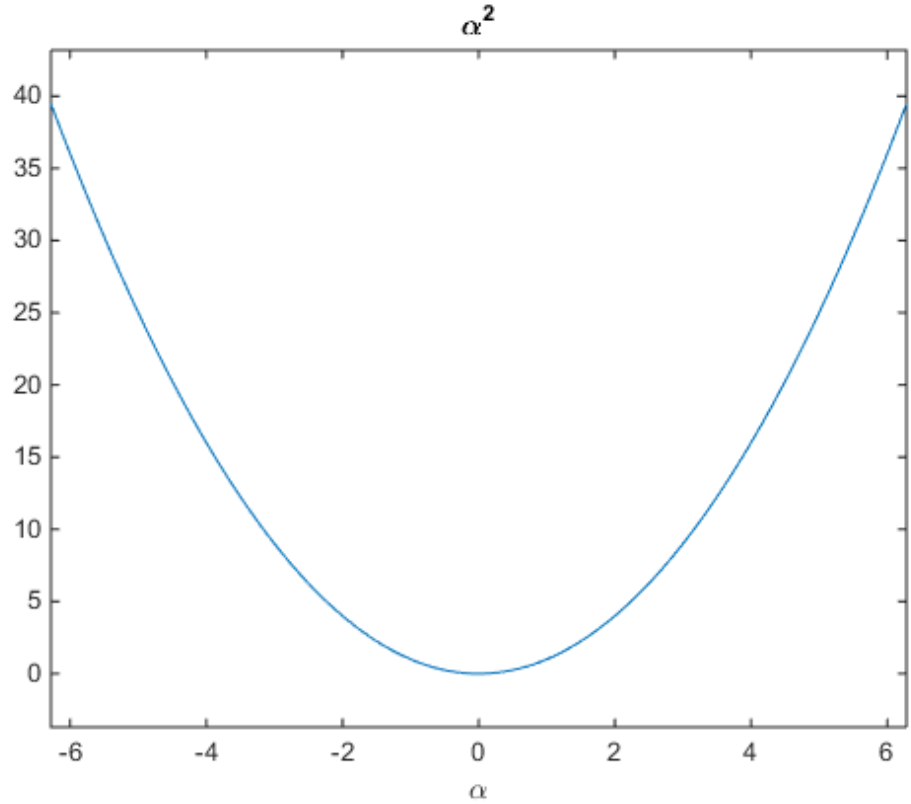
```
texlabel(lambda12,'literal')

ans =
{lambda12}
```

Insert TeX String in a Figure

Use `texlabel` to generate a TeX string that `text` inserts into a figure.

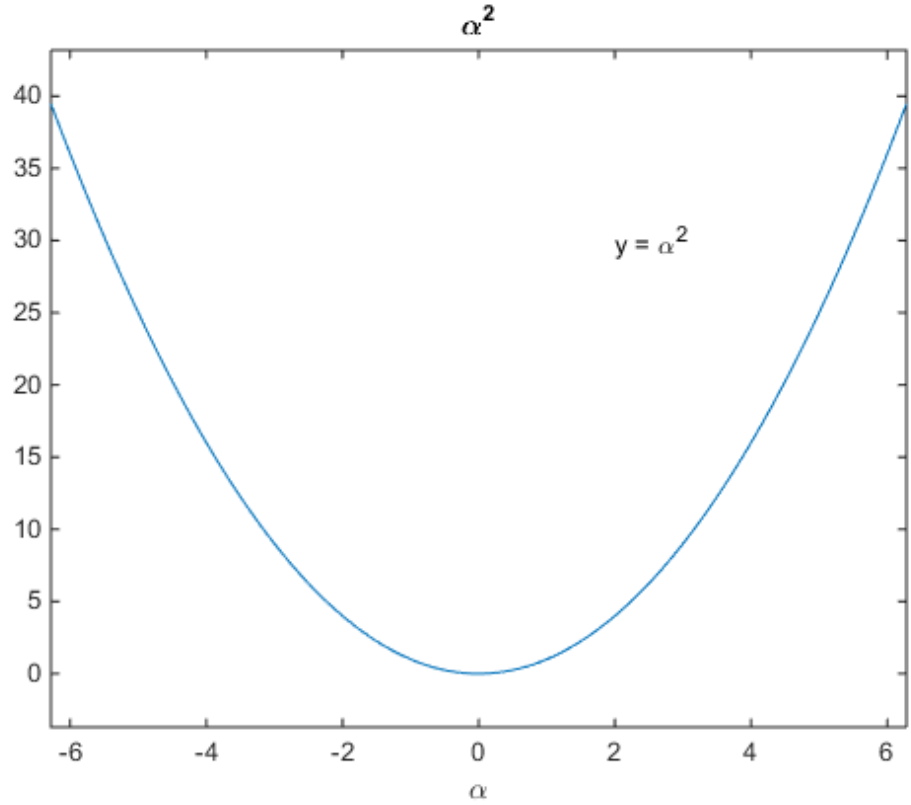
Plot `y = x^2`.

```
syms alpha
expr = alpha^2;
ezplot(expr)
```



Display the plotted expression on the plot.

```
expr = texlabel(expr);
text(2,30,['y = ' expr]);
```



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`

toeplitz

Symbolic Toeplitz matrix

Syntax

```
toeplitz(c,r)
toeplitz(r)
```

Description

`toeplitz(c,r)` generates a nonsymmetric Toeplitz matrix having c as its first column and r as its first row. If the first elements of c and r are different, `toeplitz` issues a warning and uses the first element of the column.

`toeplitz(r)` generates a symmetric Toeplitz matrix if r is real. If r is complex, but its first element is real, then this syntax generates the Hermitian Toeplitz matrix formed from r. If the first element of r is not real, then the resulting matrix is Hermitian off the main diagonal, meaning that T_{ij} = conjugate(T_{ji}) for $i \neq j$.

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, i]
[  2, 1, 2]
[ -i, 2, 1]
```

If the first element of the vector is real, then the resulting Toeplitz matrix is Hermitian:

```
logical(T == T')
```

```
ans =
     1     1     1
     1     1     1
     1     1     1
```

If the first element is not real, then the resulting Toeplitz matrix is Hermitian off the main diagonal:

```
T = toeplitz(sym([i, 2, 1]))
```

```
T =
[ i, 2, 1]
[ 2, i, 2]
[ 1, 2, i]
```

```
logical(T == T')
```

```
ans =
     0     1     1
     1     0     1
     1     1     0
```

Generate a Toeplitz matrix using these vectors to specify the first column and the first row. Because the first elements of these vectors are different, `toeplitz` issues a warning and uses the first element of the column:

```
syms a b c
```

```
toeplitz([a b c], [1 b/2 a/2])
```

```
Warning: First element of input column does not match first element of input row.
Column wins diagonal conflict. [linalg::toeplitz]

ans =
[ a, b/2, a/2]
[ b,   a, b/2]
[ c,   b,   a]
```

More About

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

`linalg::toeplitz` | `toeplitz`

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

Compute the triangular pulse function for these numbers. Because these numbers are not symbolic objects, you get floating-point results:

```
[triangularPulse(-2, 0, 2, -3)
triangularPulse(-2, 0, 2, -1/2)
triangularPulse(-2, 0, 2, 0)
triangularPulse(-2, 0, 2, 3/2)
triangularPulse(-2, 0, 2, 3)]
```

```
ans =
         0
    0.7500
    1.0000
    0.2500
         0
```

Compute the triangular pulse function for the numbers converted to symbolic objects:

```
[triangularPulse(sym(-2), 0, 2, -3)
triangularPulse(-2, 0, 2, sym(-1/2))
triangularPulse(-2, sym(0), 2, 0)
triangularPulse(-2, 0, 2, sym(3/2))
triangularPulse(-2, 0, sym(2), 3)]
```

```
ans =
   0
 3/4
   1
 1/4
   0
```

Compute the triangular pulse function for `a < x < b`:

```
syms a b c x
```

```
assume(a < x < b)
triangularPulse(a, b, c, x)

ans =
(a - x)/(a - b)
```

For further computations, remove the assumption:

```
syms a b x clear
```

Compute the triangular pulse function for `b < x < c`:

```
assume(b < x < c)
triangularPulse(a, b, c, x)

ans =
-(c - x)/(b - c)
```

For further computations, remove the assumption:

```
syms b c x clear
```

Compute the triangular pulse function for `a = b`:

```
syms a b c x
assume(b < c)
triangularPulse(b, b, c, x)

ans =
-((c - x)*rectangularPulse(b, c, x))/(b - c)
```

Compute the triangular pulse function for `c = b`:

```
assume(a < b)
triangularPulse(a, b, b, x)

ans =
((a - x)*rectangularPulse(a, b, x))/(a - b)
```

For further computations, remove all assumptions on `a`, `b`, and `c`:

```
syms a b c clear
```

Use `triangularPulse` with one input argument as a shortcut for computing `triangularPulse(-1, 0, 1, x)`:

```
syms x
triangularPulse(x)
```

```
ans =
triangularPulse(-1, 0, 1, x)
```

```
[triangularPulse(sym(-10))
triangularPulse(sym(-3/4))
triangularPulse(sym(0))
triangularPulse(sym(2/3))
triangularPulse(sym(1))]
```

```
ans =
   0
 1/4
   1
 1/3
   0
```

Use `triangularPulse` with three input arguments as a shortcut for computing `triangularPulse(a, (a + c)/2, c, x)`:

```
syms a c x
triangularPulse(a, c, x)
```

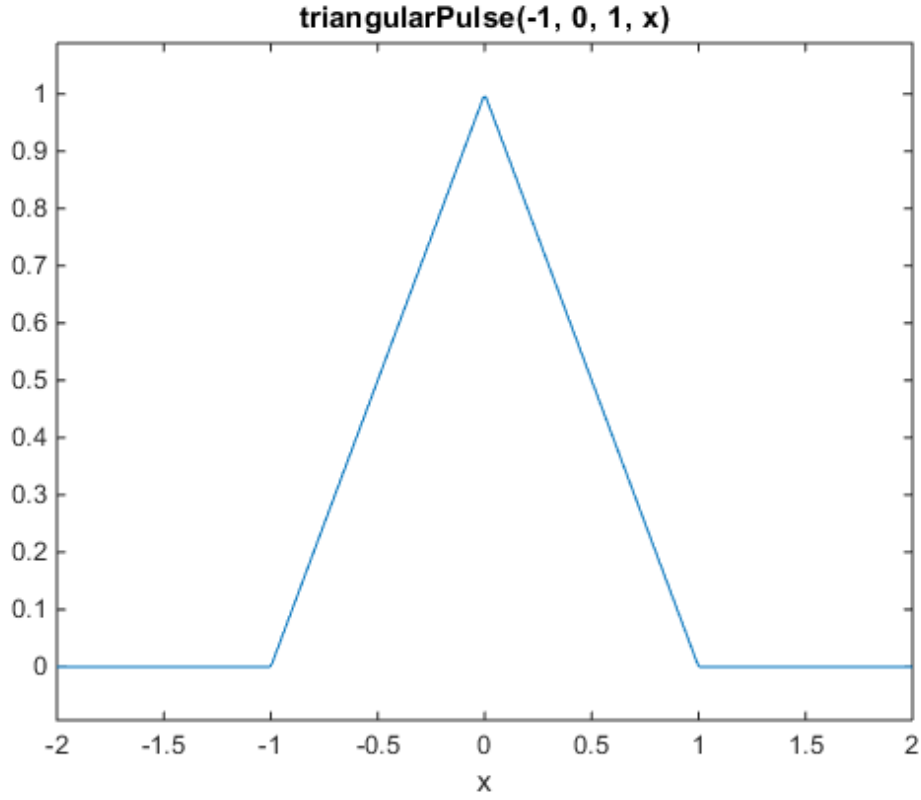
```
ans =
triangularPulse(a, a/2 + c/2, c, x)
```

```
[triangularPulse(sym(-10), 10, 3)
triangularPulse(sym(-1/2), -1/4, -2/3)
triangularPulse(sym(2), 4, 3)
triangularPulse(sym(2), 4, 6)
triangularPulse(sym(-1), 4, 0)]
```

```
ans =
 7/10
    0
    1
    0
  2/5
```

Plot the triangular pulse function:

```
syms x
ezplot(triangularPulse(x), [-2, 2])
```



Call `triangularPulse` with infinities as its rising and falling edges:

```
syms x
triangularPulse(-1, 0, inf, x)
triangularPulse(-inf, 0, 1, x)
triangularPulse(-inf, 0, inf, x)

ans =
heaviside(x) + (x + 1)*rectangularPulse(-1, 0, x)

ans =
heaviside(-x) - (x - 1)*rectangularPulse(0, 1, x)

ans =
1
```

More About

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`

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`

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`

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`
`uint64`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	8	`uint64`

See Also

`sym` | `vpa` | `single` | `int8` | `int16` | `int32` | `int64` | `double`

vectorPotential

Vector potential of vector field

Syntax

```
vectorPotential(V,X)
vectorPotential(V)
```

Description

`vectorPotential(V,X)` computes the vector potential of the vector field V with respect to the vector X in Cartesian coordinates. The vector field V and the vector X are both three-dimensional.

`vectorPotential(V)` returns the vector potential V with respect to a vector constructed from the first three symbolic variables found in V by `symvar`.

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
```

More About

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`

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`

vpa

Variable-precision arithmetic

Syntax

```
R = vpa(A)
R = vpa(A,d)
```

Description

`R = vpa(A)` uses variable-precision arithmetic (VPA) to compute each element of `A` to at least `d` decimal digits of accuracy, where `d` is the current setting of `digits`.

`R = vpa(A,d)` uses at least `d` significant (nonzero) digits, instead of the current setting of `digits`.

Input Arguments

`A`

Symbolic object, string, or numeric expression.

`d`

Integer greater than 1 and smaller than $2^{29}+1$.

Output Arguments

`R`

Symbolic object representing a floating-point number

Examples

Approximate the following expressions with the 25 digits precision:

```
old = digits(25);
q = vpa('1/2')
p = vpa(pi)
w = vpa('(1+sqrt(5))/2')
digits(old)

q =
0.5

p =
3.14159265358979323846264

w =
1.6180339887498948482045
```

Solve the following equation:

```
syms x
y = solve(x^2 - 2,x)

y =
  2^(1/2)
 -2^(1/2)
```

Approximate the solutions with floating-point numbers:

```
vpa(y(1))
vpa(y(2))

ans =
1.4142135623730950488016887242097

ans =
-1.4142135623730950488016887242097
```

Use the vpa function to approximate elements of the following matrices:

```
A = vpa(hilb(2), 25)
B = vpa(hilb(2), 5)

A =
[ 1.0,                          0.5]
[ 0.5, 0.33333333333333333333333333]

B =
[ 1.0,     0.5]
[ 0.5, 0.33333]
```

The vpa function lets you specify a number of significant (nonzero) digits that is different from the current digits setting. For example, compute the ratio 1/3 and the ratio 1/3000 with 4 significant digits:

```
vpa(1/3, 4)
vpa(1/3000, 4)
```

```
ans =
0.3333

ans =
0.0003333
```

The number of digits that you specify by the vpa function or the digits function is the minimal number of digits. Internally, the toolbox can use more digits than you specify. These additional digits are called guard digits. For example, set the number of digits to 4, and then display the floating-point approximation of 1/3 using 4 digits:

```
old = digits;
digits(4)
a = vpa(1/3, 4)
```

```
a =
0.3333
```

Now, display a using 20 digits. The result shows that the toolbox internally used more than 4 digits when computing a. The last digits in the following result are incorrect because of the round-off error:

```
vpa(a, 20)
digits(old)
```

```
ans =
0.33333333333303016843
```

Hidden round-off errors can cause unexpected results. For example, compute the number 1/10 with the default 32 digits accuracy and with the 10 digits accuracy:

```
a = vpa(1/10, 32)
b = vpa(1/10, 10)
```

```
a =
0.1

b =
0.1
```

Now, compute the difference `a - b`:

```
a - b
```

```
ans =
0.000000000000000000086736173798840354720600815844403
```

The difference is not equal to zero because the toolbox approximates the number `b=0.1` with 32 digits. This approximation produces round-off errors because the floating point number 0.1 is different from the rational number 1/10. When you compute the difference `a - b`, the toolbox actually computes the difference as follows:

```
a - vpa(b, 32)
```

```
ans =
0.000000000000000000086736173798840354720600815844403
```

Suppose you convert a number to a symbolic object, and then perform VPA operations on that object. The results can depend on the conversion technique that you used to convert a floating-point number to a symbolic object. The `sym` function lets you choose the conversion technique by specifying the optional second argument, which can be `'r'`, `'f'`, `'d'` or `'e'`. The default is `'r'`. For example, convert the constant π=3.141592653589793... to a symbolic object:

```
r = sym(pi)
f = sym(pi, 'f')
d = sym(pi, 'd')
e = sym(pi, 'e')
```

```
r =
pi

f =
884279719003555/281474976710656

d =
3.14159265358979311599796346854 42

e =
pi - (198*eps)/359
```

Compute these numbers with the 4 digits VPA precision. Three of the four numeric approximations give the same result:

```
vpa(r, 4)
```

```
vpa(f, 4)
vpa(d, 4)
vpa(e, 4)

ans =
3.142

ans =
3.142

ans =
3.142

ans =
3.142 - 0.5515*eps
```

Now, increase the VPA precision to 40 digits. The numeric approximation of 1/10 depends on the technique that you used to convert 1/10 to the symbolic object:

```
vpa(r, 40)
vpa(f, 40)
vpa(d, 40)
vpa(e, 40)

ans =
3.141592653589793238462643383279502884197

ans =
3.141592653589793115997963468544185161591

ans =
3.141592653589793115997963468544185161591

ans =
3.141592653589793238462643383279502884197 -...
0.551532033426183844011142061281337047353 8*eps
```

More About

Tips

- The toolbox increases the internal precision of calculations by several digits (guard digits).

- When you apply `vpa` to a numeric expression, such as `1/3`, `2^(-5)`, or `sin(pi/4)`, it is evaluated to a double-precision number. Then, `vpa` is applied to that double-precision number. For more accurate results, convert numeric expressions to symbolic expressions. For example, to approximate `exp(1)` use `vpa(sym(exp(1)))`.
- If the value d is not an integer, `vpa` rounds it to the nearest integer.

- "Control Accuracy of Variable-Precision Computations"
- "Recognize and Avoid Round-Off Errors"
- "Improve Performance of Numeric Computations"
- "Choose the Arithmetic"

See Also

`digits` | `double`

vpasolve

Numeric solver

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 a 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 =
 - 0.880113771260681698178751904578 35 - 0.763315833877154525129784681022 63*i
   0.505113771260681698178751904578 35 + 0.815989650689463128532270678906 56*i
   0.505113771260681698178751904578 35 - 0.815989650689463128532270678906 56*i
 - 0.880113771260681698178751904578 35 + 0.763315833877154525129784681022 63*i
```

Solve a 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 a Structure Array

When solving a system of equations, use one output argument to return the solutions in the form of a structure array:

```
syms x y
S = vpasolve([x^3 + 2*x == y, y^2 == x], [x, y])
```

```
S =
    x: [6x1 sym]
    y: [6x1 sym]
```

Display solutions by accessing the elements of the structure array S:

```
S.x
```

```
ans =
                                                                    0
                                          0.23657429427733416176148715217768
 - 0.28124065338711968666197895499453 + 1.23487242364701420748598945319460*i
   0.16295350624845260578123537890613 + 1.61515446505553669178865854179260*i
   0.16295350624845260578123537890613 - 1.61515446505553669178865854179260*i
 - 0.28124065338711968666197895499453 - 1.23487242364701420748598945319460*i
```

```
S.y
```

```
ans =
                                                                    0
                                          0.48638903593454300001655725369801
   0.70187356885586188630668751791218 + 0.87969719792982402287026727381769*i
 - 0.94506808682313338631496614476119 - 0.85451751443904587692179191887616*i
 - 0.94506808682313338631496614476119 + 0.85451751443904587692179191887616*i
   0.70187356885586188630668751791218 - 0.87969719792982402287026727381769*i
```

Assign Solutions to Variables When Solving a 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 =
88.907072096591148648492807746810
```

```
sol_y =
```

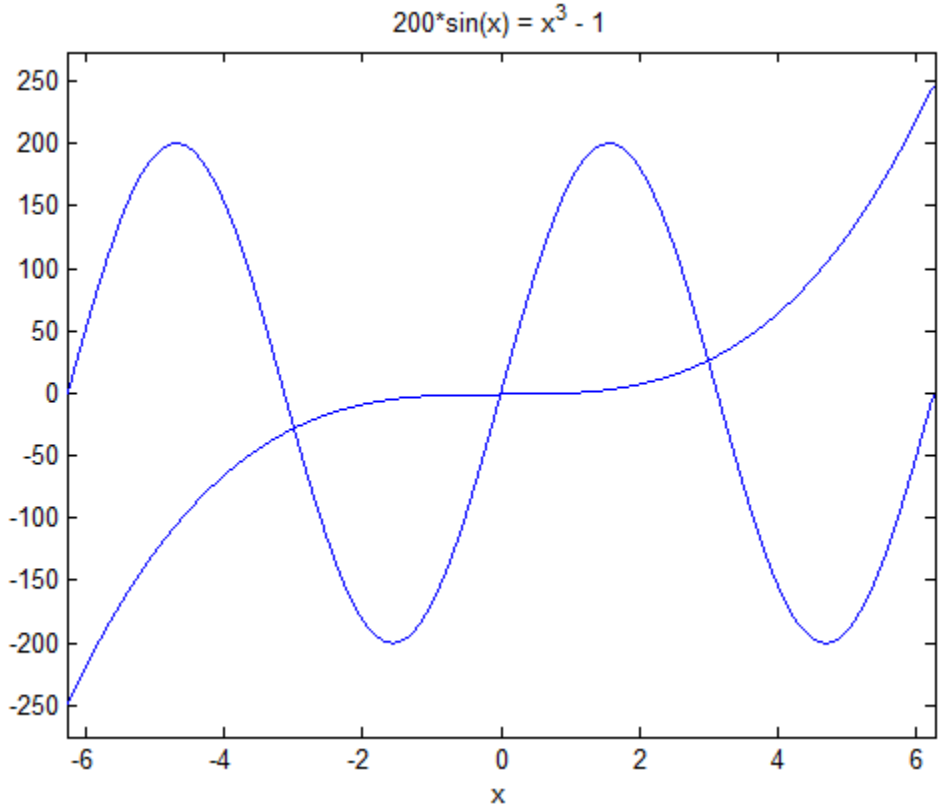
```
0.00000000000013470479710676694388973703681918
```

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
ezplot(200*sin(x))
hold on
ezplot(x^3 - 1)
title('200*sin(x) = x^3 - 1')
```



This equation has three solutions. If you do not specify the initial guess (zero-approximation), `vpasolve` returns the first solution that it finds:

```
vpasolve(200*sin(x) == x^3 - 1, x)
```

```
ans =
-0.0050000214585835715725440675982988
```

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 of an Equation

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 =
                                        -1.205249716379906069588839726434 1
                                         1.205249716379906069588839726434 1
 - 0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
 - 0.770614313708030291274954267474 28 - 0.859152076039938188593211427571 64*i
                                          1.078904602033826530804743628420 5*i
                                         -1.078904602033826530804743628420 5*i
   0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
   0.770614313708030291274954267474 28 - 0.859152076039938188593211427571 64*i
```

Suppose you need only real solutions of this equation. You cannot use assumptions on variables because `vpasolve` ignores them.

```
assume(x, 'real')
vpasolve(x^8 - x^2 == 3, x)
```

```
ans =
                                        -1.205249716379906069588839726434 1
```

```
                                          1.205249716379906069588839726434 1
 - 0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
 - 0.770614313708030291274954267474 28 - 0.859152076039938188593211427571 64*i
                                           1.078904602033826530804743628420 5*i
                                          -1.078904602033826530804743628420 5*i
   0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
   0.770614313708030291274954267474 28 - 0.859152076039938188593211427571 64*i
```

Specify the search range to restrict the returned results to particular ranges. For example, to return only real solutions of this equation, specify the search interval as `[-Inf Inf]`:

```
vpasolve(x^8 - x^2 == 3, x, [-Inf Inf])
```

```
ans =
 -1.205249716379906069588839726434 1
  1.205249716379906069588839726434 1
```

Return only nonnegative solutions:

```
vpasolve(x^8 - x^2 == 3, x, [0 Inf])
```

```
ans =
1.205249716379906069588839726434 1
```

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 =
 - 0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
   0.770614313708030291274954267474 28 + 0.859152076039938188593211427571 64*i
```

Find Multiple Solutions for a 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.

More About

Tips

- `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`

whittakerM

Whittaker M function

Syntax

```
whittakerM(a,b,z)
```

Description

`whittakerM(a,b,z)` returns the value of the Whittaker M function.

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
simplify(diff(whittakerM(a,b,z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(a,b,z)) == 0

ans =
     1
```

Verify that `whittakerM(-a,-b,-z)` also is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerM(-a,-b,-z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerM(-a,-b,-z)) == 0

ans =
     1
```

Compute the Whittaker M function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[whittakerM(1, 1, 1), whittakerM(-2, 1, 3/2 + 2*i),...
whittakerM(2, 2, 2), whittakerM(3, -0.3, 1/101)]

ans =
   0.7303            -9.2744 + 5.4705i   2.6328             0.3681
```

Compute the Whittaker M function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `whittakerM` returns unresolved symbolic calls.

```
[whittakerM(sym(1), 1, 1), whittakerM(-2, sym(1), 3/2 + 2*i),...
whittakerM(2, 2, sym(2)), whittakerM(sym(3), -0.3, 1/101)]

ans =
[ whittakerM(1, 1, 1), whittakerM(-2, 1, 3/2 + 2*i),
whittakerM(2, 2, 2), whittakerM(3, -3/10, 1/101)]
```

For symbolic variables and expressions, `whittakerM` also returns unresolved symbolic calls:

```
syms a b x y
[whittakerM(a, b, x), whittakerM(1, x, x^2),...
whittakerM(2, x, y), whittakerM(3, x + y, x*y)]

ans =
[ whittakerM(a, b, x), whittakerM(1, x, x^2),...
whittakerM(2, x, y), whittakerM(3, x + y, x*y)]
```

The Whittaker M function has special values for some parameters:

```
whittakerM(sym(-3/2), 1, 1)

ans =
exp(1/2)

syms a b x
whittakerM(0, b, x)

ans =
4^b*x^(1/2)*gamma(b + 1)*besseli(b, x/2)

whittakerM(a + 1/2, a, x)

ans =
x^(a + 1/2)*exp(-x/2)

whittakerM(a, a - 5/2, x)

ans =
(2*x^(a - 2)*exp(-x/2)*(2*a^2 - 7*a + x^2/2 -...
x*(2*a - 3) + 6))/pochhammer(2*a - 4, 2)
```

Differentiate the expression involving the Whittaker M function:

```
syms a b z
diff(whittakerM(a,b,z), z)

ans =
(whittakerM(a + 1, b, z)*(a + b + 1/2))/z -...
(a/z - 1/2)*whittakerM(a, b, z)
```

Compute the Whittaker M function for the elements of matrix `A`:

```
syms x
A = [-1, x^2; 0, x];
whittakerM(-1/2, 0, A)
```

```
ans =
[ exp(-1/2)*i, exp(x^2/2)*(x^2)^(1/2)]
[           0,       x^(1/2)*exp(x/2)]
```

More About

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`

whittakerW

Whittaker W function

Syntax

```
whittakerW(a,b,z)
```

Description

`whittakerW(a,b,z)` returns the value of the Whittaker W function.

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
simplify(diff(whittakerW(a, b, z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(a, b, z)) == 0

ans =
     1
```

Verify that `whittakerW(-a, -b, -z)` also is a valid solution of this differential equation:

```
syms a b z
simplify(diff(whittakerW(-a, -b, -z), z, 2) +...
(-1/4 + a/z + (1/4 - b^2)/z^2)*whittakerW(-a, -b, -z)) == 0

ans =
     1
```

Compute the Whittaker W function for these numbers. Because these numbers are not symbolic objects, you get floating-point results.

```
[whittakerW(1, 1, 1), whittakerW(-2, 1, 3/2 + 2*i),...
whittakerW(2, 2, 2), whittakerW(3, -0.3, 1/101)]

ans =
   1.1953            -0.0156 - 0.0225i   4.8616            -0.1692
```

Compute the Whittaker W function for the numbers converted to symbolic objects. For most symbolic (exact) numbers, `whittakerW` returns unresolved symbolic calls.

```
[whittakerW(sym(1), 1, 1), whittakerW(-2, sym(1), 3/2 + 2*i),...
whittakerW(2, 2, sym(2)), whittakerW(sym(3), -0.3, 1/101)]

ans =
[ whittakerW(1, 1, 1), whittakerW(-2, 1, 3/2 + 2*i),
whittakerW(2, 2, 2), whittakerW(3, -3/10, 1/101)]
```

For symbolic variables and expressions, `whittakerW` also returns unresolved symbolic calls:

```
syms a b x y
[whittakerW(a, b, x), whittakerW(1, x, x^2),...
whittakerW(2, x, y), whittakerW(3, x + y, x*y)]

ans =
[ whittakerW(a, b, x), whittakerW(1, x, x^2),
whittakerW(2, x, y), whittakerW(3, x + y, x*y)]
```

The Whittaker W function has special values for some parameters:

```
whittakerW(sym(-3/2), 1/2, 0)

ans =
4/(3*pi^(1/2))

syms a b x
whittakerW(0, b, x)

ans =
(x^(b + 1/2)*besselk(b, x/2))/(pi^(1/2)*x^b)

whittakerW(a, -a + 1/2, x)

ans =
x^(1 - a)*x^(2*a - 1)*exp(-x/2)

whittakerW(a - 1/2, a, x)

ans =
(x^(a + 1/2)*exp(-x/2)*exp(x)*igamma(2*a, x))/x^(2*a)
```

Differentiate the expression involving the Whittaker W function:

```
syms a b z
diff(whittakerW(a,b,z), z)

ans =
- (a/z - 1/2)*whittakerW(a, b, z) -...
whittakerW(a + 1, b, z)/z
```

Compute the Whittaker W function for the elements of matrix `A`:

```
syms x
A = [-1, x^2; 0, x];
whittakerW(-1/2, 0, A)

ans =
```

```
[ -exp(-1/2)*(pi*i + ei(1))*i,
exp(x^2)*exp(-x^2/2)*expint(x^2)*(x^2)^(1/2)]
[  0,
x^(1/2)*exp(-x/2)*exp(x)*expint(x)]
```

More About

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`

wrightOmega

Wright omega function

Syntax

```
wrightOmega(x)
wrightOmega(A)
```

Description

`wrightOmega(x)` computes the Wright omega function of x.

`wrightOmega(A)` computes the Wright omega function of each element of A.

Input Arguments

x

Number, symbolic variable, or symbolic expression.

A

Vector or matrix of numbers, symbolic variables, or symbolic expressions.

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
```

More About

Wright omega Function

The Wright omega function is defined in terms of the Lambert W function:

$$\omega(x) = W_{\left\lceil \frac{\mathrm{Im}(x)-\pi}{2\pi} \right\rceil}\left(e^{x}\right)$$

The Wright omega function $\omega(x)$ is a solution of the equation $Y + \log(Y) = X$.

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`

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 `logical` or `isAlways`. If only one inequality is valid, the expression with `xor` evaluates to logical `1`. If both inequalities are valid, the expression with `xor` evaluates to logical `0`.

```
logical(x1)
isAlways(x2)

ans =
     1

ans =
     0
```

Note that `simplify` does not simplify these logical expressions to logical `1` or `0`. Instead, they return *symbolic* values `TRUE` or `FALSE`.

```
s1 = simplify(x1)
s2 = simplify(x2)

s1 =
TRUE

s2 =
FALSE
```

Convert symbolic `TRUE` or `FALSE` to logical values using `logical`:

```
logical(s1)
logical(s2)

ans =
     1

ans =
     0
```

More About

Tips

- If you call `simplify` for a logical expression containing symbolic subexpressions, you can get symbolic values `TRUE` or `FALSE`. These values are not the same as logical `1` (`true`) and logical `0` (`false`). To convert symbolic `TRUE` or `FALSE` to logical values, use `logical`.
- `assume` and `assumeAlso` do not accept assumptions that contain `xor`.

See Also

`all` | `and` | `any` | `isAlways` | `logical` | `not` | `or`

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 the 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 4-1160.

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 the 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 the 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`.

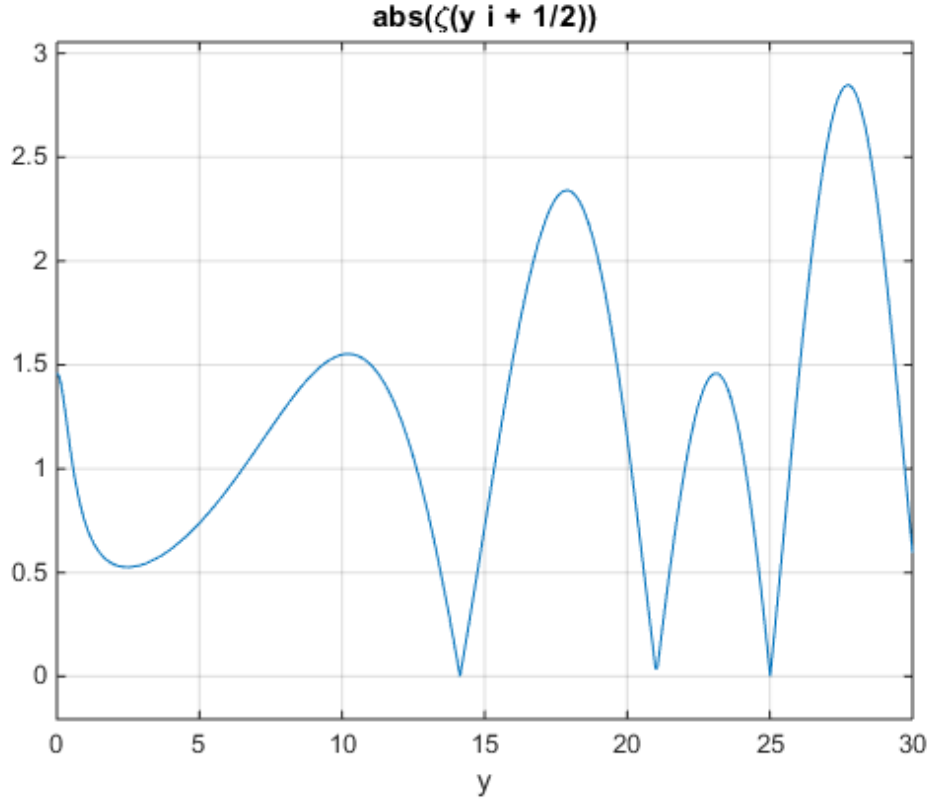
```
expr = vpa(expr)
```

```
expr =
-0.072640849891321371962446167811 77
```

Plot Zeros of the 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.

```
syms y;
ezplot(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.

More About

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`

ztrans

Z-transform

Syntax

```
ztrans(f,trans_index,eval_point)
```

Description

`ztrans(f,trans_index,eval_point)` computes the Z-transform of f with respect to the transformation index trans_index at the point eval_point.

Input Arguments

`f`

Symbolic expression, symbolic function, or vector or matrix of symbolic expressions or functions.

`trans_index`

Symbolic variable representing the transformation index. This variable is often called the "discrete time variable".

Default: The variable `n`. If f does not contain `n`, then the default variable is determined by `symvar`.

`eval_point`

Symbolic variable or expression representing the evaluation point. This variable is often called the "complex frequency variable".

Default: The variable `z`. If `z` is the transformation index of f, then the default evaluation point is the variable `w`.

Examples

Compute the Z-transform of this expression with respect to the transformation index `k` at the evaluation point `x`:

```
syms k x
f = sin(k);
ztrans(f, k, x)
```

```
ans =
(x*sin(1))/(x^2 - 2*cos(1)*x + 1)
```

Compute the Z-transform of this expression calling the `ztrans` function with one argument. If you do not specify the transformation index, `ztrans` uses the variable `n`.

```
syms a n x
f = a^n;
ztrans(f, x)
```

```
ans =
-x/(a - x)
```

If you also do not specify the evaluation point, `ztrans` uses the variable `z`:

```
ztrans(f)
```

```
ans =
-z/(a - z)
```

Compute the following Z-transforms that involve the Heaviside function and the binomial coefficient:

```
syms n z
ztrans(heaviside(n - 3), n, z)
```

```
ans =
(1/(z - 1) + 1/2)/z^3
```

```
ztrans(nchoosek(n, 2)*heaviside(5 - n), n, z)
```

```
ans =
z/(z - 1)^3 + 5/z^5 + (6*z - z^6/(z - 1)^3 + 3*z^2 + z^3)/z^5
```

If `ztrans` cannot find an explicit representation of the transform, it returns an unevaluated call:

```
syms f(n) z
F = ztrans(f, n, z)

F =
ztrans(f(n), n, z)
```

`iztrans` returns the original expression:

```
iztrans(F, z, n)

ans =
f(n)
```

Find the Z-transform of this matrix. Use matrices of the same size to specify the transformation variable and evaluation point.

```
syms a b c d w x y z
ztrans([exp(x), 1; sin(y), i*z],[w, x; y, z],[a, b; c, d])

ans =
[                (a*exp(x))/(a - 1),       b/(b - 1)]
[ (c*sin(1))/(c^2 - 2*cos(1)*c + 1), (d*i)/(d - 1)^2]
```

When the input arguments are nonscalars, `ztrans` acts on them element-wise. If `ztrans` is called with both scalar and nonscalar arguments, then `ztrans` expands the scalar arguments into arrays of the same size as the nonscalar arguments with all elements of the array equal to the scalar.

```
syms w x y z a b c d
ztrans(x,[x, w; y, z],[a, b; c, d])

ans =
[   a/(a - 1)^2, (b*x)/(b - 1)]
[ (c*x)/(c - 1), (d*x)/(d - 1)]
```

Note that nonscalar input arguments must have the same size.

When the first argument is a symbolic function, 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]
```

More About

Z-Transform

The Z-transform of the expression $f = f(n)$ is defined as follows:

$$F(z) = \sum_{n=0}^{\infty} \frac{f(n)}{z^n}.$$

Tips

- If you call `ztrans` with two arguments, it assumes that the second argument is the evaluation point eval_point.
- If f is a matrix, `ztrans` acts element-wise on all components of the matrix.
- If eval_point is a matrix, `ztrans` acts element-wise on all components of the matrix.
- To compute the inverse Z-transform, use `iztrans`.
- "Compute Z-Transforms and Inverse Z-Transforms" on page 2-197

See Also

`fourier` | `ifourier` | `ilaplace` | `iztrans` | `kroneckerDelta` | `laplace`