Built-In Functions in MATLAB

August 11, 2023

Built-In Functions in MATLAB

Random Functions

In engineering calculations, randomly generated numbers are frequently provided in order to simulate real-world measurements and data. The different MATLAB functions intended for this are as follows:

`rand()`

- The output of the **`rand()`** function is a matrix consisting of values from a standard uniform distribution in the interval **between 0 and 1**. Both dimensions of the matrix are equal to the value passed as an argument. If 1 or no argument is passed, a scalar is returned.

[ ]:
```
rand(3)
```

- The syntax for specifying the dimensions of the random number matrix is **`rand(M,N)`**, where M sets the number of rows and N is for the columns.

[ ]:
```
rand(5,3)
```

`randn()`

- The output of the **`randn()`** function is a matrix consisting of values from a *Gaussian distribution* or generated random data that are symmetrically distributed about a **mean** of **0**, with a **variance** of **1**. Both dimensions of the matrix are equal to the value passed as an argument. If 1 or no argument is passed, a scalar is returned.

[ ]:
```
randn(3)
```

- The syntax for specifying the matrix dimensions using the **`rand()`** function also applies to **`randn()`**.

[ ]:
```
randn(5,3)
```

Trigonometric Functions

The native functions in MATLAB consist of a comprehensive set of commands that are useful in trigonometric ratio calculations. These include standard trigonometric functions for both data in radians and degrees, functions for the inverse form, and hyperbolic trigonometric functions.

`sin()`

- The **sine** of an input value in *radians* can be obtained in MATLAB with the **`sin()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

[ ]:
```
x = 0;
e = exp(1);
l = log(e.^x);

y = sin(e.^l - e.^x)
```

`cos()`

- The **cosine** of an input value in *radians* can be obtained in MATLAB with the **`cos()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

[ ]:
```
c = cos(e.^l - e.^x)
```

`tan()`

- The **tangent** of an input value in *radians* can be obtained in MATLAB with the **`tan()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

[ ]:
```
j = 2;
i = -2;
p = pi;

t = tan((j.^i + (e - c*e))*p)
```

`csc()`

- The **cosecant** of an input value in *radians* can be obtained in MATLAB with the **`csc()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

```
[ ]: cs = csc((j.^i + (e - c*e))*p)
```

`sec()`

- The **secant** of an input value in *radians* can be obtained in MATLAB with the **`sec()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

```
[ ]: sc = sec((j.^i + (e - c*e))*p)
```

If the expression given by $j.\hat{}i + (e - c*e)) * p$ will be evaluated, the result is $\frac{\pi}{4}$. Recall that angles with a sum equal to 90° are known as *complementary angles*. Thus, the complementary angle of $\frac{\pi}{4}$ radians or 45° is also $\frac{\pi}{4}$ radians.

```
[1]: cs = csc((j.^i + (e - c*e))*p);
     sc = sec((j.^i + (e - c*e))*p);

     cs = round(cs, 6);
     sc = round(sc, 6);

     sc == cs
```

```
ans =

  logical

   1
```

Notice that comparing the cosecant and secant of $j.\hat{}i + (e - c*e)) * p$ using the **`equal to (==)`** operator produced a logical 1 output value. The values of cosecant and secant are the reciprocal of the sine and cosine, respectively. Since the sine of an angle is equal to the cosine of its complementary angle, or simply complement, the sine and cosine of $\frac{\pi}{4}$ radians are equivalent. Likewise, the value that the **`csc()`** function will return as the cosecant of $\frac{\pi}{4}$ is approximately equal to the result of **`sec()`** with $\frac{\pi}{4}$ as the argument.

`cot()`

- The **cotangent** of an input value in *radians* can be obtained in MATLAB with the **`cot()`** function. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in a different unit of measure. This function is applied *element-wise* on matrices and arrays.

```
[ ]: co_t = cot((j.^i + (e - c*e))*p)
```

- ***Angle Expressed in Degrees***
  - If the angle supplied as input is in *degrees*, there is only minimal difference in how the previous six functions would be entered in the command window. The difference being the function name or identifier ends in **d**. In order to avoid the need for conversion and ensure the accuracy of the output, the data that will be passed as an argument should not be expressed in radians.

Demonstrated below are some examples of executing the MATLAB trigonometric functions which return output values with degree as the unit of measure.

```
[ ]: c_deg = cosd(90)
     t_deg = tand(210)
     cs_deg = cscd(300)
```

- ***Multiples of π***
  - For an input value that is equal to a number multiplied by π, the syntax **`sinpi(n)`** and **`cospi(n)`** can be used instead of **`sin(n*pi)`** and **`cos(n*pi)`** to find the sine or cosine more accurately because the floating-point value of the *pi* constant in MATLAB is an *approximation of π*.

Non-sinpi function syntax:

```
[ ]: sin(2*pi)
```

With the **`sinpi()`** function:

```
[ ]: sinpi(2)
```

- ***Inverse Trigonometric Functions***
  - In order to find the output value for the inverse of each of the trigonometric ratios, which can be referred to as *arcsin*, *arccosine*, *arctangent*, *arccosecant*, *arcsecant*, and *arccotangent*, the trigonometric functions should be entered with their function name or identifier starting in **a**. The output value is expressed in *radians* and the argument that must be passed to **`asin()`** and **`acos()`** must be between **-1** and **1**.

```
[ ]: arc_s = asin((j.^i + (e - c*e))*p)
     arc_c = acos((j.^i + (e - c*e))*p)
     arc_t = atan((j.^i + (e - c*e))*p)
```

```
[ ]: arc_cs = acsc((c*t)+cs)
     arc_sc = asec((c*t)+cs)
     arc_tc = acot((c*t)+cs)
```

- ***Hyperbolic Functions***

  - In order to find the output value for the trigonometric functions defined using a hyperbola, or the *hyperbolic functions*, the trigonometric functions should be entered with their function name or identifier ending in **h**.

```
[ ]: h_s = sinh((j.^i + (e - c*e))*p)
     h_c = cosh((j.^i + (e - c*e))*p)
     h_t = tanh((j.^i + (e - c*e))*p)
```

```
[ ]: h_cs = csch((j.^i + (e - c*e))*p)
     h_sc = sech((j.^i + (e - c*e))*p)
     h_tc = coth((j.^i + (e - c*e))*p)
```

- ***Degrees/Radians Conversion***

  - The built-in MATLAB functions also include commands that reduce the steps for converting between degrees and radians by making it unnecessary to enter the corresponding expression in order to execute the conversion. An angle in degrees can be converted to radians and the other way around with the **`deg2rad()`** and **`rad2deg()`** functions.

```
[ ]: z_deg = rad2deg((j.^i + (e - c*e))*p)

     z_rad = deg2rad(z_deg)
```

Data Analysis Functions

`sort()`

- When data comprising multiple elements, such as a matrix or array, is passed as the argument to **`sort()`**, the function returns an output containing elements that have been arranged or **sorted** in either ascending or descending order. By default, ascending order is applied.

```
[2]: M = [ 4 90 85 75
            2 55 65 75
            3 78 82 79
            1 84 92 93 ];

     M_ASC = sort(M)
```

```
M_ASC =

     1    55    65    75
     2    78    82    75
     3    84    85    79
     4    90    92    93
```

As shown in the example above, each element will undergo a change in position depending on whether it is greater than or less than the other values in the same column. The syntax for specifying *descending* as the desired arrangement of the elements is as follows:

```
[ ]: M_DESC = sort(M, 'descend')
```

`sortrows()`

- When a matrix consisting of multiple rows is passed as the argument to **`sortrows()`**, the ascending order of the elements in column 1 of the original matrix will determine the position of an entire row within the output. However, the initial order of the elements in each row will be maintained.

```
[3]: M_COL1 = sortrows(M)
```

```
M_COL1 =

     1    84    92    93
     2    55    65    75
     3    78    82    79
     4    90    85    75
```

You can also provide a different column number wherein the elements will be sorted. In the example below, the arrangement of the rows in the result will depend on the ascending order of the elements at column 3.

```
[ ]: M_COL3 = sortrows(M, 3)
```

`numel()`

- The **`numel()`** function can be used to obtain and store the number of elements in a matrix.

```
[4]: M_NUMEL = numel(M)
```

```
M_NUMEL =

    16
```

`std()`

- There is a built-in data analysis function in MATLAB that is useful in measuring the *spread* of the data within a matrix. In statistics, this value is referred to as the **standard deviation** and the computation for it can be executed with the **`std()`** function.

```
[5]: SD = std(M)
```

```
SD =

    1.2910   15.3052   11.4601    8.5440
```

`var()`

- There is also a data analysis function in MATLAB that can be used to find the *square of the standard deviation* or the average of the squared differences from the mean of the matrix elements. In statistics, this value is referred to as the **variance** and it can be calculated with the **`var()`** function.

```
[6]: SD_SQUARED = var(M)
```

```
SD_SQUARED =

    1.6667  234.2500  131.3333   73.0000
```

Notice that **`std()`** and **`var()`** returned the standard deviation and variance, respectively, of the elements in each column.

Complex Numbers Functions

Another category of predefined MATLAB functions that will be covered in this chapter is the collection of functions which provide convenience in calculations involving numbers with an *imaginary component*. The commands listed below are intended for **complex numbers**.

`angle()`

- This function returns the angle from the horizontal expressed in *radians* when the polar coordinate form of a complex number is passed as the input value.

```
[7]: el = 1;
     ex = 3;
     num = el + ex*j;

     num_angle = angle(num)
num_angle =

    1.2490
```

complex()

- The syntax for generating a complex number by executing the **complex** function is **`complex(a,b)`**, where a is a real number and b is the value of the imaginary component.

[8]:
```
num = complex(el,ex)
```

```
num =

   1.0000 + 3.0000i
```

real()

- In order to access the **real component** of a complex number, enter the **`real()`** function in the command window, with a complex number provided as the input value, either directly or stored in a variable.

[9]:
```
real(1+3j);

el = real(num);
```

imag()

- In order to access the **imaginary component** of a complex number, enter the **`imag()`** function in the command window, with a complex number provided as the input value, either directly or stored in a variable.

[10]:
```
imag(1+3j);

ex = imag(num);
```

`isreal()`

- The **`isreal()`** function verifies whether or not the value that was passed as an argument is a real number. The output is 1 (True) if this condition is satisfied. Otherwise, the result is 0 (False).

[11]:
```
imag_zero = el + ex*i*0;

ec = isreal(imag_zero)
```

```
ec =

  logical

   1
```

`conj()`

- The **`conj()`** function produces the **conjugate** of the input value, or a complex number consisting of the same real part and an imaginary component with a *sign which is the opposite* of that of its counterpart in the original complex number.

[12]:
```
ep = conj(num)
```

```
ep =

   1.0000 - 3.0000i
```

The following example shows a comparison of the output values produced from certain complex number functions and operations.

The **`abs()`** function is first used in calculating the absolute value of a complex number, which applies the Pythagorean theorem.

[13]:
```
L = 1 + i;

L_ABS = abs(L)
```

```
L_ABS =

    1.4142
```

The complex number L is then multiplied by its conjugate.

[14]:
```
M = conj(L);
N = L * M;
```

Executing **`sqrt()`** in order to obtain the squareroot of the product of L and its conjugate reveals that the result is equal to the value returned by the **`abs()`** function when L was passed as the argument.

[15]:
```
N_SQRT = sqrt(N)
```

```
N_SQRT =

    1.4142
```