MATLAB - ALGEBRA

So far, we have seen that all the examples work in MATLAB as well as its GNU, alternatively called Octave. But for solving basic algebraic equations, both MATLAB and Octave are little different, so we will try to cover MATLAB and Octave in separate sections.

We will also discuss factorizing and simplification of algebraic expressions.

Solving Basic Algebraic Equations in MATLAB

The **solve** function is used for solving algebraic equations. In its simplest form, the solve function takes the equation enclosed in quotes as an argument.

For example, let us solve for x in the equation x-5 = 0

```
solve('x-5=0')
```

MATLAB will execute the above statement and return the following result −

```
ans =
 5
```

You can also call the solve function as −

```
y = solve('x-5 = 0')
```

MATLAB will execute the above statement and return the following result −

```
y =
 5
```

You may even not include the right hand side of the equation −

```
solve('x-5')
```

MATLAB will execute the above statement and return the following result −

```
ans =
 5
```

If the equation involves multiple symbols, then MATLAB by default assumes that you are solving for x, however, the solve function has another form −

```
solve(equation, variable)
```

where, you can also mention the variable.

For example, let us solve the equation v – u – $3t^2$ = 0, for v. In this case, we should write −

```
solve('v-u-3*t^2=0', 'v')
```

MATLAB will execute the above statement and return the following result −

```
ans =
 3*t^2 + u
```

Solving Basic Algebraic Equations in Octave

The **roots** function is used for solving algebraic equations in Octave and you can write above examples as follows:

For example, let us solve for x in the equation x-5 = 0

```
roots([1, -5])
```

Octave will execute the above statement and return the following result −

```
ans = 5
```

You can also call the solve function as −

```
y = roots([1, -5])
```

Octave will execute the above statement and return the following result −

```
y = 5
```

Solving Quadratic Equations in MATLAB

The **solve** function can also solve higher order equations. It is often used to solve quadratic equations. The function returns the roots of the equation in an array.

The following example solves the quadratic equation x^2 -7x +12 = 0. Create a script file and type the following code −

```
eq = 'x^2 -7*x + 12 = 0';
s = solve(eq);
disp('The first root is: '), disp(s(1));
disp('The second root is: '), disp(s(2));
```

When you run the file, it displays the following result −

```
The first root is:
   3
The second root is:
   4
```

Solving Quadratic Equations in Octave

The following example solves the quadratic equation x^2 -7x +12 = 0 in Octave. Create a script file and type the following code −

```
s = roots([1, -7, 12]);

disp('The first root is: '), disp(s(1));
disp('The second root is: '), disp(s(2));
```

When you run the file, it displays the following result −

```
The first root is:
 4
The second root is:
 3
```

Solving Higher Order Equations in MATLAB

The **solve** function can also solve higher order equations. For example, let us solve a cubic equation as $x-3^2x-7=0$

```
solve('(x-3)^2*(x-7)=0')
```

MATLAB will execute the above statement and return the following result −

```
ans =
  3
  3
  7
```

In case of higher order equations, roots are long containing many terms. You can get the numerical value of such roots by converting them to double. The following example solves the fourth order equation $x^4 - 7x^3 + 3x^2 - 5x + 9 = 0$.

Create a script file and type the following code −

```
eq = 'x^4 - 7*x^3 + 3*x^2 - 5*x + 9 = 0';
s = solve(eq);
disp('The first root is: '), disp(s(1));
disp('The second root is: '), disp(s(2));
disp('The third root is: '), disp(s(3));
disp('The fourth root is: '), disp(s(4));
% converting the roots to double type
disp('Numeric value of first root'), disp(double(s(1)));
disp('Numeric value of second root'), disp(double(s(2)));
disp('Numeric value of third root'), disp(double(s(3)));
disp('Numeric value of fourth root'), disp(double(s(4)));
```

When you run the file, it returns the following result −

```
The first root is:
6.630396332390718431485053218985
 The second root is:
1.059780463302589629168277249885
 The third root is:
- 0.345088397846654030326665234486 75 - 1.077836295463017659683110926979 3*i
 The fourth root is:
- 0.345088397846654030326665234486 75 + 1.077836295463017659683110926979 3*i
Numeric value of first root
    6.6304
Numeric value of second root
    1.0598
Numeric value of third root
  -0.3451 - 1.0778i
Numeric value of fourth root
  -0.3451 + 1.0778i
```

Please note that the last two roots are complex numbers.

Solving Higher Order Equations in Octave

The following example solves the fourth order equation $x^4 - 7x^3 + 3x^2 - 5x + 9 = 0$.

Create a script file and type the following code −

```
v = [1, -7,  3, -5, 9];

s = roots(v);
% converting the roots to double type
disp('Numeric value of first root'), disp(double(s(1)));
disp('Numeric value of second root'), disp(double(s(2)));
disp('Numeric value of third root'), disp(double(s(3)));
```

```
disp('Numeric value of fourth root'), disp(double(s(4)));
```

When you run the file, it returns the following result −

```
Numeric value of first root
 6.6304
Numeric value of second root
-0.34509 + 1.07784i
Numeric value of third root
-0.34509 - 1.07784i
Numeric value of fourth root
 1.0598
```

Solving System of Equations in MATLAB

The **solve** function can also be used to generate solutions of systems of equations involving more than one variables. Let us take up a simple example to demonstrate this use.

Let us solve the equations −

5x + 9y = 5

3x – 6y = 4

Create a script file and type the following code −

```
s = solve('5*x + 9*y = 5','3*x - 6*y = 4');
s.x
s.y
```

When you run the file, it displays the following result −

```
ans =
 22/19
ans =
-5/57
```

In same way, you can solve larger linear systems. Consider the following set of equations −

x + 3y -2z = 5

3x + 5y + 6z = 7

2x + 4y + 3z = 8

Solving System of Equations in Octave

We have a little different approach to solve a system of 'n' linear equations in 'n' unknowns. Let us take up a simple example to demonstrate this use.

Let us solve the equations −

5x + 9y = 5

3x – 6y = 4

Such a system of linear equations can be written as the single matrix equation Ax = b, where A is the coefficient matrix, b is the column vector containing the right-hand side of the linear equations and x is the column vector representing the solution as shown in the below program −

Create a script file and type the following code −

```
A = [5, 9; 3, -6];
b = [5;4];
A \ b
```

When you run the file, it displays the following result −

```
ans =

   1.157895
  -0.087719
```

In same way, you can solve larger linear systems as given below −

x + 3y -2z = 5

3x + 5y + 6z = 7

2x + 4y + 3z = 8

Expanding and Collecting Equations in MATLAB

The **expand** and the **collect** function expands and collects an equation respectively. The following example demonstrates the concepts −

When you work with many symbolic functions, you should declare that your variables are symbolic.

Create a script file and type the following code −

```
syms x %symbolic variable x
syms y %symbolic variable x
% expanding equations
expand((x-5)*(x+9))
expand((x+2)*(x-3)*(x-5)*(x+7))
expand(sin(2*x))
expand(cos(x+y))

% collecting equations
collect(x^3 *(x-7))
collect(x^4*(x-3)*(x-5))
```

When you run the file, it displays the following result −

```
ans =
x^2 + 4*x - 45
ans =
x^4 + x^3 - 43*x^2 + 23*x + 210
ans =
2*cos(x)*sin(x)
ans =
cos(x)*cos(y) - sin(x)*sin(y)
ans =
x^4 - 7*x^3
ans =
x^6 - 8*x^5 + 15*x^4
```

Expanding and Collecting Equations in Octave

You need to have **symbolic** package, which provides **expand** and the **collect** function to expand and collect an equation, respectively. The following example demonstrates the concepts −

When you work with many symbolic functions, you should declare that your variables are symbolic but Octave has different approach to define symbolic variables. Notice the use of **Sin** and **Cos**, which are also defined in symbolic package.

Create a script file and type the following code −

```
% first of all load the package, make sure its installed.
pkg load symbolic

% make symbols module available
```

```
symbols

% define symbolic variables
x = sym ('x');
y = sym ('y');
z = sym ('z');

% expanding equations
expand((x-5)*(x+9))
expand((x+2)*(x-3)*(x-5)*(x+7))
expand(Sin(2*x))
expand(Cos(x+y))

% collecting equations
collect(x^3 *(x-7), z)
collect(x^4*(x-3)*(x-5), z)
```

When you run the file, it displays the following result −

```
ans =

-45.0+x^2+(4.0)*x
ans =

210.0+x^4-(43.0)*x^2+x^3+(23.0)*x
ans =

sin((2.0)*x)
ans =

cos(y+x)
ans =

x^(3.0)*(-7.0+x)
ans =

(-3.0+x)*x^(4.0)*(-5.0+x)
```

Factorization and Simplification of Algebraic Expressions

The **factor** function factorizes an expression and the **simplify** function simplifies an expression. The following example demonstrates the concept −

Example

Create a script file and type the following code −

```
syms x
syms y
factor(x^3 - y^3)
factor([x^2-y^2,x^3+y^3])
simplify((x^4-16)/(x^2-4))
```

When you run the file, it displays the following result −

```
 ans =
 (x - y)*(x^2 + x*y + y^2)
 ans =
 [ (x - y)*(x + y), (x + y)*(x^2 - x*y + y^2)]
 ans =
 x^2 + 4
```