Linear Algebra

Solving a Linear System

A linear algebraic equation is an equation of the system

$a_1 x_1+a_2 x_2+a_3 x_3+\cdots +a_n x_n=b$

where a's are **constant coefficients**, the x's are the **unknowns,** and b is a **constant.** A solution is a sequence of numbers s_1,s_2, and $s_3$that satisfies the equation.

Example

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 – 3t2 = 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

Live Demo

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

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

```
ans = 5
```

You can also call the solve function as –

Live Demo

```
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 x2 -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 x2 -7x +12 = 0 in Octave. Create a script file and type the following code –

Live Demo

```
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)2(x-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 x4 - 7x3 + 3x2 - 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:
1.059780463302589629168277249988 5
 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 x4 – 7x3 + 3x2 – 5x + 9 = 0.

Create a script file and type the following code –

Live Demo

```
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 –

Live Demo

```
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
```

Eigenvalues and Eigenvectors

An eigenvalues and eigenvectors of the square matrix A are a scalar λ and a nonzero vector v that satisfy

Av = λv

In this equation, **A** is a n-by-n matrix, **v** is non-zero n-by-1 vector, and λ is the scalar (which might be either real or complex). Any value of the λ for which this equation has a solution known as eigenvalues of the matrix **A**. It is also called the **characteristic value.** The vector, **v**, which corresponds to this equation, is called eigenvectors**.**

```
>> A=[0 1;-2 -3]
A =
     0   1
    -2  -3
>> [v,d]=eig(A)
v =
    0.7071  -0.4472
   -0.7071   0.8944
d =
    -1   0
     0  -2
```

MATLAB Polynomial

MATLAB performs, polynomials as row vectors, including coefficients ordered by descending powers. For example, the equation $P(x) = x^4 + 7x^3 - 5x + 9$ can be represented as -

p = [1 7 0 -5 9];

Polynomial Functions

polyfit

Given two vector x and y, the command a=polyfit (x, y, n) fits a polynomial of order n through the data points (x_i,y_i) and returns (n+1) coefficients of the power of x in the row vector a. The coefficients are arranged in the decreasing order of the power of x, i.e.,$a=[a_n\ a_{n-1}...a_1\ a_0]$.

polyval

Given a data vector x and the coefficients of a polynomial in a row vector a the command y=polyval(a, x) evaluates the polynomials at the data points x_i and generates the values y_i such that

```
m= [5 10 20 50 100];              % mass data (g)
d= [15 .5 33 .07 53 .39 140 .24 301 .03];      % displacement data (mm)
g=9.81;                  % g= 9.81 m/s^2
F= m/1000*g;                % compute spring force (N)
a= polyfit (d, F, 1);              % fit a line (1st order polynomial)
d_fit=0:10:300;              % make a finer grid on d
F_fit=polyval (a, d_fit);           % calculate the polynomial at new points
plot (d, F,'o',d_fit,F_fit)          % plot data and the fitted curve
xlabel ('Displacement \delta (mm)'), ylabel ('Force (N)')
k=a(1);                % Find the spring constant
text (100,.32, ['\leftarrow Spring Constant K=', num2str(k),'N/mm']);
```

Least squares curve fitting

- The procedure of least square curve fit can simply be implemented in MATLAB, because the technique results in a set of linear equations that need to be solved. Most of the curve fits are polynomial curve fits or exponential curve fits (including power laws, e.g., $y=ax^b$). Two most commonly used functions are: $y=ae^{bx}$
- $y=cx^d$

Example: Create a script file

```
% EXPFIT: Exponential curve fit example
% For the following input for (t, p) fit an exponential curve
%   p=p0 * exp(-t/tau).
% The problem is resolved by taking a log and then using the linear.
% fit (1st order polynomial)

% original input
t=[0  0.5  1  5  10  20];
p=[760  625  528  85  14  0  .16];

```

```
% Prepare new input for linear fit
tbar= t;          % no change in t is needed
pbar=log (p);          % Fit a 1st order polynomial through (tbar, pbar)
a=polyfit (tbar, pbar, 1);  % output is a=a1 a0]

% Evaluate constants p0 and tau
p0 = exp (a(2));       % a(2)=a0=log (p0)
tau=-1/a(1);          % a1=-1/tau

% (a) Plot the new curve and the input on linear scale
tnew= linspace (0, 20, 20); % generate more refined t
pnew=p0*exp(-tnew/tau);  % calculate p at new t
plot (t, p, '0', tnew, pnew),  grid
xlabel ('Time' (sec)'), ylabel ('Pressure ((torr)')

% (b) Plot the new curve and the input on semilog scale
lpnew = exp(polyval (a, tnew));
semilogy (t, p, 'o', tnew, lpnew), grid
xlabel ('Time (sec)'), ylabel ('Pressure (torr)')

% Note: We only need one plot; we can select (a) or (b).
```

Differential

MATLAB provides the **diff** command for computing symbolic derivatives. In its simplest form, you pass the function you want to differentiate to diff command as an argument.

For example, let us compute the derivative of the function f(t) = 3t2 + 2t-2

Example

Create a script file and type the following code into it –

```
syms t
f = 3*t^2 + 2*t^(-2);
diff(f)
```

When the above code is compiled and executed, it produces the following result –

```
ans =
6*t - 4/t^3
```

Integration

Integration deals with two essentially different types of problems.

- In the first type, derivative of a function is given and we want to find the function. Therefore, we basically reverse the process of differentiation. This reverse process is known as anti-differentiation, or finding the primitive function, or finding an **indefinite integral**.
- The second type of problems involve adding up a very large number of very small quantities and then taking a limit as the size of the quantities approaches zero, while the number of terms tend to infinity. This process leads to the definition of the **definite integral**.