

# MATLAB Description

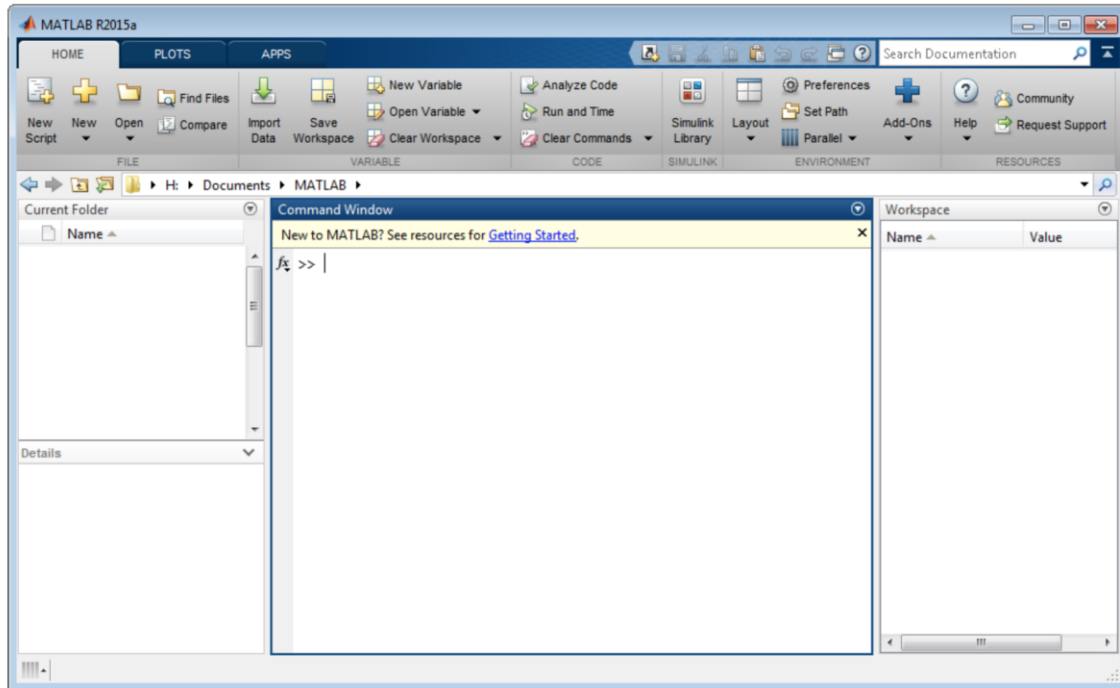
MATLAB® is a high-level language and interactive environment for numerical computation, visualization, and programming. Using MATLAB, you can analyze data, develop algorithms, and create models and applications. The language, tools, and builtin math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java® . You can use MATLAB for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology. More than a million engineers and scientists in industry and academia use MATLAB, the language of technical computing.

## Key Features

- High-level language for numerical computation, visualization, and application development
- Interactive environment for iterative exploration, design, and problem solving
- Mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving ordinary differential equations
- Built-in graphics for visualizing data and tools for creating custom plots
- Development tools for improving code quality and maintainability and maximizing performance
- Tools for building applications with custom graphical interfaces
- Functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET, and Microsoft® Excel®

# Desktop Basics

When you start MATLAB, the desktop appears in its default layout. The desktop includes these panels:



- Current Folder — Access your files.
- Command Window — Enter commands at the command line, indicated by the prompt (`>>`).
- Workspace — Explore data that you create or import from files.

## Commands Tutorial

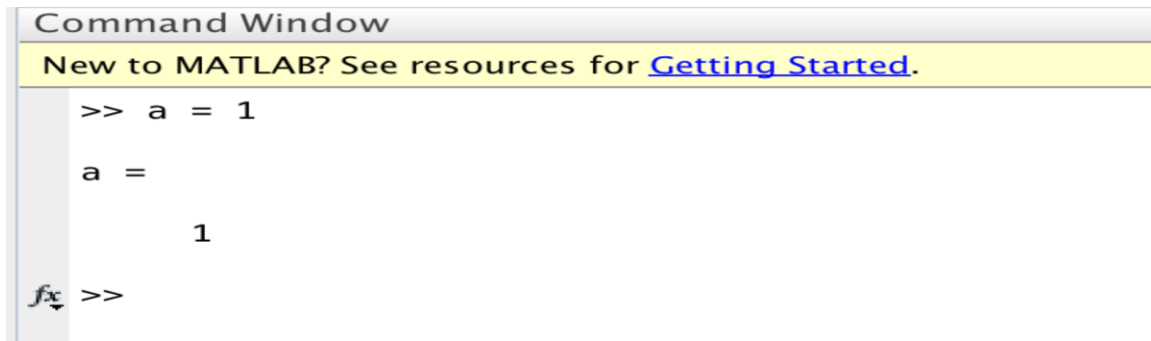
You need a small number of basic commands to start using MATLAB. This short tutorial describes those fundamental commands. You need to create vectors and matrices to change them and to operate with them. Those are all short high level commands because MATLAB constantly works with matrices. I believe that you will like the power that this software gives to do linear algebra by a series of short instructions.

## 1. Basic Variables:

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named **a** by typing this statement at the command line:

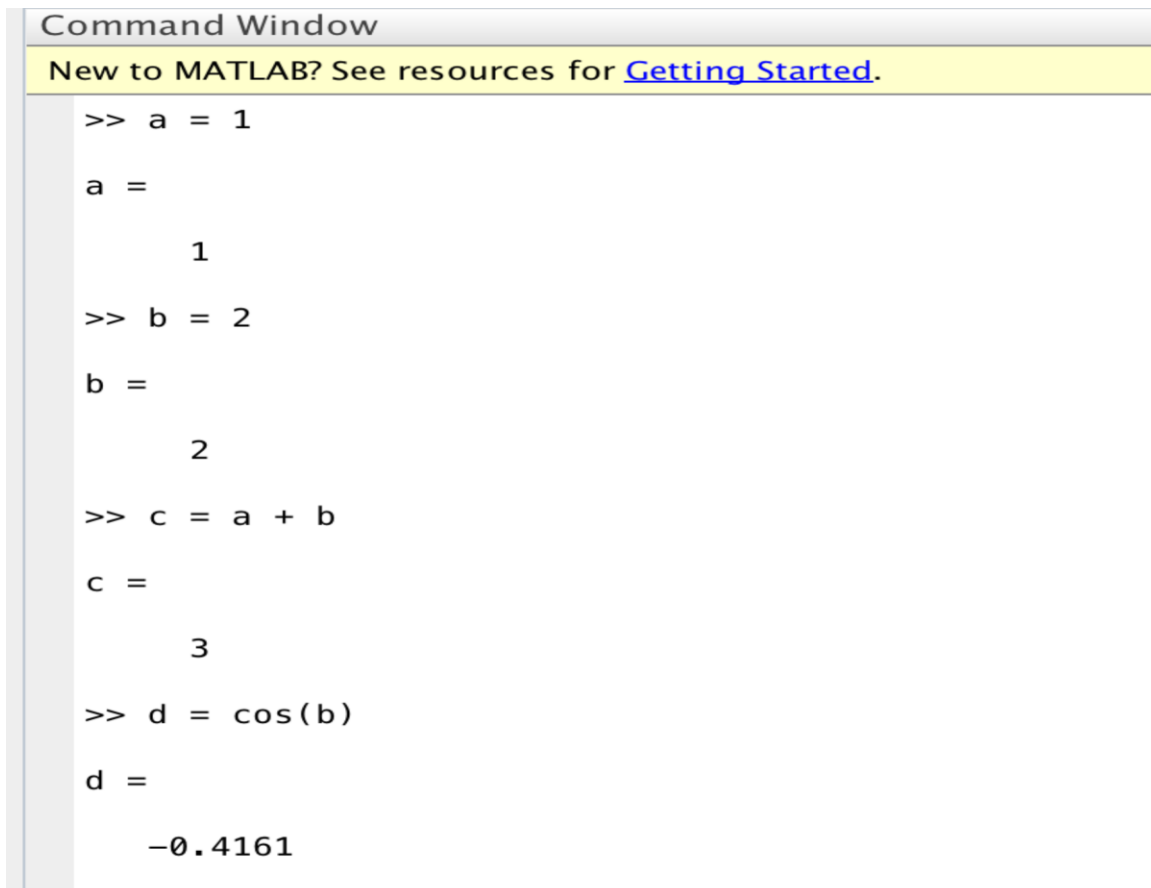
```
a=1
```

MATLAB adds variable **a** to the workspace and displays the result in the Command Window.



```
Command Window
New to MATLAB? See resources for Getting Started.
>> a = 1
a =
    1
fx >>
```

Create a few more variables.



```
Command Window
New to MATLAB? See resources for Getting Started.
>> a = 1
a =
    1
>> b = 2
b =
    2
>> c = a + b
c =
    3
>> d = cos(b)
d =
 -0.4161
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

```
>> e = a * b;  
fx >>
```

By the way, you can recall previous commands by pressing the up- and down-arrow keys,  $\uparrow$  and  $\downarrow$ . Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command `b = 2`, type `b`, and then press the up-arrow key.

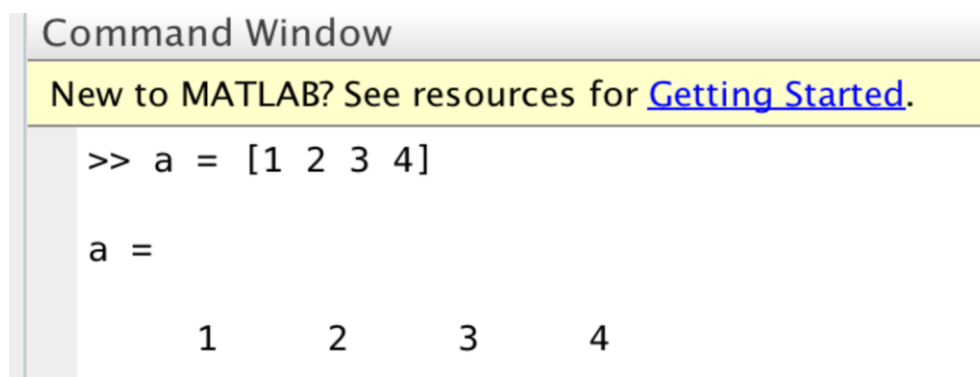
## 2. Matrices and Arrays

MATLAB is an abbreviation for "matrix laboratory." While other programming languages mostly work with numbers one at a time, MATLAB® is designed to operate primarily on whole matrices and arrays.

All MATLAB variables are multidimensional arrays, no matter what type of data. A matrix is a two-dimensional array often used for linear algebra.

### Array Creation

To create an array with four elements in a single row, separate the elements with either a comma (,) or a space.

A screenshot of the MATLAB Command Window. At the top, there is a title bar that says "Command Window". Below it is a yellow banner with the text "New to MATLAB? See resources for [Getting Started](#)." The command prompt shows the user entering the command `>> a = [1 2 3 4]`. Below the command, the output shows `a =` followed by the numbers 1, 2, 3, and 4, each centered under its respective column.

```
Command Window  
New to MATLAB? See resources for Getting Started.  
>> a = [1 2 3 4]  
a =  
1      2      3      4
```

This type of array is a *row vector*.

To create a matrix that has multiple rows, separate the rows with semicolons.

```
>> a = [1,2,3;4,5,6;7,8,10]
```

```
a =
```

```
     1     2     3
     4     5     6
     7     8    10
```

```
fx >>
```

Another way to create a matrix is to use a function, such as ones, zeros, or rand. For example, create a 5-by-1 column vector of zeros.

```
>> zeros(5,1)
```

```
ans =
```

```
0
0
0
0
0
```

### Matrix and Array Operations

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

```
>> a + 10

ans =

    11    12    13
    14    15    16
    17    18    20
```

*fx* >>

To transpose a matrix, use a single quote ('):

```
>> a'

ans =

     1     4     7
     2     5     8
     3     6    10
```

*fx* >>

You can perform standard matrix multiplication, which computes the inner products between rows and columns, using the \* operator. For example, confirm that a matrix times its inverse returns the identity matrix:

```
>> P = a * inv(a)

P =

    1.0000   -0.0000   -0.0000
    0.0000    1.0000   -0.0000
    0.0000   -0.0000    1.0000
```

*fx* >>

Notice that p is not a matrix of integer values. MATLAB stores numbers as floating-point values, and arithmetic operations are sensitive to small differences between the actual value and its floating-point representation.

To perform element-wise multiplication rather than matrix multiplication,

use the .\* operator:

```
>> P = a .* a  
  
P =  
  
     1     4     9  
    16    25    36  
    49    64   100
```

### Concatenation

Concatenation is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets [] is the concatenation operator.

```
>> A = [a,a]  
  
A =  
  
     1     2     3     1     2     3  
     4     5     6     4     5     6  
     7     8    10     7     8    10  
  
fx >>
```

Concatenating arrays next to one another using commas is called horizontal concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate vertically using semicolons.

```
>> A = [a;a]  
  
A =  
  
     1     2     3  
     4     5     6  
     7     8    10  
     1     2     3  
     4     5     6  
     7     8    10  
  
fx >>
```

### 3. Array Indexing


Every variable in MATLAB® is an array that can hold many numbers. When you want to access selected elements of an array, use indexing.

For example, consider the 4-by-4 magic square A:

```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```



```
>>
```

There are two ways to refer to a particular element in an array. The most common way is to specify row and column subscripts, such as

```
>> A(4,2)

ans =

    14
```

Less common, but sometimes useful, is to use a single subscript that traverses down each column in order:

```
>> A(8)

ans =

    14
```

Using a single subscript to refer to a particular element in an array is called linear indexing.



To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form **start:end**. For example, list the elements in the first three rows and the second column of A:

```
>> A(1:3,2)
```

```
ans =
```

```
    2
   11
    7
```

The colon alone, without start or end values, specifies all of the elements in that dimension. For example, select all the columns in the third row of A:

```
>> A(3,:) 
```

```
ans =
```

```
    9    7    6   12
```

```
fx >>
```

The colon operator also allows you to create an equally spaced vector of values using the more general form **start:step:end**.

```
>> B = 0 : 10 : 100
```

```
B =
```

```
    0    10    20    30    40    50    60    70    80    90   100
```

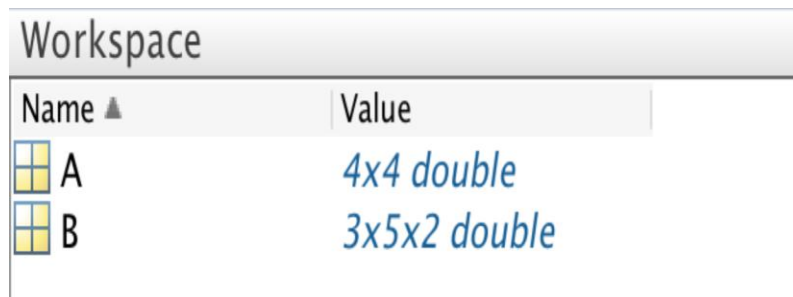
```
fx >>
```

## 4. Workspace Variables

The workspace contains variables that you create within or import into MATLAB from data files or other programs. For example, these statements create variables A and B in the workspace.

```
A = magic(4);  
B = rand(3,5,2);
```

The variables also appear in the Workspace pane on the desktop.



The screenshot shows the MATLAB Workspace pane. It has a title bar labeled "Workspace". Below the title bar is a table with two columns: "Name" and "Value". There are two rows of variables. The first row is for variable "A", which is a 4x4 double matrix, represented by a 4x4 grid icon. The second row is for variable "B", which is a 3x5x2 double array, represented by a 3x5x2 grid icon.

Name ▲	Value
A	4x4 double
B	3x5x2 double

## 5. Calling Functions

MATLAB® provides a large number of functions that perform computational tasks. Functions are equivalent to subroutines or methods in other programming languages.

To call a function, such as max, enclose its input arguments in parentheses:

```
>> A = [1 3 5];  
>> max(A)  
  
ans =  
  
5
```

When there are multiple output arguments, enclose them in square brackets:

```
>> [maxA,location]=max(A)

maxA =

     5

location =

     3
```

## 6. 2-D and 3-D Plots

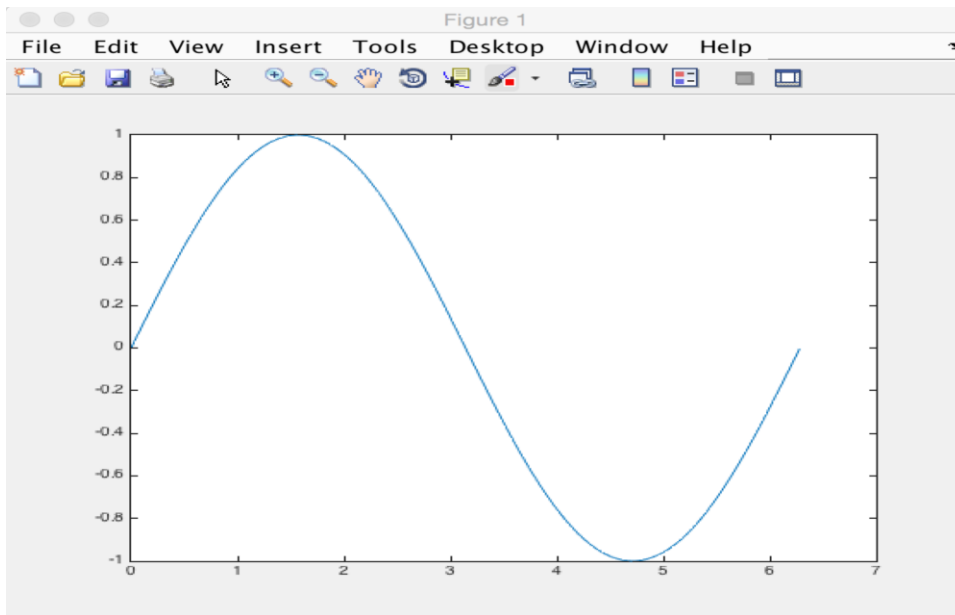
### Line Plots

To create two-dimensional line plots, use the plot function. For example, plot the value of the sine function from 0 to  $2\pi$  :

Command Window

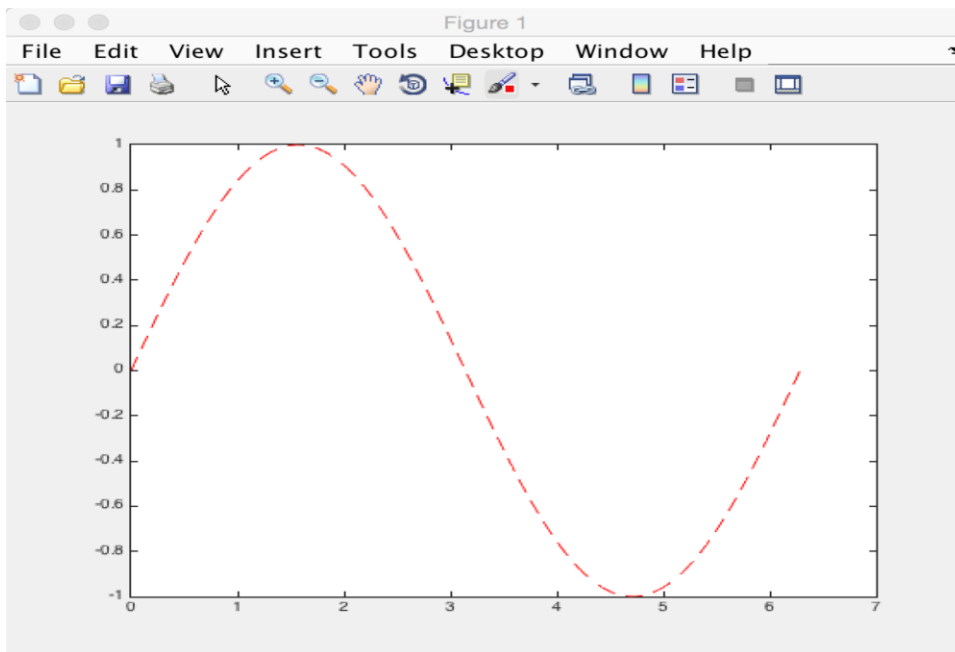
New to MATLAB? See resources for [Getting Started](#).

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
fx >>
```



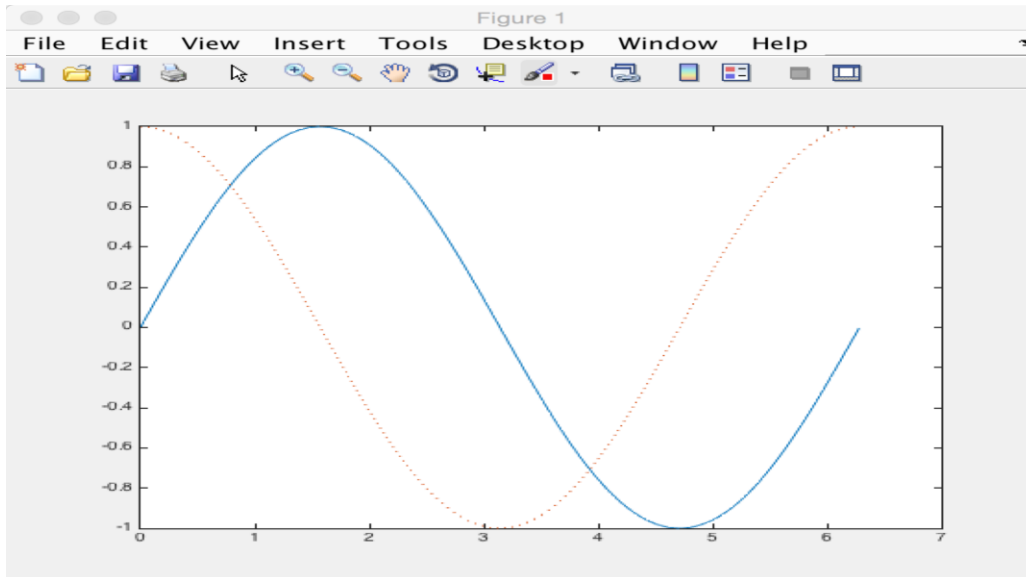
By adding a third input argument to the plot function, you can plot the same variables using a red dashed line.

**plot(x,y,'r--')**



By default, MATLAB® clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot. To add plots to an existing figure, use hold.

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
>> hold on
>> y2 = cos(x);
>> plot(x,y2, ':')
fx >>
```



Until you use hold off or close the window, all plots appear in the current figure window.

### 3-D Plots

For convenience, we first introduce a function **meshgrid**:

**[X,Y] = meshgrid(xgv,ygv)** replicates the grid vectors **xgv** and **ygv** to produce a full grid. This grid is represented by the output coordinate arrays X and Y. The output coordinate arrays X and Y contain copies of the grid vectors **xgv** and **ygv** respectively. The size of the output arrays are determined by the length of the grid vectors. For grid vectors **xgv** and **ygv** of length M and N respectively, X and Y will have N rows and M columns. Create a full grid from two monotonically increasing grid vectors:

```
>> [X,Y] = meshgrid(1:3,10:14)
```

```
X =
```

```
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
```

```
Y =
```

```
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

$[X,Y] = \text{meshgrid}(gv)$  is the same as  $[X,Y] = \text{meshgrid}(gv,gv)$ . In other words, you can reuse the same grid vector in each respective dimension. The dimensionality of the output arrays is determined by the number of output arguments.

Now let's back to 3-D plot.

Three-dimensional plots typically display a surface defined by a function in two variables,  $z = f(x,y)$ .

To evaluate  $z$ , first create a set of  $(x,y)$  points over the domain of the function using `meshgrid`.

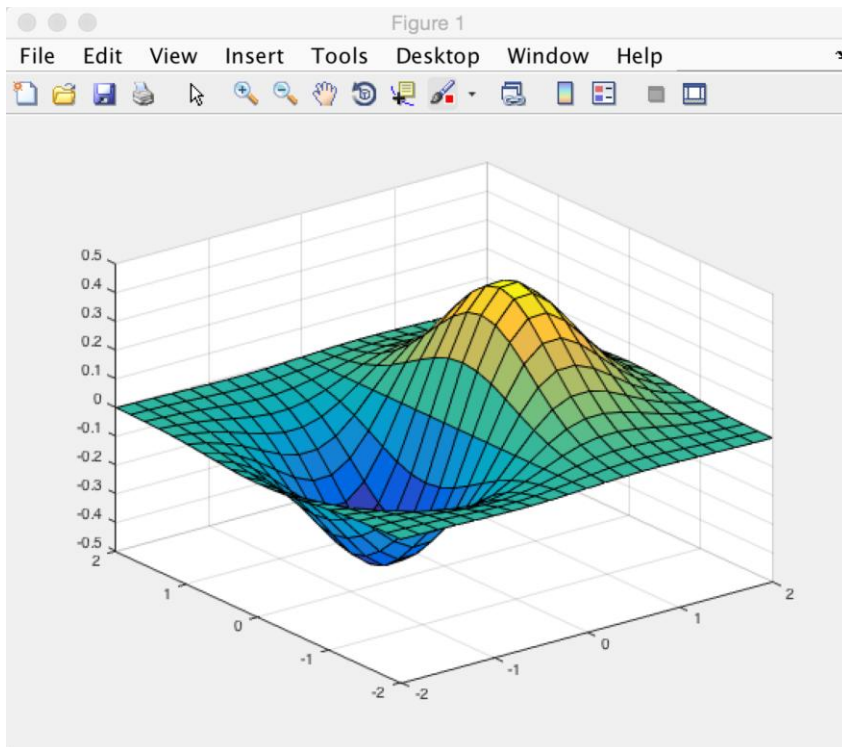
```
[X,Y] = meshgrid(-2:.2:2);  
Z = X .* exp(-X.^2 - Y.^2);
```

Then, create a surface plot.

```
surf(X,Y,Z)
```

```
>> [X,Y] = meshgrid(-2:.2:2);  
>> Z = X .* exp(-X.^2 - Y.^2);  
>> surf(X,Y,Z)
```

```
fx >>
```



Both the `surf` function and its companion `mesh` display surfaces in three dimensions. `surf` displays both the connecting lines and the faces of the surface in color. `mesh` produces wireframe surfaces that color only the lines connecting the defining points.

## 7. Programming and Scripts

The simplest type of MATLAB program is called a script. A script is a file with a `.m` extension that contains multiple sequential lines of MATLAB commands and function calls. You can run a script by typing its name at the command line.

### Sample Script

To create a script, use the **`edit`** command,

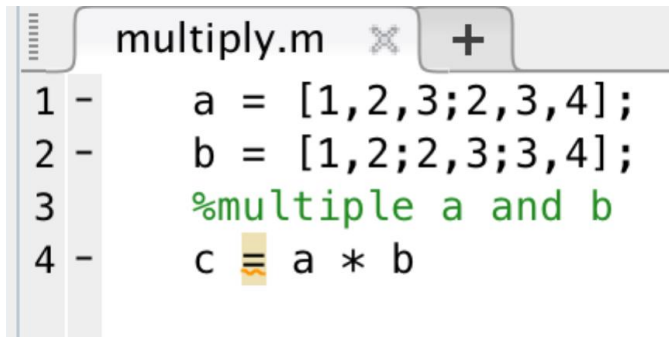
**`edit multiply`**

This opens a blank file named **`multiply.m`**. Enter some code:

```
a = [1,2,3;2,3,4];  
b = [1,2;2,3;3,4];
```

**c = a \* b**

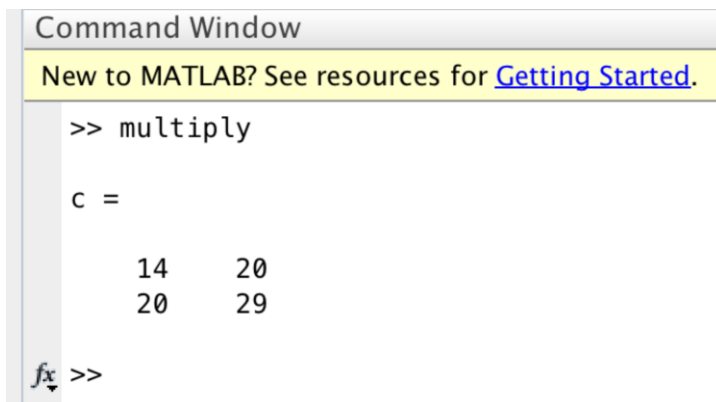
Whenever you write code, it is a good practice to add comments that describe the code. Comments allow others to understand your code, and can refresh your memory when you return to it later. Add comments using the percent (%) symbol.



```
1 - a = [1,2,3;2,3,4];
2 - b = [1,2;2,3;3,4];
3   %multiple a and b
4 - c = a * b
```

Save the file in the current folder. To run the script, type its name at the command line:

**multiply**



```
Command Window
New to MATLAB? See resources for Getting Started.

>> multiply

c =

    14    20
    20    29

fx >>
```

## Loops and Conditional Statements

Within a script, you can loop over sections of code and conditionally execute sections using the keywords **for**, **while**, **if**, and **switch**.

For example, create a script named **calcmean.m** that uses a **for** loop to calculate the mean of five random samples and the overall mean.



```
calcmean.m  x  +
1 - nsamples = 5;
2 - npoints = 50;
3 - for k = 1:nsamples
4 -     currentData = rand(npoints,1);
5 -     sampleMean(k) = mean(currentData);
6 - end
7 - overallMean = mean(sampleMean)
8
```

Now, modify the **for** loop so that you can view the results at each iteration. Display text in the Command Window that includes the current iteration number, and remove the semicolon from the assignment to sampleMean.

```
calcmean.m  x  +
1 - nsamples = 5;
2 - npoints = 50;
3 - for k = 1:nsamples
4 -     iterationString = ['Iteration #',int2str(k)];
5 -     disp(iterationString)
6 -     currentData = rand(npoints,1);
7 -     sampleMean(k) = mean(currentData)
8 - end
9 - overallMean = mean(sampleMean)
```

When you run the script, it displays the intermediate results, and then calculates the overall mean.

```
Command Window
New to MATLAB? See resources for Getting Started.

>> calcmean
Iteration #1

sampleMean =

    0.4843

Iteration #2

sampleMean =

    0.4843    0.4915

Iteration #3

sampleMean =

    0.4843    0.4915    0.4305

Iteration #4

sampleMean =

    0.4843    0.4915    0.4305    0.5014

Iteration #5

sampleMean =

    0.4843    0.4915    0.4305    0.5014    0.5021

overallMean =

    0.4820
```

In the Editor, add conditional statements to the end of **calcmean.m** that display a different message depending on the value of overallMean.

```
calcmean.m x +
1 - nsamples = 5;
2 - npoints = 50;
3 - for k = 1:nsamples
4 -     iterationString = ['Iteration #',int2str(k)];
5 -     disp(iterationString)
6 -     currentData = rand(npoints,1);
7 -     sampleMean(k) = mean(currentData)
8 - end
9 - overallMean = mean(sampleMean)
10
11 - if overallMean < .49
12 -     disp('Mean is less than expected')
13 - elseif overallMean > .51
14 -     disp('Mean is greater than expected')
15 - else
16 -     disp('Mean is within the expected range')
17 - end
```

Run calcmean and verify that the correct message displays for the calculated overallMean. For example:

```
>> calcmean
Iteration #1
sampleMean =
    0.5403    0.4174    0.4447    0.4977    0.5112
Iteration #2
sampleMean =
    0.5403    0.5026    0.4447    0.4977    0.5112
Iteration #3
sampleMean =
    0.5403    0.5026    0.5602    0.4977    0.5112
Iteration #4
sampleMean =
    0.5403    0.5026    0.5602    0.4630    0.5112
Iteration #5
sampleMean =
    0.5403    0.5026    0.5602    0.4630    0.5519
overallMean =
    0.5236
Mean is greater than expected
```

## Script Locations

MATLAB looks for scripts and other files in certain places. To run a script, the file must be in the current folder or in a folder on the search path.

By default, the MATLAB folder that the MATLAB Installer creates is on the search path. If you want to store and run programs in another folder, add it to the search path. Select the folder in the Current Folder browser, right-click, and then select **Add to Path**.

## 8. Some useful functions

### How to create special Matrices

**diag(v)** produces the diagonal matrix with vector v on its diagonal;

**ones(n)** gives an n by n matrix of ones;

**zeros(n)** gives an n by n matrix of zeros;

**rand(n)** gives an n by n matrix with random entries between 0 and 1 (uniform distribution);

**randn(n)** gives an n by n matrix with normally distributed entries (mean 0 and variance 1);

**ones(m,n)** **zeros(m,n)** **rand(m,n)** give m by n matrices.

### How to change entries in a given matrix A

$A(3,2) = 7$  resets the (3,2) entry to equal 7;

$A(3,:) = v$  resets the third row to equal v;

$A(:,2) = w$  resets the second column to equal w;

$A([2\ 3],:) = A([3\ 2],:)$  exchanges rows 2 and 3 of A.

### How to create sub-matrices of an m by n matrix A

$A(i,j)$  returns the (i,j) entry of the matrix A (scalar = 1 by 1 matrix);

$A(i,:)$  returns the i-th row of A (as row vector);

$A(:,j)$  returns the j-th column of A (as column vector);

$A(2:4,3:7)$  returns rows from 2 to 4 and columns from 3 to 7;

$A([2\ 4],:)$  returns rows 2 and 4 and all columns;

$A(:)$  returns one long column formed from the columns of A;

$\text{triu}(A)$  sets all entries below the main diagonal to zero;

$\text{tril}(A)$  sets all entries above the main diagonal to zero.

### Numbers and matrices associated with A

**det(A)** is the determinant (if A is a square matrix);

**rank(A)** is the rank;

**size(A)** is the pair of numbers [m,n]

**trace(A)** is the trace = sum of diagonal entries = sum of eigenvalues