



MATLAB Guide

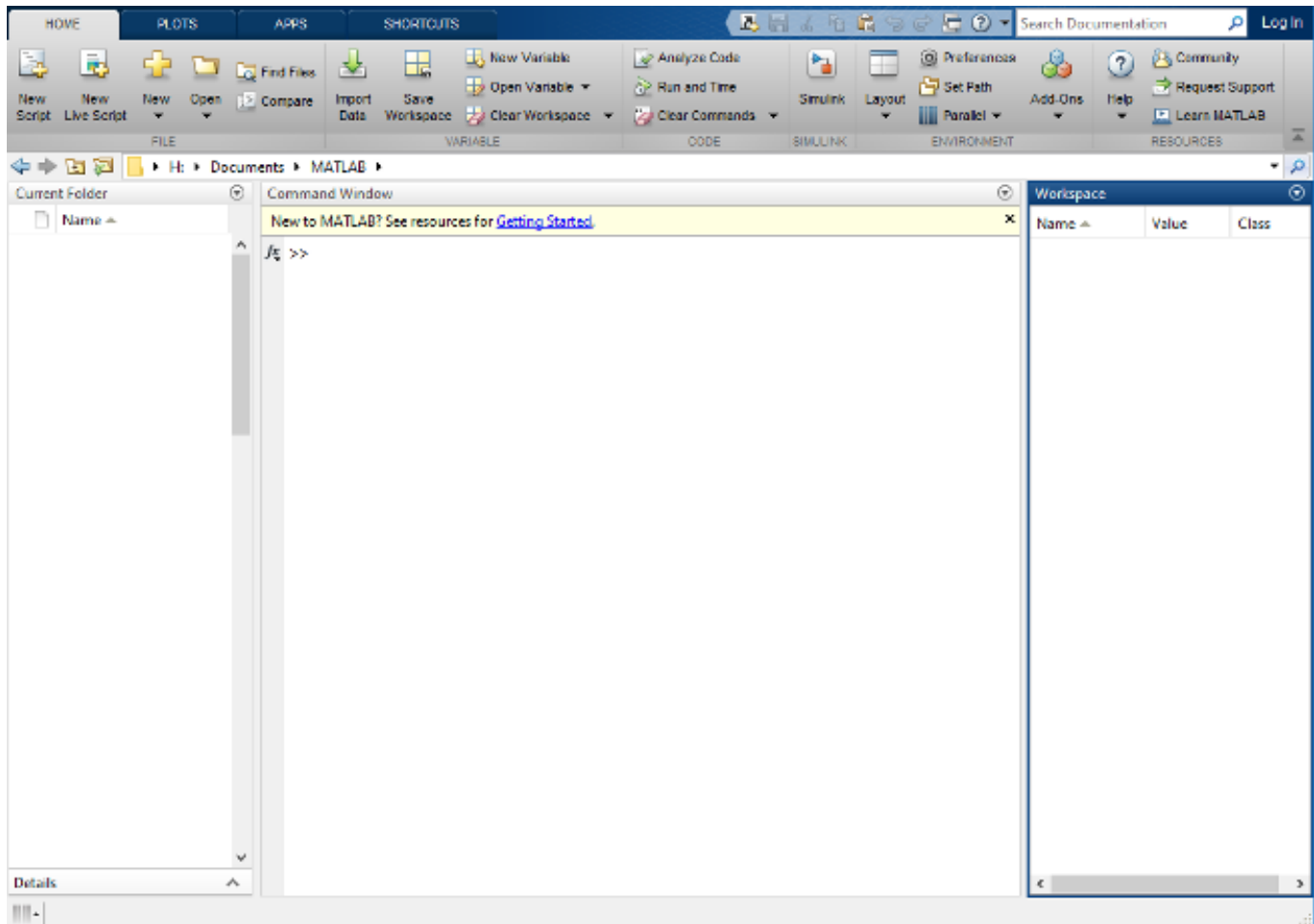
Link to the MATLAB website tutorial: <https://www.mathworks.com/help/matlab/getting-started-with-matlab.html>

Table of Contents

1. Desktop Basics	2
2. Data Types	
a. Numeric	5
b. Character/Strings	7
c. Dates and Times	8
d. Categorical Arrays	9
e. Tables	9
f. Timetables	10
g. Structures	11
h. Cell Array	11
i. Function Handles	12
j. Map Containers	13
k. Time Series	14
3. Arrays	15
4. Built in Functions	
a. Functions to Create a Matrix	18
b. Functions to Modify the Shape of a Matrix	18
c. Functions to Find the Structure or Shape of a Matrix	19
d. Functions to Determine Class	19
e. Functions to Sort and Shift Matrix Elements	19
f. Functions That Work on Diagonals of a Matrix	19
g. Functions to Change the Indexing Style	19
h. Functions for Working with Multidimensional Arrays	19
5. Plots	
a. Line Plots	20
b. Pie Charts, Bar Plots, and Histograms	22
c. Discrete Data Plots	26
d. Polar Plots	27
e. Contour Plots	28
f. Vector Fields	29
g. Surface and Mesh Plots	31
h. Volume Visualization	33
i. Animation	34
6. Loops/Indexing.....	36
7. Creating Your Own Function	37
8. Help	39
9. Debugging.....	39
10. Help	40

1. Desktop Basics

When opening up MATLAB the desktop appears in its default layout, as shown in figure 1 below.



As a starter you need only concern yourself with the **Command Window**, **Workspace**, and **Current Folder**. As you work more and more with MATLAB, you issue commands that create variables and call functions. For example, create these variables by typing them one at a time into command line, pressing enter after each line:

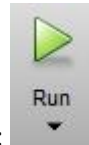
```
a = 1
b = 2
c = a + b
```

If you notice, c is now equal to 3. You'll also see that in the workspace panel on the right-hand side are the variables you just typed out. This means you have saved those values to your variables and can use them at any time.

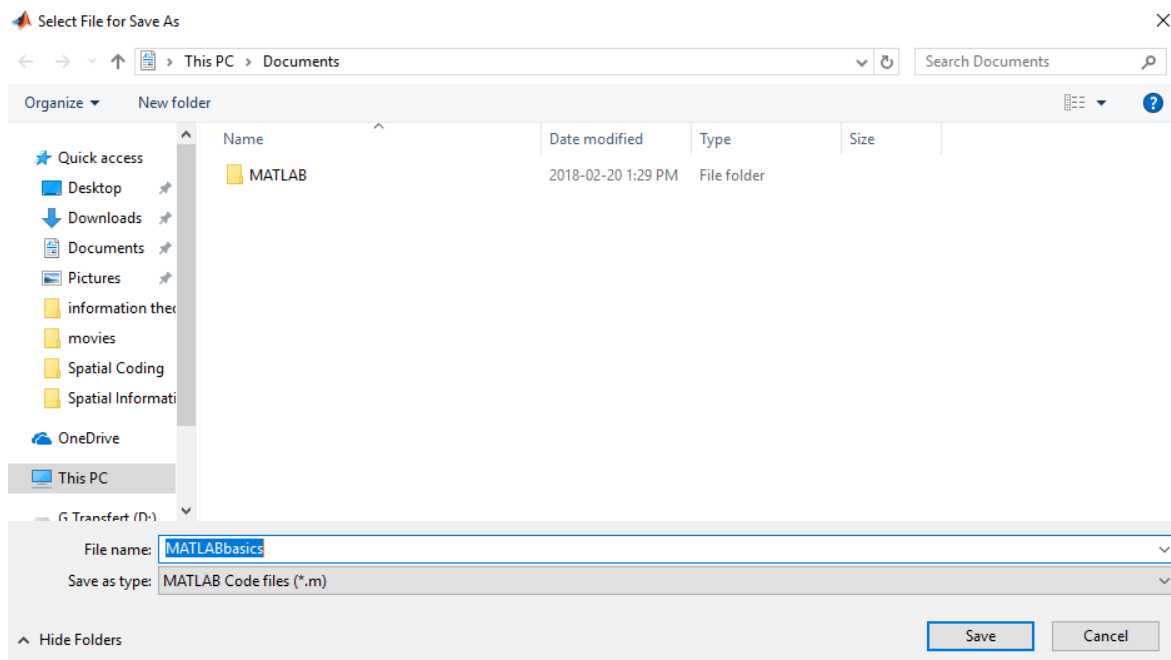
Another feature of the workspace panel is that you are able to import files for which already have variables, arrays, matrices, or structs with saved information. Of course, this is useful if you wish to save all the information you calculate one day to a script on another day (more detail on this to come).

Using the command panel is useful for executing functions and commands, but in order to create scripts and functions and save them, you need to start a New Script.

- Click "New Script", located in the top left hand corner
- Type in the same thing as previously typed in the command window



- Click "Run" button located just above your new script panel:
- MATLAB will then ask you to save your script before it executes your commands. Name the script "MATLABbasics" and type this in as the File name, just like figure below.



Now that you've run your script, you'll see in the Command Window your executed script and once again with your variables saved in the workspace.

It is important to note that every time you click Run MATLAB saves everything you've done, though it is still important to save your work regularly.

2. Data Types

Now that you've familiarized yourself with how to open and run a script, you should become aware of some of the types of variables, and data you can save. As well as some of the operations you can use in MATLAB in order to execute certain calculations.

There are several different data types you can assign to variables, depending on the type of function or script you want to create, knowing these different types can be of use.

Data Types	Description
Numeric	Integer and floating-point data
Characters and Strings	Text in character arrays and strings
Dates and Times	Arrays of date and time values that can be displayed in different formats
Categorical Arrays	Arrays of qualitative data with values from a finite set of discrete, nonnumeric data
Tables	Arrays in tabular form whose named columns can have different types
Timetables	Time-stamped data in tabular form
Structures	Arrays with named fields that can contain data of varying types and sizes
Cell Arrays	Arrays that can contain data of varying types and sizes
Function Handles	Variables that allow you to invoke a function indirectly
Map Containers	Objects with keys that index to values, where keys need not be integers
Time Series	Data vectors sampled over time

Table 1 above summarizes all the different data types you can assign variables. For most analysis you'll be using data types: **Numeric**, **Characters**, **Categorical Arrays**, **Tables**, **Structures**, and **Cell Arrays**. Therefore, though useful to familiarize yourself with all data types you'll most likely only need to know the 6 types just mentioned.

2.1 Numeric

Detailed Description: <https://www.mathworks.com/help/matlab/numeric-types.html>

Numeric classes in MATLAB include signed and unsigned integers, and single-precision and double-precision floating point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single precision arrays offer more memory-efficient storage than double precision but are more limited in variable size and precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

It is also possible for MATLAB to assign variables as infinity or NaN (Not A Number).

Examples below show some different uses of numeric assignments to variables and their corresponding outputs:

***The typical assignment for a variable (and most used) is to assign a variable a simple integer with little precision for simple calculations:

```
gamma =
    3.1000
fx >>
```

Function	Description
double()	<p>Without a specified data type MATLAB stores all variables as doubles. This means your variable will have 5 significant digits saved. Here a double will save any variable as 8 bytes, meaning MATLAB will not be able to save anything below 10^{-324} or above 10^{308}. A value outside this range will output either a $-\text{inf}$ or inf depending on your value being negative or positive.</p> <pre data-bbox="565 533 915 995"> >> gamma = 3.1132321 gamma = 3.1132 >> double(gamma) ans = 3.1132 fx >> </pre>
single()	<p>A data type specified as single will have 4 bytes saved to the variable. Since the program allocates less memory with variables assigned to this function, the minimum value you can assign to data type single is 10^{-45} and maximum value of 10^{38}.</p> <pre data-bbox="565 1205 886 1730"> >> single(3*10^39) ans = single Inf >> double(3*10^39) ans = 3.0000e+39 fx >> </pre>
int8()	<p>Similar to the above data types, with the exception that the values must be integers (no decimals) as well as how much memory is saved in this type. Variables with data type int8 saves your values with 1 byte. This</p>

	means you cannot exceed anything higher than 127 and lower than -128.
int16()	Variables with data type int16 saves your integer values with 1 byte. This means you cannot exceed anything higher than 32767 and lower than -32768.
int32()	Variables with data type int32 saves your integer values with 1 byte. This means you cannot exceed anything higher than 2147483647 and lower than -2147483648.
Nan()	This function returns the IEEE arithmetic representation for Not-A-Number. These values result from operations which have undefined numerical results.
isfloat()	<p>This function returns a true or false output (1,0) for which are type single and double. Float point numbers are by default double precision data types with 754 for double precision.</p> <pre>>> isfloat(pi) ans = logical 1</pre>

2.2 Characters and Strings

If you want a variable to save a text file of characters, you assign it similarly as a regular integer value but with single quotes around the text:

```
myText = 'Hello, world';
```

This is saved in MATLAB as an array of char values (details on arrays later), which is short for character. Each char typed in myText is seen as 2 bytes. Meaning, since there are 12 characters in myText, including the comma and space, there are a total of 24 bytes in myText.

You can concatenate character arrays with square brackets the same way as you concatenate numeric arrays:

```
otherText = 'You're right'
```

```
longText = [myText, ' - ', otherText]
```

```
longText =
'Hello, world - You're right'
```

You can also convert numerical values into character arrays using the **num2str()** function:

```
f = 71;  
c = (f-32)/1.8;  
tempText = ['Temperature is ', num2str(c), 'C']
```

```
tempText =  
'Temperature is 21.6667C'
```

2.3 Dates and Times

The date and time data types `datetime`, `duration`, and `calendarDuration` support efficient computations, comparisons, and formatted display of dates and times. Work with these arrays in the same way that you work with numeric arrays. You can add, subtract, sort, compare, concatenate, and plot date and time values. You also can represent dates and times as numeric arrays or as text.

```
>> t = datetime('now','TimeZone','local','Format','d-MMM-y HH:mm:ss Z')
```

```
t =
```

```
datetime
```

```
13-Mar-2018 14:50:17 -0400
```

```
>> Y = [2014;2013;2012];  
M = 01;  
D = [31;30;31];  
>> t = datetime(Y,M,D)
```

```
t =
```

```
3×1 datetime array
```

```
31-Jan-2014  
30-Jan-2013  
31-Jan-2012
```

```
>> t = datetime(Y,M,D,'Format','eeee, MMMM d, y')
```

```
t =
```

```
3×1 datetime array
```

```
Friday, January 31, 2014  
Wednesday, January 30, 2013  
Tuesday, January 31, 2012
```


2.4 Categorical Arrays

Categorical is a data type that assigns values to a finite set of discrete categories, such as High, Med, and Low. These categories can have a mathematical ordering that you specify, such as High > Med > Low, but it is not required. A categorical array provides efficient storage and convenient manipulation of nonnumeric data, while also maintaining meaningful names for the values. A common use of categorical arrays is to specify groups of rows in a table.

Syntax

```
B = categorical(A)
B = categorical(A,valueset)
B = categorical(A,valueset,catnames)
B = categorical(A, __ ,Name,Value)
```

```
>> Temps = [58; 72; 56; 90; 76];
Dates = {'2017-04-17'; '2017-04-18'; '2017-04-30'; '2017-05-01'; '2017-04-27'};
Stations = {'S1'; 'S2'; 'S1'; 'S3'; 'S2'};
>> Stations = categorical(Stations)
```

```
Stations =
```

```
5×1 categorical array
```

```
S1
S2
S1
S3
S2
```

```
>> categories(Stations)
```

```
ans =
```

```
3×1 cell array
```

```
{ 'S1' }
{ 'S2' }
{ 'S3' }
```

2.5 Tables

The *table* is a data type suitable for column-oriented or tabular data that is often stored as columns in a text file or in a spreadsheet. Tables consist of rows and column-oriented variables. Each variable in a table can have a different data type and a different size with the one restriction that each variable must have the same number of rows.

You can read data from a file into a table using either the [Import Tool](#) or the [readtable](#) function. Alternatively, use the [table](#) function shown below to create a table from existing workspace variables.

```
>> LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
>> T = table(Age,Height,Weight,BloodPressure,...
    'RowNames',LastName)
```

T =

5×4 [table](#)

	Age	Height	Weight	BloodPressure	
Smith	38	71	176	124	93
Johnson	43	69	163	109	77
Williams	38	64	131	125	83
Jones	40	67	133	117	75
Brown	49	64	119	122	80

2.6 Timetables

Timetable is a type of table that associates a time with each row. Like tables, timetables can store column-oriented data variables that have different data types and sizes, as long as they have the same number of rows. In addition, timetables provide time-specific functions to align, combine, and perform calculations with one or more timetables.

The *row times* of a timetable are datetime or duration values that label the rows. You can index into a timetable by row time and variable. To index into a timetable, use smooth parentheses () to return a subtable or curly braces {} to extract the contents. You can reference variables and the vector of row times using names.

```
>> Time = datetime({'2015-12-18 08:03:05';'2015-12-18 10:03:17';'2015-12-18 12:03:13'});
Temp = [37.3;39.1;42.3];
Pressure = [30.1;30.03;29.9];
WindSpeed = [13.4;6.5;7.3];
WindDirection = categorical({'NW';'N';'NW'});
TT = timetable(Time,Temp,Pressure,WindSpeed,WindDirection)
```

TT =

3×4 [timetable](#)

Time	Temp	Pressure	WindSpeed	WindDirection
18-Dec-2015 08:03:05	37.3	30.1	13.4	NW
18-Dec-2015 10:03:17	39.1	30.03	6.5	N
18-Dec-2015 12:03:13	42.3	29.9	7.3	NW

2.7 Structures

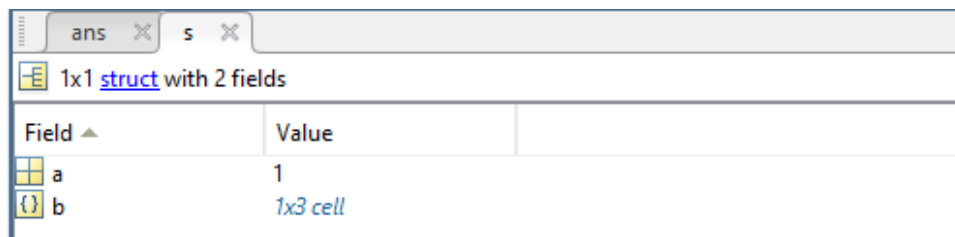
A *structure array* is a data type that groups related data using data containers called *fields*. Each field can contain any type of data. Access data in a field using dot notation of the form `structName.fieldName`.

This is very useful tool in saving data when using large programs with lots of analysis calculations.

A simple creation could be:

```
>> s.a = 1;  
s.b = {'A','B','C'}  
  
s =  
  
struct with fields:  
  
a: 1  
b: {'A' 'B' 'C'}
```

This is saved into your workspace as "s" a 1x1 struct. Double clicking on "s" you'll see your variables a and b saved as:



The image shows a MATLAB workspace window with two tabs: 'ans' and 's'. The 's' tab is active, showing a 1x1 struct with 2 fields. The fields are listed in a table:

Field	Value
a	1
b	1x3 cell

An even more complicated use of a structure array is creating a struct within a struct array:

```
s.n.a = ones(3);  
s.n.b = eye(4);  
s.n.c = magic(5);
```



The image shows a MATLAB workspace window with a table of fields and values for a structure:

Field	Value
a	1
b	1x3 cell
n	1x1 struct

2.8 Cell Arrays

A cell array is a data type with indexed data containers called cells, where each cell can contain any type of data. Cell arrays commonly contain either lists of text strings, combinations of text and numbers, or numeric arrays of different sizes. Refer to sets of cells by enclosing indices in smooth parentheses, `()`. Access the contents of cells by indexing with curly braces, `{}`:

```
>> C = {1,2,3;
        'text',rand(5,10,2),{11; 22; 33}}

C =

2×3 cell array

    {[ 1]}    {[          2]}    {[ 3]}
    {'text'}  {5×10×2 double}  {3×1 cell}
```

Another way to initialize a cell array variable is to do the following:

```
>> C = cell(3)

C =

3×3 cell array

    {0×0 double}    {0×0 double}    {0×0 double}
    {0×0 double}    {0×0 double}    {0×0 double}
    {0×0 double}    {0×0 double}    {0×0 double}
```

2.9 Function Handles

A function handle is a data type that stores an association to a function. For example, you can use a function handle to construct anonymous functions or specify call back functions. Also, you can use a function handle to pass a function to another function, or call local functions from outside the main function.

To create a function handle, precede the function name with an @sign. For example, if you wish a function called *myfunction*, create a handle named *f* as follows:

```
f = @myfunction;
```

You call a function using a handle the same way you call the function directly. For example, suppose that you have a function named *computeSquare*, defined as:

```
function y = computeSquare(x)
y = x.^2;
end
```

Create a handle and call the function to compute the square of four.

```
f = @computeSquare;  
a = 4;  
b = f(a)
```

2.10 Map Containers

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements. Unlike most array data structures in the MATLAB® software that only allow access to the elements by means of integer indices, the indices for a Map can be nearly any scalar numeric value or a character vector.

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map. Each entry of a Map contains exactly one unique key and its corresponding value. Indexing into the Map of rainfall statistics shown below with a character vector representing the month of August yields the value internally associated with that month, 37.3.

A Map object is a data structure that allows you to retrieve values using a corresponding key. Keys can be real numbers or character vectors and provide more flexibility for data access than array indices, which must be positive integers. Values can be scalar or nonscalar arrays.

```
>> keySet = {'Jan', 'Feb', 'Mar', 'Apr'};  
valueSet = [327.2, 368.2, 197.6, 178.4];  
mapObj = containers.Map(keySet,valueSet)  
  
mapObj =  
  
Map with properties:  
  
    Count: 4  
    KeyType: char  
    ValueType: double
```

You are able to use the map to look up a specific property using dot notation, or a quotation. If for example, using the example above, you wanted to look up the rainfall data for February:

```
>> rainFeb = mapObj('Feb')  
  
rainFeb =  
  
    368.2000
```

Or add more data:

```
mapObj('May') = 100.0;
```

```
>> keySet    = {'Jun','Jul','Aug'};
valueSet = [ 69.9, 32.3, 37.3];
newMap = containers.Map(keySet,valueSet);

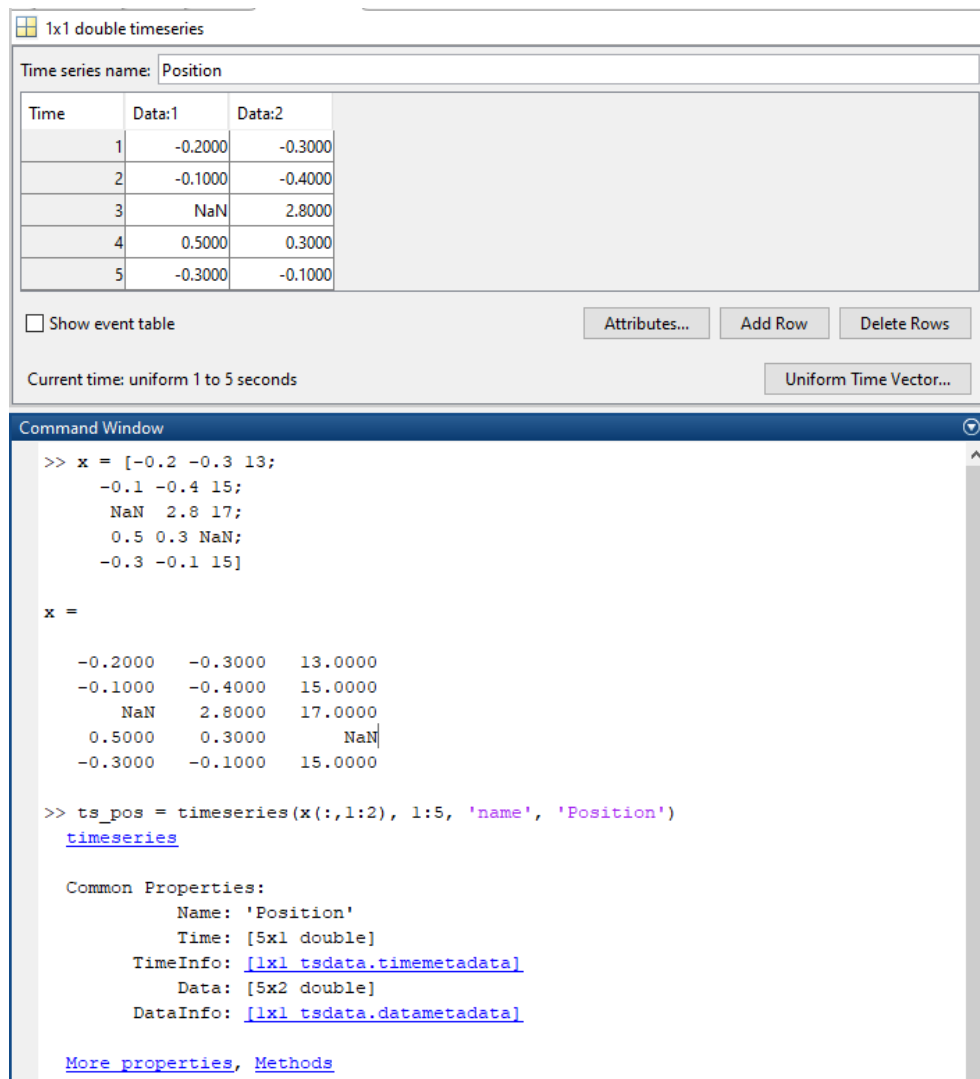
mapObj = [mapObj; newMap];
```

2.11 Time Series

Time series are data vectors sampled over time, in order, often at regular intervals. They are distinguished from randomly sampled data that form the basis of many other data analyses. Time series represent the time-evolution of a dynamic population or process. The linear ordering of time series gives them a distinctive place in data analysis, with a specialized set of techniques. Time series analysis is concerned with:

- Identifying patterns
- Modeling patterns
- Forecasting values

`ts = timeseries(tsname) creates an empty time-series object using the name, tsname, for the time-series object. This name can differ from the time-series variable name.`



3. 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.

To create an array, you can either create a row vector or a column vector:

```
>> a = [1 2 3 4]

a =

     1     2     3     4

>> a = [1; 2; 3; 4]

a =

     1
     2
     3
     4
```

Or multidimensional with both rows and vectors:

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

a =

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

You can also create an array with all ones, zeros, or random numbers. Creating an array full of zeros to initialize a variable is a good way to save computing power when filling in an array.

When applying basic calculations to your variables, you can treat MATLAB the same as you would treat a single independent variable value:


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

a =

     1     2     3
     4     5     6
     7     8    10

>> a + 10

ans =

    11    12    13
    14    15    16
    17    18    20

>> sin(a)

ans =

    0.8415    0.9093    0.1411
   -0.7568   -0.9589   -0.2794
    0.6570    0.9894   -0.5440

fx >> |
```

In addition, you can transpose a matrix by using a single quote ('):

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

a =

     1     2     3
     4     5     6
     7     8    10

>> a'

ans =

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

You can also perform element wise multiplication by using the `.*` operator:

```
>> p = a.*a

p =

     1     4     9
    16    25    36
    49    64   100
```

There are a lot of different manipulations and functions to use on arrays and matrices, but for the case of using them as large data arrays for analysis reasons you should familiarize yourself through this website with all the different operations available: <https://www.mathworks.com/help/matlab/math/summary-of-matrix-and-array-functions.html>

MATLAB is very useful in that majority of the calculations and functions are already done for you, just need to find the right function or combination of functions to perform on your data set. Later in this tutorial I will go through a practical example that will give you a general idea of the applications MATLAB has to offer.

4. Built in Functions

This section summarizes the principal functions used in creating and handling matrices. Most of these functions work on multidimensional arrays as well

4.1 Functions to Create a Matrix

Function	Description
[a,b] or [a;b]	Create a matrix from specified elements, or concatenate matrices together.
accumarray	Construct a matrix using accumulation.
blkdiag	Construct a block diagonal matrix.
cat	Concatenate matrices along the specified dimension.
diag	Create a diagonal matrix from a vector.
horzcat	Concatenate matrices horizontally.
magic	Create a square matrix with rows, columns, and diagonals that add up to the same number.
ones	Create a matrix of all ones.
rand	Create a matrix of uniformly distributed random numbers.
repmat	Create a new matrix by replicating or tiling another.
vertcat	Concatenate two or more matrices vertically.
zeros	Create a matrix of all zeros.

4.2 Functions to Modify the Shape of a Matrix

Function	Description
ctranspose	Flip a matrix about the main diagonal and replace each element with its complex conjugate.
flip	Flip a matrix along the specified dimension.
fliplr	Flip a matrix about a vertical axis.
flipud	Flip a matrix about a horizontal axis.
reshape	Change the dimensions of a matrix.
rot90	Rotate a matrix by 90 degrees.
transpose	Flip a matrix about the main diagonal.

4.3 Functions to Find the Structure or Shape of a Matrix

Function	Description
isempty	Return true for 0-by-0 or 0-by-n matrices.
isscalar	Return true for 1-by-1 matrices.
issparse	Return true for sparse matrices.
isvector	Return true for 1-by-n matrices.
length	Return the length of a vector.
ndims	Return the number of dimensions in a matrix.
numel	Return the number of elements in a matrix.
size	Return the size of each dimension.

4.4 Functions to Determine Class

Function	Description
iscell	Return true if the matrix is a cell array.
ischar	Return true if matrix elements are character arrays.
isfloat	Determine if input is a floating point array.
isinteger	Determine if input is an integer array.
islogical	Return true if matrix elements are logicals.
isnumeric	Return true if matrix elements are numeric.
isreal	Return true if matrix elements are real numbers.
isstruct	Return true if matrix elements are MATLAB® structures.

4.5 Functions to Sort and Shift Matrix Elements

Function	Description
circshift	Circularly shift matrix contents.
issorted	Return true if the matrix elements are sorted.
sort	Sort elements in ascending or descending order.
sortrows	Sort rows in ascending order.

4.6 Functions That Work on Diagonals of a Matrix

Function	Description
blkdiag	Construct a block diagonal matrix.
diag	Return the diagonals of a matrix.
trace	Compute the sum of the elements on the main diagonal.
tril	Return the lower triangular part of a matrix.
triu	Return the upper triangular part of a matrix.

4.7 Functions to Change the Indexing Style

Function	Description
ind2sub	Convert a linear index to a row-column index.
sub2ind	Convert a row-column index to a linear index.

4.8 Functions for Working with Multidimensional Arrays

Function	Description
cat	Concatenate arrays.
circshift	Shift array circularly.
ipermute	Inverse permute array dimensions.
ndgrid	Generate arrays for n-dimensional functions and interpolation.

ndims	Return the number of array dimensions.
permute	Permute array dimensions.
shiftdim	Shift array dimensions.
squeeze	Remove singleton dimensions.

5. Plots

There are various functions you can use to plot data in MATLAB. The tables below describe the different types and how to use them.

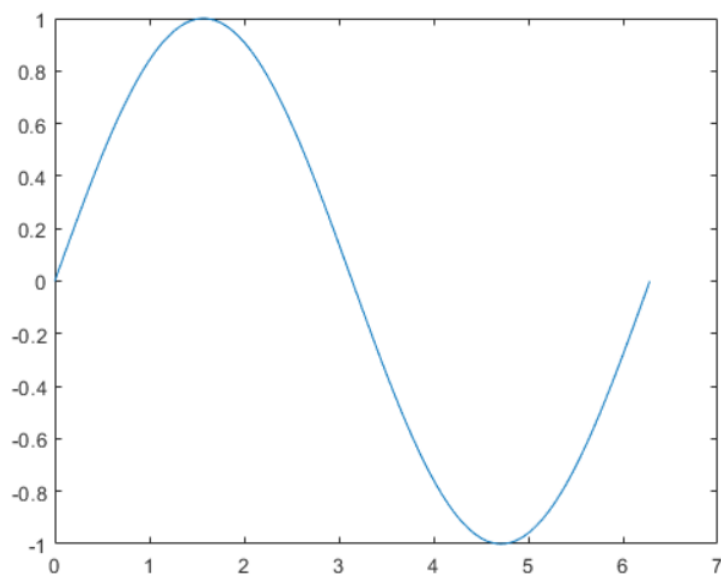
5.1 Line Plots

Linear, log-log, semi-log, error bar plots.

plot	2-D line plot
plot3	3-D line plot
loglog	Log-log scale plot
semilogx	Semilogarithmic plot
semilogy	Semilogarithmic plot
errorbar	Line plot with error bars
fplot	Plot expression or function
fplot3	3-D parametric curve plotter
fimplicit	Plot implicit function

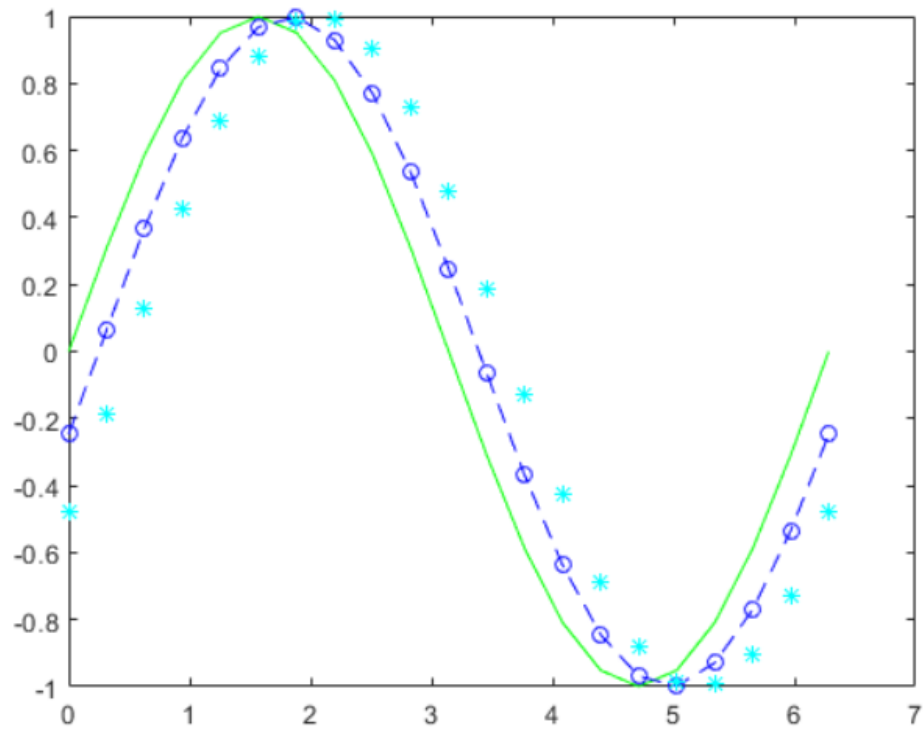
Looking more specifically at the "plot" function, we can perform this simple example:

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



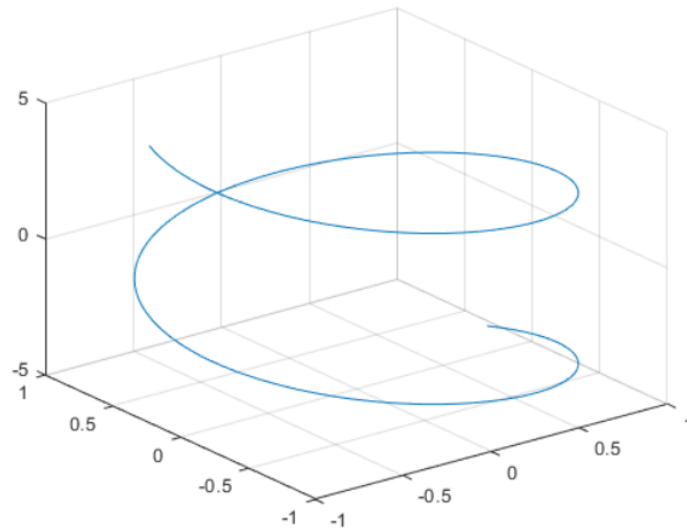
If you wanted to specify line style, color, and marker:

```
>> x = 0:pi/10:2*pi;  
y1 = sin(x);  
y2 = sin(x-0.25);  
y3 = sin(x-0.5);  
  
figure  
plot(x,y1,'g',x,y2,'b--o',x,y3,'c*')
```



You can also plot in 3D:

```
>> xt = @(t) sin(t);  
yt = @(t) cos(t);  
zt = @(t) t;  
fplot3(xt,yt,zt)
```



5.2 Pie Charts, Bar Plots, and Histograms

These are plots you would use when wanting to proportion and distribute data.

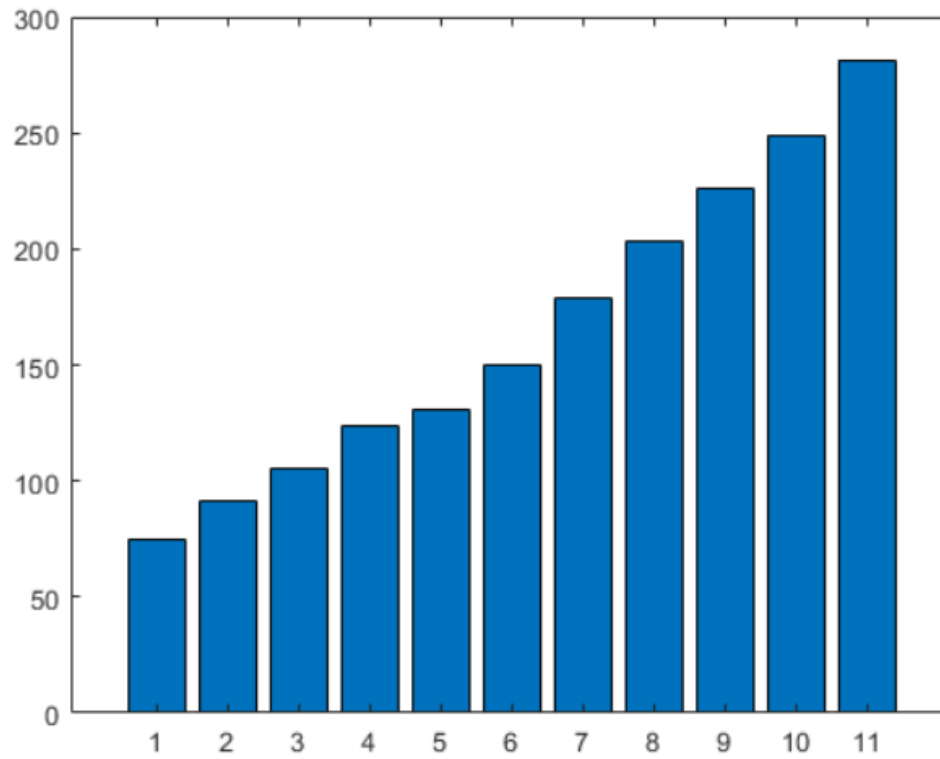
bar	Bar graph
bar3	Plot 3-D bar graph
barh	Plot bar graph horizontally
bar3h	Plot horizontal 3-D bar graph
histogram	Histogram plot
histogram2	Bivariate histogram plot
morebins	Increase number of histogram bins
fewerbins	Decrease number of histogram bins
histcounts	Histogram bin counts
histcounts2	Bivariate histogram bin counts
binscatter	Binned scatter plot
rose	Angle histogram plot
pareto	Pareto chart
area	Filled area 2-D plot
pie	Pie chart
pie3	3-D pie chart

Properties

Bar Properties	Bar chart appearance and behavior
Area Properties	Area chart appearance and behavior
Histogram Properties	Histogram appearance and behavior
Histogram2 Properties	Histogram2 appearance and behavior
Binscatter Properties	Binscatter appearance and behavior

Using a simple "bar()" we get the following example:

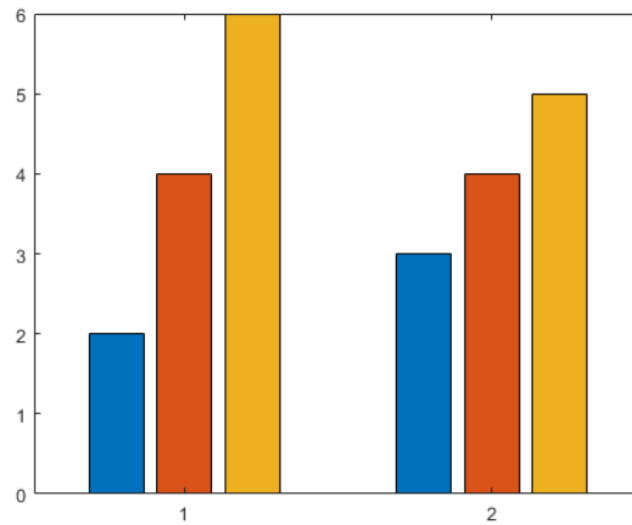
```
>> y = [75 91 105 123.5 131 150 179 203 226 249 281.5];  
bar(y)
```



As well, similarly to using plot, you can change properties for a specific bar series by indexing into an object array:

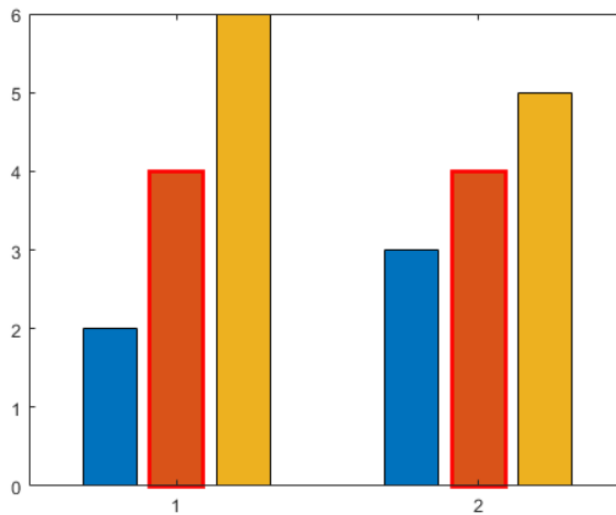
If we first define an array and display its bar output:

```
>> y = [2 4 6; 3 4 5];  
b = bar(y);
```



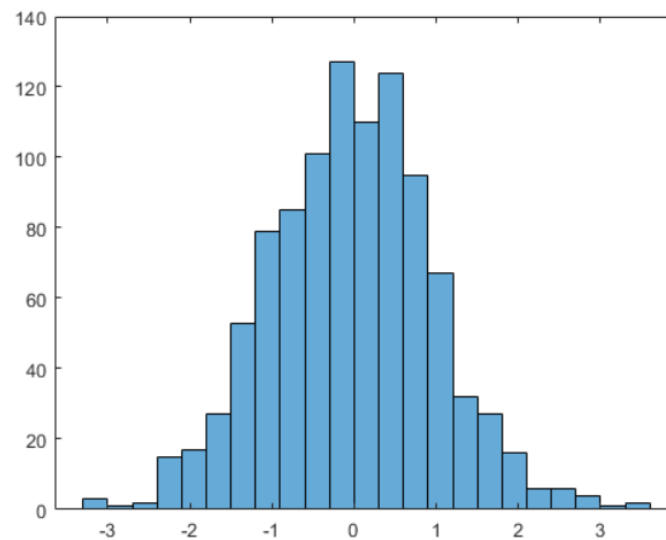
And now changing properties of the bars representing the second column of y using b(2).

```
>> b(2).LineWidth = 2;  
b(2).EdgeColor = 'red';
```

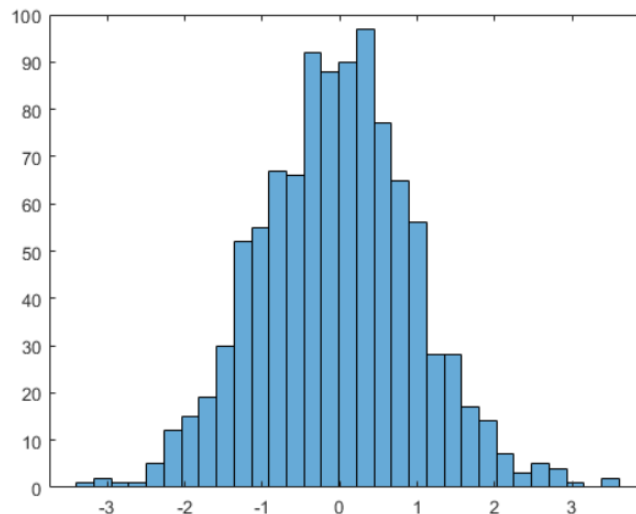


A function more likely to be of use to data analysis would be the "histogram()" function. An example below shows 1000 randomly generated numbers and some ways you can alter your graph:

```
>> X = randn(1000,1);  
h = histogram(X)  
  
h =  
Histogram with properties:  
  
    Data: [1000x1 double]  
   Values: [1x23 double]  
  NumBins: 23  
 BinEdges: [1x24 double]  
BinWidth: 0.3000  
BinLimits: [-3.3000 3.6000]  
Normalization: 'count'  
  FaceColor: 'auto'  
 EdgeColor: [0 0 0]  
  
Show all properties
```



```
h.NumBins = 31;
```



5.3 Discrete Data Plots

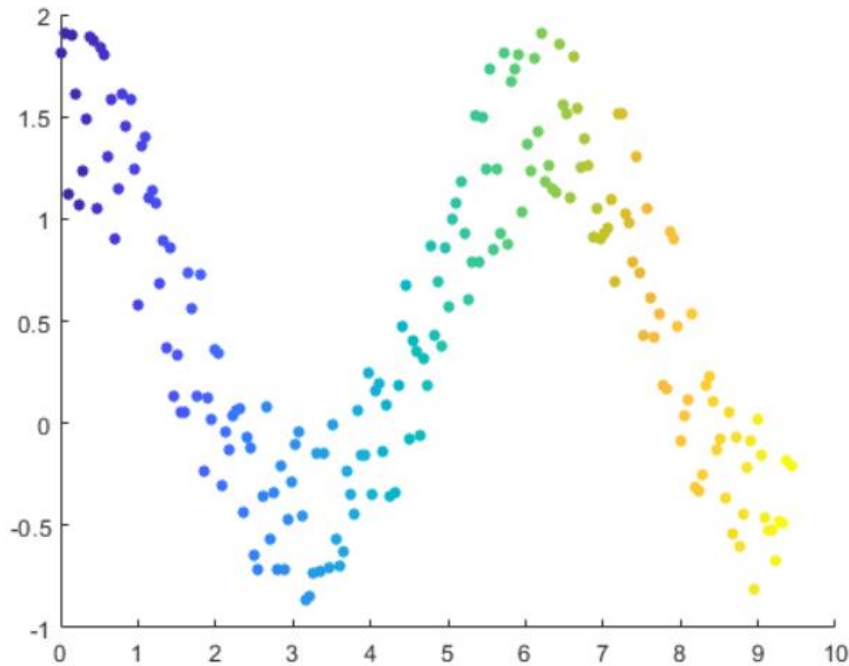
These types of plots are useful for conducting exploratory analysis and communicate experimental results. Majority of the time an experimenter would use this as a way to compare two variables of data in a scatter plot in order to discover their correlations.

stem	Plot discrete sequence data
stairs	Stairstep graph
stem3	Plot 3-D discrete sequence data
scatter	Scatter plot
scatter3	3-D scatter plot
spy	Visualize sparsity pattern
plotmatrix	Scatter plot matrix
heatmap	Create heatmap chart
sortx	Sort elements in heatmap row
sorty	Sort elements in heatmap column
wordcloud	Create word cloud chart from text data
geobubble	Visualize data values at specific geographic locations
geolimits	Set or query geographic limits

Properties

Stem Properties	Stem chart appearance and behavior
Stair Properties	Stair chart appearance and behavior
Scatter Properties	Scatter chart appearance and behavior
HeatmapChart Properties	Heatmap chart appearance and behavior
WordCloudChart Properties	Control word cloud chart appearance and behavior
GeographicBubbleChart Properties	Control geographic bubble chart appearance and behavior

```
>> x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
sz = 25;
c = linspace(1,10,length(x));
scatter(x,y,sz,c,'filled')
```

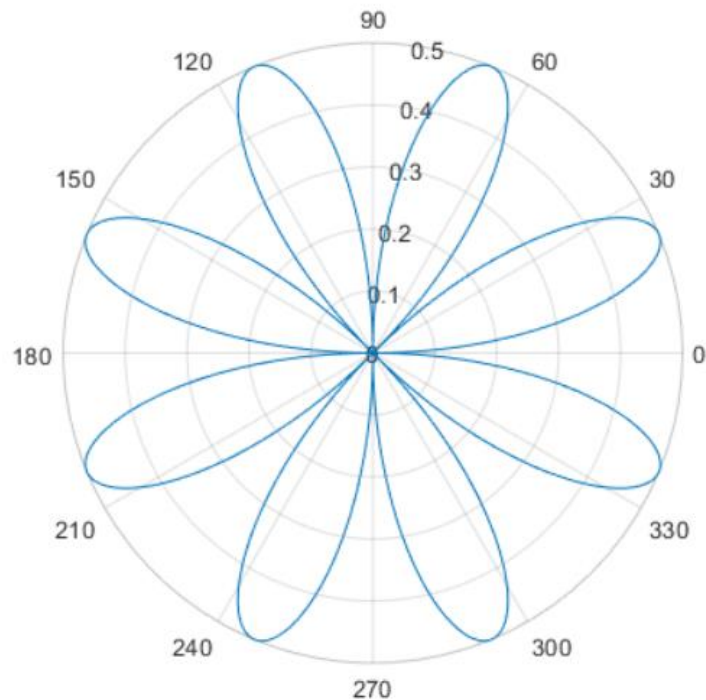


5.4 Polar Plots

polarplot	Plot line in polar coordinates
polarscatter	Scatter chart in polar coordinates
polarhistogram	Histogram chart in polar coordinates
compass	Plot arrows emanating from origin
ezpolar	Easy-to-use polar coordinate plotter

`polarplot(theta,rho)` plots a line in polar coordinates, with `theta` indicating the angle in radians and `rho` indicating the radius value for each point. The inputs must be vectors with equal length or matrices with equal size. If the inputs are matrices, then `polarplot` plots columns of `rho` versus columns of `theta`. Alternatively, one of the inputs can be a vector and the other a matrix as long as the vector is the same length as one dimension of the matrix.

```
>> theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
```



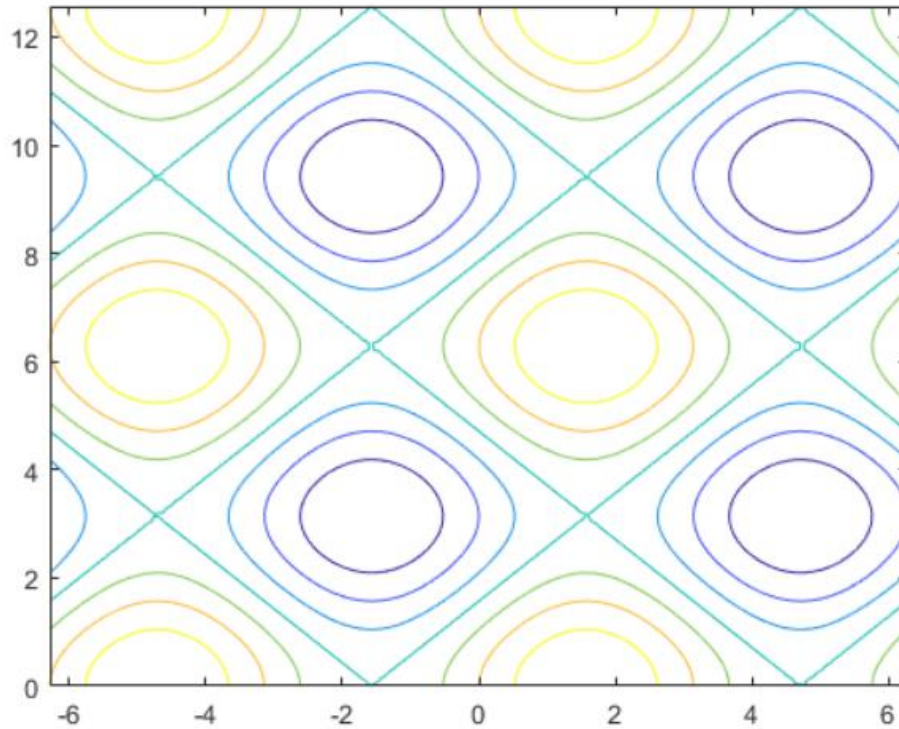
5.5 Contour Plots

`contour(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the `x-y` plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The `x` values correspond to the column indices of `Z` and the `y` values correspond to the row indices of `Z`. The contour levels are chosen automatically.

contour	Contour plot of matrix
contourf	Filled 2-D contour plot
contourc	Low-level contour plot computation
contour3	3-D contour plot
contourslice	Draw contours in volume slice planes
clabel	Label contour plot elevation
fcontour	Plot contours

```
>> x = linspace(-2*pi,2*pi);
y = linspace(0,4*pi);
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);

figure
contour(X,Y,Z)
```

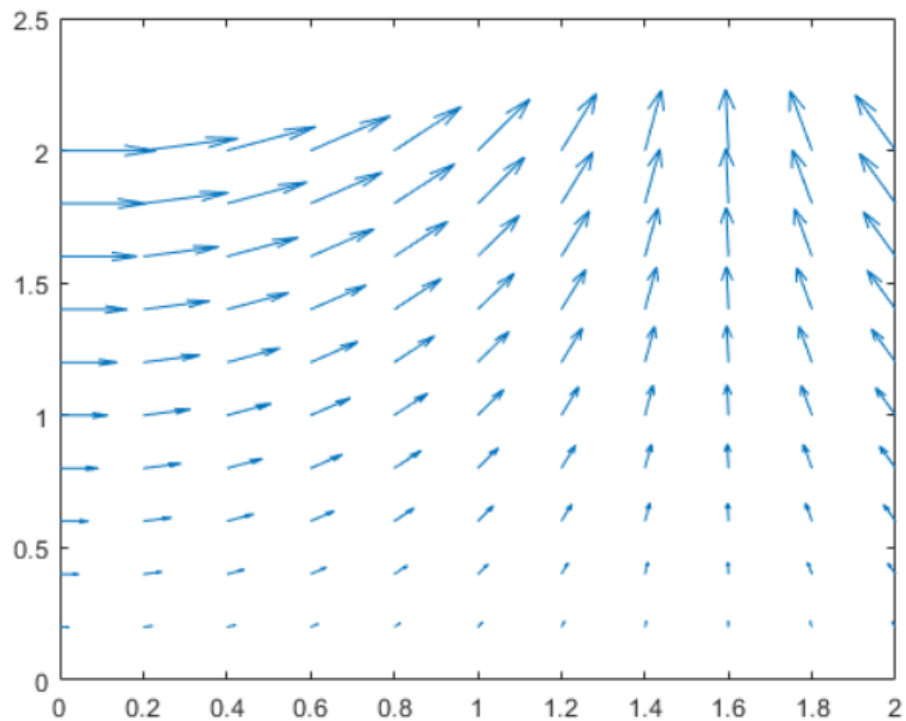


5.6 Vector Fields

The more useful function from the list below is a quiver plot. In representing field vectors a quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y) .

<u>feather</u>	Plot velocity vectors
<u>quiver</u>	Quiver or velocity plot
<u>compass</u>	Plot arrows emanating from origin
<u>quiver3</u>	3-D quiver or velocity plot
<u>streamslice</u>	Plot streamlines in slice planes
<u>streamline</u>	Plot streamlines from 2-D or 3-D vector data

```
>> [x,y] = meshgrid(0:0.2:2,0:0.2:2);  
u = cos(x).*y;  
v = sin(x).*y;
```



5.7 Surface and Mesh Plots

These plots generate two and three-dimensional surface plots. The functions plot the values in the inputted matrices in grid planes, with addition that you are able to specify colour, rows/columns, transparency, and returning chart surface object.

surf	Surface plot
surfc	Contour plot under a 3-D shaded surface plot
surface	Create surface object
surf1	Surface plot with colormap-based lighting
surfnorm	Compute and display 3-D surface normals
mesh	Mesh plot
meshc	Plot a contour graph under mesh graph
meshz	Plot a curtain around mesh plot
hidden	Remove hidden lines from mesh plot
fsurf	Plot 3-D surface
fmesh	Plot 3-D mesh
fimplicit3	Plot 3-D implicit function
waterfall	Waterfall plot
ribbon	Ribbon plot
contour3	3-D contour plot
peaks	Example function of two variables
cylinder	Generate cylinder
ellipsoid	Generate ellipsoid
sphere	Generate sphere
pcolor	Pseudocolor (checkerboard) plot
surf2patch	Convert surface data to patch data

Properties

Surface Properties	Chart surface appearance and behavior
Surface Properties	Primitive surface appearance and behavior
FunctionSurface Properties	Surface chart appearance and behavior
ImplicitFunctionSurface Properties	Implicit surface chart appearance and behavior
ParameterizedFunctionSurface Properties	Parameterized surface chart appearance and behavior

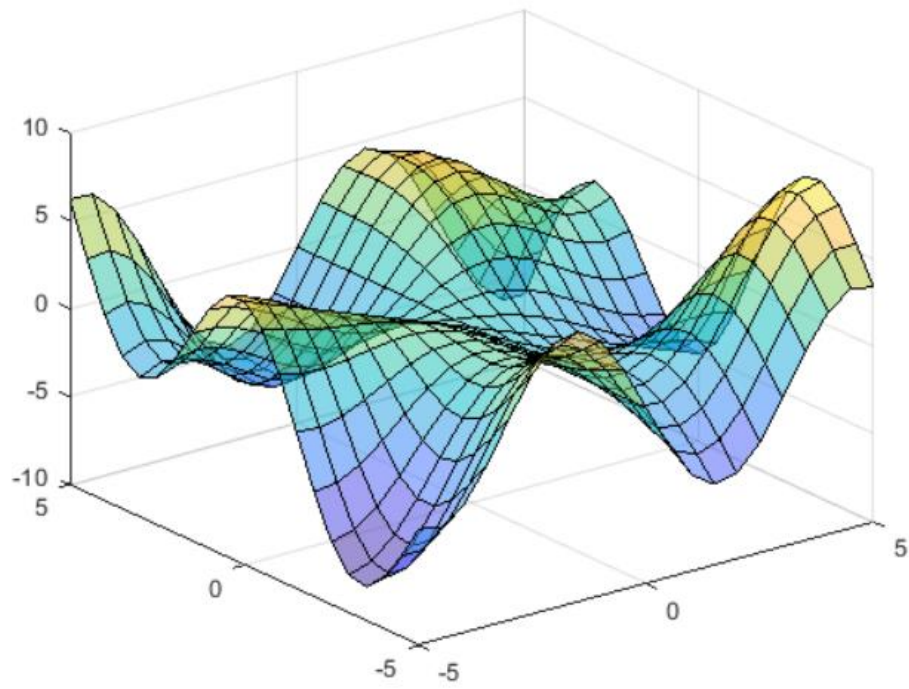
```
>> [X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
s = surf(X,Y,Z, 'FaceAlpha',0.5)
```

s =

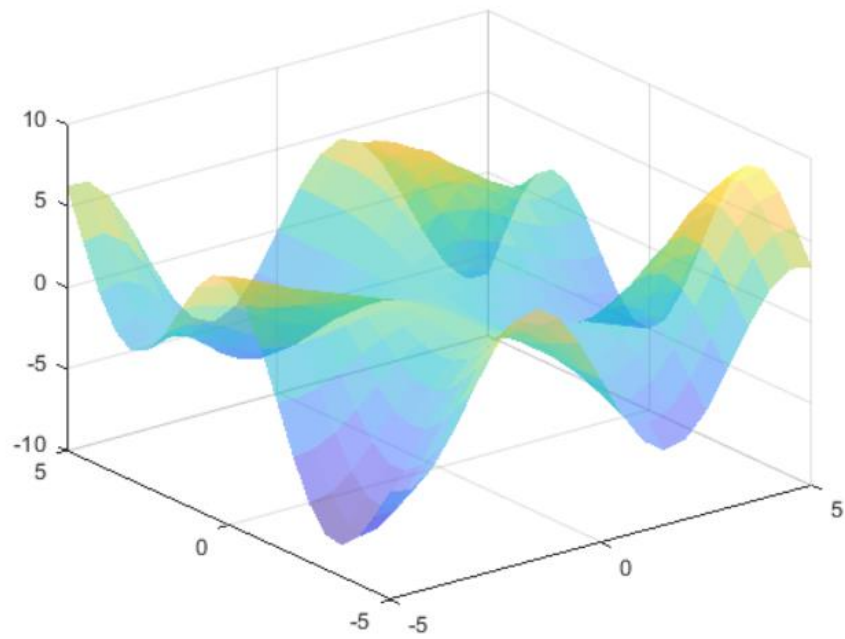
[Surface](#) with properties:

```
EdgeColor: [0 0 0]
LineStyle: '-'
FaceColor: 'flat'
FaceLighting: 'flat'
FaceAlpha: 0.5000
XData: [21×21 double]
YData: [21×21 double]
ZData: [21×21 double]
CData: [21×21 double]
```

Show [all properties](#)



```
>> s.EdgeColor = 'none';
```



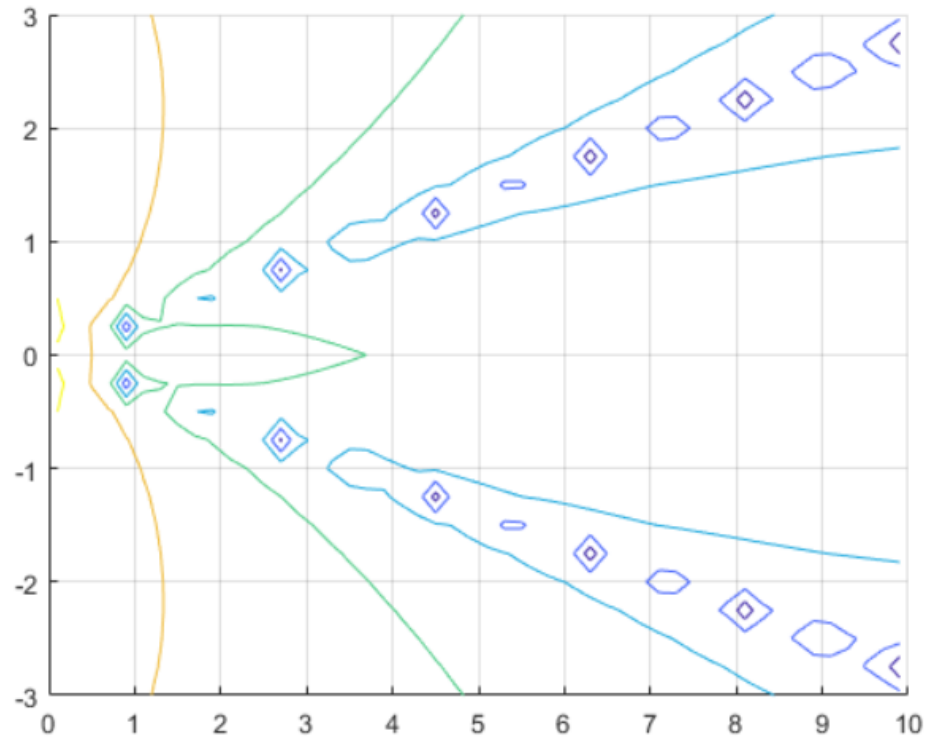
5.8 Volume Visualization

These plots represent gridded volume data as iso, slice, and stream plots:

contourslice	Draw contours in volume slice planes
flow	Simple function of three variables
isocaps	Compute isosurface end-cap geometry
isocolors	Calculate isosurface and patch colors
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
reducepatch	Reduce number of patch faces
reducevolume	Reduce number of elements in volume data set
shrinkfaces	Reduce size of patch faces
slice	Volume slice planes
smooth3	Smooth 3-D data
subvolume	Extract subset of volume data set
volumebounds	Coordinate and color limits for volume data
coneplot	Plot velocity vectors as cones in 3-D vector field
curl	Compute curl and angular velocity of vector field
divergence	Compute divergence of vector field
interpstreamspeed	Interpolate stream-line vertices from flow speed
stream2	Compute 2-D streamline data
stream3	Compute 3-D streamline data
streamline	Plot streamlines from 2-D or 3-D vector data
streamparticles	Plot stream particles
streamribbon	3-D stream ribbon plot from vector volume data
streamslice	Plot streamlines in slice planes
streamtube	Create 3-D stream tube plot

An example of a `contourslice()` would be:

```
>> [X,Y,Z,V] = flow;  
zslice = 0;  
contourslice(X,Y,Z,V, [], [], zslice)  
grid on
```



5.9 Animation

These functions are a way for you to produce an animating plot.

movie	Play recorded movie frames
getframe	Capture axes or figure as movie frame
frame2im	Return image data associated with movie frame
im2frame	Convert image to movie frame
animatedline	Create animated line
addpoints	Add points to animated line
getpoints	Return points that define animated line
clearpoints	Clear points from animated line
comet	2-D comet plot
comet3	3-D comet plot
drawnow	Update figures and process callbacks
refreshdata	Refresh data in graph when data source is specified

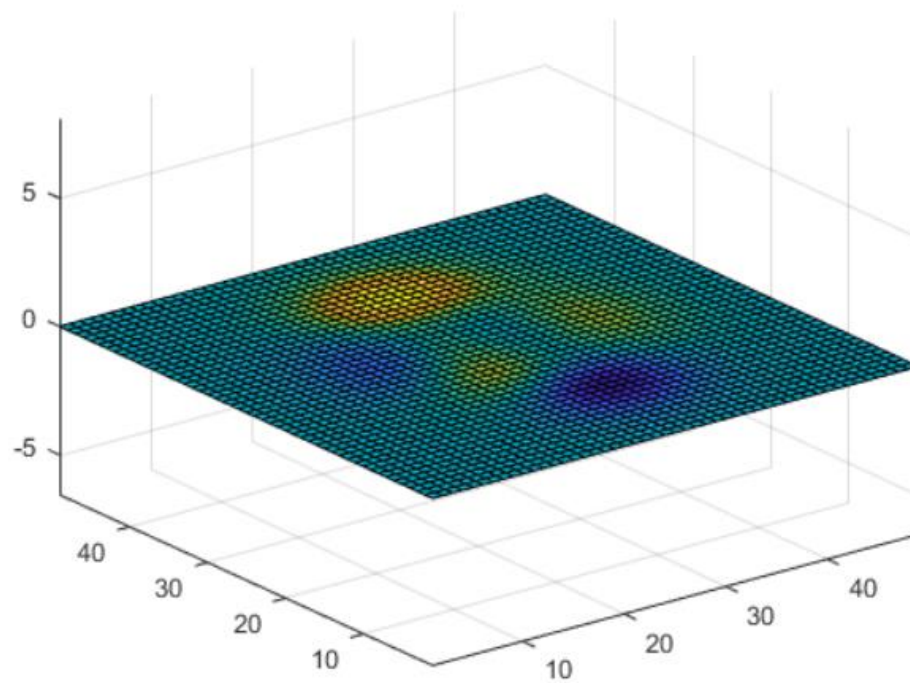
If we look at the movie function we see that it plays the movie defined by a matrix who's columns are movie frames (usually produces by "getframe"):

```

>> figure
Z = peaks;
surf(Z)
axis tight manual
ax = gca;
ax.NextPlot = 'replaceChildren';

loops = 40;
F(loops) = struct('cdata',[],'colormap',[]);
for j = 1:loops
    X = sin(j*pi/10)*Z;
    surf(X,Z)
    drawnow
    F(j) = getframe;
end

```



6. Loops/Indexing

Programming loops is a great way of cycling through and manipulating changes in your data. You use your loop control statements in order to repeatedly execute a block of code with either a "for" loop or a "while" loop. As soon as the statements for the loop are no longer satisfied, the loop will end.

A simple loop used to calculate five values:

```
>> x = ones(1,10);  
for n = 2:6  
    x(n) = 2 * x(n - 1);  
end
```

The above code would initialize x to an array of ten values all equal to one:

```
x =  
  
    1    1    1    1    1    1    1    1    1    1
```

From here, the loop begins where n is initialized to 2, and is expected to run until it reaches a value of 6. Therefore, the first iteration of the loop n = 2, the second iteration n = 3, then n = 4...until the last iteration where n = 6. This means the loop will execute 5 times. With each iteration the x array will save new calculated elements at each position in the array. For example:

$$x(2) = 2 * x(2-1) = 2$$

$$x(3) = 2 * x(3-1) = 4$$

$$x(4) = 2 * x(4-1) = 8$$

$$x(5) = 2 * x(5-1) = 16$$

$$x(6) = 2 * x(6-1) = 32$$

So now when you look at the x array, where you previously had only ones, you now have:

```
x =  
  
    1    2    4    8   16   32    1    1    1    1
```

You can also do the same calculations you do with loops with vectorization and indexing. Depending on if it's possible, this method can save you a lot of computational power:

```
>> i = 0;  
for t = 0:.01:10  
    i = i + 1;  
    y(i) = sin(t);  
end
```

Becomes:

```
>> t = 0:.01:10;  
y = sin(t);
```

Another loop you can use is a while loop. A while loop will continue to execute if the control statement holds true. For example, find the first integer "n" for which factorial(n) is a 100-digit number:

```
>> n = 1;  
nFactorial = 1;  
while nFactorial < 1e100  
    n = n + 1;  
    nFactorial = nFactorial * n;  
end
```

Once you feel comfortable using loops, you can try programming nested loops; that is loops within loops. This is most common when dealing with 2D and 3D spaces.

```
>> A = zeros(5,100);  
for m = 1:5  
    for n = 1:100  
        A(m, n) = 1/(m + n - 1);  
    end  
end
```

7. Creating your own function

Learning to create your own function is a very useful tool in that once a function is created and saved, you can minimize the amount of code written for a script. Functions are used in the same way you call an already created MATLAB function (example: `zeros()`), but in this case your input arguments, output arguments, and functionality is completely designed by you.

As a simple example on making a function to compute the factorial of a number:

```
function f = fact(n)  
    f = prod(1:n);  
end  
  
x = 5;  
y = fact(x)
```

In order to save and use functions, you need to either open a new script and save in the same folder you save the script you intend on working on, or include them at the end of a script file:

```
x = 3;
y = 2;
z = perm(x,y)

function p = perm(n,r)
    p = fact(n)*fact(n-r);
end

function f = fact(n)
    f = prod(1:n);
end
```

Syntax for various functions:

function keyword (required)	Use lowercase characters for the keyword.
Output arguments (optional)	<p>If your function returns one output, you can specify the output name after the function keyword.</p> <pre>function myOutput = myFunction(x)</pre> <p>If your function returns more than one output, enclose the output names in square brackets.</p> <pre>function [one,two,three] = myFunction(x)</pre> <p>If there is no output, you can omit it.</p> <pre>function myFunction(x)</pre> <p>Or you can use empty square brackets.</p> <pre>function [] = myFunction(x)</pre>
Function name (required)	<p>Valid function names follow the same rules as variable names. They must start with a letter, and can contain letters, digits, or underscores.</p> <p>Note</p> <p>To avoid confusion, use the same name for both the function file and the first function within the file. MATLAB associates your program with the <i>file</i> name, not the function name. Script files cannot have the same name as a function in the file.</p>
Input arguments (optional)	<p>If your function accepts any inputs, enclose their names in parentheses after the function name. Separate inputs with commas.</p> <pre>function y = myFunction(one,two,three)</pre> <p>If there are no inputs, you can omit the parentheses.</p>

Here is an example of a function with multiple outputs:

```
function [m,s] = stat(x)
    n = length(x);
    m = sum(x)/n;
    s = sqrt(sum((x-m).^2/n));
end

values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat(values)

ave = 47.3400
stdev = 29.4124
```

Another option is to call a function within another function:

```
function [m,s] = stat2(x)
    n = length(x);
    m = avg(x,n);
    s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
    m = sum(x)/n;
end
```

Now if you are getting more comfortable with functions, you can try creating a recursive function. That is a function calling itself in order to compute something:

```
function [ Result ] = Factorial2( Value )
%Factorial2 - Calculates the value of n!
% Outputs the factorial value of the input number.
if Value > 1
    Result = Factorial2(Value - 1) * Value;
else
    Result = 1;
end
end
```

8. Examples

Now that we have some MATLAB basics behind us, let's go through some practical examples, starting from simple to more complicated ones.

EXAMPLE 1:

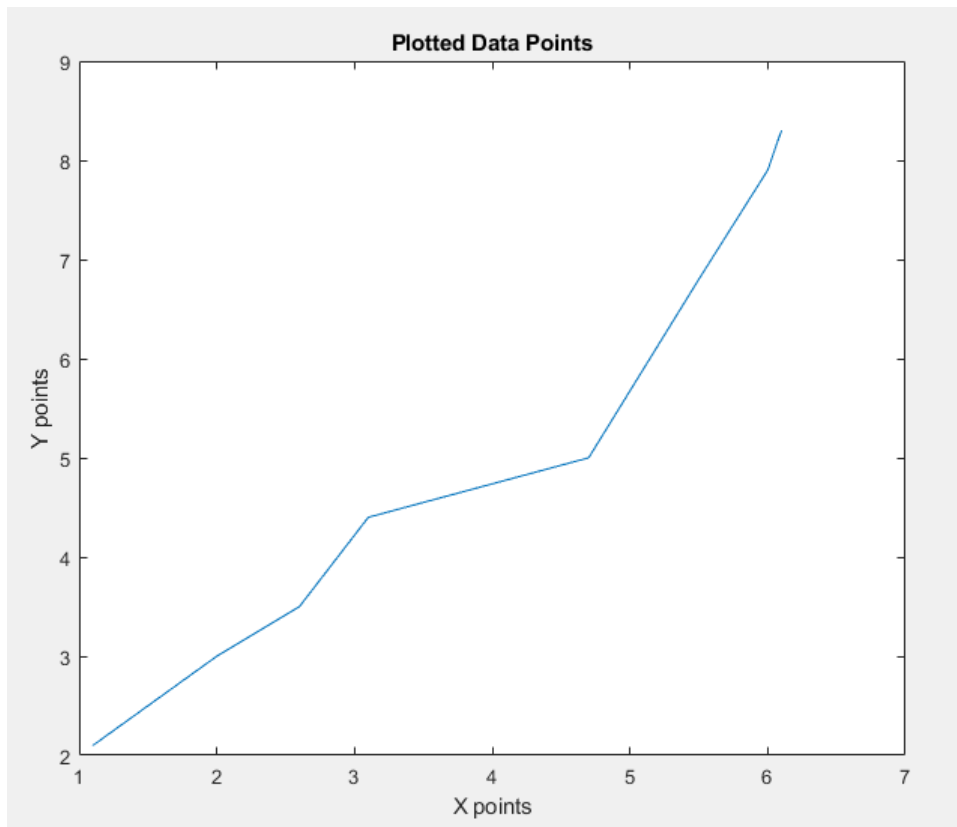
Let's start by analyzing two data arrays of x and y positions, and use MATLAB functions to find their linear equations, and finally make a function to calculate the associate error.

```
%let's produce two arrays and call them x and y:
x = [1.1 2.0 2.6 3.1 4.7 5.5 6.0 6.1];
y = [2.1 3.0 3.5 4.4 5.0 6.8 7.9 8.3];

%now let's perform some statistics on each data set and plot their outputs:
x_max = max(x); %returns 6
y_max = max(y); %returns 9.8

x_min = min(x); %returns 1.1
y_min = min(y); %returns 2.1

%now we can plot the data and find its the maximum point and minimum point
figure
plot(x,y)
title("Plotted Data Points");
xlabel("X points");
ylabel("Y points");
```




```

%the find() functions returns the indices of the max element in the y
%array. In this case the max element is 9.8, which is at position 6 in the
%above array. Therefore maxPoint will be 6.
maxPoint = find(max(y)==y)

%now that we know the max y value to be at the 6th position in the array,
%we find what the values are at this position in both the x and y array to
%find the point.
maxPoint_x = x(maxPoint); %maxPoint_x = 5.5 at the 6th position
maxPoint_y = y(maxPoint); % maxPoint_y = 9.8 at 6th position

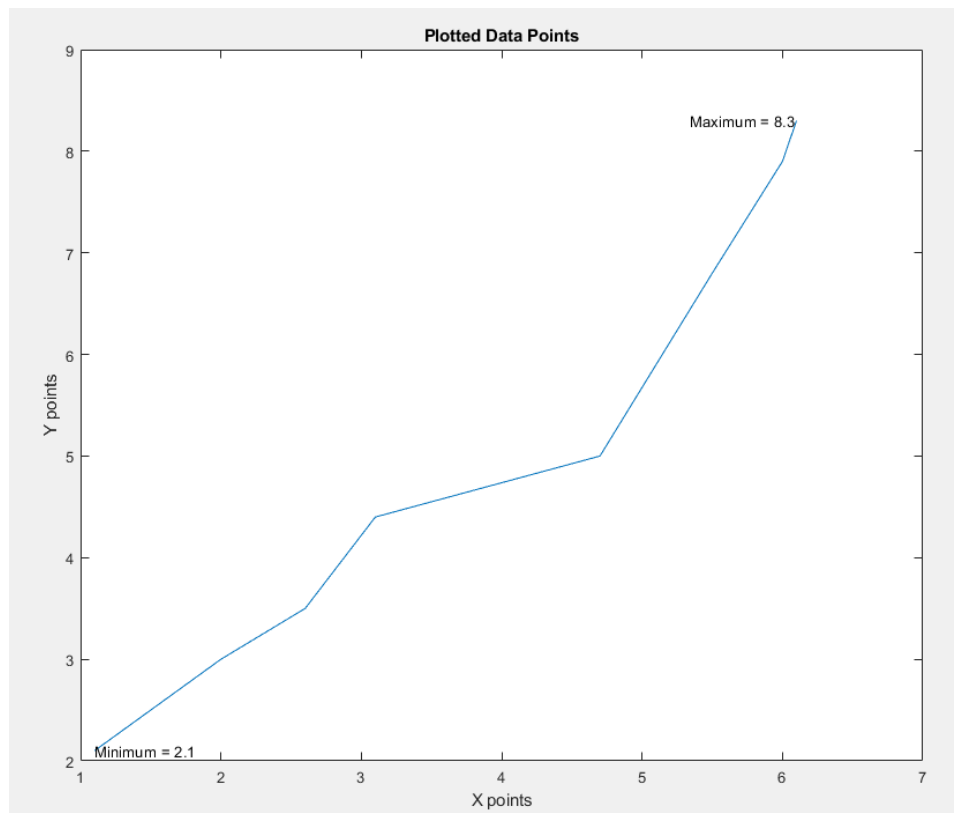
%now lets do this for the minimum point:
minPoint = find(min(y)==y);
minPoint_x = x(minPoint);
minPoint_y = y(minPoint);

%now that we have the maximum and minimum points, let's display them on
%our plot:

strmin = ['Minimum = ',num2str(minPoint_y)];
text(minPoint_x,minPoint_y,strmin,'HorizontalAlignment','left');

strmax = ['Maximum = ',num2str(maxPoint_y)];
text(maxPoint_x,maxPoint_y,strmax,'HorizontalAlignment','right');

```



```

%now let's add a linear fit to the data:
figure
scatter(x,y);
lsline
title("Scatter Plot with Linear Regression From Data Points");
xlabel("X points");
ylabel("Y points");

%now that you can see the linear fit with the use of "lsline" let's use the
%polyfit function to calculate the equation for the linear fit:
%if you remember the equation y = mx+b, polyfit works such that you enter
%in your x and y point, and the degree of the curve. In this case, since
%the data seems to be producing a linear fit, we say that the curve is of
%order 1. Polyfit then produces the coefficients for the line equation in
%the order of:
          %x^1*C1 + X^0*C2
%if in the event you were dealing with a quadratic equation you could say
%this:
          %polyfit(x,y,2), and the coefficients would save in the
          %order:
          %x^2*C1 + X^1*C2 + X^0*C3|

p = polyfit(x,y,1);
f1= @(x) p(1) + p(2);

```

Now that we have a arrays and linear equation, making a function for the percent array or our data, we first need to understand this equation:

$$ERR - a = SLOPE \cdot ERROR = S * \sqrt{\frac{n}{(n \sum x_i^2) - (\sum x_i)^2}}$$

$$ERR - b = INTERCEPT \cdot ERROR = S * \sqrt{\frac{\sum x_i^2}{(n \sum x_i^2) - (\sum x_i)^2}}$$

Where n = the number of points

xi = each element in array x

And S = the square root of the quantity found by dividing the sum of the squares of the deviations from the best fit line:

$$S = \sqrt{\frac{\sum (y_i - ax_i - b)^2}{n - 2}}$$

And a and b are coefficients of the linear fit line equations.

```

p = polyfit(x,y,1);
f1= @(x) p(1) + p(2);
[InterceptError, SlopeError] = PercentError(x,y,p);

%now the relative error would therefore be:
relative_slope_error = p(1)*SlopeError;
relative_intercept_error = p(2)*InterceptError;

function [InterceptError, SlopeError] = PercentError(x,y, p)

    %returns the length of array x, which in this case is 8.
    n = length(x);
    a = p(1); %returns the first coefficient in the linear equation
    b = p(2); %returns the second coefficient in the linear equation

    S = sqrt( (sum(y - a*x - b)^2)/(n-2) );
    InterceptError = S*sqrt( sum(x.^2)/[ n*sum(x.^2) - sum(x)^2 ] ) * b;
    SlopeError = S*sqrt( n/[ n*sum(x.^2) - sum(x)^2 ] ) * a;

end

```

EXAMPLE 2:

As you saw from the example above we were able to declare arrays, use functions, and create our own function and implement it with our data. Now we are going to increase the complexity of the example to include loops, structures, and matrices.

In this example we will be defining a class of students with grades in letter and percent, and attendance. We will calculate whether or not students will pass their course based on not only passing every class but also not missing more than 25% of their classes.

This script will be complicated as there will be multiple functions, and conditional statements. We will make it so there is an already defined array of students, but after running you have the option to include additional students into the class:

```
class.students = {'john'; 'sarah'; 'bill'; 'charlie'; 'sophie'};
class.grades = [93; 72; 51; 80; 49.5];
```

```
%we will have to look at attendance for a semester of classes. Typically a
%class lecture is twice a week, and there are 16 weeks a semester.
%Therefore there needs to be 32 attendance columns, corresponding to the
%date of the class.
```

```
NumberOfStudents = length(strlength(class.students));
```

```
for i = 1:NumberOfStudents
    for j = 1:1:32;
        if randi([1,10]) <= 7
            class.attendance(i,j) = 'P';
        else
            class.attendance(i,j) = 'A';
        end
    end
end
class.attendance = char(class.attendance);
AverageAttendance = mean(class.attendance,2);

%now to define a students letter grade:
%{
    remembering these conditions:
                                85 > A
                                73-85 = B
                                67-72 = C+
                                60-66 = C
                                50-59 = C-
                                50 < F
%}
for i = 1: NumberOfStudents
    if class.grades(i) > 85
        class.letterGrade(i) = 'A';

    elseif class.grades(i) > 72 & class.grades(i) < 86
        class.letterGrade(i) = 'B';

    elseif class.grades(i) > 66 & class.grades(i) < 73
        class.letterGrade(i) = 'C+';

    elseif class.grades(i) > 59 & class.grades(i) < 67
        class.letterGrade(i) = 'C';

    elseif class.grades(i) > 50 & class.grades(i) < 60
        class.letterGrade(i) = 'C-';
    else
        class.letterGrade(i) = 'F';
    end
end
Grades = class.grades;
LetterGrade = class.letterGrade;
```

```

%If students miss more than 25% of their classes that semester they
%cannot fully pass their course. Let's make calculations based on the
%attendance to see if everyone gets to pass:
for i= 1: NumberOfStudents
    if AverageAttendance(i) > 74 & LetterGrade(i) ~= 'F'
        class.Pass(i) = 'P';
    else
        class.Pass(i) = 'F';
    end
end
PassOrFail = class.Pass';

%prints out the table of information
T = table(Grades, AverageAttendance, LetterGrade, PassOrFail, ...
    'VariableNames',{'Grades' 'AverageAttendance' 'LetterGrade' 'PassFail'}, ...
    'RowNames', class.students)

```

This example so far gives us:

T =

5×4 [table](#)

	Grades	AverageAttendance	LetterGrade	PassFail
john	93	76.25	"A"	P
sarah	72	76.25	"C+"	P
bill	51	72.5	"C-"	F
charlie	80	77.188	"B"	P
sophie	49.5	74.375	"F"	F

Where all this information is saved in our class struct:

{}	students	5x1 cell
	grades	[93;72;51;80;49.5000]
c/h	attendance	5x32 char
str	letterGrade	5x1 string
c/h	Pass	'PPFPF'

Finally, now that we have started this class. We will change the structure to add the functions that give the user the option to add/drop students with their grades etc:

```

%initializing the students with their grades:
class.students = {'john'; 'sarah'; 'bill'; 'charlie'; 'sophie'};
class.grades = [93; 72; 51; 80; 49.5];

%we will have to look at attendance for a semester of classes. Typically a
%class lecture is twice a week, and there are 16 weeks a semester.
%Therefore there needs to be 32 attendance columns, corresponding to the
%date of the class.
NumberOfStudents = length(strlength(class.students));
class.attendance = zeros(NumberOfStudents,32);
for i = 1:NumberOfStudents
    class.attendance = char(class.attendance);
    AverageAttendance = mean(class.attendance,2);

%now to define a students letter grade:
class.letterGrade = zeros(NumberOfStudents,1);
for i = 1: NumberOfStudents
    letterGrade = percent(class.grades(i));
    class.letterGrade(i) = letterGrade;
end
class.letterGrade = char(class.letterGrade);
Grades = class.grades;

%If students miss more than 25% of their classes that semester they
%cannot fully pass their course. Let's make calulations based on the
%attendnace to see if everyone gets to pass:
class.Pass = zeros(NumberOfStudents,1);
for i= 1: NumberOfStudents
    if AverageAttendance(i) > 74 & class.letterGrade(i) ~= 'F'
        class.Pass(i) = 'P';
    else
        class.Pass(i) = 'F';
    end
end
PassOrFail = char(class.Pass);

%prints out the table of information
T = table(Grades, AverageAttendance, class.letterGrade, PassOrFail, ...
    'VariableNames',{'Grades' 'AverageAttendance' 'LetterGrade' 'PassFail'}, ...
    'RowNames', class.students)

%finially, if you wish to add students to the class:
Yes_No = input('Do you want to add another student?(Y/N): ','s') ;

%this loop enters into the functions designed to add students to the class:
while Yes_No == 'y' || Yes_No == 'Y'
    class = addStudents(class);
    Yes_No = input('Do you want to add another student?(Y/N): ', 's') ;
end

```

```

%this will add students to the class name
function class = addStudents (class)

    name = input('Whats the name of the student: ', 's');
    class.students = [class.students; {name}] ;
    class = addAttendance(class);
    class = addGrades(class);
    NumberOfStudents = length(strlength(class.students));

    for j = 1:1:32;
        if randi([1,10]) <=7
            class.attendance(NumberOfStudents,j) = 'P';
        else
            class.attendance(NumberOfStudents,j) = 'A';
        end
    end

    class.attendance = char(class.attendance);
    AverageAttendance = mean(class.attendance,2);

    class.letterGrade = zeros(NumberOfStudents,1);
    for i = 1: NumberOfStudents
        letterGrade = percent(class.grades(i));
        class.letterGrade(i) = letterGrade;
    end
    class.letterGrade = char(class.letterGrade);
    Grades = class.grades;

    if AverageAttendance(length(strlength(class.students))) > 74 & class.letterGrade(length(strlength(class.students))) ~= 'F'
        class.Pass(i) = 'P';
    else
        class.Pass(i) = 'F';
    end
    PassOrFail = char(class.Pass);

    %prints out the table of information
    T = table(Grades, AverageAttendance, class.letterGrade, PassOrFail, ...
        'VariableNames',{'Grades' 'AverageAttendance' 'LetterGrade' 'PassFail'}, ...
        'RowNames', class.students)
end

```

9. Debugging

10. Help

Something MATLAB offers for its users which other programming languages don't, is a hugely detailed database full of instructions, tutorials, and examples of how to write in MATLAB. One of the ways to understand how to do something when programming, if for example you didn't know how a specific function worked, is going to the MATLAB prompt and typing "help" and then the function you are confused about right next to it:

```
>> help ones
ones    Ones array.
       ones(N) is an N-by-N matrix of ones.

       ones(M,N) or ones([M,N]) is an M-by-N matrix of ones.

       ones(M,N,P,...) or ones([M N P ...]) is an M-by-N-by-P-by-... array of
       ones.

       ones(SIZE(A)) is the same size as A and all ones.

       ones with no arguments is the scalar 1.

       ones(..., CLASSNAME) is an array of ones of class specified by the
       string CLASSNAME.

       ones(..., 'like', Y) is an array of ones with the same data type, sparsity,
       and complexity (real or complex) as the numeric variable Y.

       Note: The size inputs M, N, and P... should be nonnegative integers.
       Negative integers are treated as 0.

       Example:
           x = ones(2,3,'int8');

       See also eye, zeros.

       Reference page for ones
       Other functions named ones
```

As you can see MATLAB outputs the specifics for the function including a link if you inquired to understand other functions similar to the typed in function.

Documentation ▾

