

# Matlab Bootcamp

## Recap

### Numeric Arrays



- e.g. list of phone numbers
- `B = [4670456, 7640600, 123456];`

### Matrices

- e.g. list of GPS coordinates
- `A = [42.3086,-83.6921; 82.208, 23.692];`

### Cell Arrays

- e.g. list of names
- `C = {'Duck','Goose','Crane'};`

## Matrix Manipulation

### Adding a scalar to an array

```
A = [1;2;3;4];  
A + 3
```

```
ans = 4x1  
    4  
    5  
    6  
    7
```

### Adding two arrays

```
B = [3;4;5;6];  
A + B
```

```
ans = 4x1  
    4  
    6  
    8  
   10
```

### Element-wise multiplication

```
A .* B
```

```
ans = 4x1  
    3  
    8  
   15  
   24
```

### Matrix multiplication

```
C = [5 10 15 20];
```

A\*C

```
ans = 4x4
      5      10      15      20
     10      20      30      40
     15      30      45      60
     20      40      60      80
```

## Organizing variables into structures

### Structures

Structure arrays can be used to group related data together.

```
x.age = [10,15,20];
x.heartrate = [62,71,84];
x.weight = [78,122,159];
x
```

```
x = struct with fields:
    age: [10 15 20]
heartrate: [62 71 84]
    weight: [78 122 159]
```

### Tables

Tables are used for storing data in rows and column-oriented variables.

- Tables can contain different data types, such as strings and doubles
- Each variable in the table must have the same number of rows

Use *readtable* to import example patient data and then *summary* to examine its contents:

```
T=readtable('patients.dat');
summary(T)
```

Variables:

**LastName:** 100×1 cell array of character vectors

**Gender:** 100×1 cell array of character vectors

**Age:** 100×1 double

Values:

Min	25
Median	39
Max	50

**Location:** 100×1 cell array of character vectors

**Height:** 100×1 double

Values:

```

Min      60
Median   67
Max      72

```

**Weight:** 100×1 double

Values:

```

Min      111
Median   142.5
Max      202

```

**Smoker:** 100×1 double

Values:

```

Min      0
Median   0
Max      1

```

**Systolic:** 100×1 double

Values:

```

Min      109
Median   122
Max      138

```

**Diastolic:** 100×1 double

Values:

```

Min      68
Median   81.5
Max      99

```

**SelfAssessedHealthStatus:** 100×1 cell array of character vectors

Display data for the first four patients:

```
T(1:4,:)
```

```
ans = 4×10 table
```

...

	LastName	Gender	Age	Location	Height	Weight	Smoker	Systolic
1	'Smith'	'Male'	38	'County Ge...	71	176	1	124
2	'Johnson'	'Male'	43	'VA Hospital'	69	163	0	109
3	'Williams'	'Female'	38	'St. Mary'...	64	131	0	125
4	'Jones'	'Female'	40	'VA Hospital'	67	133	0	117

Now create a table that only includes the patient age, height and weight:

```

T2=table(T.Age,T.Height,T.Weight);
T2.Properties.VariableNames={'Age','Height','Weight'}

```

```
T2 = 100×3 table
```

	Age	Height	Weight
1	38	71	176
2	43	69	163
3	38	64	131
4	40	67	133
5	49	64	119
6	46	68	142
7	33	64	142
8	40	68	180
9	28	68	183
10	31	66	132

⋮

## Framington Heart Disease Dataset

This dataset comes from a landmark study that analyzed ~14,000 people from three generations. The findings have informed the understanding of factors that impact cardiovascular health.

Import the dataset and determine its size:

```
framingham = readtable('frmgham2.xls');
size(framingham)
```

```
ans = 1x2
      11627      39
```

The output of `size` indicates that the dataset includes 11,627 rows and 39 columns. Display the names of the 39 columns:

```
framingham.Properties.VariableNames
```

```
ans = 1x39 cell array
      {'RANDID'}      {'SEX'}      {'TOTCHOL'}      {'AGE'}      {'SYSBP'}      {'DIABP'}      {'CURSMOKE'}      {'CIGPDAY'}
```

## Extracting Simple Properties

Determine the mean, median and range of BMI:

```
BMI=framingham.BMI;
mean(BMI)
```

```
ans = NaN
```

```
median(BMI)
```

```
ans = NaN
```

```
range(BMI)
```

```
ans = 42.3700
```

The mean and median functions return *NaN* because there is data missing from the BMI array. MATLAB also has functions that ignore these missing data points:

```
nanmean(BMI)
```

```
ans = 25.8773
```

```
nanmedian(BMI)
```

```
ans = 25.4800
```

We will now look at extracting data from the framingham data set based on certain criteria. For example, what if we only want to examine individuals who smoke?

```
rows = framingham.CURSMOKE==1;  
T = framingham(rows,:)
```

```
T = 5029×39 table
```

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	9428	1	245	48	127.5000	80.0000	1	20
2	9428	1	283	54	141.0000	89.0000	1	30
3	10552	2	225	61	150.0000	95.0000	1	30
4	10552	2	232	67	183.0000	109.0000	1	20
5	11252	2	285	46	130.0000	84.0000	1	23
6	11252	2	343	51	109.0000	77.0000	1	30
7	11252	2	NaN	58	155.0000	90.0000	1	30
8	12806	2	313	45	100.0000	71.0000	1	20
9	12806	2	NaN	51	109.5000	72.5000	1	30
10	12806	2	320	57	110.0000	46.0000	1	30

⋮

Similarly, we can pick out individuals with a BMI above 30:

```
rows2 = framingham.BMI>35;  
T2 = framingham(rows2,:)
```

```
T2 = 318×39 table
```

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	12629	2	220	70	149.0000	81.0000	0	0
2	43522	2	NaN	55	129.0000	76.0000	0	0

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
3	82188	1	225	37	124.5000	92.5000	0	0
4	82188	1	244	43	156.0000	109.0000	0	0
5	82188	1	226	49	190.0000	123.0000	0	0
6	83398	1	178	52	160.0000	98.0000	0	0
7	83398	1	155	58	173.0000	90.0000	0	0
8	83398	1	NaN	64	205.0000	90.0000	0	0
9	174973	2	206	42	130.0000	80.0000	1	3
10	174973	2	208	48	122.0000	74.0000	1	3

⋮

This selection criteria method can also be applied to multiple variables at once:

```
rows3 = framingham.CURSMOKE==1 & framingham.BMI>35;
T3 = framingham(rows3,:)
```

T3 = 84×39 table

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	174973	2	206	42	130.0000	80.0000	1	3
2	174973	2	208	48	122.0000	74.0000	1	3
3	202101	2	326	61	200.0000	104.0000	1	1
4	610021	2	180	60	200.0000	122.5000	1	20
5	935116	1	229	44	177.5000	120.0000	1	10
6	968222	1	153	56	182.0000	95.0000	1	1
7	977985	2	268	48	117.5000	80.0000	1	10
8	977985	2	247	60	164.0000	104.0000	1	10
9	1186959	2	NaN	73	200.0000	100.0000	1	2
10	1225217	2	233	55	128.0000	94.0000	1	20

⋮

By scrolling through the CURSMOKE and BMI columns, we can confirm that the new table only includes data for individuals that both smoked and have a BMI above 35. Now create a table which only includes the age, cholesterol and heartrate for these individuals:

```
vars = {'AGE', 'TOTCHOL', 'HEARTRTE'};
T4 = framingham(rows3,vars)
```

T4 = 84×3 table

	AGE	TOTCHOL	HEARTRTE
1	42	206	70

	AGE	TOTCHOL	HEARTRTE
2	48	208	75
3	61	326	57
4	60	180	88
5	44	229	104
6	56	153	75
7	48	268	72
8	60	247	78
9	73	NaN	NaN
10	55	233	80

⋮

## Manipulating Datasets

As seen earlier in the BMI array, there are missing values throughout the dataset, which are signified by *NaN*. It is important to know how to both find and replace these missing values.

We can use the *find* and *isnan* functions to determine the rows where data is missing:

```
rows=find(isnan(BMI))
```

```
rows = 52x1
      2
     265
     347
     433
     795
    1216
    1300
    1527
    1894
    2631
      ⋮
```

Another option is to use the *ismissing* function, which creates an array of the same size that is being searched. It fills in the corresponding elements of the array with zeros, where data is found, and ones, where data is missing.

```
ismissing(BMI)
```

```
ans = 11627x1 logical array
      0
      1
      0
      0
      0
      0
      0
      0
      0
      0
      0
      0
```

⋮

Now we want to remove the rows in the BMI array with missing data:

```
BMI(find(isnan(BMI)))=[];
```

To confirm that this worked, we can check that the size of the BMI array has decreased and that the *mean* function now works:

```
mean(BMI)
```

```
ans = 25.8773
```

Now that the missing data points have been removed, sort the BMI array:

```
sort(BMI)
```

```
ans = 11575x1
    14.4300
    14.5300
    15.1600
    15.3200
    15.3300
    15.5400
    15.6400
    15.9200
    15.9600
    15.9600
    ⋮
    ⋮
```

We see that the default setting is to sort in ascending order, however there are multiple ways to use *sort*.

```
sort(BMI, 'descend')
```

```
ans = 11575x1
    56.8000
    56.8000
    56.8000
    55.3100
    52.9400
    51.2800
    48.6400
    47.2200
    46.5200
    45.9800
    ⋮
    ⋮
```

Next, let's look at the outliers in the BMI array:

```
[TF,L,U,C] = isoutlier(BMI)
```

```
TF = 11575x1 logical array
    0
    0
    0
    0
    0
```



•

•

•

Similar to the *ismissing* function, *isoutlier* creates a new array, *TF*, and identify outlier points with ones. We can also see the lower and upper thresholds and the center value used to determine the outliers (variables L, U, and C).

We can also add new data to our table. Add the *DIABETES* data to table *T4*:

```
T4.DIABETES = framingham.DIABETES(rows3)
```

```
T4 = 84x4 table
```

	AGE	TOTCHOL	HEARTRTE	DIABETES
1	42	206	70	0
2	48	208	75	0
3	61	326	57	0
4	60	180	88	1
5	44	229	104	0
6	56	153	75	1
7	48	268	72	0
8	60	247	78	0
9	73	NaN	NaN	0
10	55	233	80	0
⋮				

## Simple Analyses of the Dataset



## Importing and Exporting Datasets

## Importing

As of MATLAB 2019, the *readmatrix* function is recommended for importing datasets as opposed to previously used functions *csvread*, *xlsread*, and *dlmread*. *readmatrix* can be used with many different file types, such as .txt, .dat, .csv, and .xls. However, only numeric data can be imported.

```
M = readmatrix('csv1.csv')
```

$$M = 5 \times 2$$

```

1      6
2      7
3      8
4      9
5     10

```

If you wish to create a table instead of an array, the *readtable* function should be used:

```
T = readtable("csv2.csv")
```

```
T = 5x2 table
```

	Numbers	Letters
1	1	'A'
2	2	'B'
3	3	'C'
4	4	'D'
5	5	'E'

## Exporting

Similar to importing, the functions *xlswrite*, *dlmwrite*, and *csvwrite* are no longer recommended in MATLAB 2019. Instead, you should use *writetable* or *writematrix*.

Create a matrix of random integers and write to a .xls file:

```
M = magic(4)
```



```
M = 4x4
```

```

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

```

```

writematrix(M,'M.xls')
readmatrix("M.xls")

```

```
ans = 4x4
```

```

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

```

Now we will create a table and use *writetable* to export it to a .txt file:

```

TEMP = [71;73;69;73;80];
WIND = [15;17;21;12;8];
RAIN = [0.05;0;0.14;0.02;0];
DAY = {'M';'T';'W';'Th';'F'};
T = table(TEMP,WIND,RAIN,'RowNames',DAY)

```

```
T = 5x3 table
```

	TEMP	WIND	RAIN
1 M	71	15	0.0500
2 T	73	17	0
3 W	69	21	0.1400
4 Th	73	12	0.0200
5 F	80	8	0

```
writetable(T, 'T.txt', 'WriteRowNames', true)
readtable('T.txt')
```

ans = 5×4 table

	Row	TEMP	WIND	RAIN
1	'M'	71	15	0.0500
2	'T'	73	17	0
3	'W'	69	21	0.1400
4	'Th'	73	12	0.0200
5	'F'	80	8	0

## Handling Big Datasets

### Head and Tail

The *head* and *tail* functions can be used to view the first and last rows of a table or array. The default for *heads* and *tails* is to display the first and last eight rows, respectively. However, the number of rows can be altered for each function. Consider the framingham dataset we used earlier:

```
first_rows = head(framingham, 5)
```

first\_rows = 8×39 table

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	2448	1	195	39	106.0000	70.0000	0	0
2	2448	1	209	52	121.0000	66.0000	0	0
3	6238	2	250	46	121.0000	81.0000	0	0
4	6238	2	260	52	105.0000	69.5000	0	0
5	6238	2	237	58	108.0000	66.0000	0	0
6	9428	1	245	48	127.5000	80.0000	1	20
7	9428	1	283	54	141.0000	89.0000	1	30
8	10552	2	225	61	150.0000	95.0000	1	30

```
last_rows = tail(framingham, 3)
```

last\_rows = 8×39 table

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	9995546	2	269	52	133.5000	83	0	0
2	9995546	2	265	58	140.0000	83	0	0
3	9998212	1	185	40	141.0000	98	0	0
4	9998212	1	173	46	126.0000	82	0	0
5	9998212	1	153	52	143.0000	89	0	0
6	9999312	2	196	39	133.0000	86	1	30
7	9999312	2	240	46	138.0000	79	1	20
8	9999312	2	NaN	50	147.0000	96	1	10

## Textscan

*textscan* can be used to read data from a text file or string and insert it into a cell array. Below, we use *textscan* to read the string *chr* and use ' %f ' to indicate that we want to output double-precision floating-point numbers.

```
chr = '0.41 8.24 3.57 6.24 9.27';
C = textscan(chr, '%f');
celldisp(C)
```

```
C{1} =
    0.4100
    8.2400
    3.5700
    6.2400
    9.2700
```

## Storing Big Datasets

### Datastore



Datastores allow you to work with large data sets in small pieces which fit into the memory, as opposed to loading the entire data set into the memory at once. Datastores also let you import and process data that is stored in multiple locations as a single database. However, each file in a datastore must contain the same type of data in the same order.

There are several different kind of datastores. For example, tabular text datastores are for text files containing column-oriented data, such as CSV files:

```
ds = tabularTextDatastore('airlinesmall.csv')
```

```
ds =
    TabularTextDatastore with properties:
        Files: {
            '/Applications/MATLAB_R2019a.app/toolbox/matlab/demos/airlinesmall.csv'
```

```

    }
    FileEncoding: 'UTF-8'
  AlternateFileSystemRoots: {}
  ReadVariableNames: true
  VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
  DatetimeLocale: en_US

Text Format Properties:
  NumHeaderLines: 0
  Delimiter: ',',
  RowDelimiter: '\r\n'
  TreatAsMissing: ''
  MissingValue: NaN

Advanced Text Format Properties:
  TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
  TextType: 'char'
  ExponentCharacters: 'eEdD'
  CommentStyle: ''
  Whitespace: ' \b\t'
  MultipleDelimitersAsOne: false

Properties that control the table returned by preview, read, readall:
  SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
  SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
  ReadSize: 20000 rows

```

```
preview(ds)
```

```
ans = 8×29 table
```

...

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime
1	1987	10	21	3	642	630	735
2	1987	10	26	1	1021	1020	1124
3	1987	10	23	5	2055	2035	2218
4	1987	10	23	5	1332	1320	1431
5	1987	10	22	4	629	630	746
6	1987	10	28	3	1446	1343	1547
7	1987	10	8	4	928	930	1052
8	1987	10	10	6	859	900	1134

We can use *SelectVariableNames* to specify the columns that we want to look at:

```
ds.SelectedVariableNames = {'DepTime', 'DepDelay'};
preview(ds)
```

```
ans = 8×2 table
```

	DepTime	DepDelay
1	642	12
2	1021	1
3	2055	20
4	1332	12

	DepTime	DepDelay
5	629	-1
6	1446	63
7	928	-2
8	859	-1

If all of the data in the datastore fit in memory, you can use the *readall* function. Otherwise, you must read the data in smaller sections which do fit in the memory. The default setting reads the datastore 20,000 rows at a time, but this can be adjusted:

```
ds.ReadSize = 10000;
```

```
ds =
    TabularTextDatastore with properties:

        Files: {
                '/Applications/MATLAB_R2019a.app/toolbox/matlab/demos/airlinesmall.csv'
            }
        FileEncoding: 'UTF-8'
    AlternateFileSystemRoots: {}
        ReadVariableNames: true
        VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
        DatetimeLocale: en_US

    Text Format Properties:
        NumHeaderLines: 0
        Delimiter: ','
        RowDelimiter: '\r\n'
        TreatAsMissing: ''
        MissingValue: NaN


    Advanced Text Format Properties:
        TextscanFormats: {'%*q', '%*q', '%*q' ... and 26 more}
        TextType: 'char'
        ExponentCharacters: 'eEdD'
        CommentStyle: ''
        Whitespace: ' \b\t'
        MultipleDelimitersAsOne: false

    Properties that control the table returned by preview, read, readall:
        SelectedVariableNames: {'DepTime', 'DepDelay'}
        SelectedFormats: {'%f', '%f'}
        ReadSize: 10000 rows
```



## Tall Arrays

Tall arrays are used in tandem with datastores by enabling you to work with out-of-memory data. Tall arrays use many of the same core functions as in-memory MATLAB arrays. MATLAB handles the large data sets in small pieces while the processing is done in the background. Unlike in-memory arrays, tall arrays remain unevaluated until the calculations are called for with the *gather* command, thereby speeding up your work with large datasets.

**Create a spreadsheet datastore that points to a tabular file of airline flight data. For folders that contain a collection of files, you can specify the entire folder location, or use the wildcard  character, '\*.\*.csv', to include multiple files with the same file extension in the datastore. Clean the data by treating 'NA'**

values as missing data so that `datastore` replaces them with `NaN` values. Also, set the format of a few text variables to `%s` so that `datastore` reads them as cell arrays of character vectors.

```
ds = datastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'TailNum')} = '%s';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'CancellationCode')} = '%s';
```

Create a tall table from the datastore. When you perform calculations on this tall table, the underlying datastore reads chunks of data and passes them to the tall table to process. Neither the datastore nor the tall table retain any of the underlying data.

```
tt = tall(ds)
```

When you extract a variable from a tall table or tall timetable, the result is a tall array of the appropriate underlying data type. A tall array can be a numeric, logical, datetime, duration, calendar duration, categorical, string, or cell array. Also, you can convert an in-memory array `A` into a tall array with `tA = tall(A)`. The in-memory array `A` must be one of the supported data types.

Extract the arrival delay `ArrDelay` from the tall timetable `TT`. This creates a new tall array variable with underlying data type double.

```
a = TT.ArrDelay
```

One important aspect of tall arrays is that as you work with them, most operations are not performed immediately. These operations appear to execute quickly, because the actual computation is deferred until you specifically request that the calculations be performed. You can trigger evaluation of a tall array with either the `gather` function (to bring the result into memory) or the `write` function (to write the result to disk). This deferred evaluation is important because even a simple command like `size(X)` executed on a tall array with a billion rows is not a quick calculation.

As you work with tall arrays, MATLAB keeps track of all of the operations to be carried out. This information is then used to optimize the number of passes through the data that will be required when you request output with the `gather` function. Thus, it is normal to work with unevaluated tall arrays and request output only when you require it. For more information, see [Deferred Evaluation of Tall Arrays](#).

Calculate the mean and standard deviation of the arrival delay. Use these values to construct the upper and lower thresholds for delays that are within one standard deviation of the mean. Notice that the result of each operation indicates that the array has not been calculated yet.

```
m = mean(a, 'omitnan')
s = std(a, 'omitnan')
one_sigma_bounds = [m-s m m+s]
```

The benefit of delayed evaluation is that when the time comes for MATLAB to perform the calculations, it is often possible to combine the operations in such a way that the number of passes through the data is minimized. So even if you perform many operations, MATLAB only makes extra passes through the data when absolutely necessary.

The `gather` function forces evaluation of all queued operations and brings the resulting output into memory. For this reason, you can think of `gather` as a bridge between tall arrays and in-memory arrays. For example, you cannot control `if` or `while` loops using a tall logical array, but once the array is evaluated with `gather` it becomes an in-memory logical array that you can use in these contexts.

```
sig1 = gather(one_sigma_bounds)
```

You can specify multiple inputs and outputs to `gather` if you want to evaluate several tall arrays at once. This technique is faster than calling `gather` multiple times. For example, calculate the minimum and maximum arrival delay. Computed separately, each value requires a pass through the data to calculate for a total of two passes. However, computing both values simultaneously requires only one pass through the data.

```
[max_delay, min_delay] = gather(max(a), min(a))
```

## Sparces Matrices

Sparse matrices are useful when your dataset is comprised of mostly zeros. As opposed to normal matrices which store every elemnt in the matrix, sparce matrices only store the nonzero elements along with their row indices. Sparce matrices therefor require much less memory for storage than full matrices.

The first step in creating a sparse matrix is determining the density on nonzero elements. The lower the density, the more it makes sense to create a sparce matrix. Consider the below matrix A:

```
A = [0 0 0 5; 0 2 0 0; 1 3 0 0; 0 0 4 0]
```

```
A = 4x4
    0     0     0     5
    0     2     0     0
    1     3     0     0
    0     0     4     0
```

```
nnz(A) / prod(size(A))
```

```
ans = 0.3125
```

```
S = sparse(A)
```

```
S =
(3,1)    1
(2,2)    2
(3,2)    3
(4,3)    4
(1,4)    5
```

The output of `sparse` shows the nonzero elements and their respective indeces, sorted by column. You can then convert the sparse matrix back into the full matrix with the `full` command:

```
A = full(S)
```

```
A = 4x4
    0     0     0     5
    0     2     0     0
```



1	3	0	0
0	0	4	0

It is also possible to create a sparse matrix directly from the nonzero elements, without needing the full matrix:



$S = \text{sparse}(i, j, s, m, n);$

- $i, j$  = row and column indices, respectively
- $s$  = vector of nonzero values with indices  $i, j$
- $m, n$  = row and column dimensions of resulting matrix, respectively

```
S = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)
```

```
S =
(3,1)      1
(2,2)      2
(3,2)      3
(4,3)      4
(1,4)      5
```



## Plotting Datasets

We will quickly run through some of the plotting options in MATLAB, starting with a **2-D line plot**:

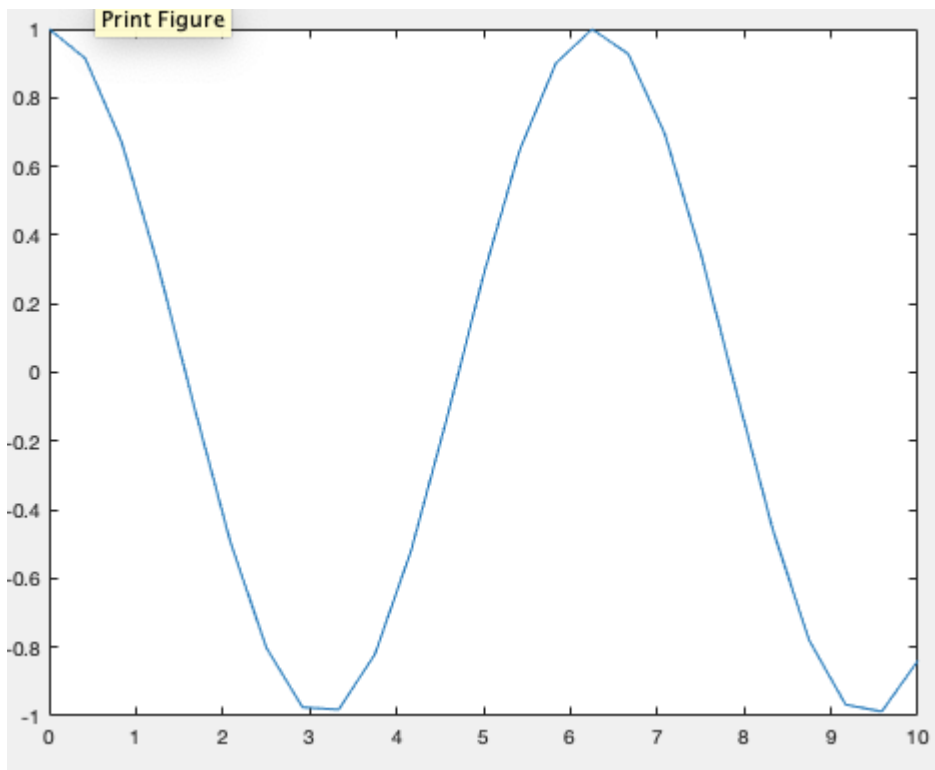
```
x = linspace(0,10,25);
y = cos(x);
```

After creating these variables, you can select them in the Workspace, go to Plots tab, and select the 2-D line plot.



MATLAB will then create the plot for you and display the plotting command in the command line:

```
plot(x,y)
```



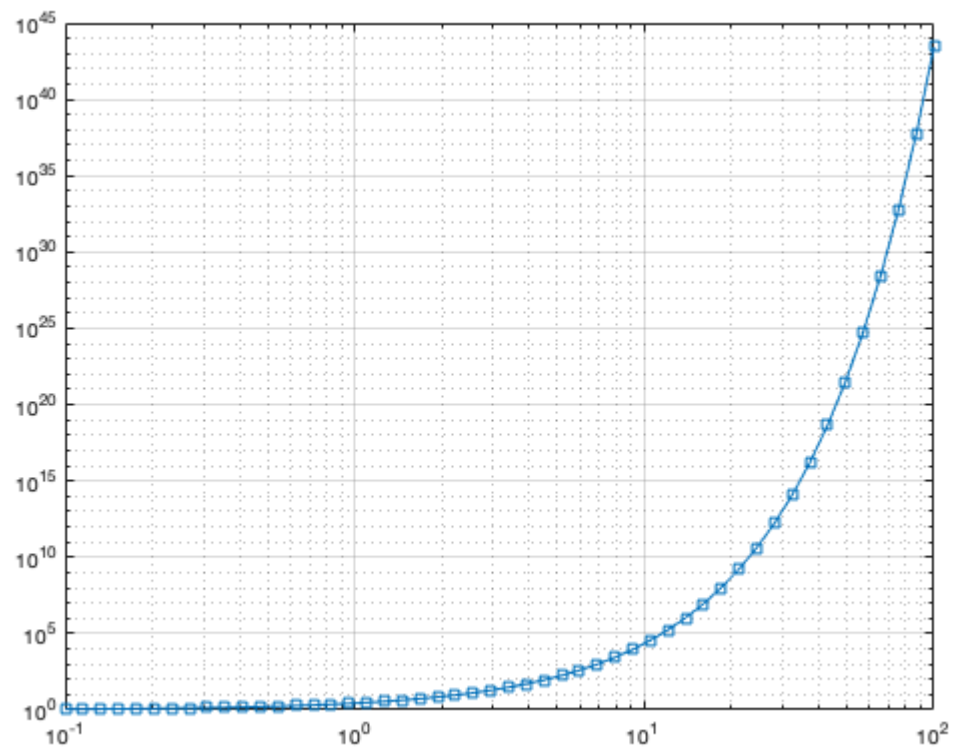
Once the figure has been created, you can use the Insert tab to add elements such as axis labels and titles. When you are done creating your plot, go to the File tab to generate the code.

If you do not wish to use the Plots tab, you can directly make the plots in your script with the appropriate command.

### Log-Log

```
l1 = logspace(-1,2);  
l2 = exp(l1);  
loglog(l1,l2,'-s')  
grid on
```

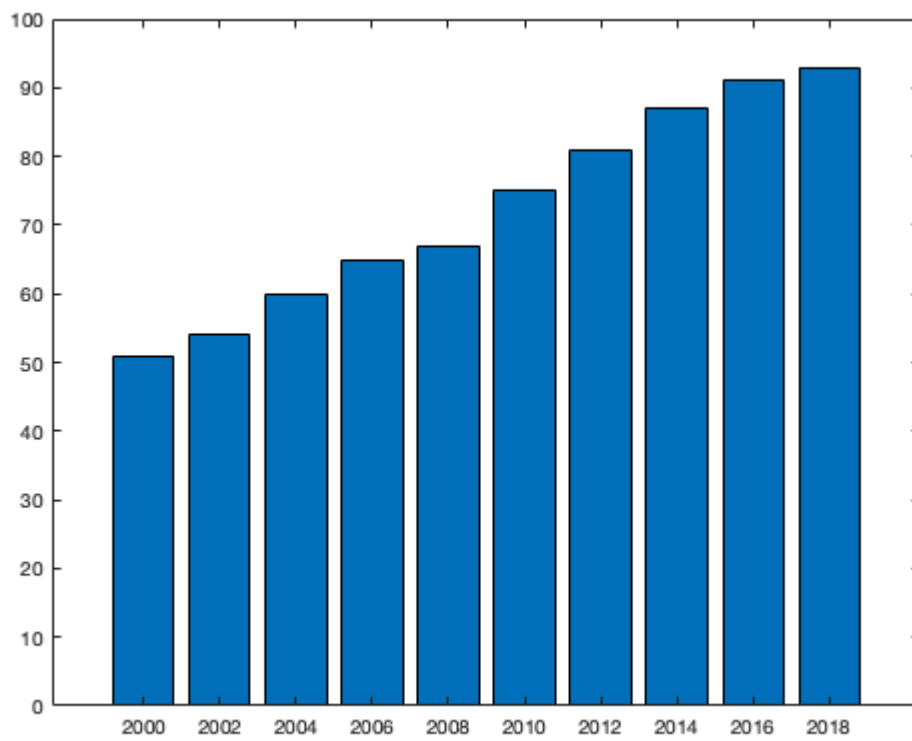




## Bar Graphs

```
x = 2000:2:2018;  
y = [51 54 60 65 67 75 81 87 91 93];  
bar(x,y)
```

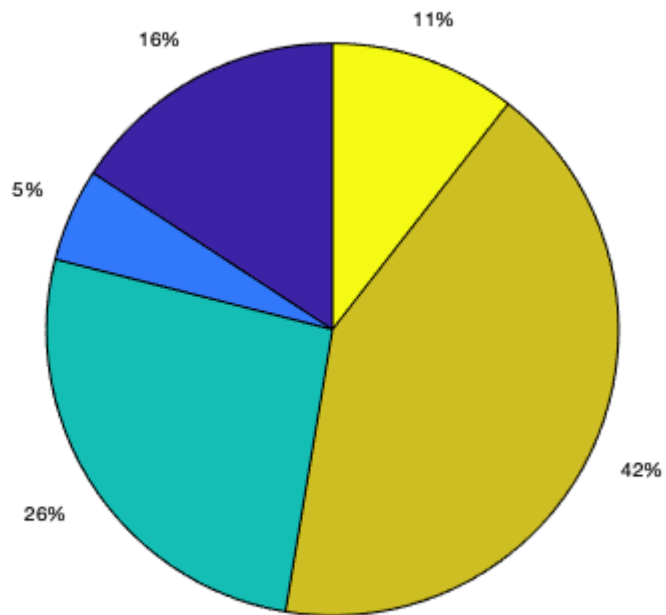




### Pie Chart

```
p = [3 1 5 8 2];  
pie(p)
```



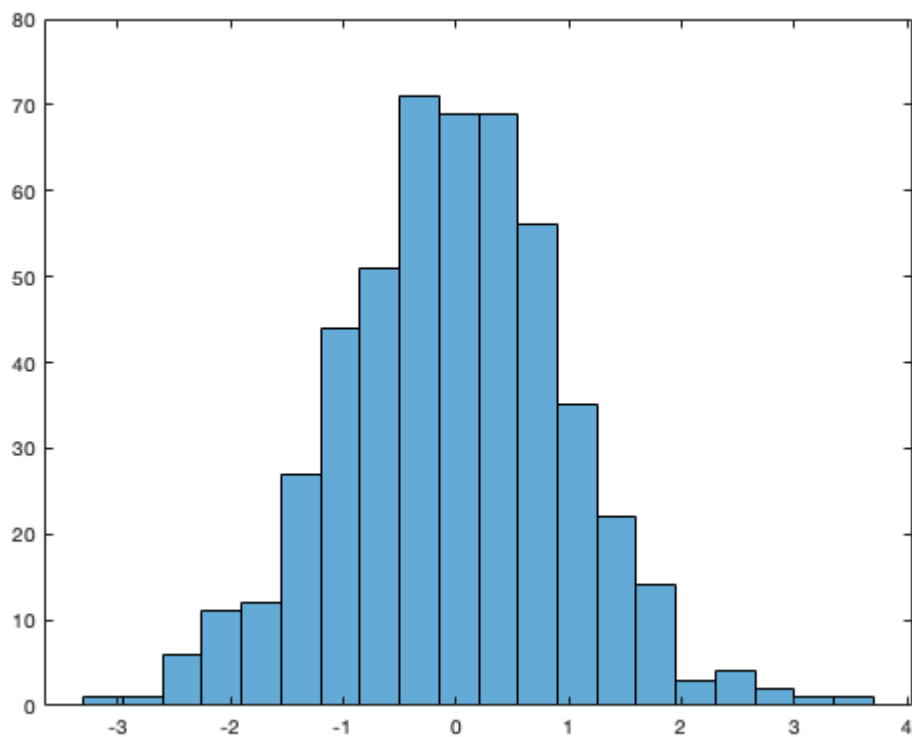


## Histograms

Plot a histogram of 500 random numbers into 20 equally spaced bins:

```
x = randn(500,1);  
nbins = 20;  
h = histogram(x,nbins)
```





h =

Histogram with properties:

```

    Data: [500x1 double]
    Values: [1 1 6 11 12 27 44 51 71 69 69 56 35 22 14 3 4 2 1 1]
    NumBins: 20
    BinEdges: [-3.3000 -2.9500 -2.6000 -2.2500 -1.9000 -1.5500 -1.2000 -0.8500 -0.5000 -0.1500 0.2000]
    BinWidth: 0.3500
    BinLimits: [-3.3000 3.7000]
    Normalization: 'count'
    FaceColor: 'auto'
    EdgeColor: [0 0 0]

```

Show all properties

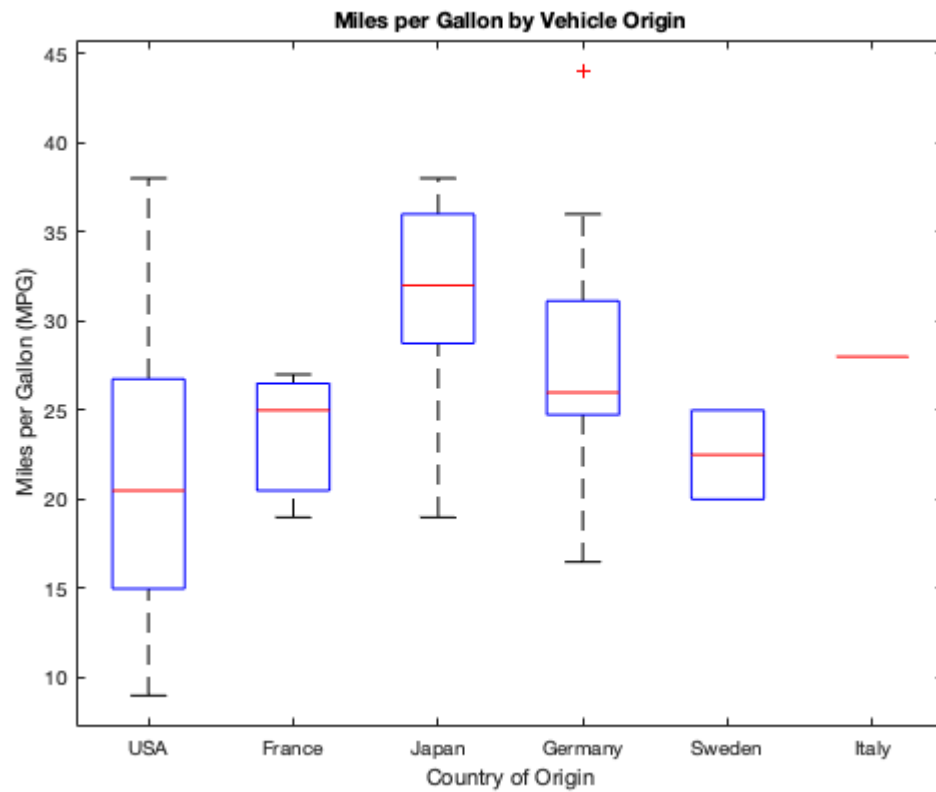
## Boxplots

```

load carsmall
boxplot(MPG,Origin)
title('Miles per Gallon by Vehicle Origin')
xlabel('Country of Origin')
ylabel('Miles per Gallon (MPG)')

```

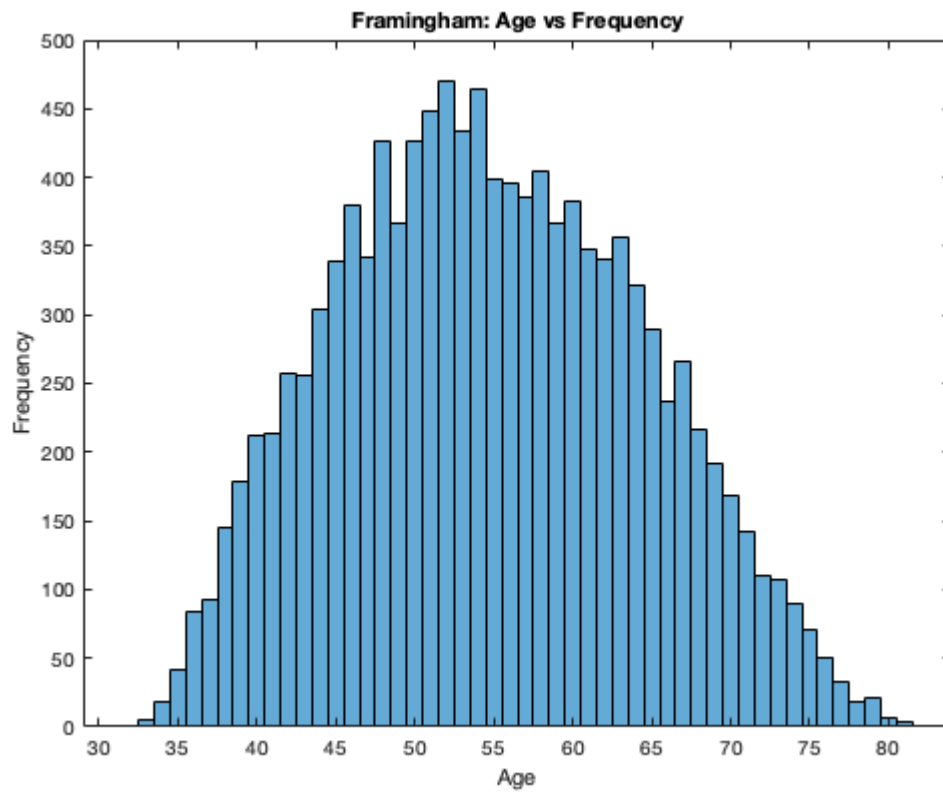




## Practice with Framingham Data

### Plot Age vs Frequency

```
age = framingham.AGE;  
histogram(age)  
  
title('Framingham: Age vs Frequency')  
xlabel('Age')  
ylabel('Frequency')
```



**Plot Education vs Income**



**Boxplot Race vs Heart rate**

**Distribution of Heart rate**

**Education vs BMI - what is best way to show relationship?**