Introduction to Computing
I - MATLAB

Jonathan Mascie-Taylor
(Slides originally by Quentin CAUDRON)

Centre for Complexity Science,
University of Warwick

Outline

Sessions

- Monday, 23 September, 13:00 - 17:00 (Laptops & MATLAB)
- Tuesday, 24 September, 09:00 - 12:30 (C)
- Wednesday, 25 September, 09:00 - 12:30 (C)
- Friday, 27 September, 10:00 - 17:00 (C & HPC)

Source Codes and Slides

These slides are written with the aim of being used as reference notes. Some points are single-word reminders, some are full paragraphs. Please make your own notes !

Slides will be uploaded onto `http://go.warwick.ac.uk/jonathanmascietaylor` under **Intro to Computing** after each session. Source code for the exercises done in the sessions will also be available.

This course was originally taught by Quentin CAUDRON who created the majority of the material.

About this Course

This course is aimed at those with absolutely no programming experience. If you have written code before, you will probably be bored. Please help your less knowledgeable course mates...

We will be covering introductions to MATLAB and C programming, with scientific goals in mind.

Course Content Overview

- Programming, compiling, languages
- Introductory C and MATLAB
- Brief intro to Linux environments and High-performance computing

Computation - Possibilities and Purpose

- 66239 vehicles in a desert battle simulation
- Solving the Mystery of Car Battery's Current
- Blue Brain - molecular-level simulation of the human brain
- Weather Prediction

These are some examples of countless applications of computing. Computer simulations are used in a huge number of diverse fields :

- Engineering : aerodynamics, large-scale interacting systems
- Meteorology : short-term weather forecasting, climate change
- Computational chemistry and bioinformatics : medicine, materials modelling
- Finance: risk prediction, portfolio construction

Source Code

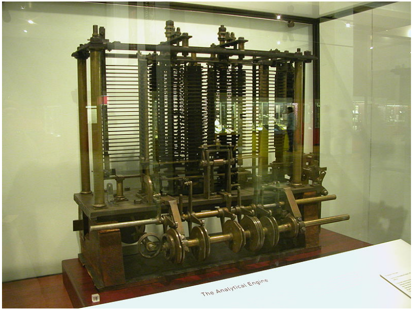
Source code is a set of instructions, saved in a plain text file, readable and changeable by humans.

However, computers can only really understand machine code.

Modern programming aims to make it easy to give a computer instructions, whilst keeping computation fast and efficient. There is, in general, a trade-off between ease of coding (high-level) and efficiency (low-level).

A Brief History

Pre-1940s : It is difficult to identify the first “programming language”. Perhaps the first computer program was written by Ada Lovelace in 1843 - a complete method for calculating Bernoulli numbers on Charles Babbage’s Analytical Engine.



A Brief History

1940s : The first real electrically-powered computers are born. Their processors recognise machine code, a set of instructions executed directly by the CPU, and hence specific to the computer architecture being coded for.

```
0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011
```

A Brief History

1940s : Assembly languages are developed. They are specific to the particular computer and are very difficult to write, but are at least beginning to look human-friendly. They require converting into machine code.

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

A Brief History

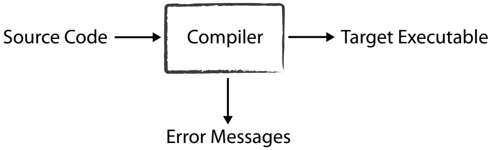
1950s onwards : High-level languages like Fortan (1955), Pascal (1970), C (1972), C++ (1980), Python (1991) and Java (1995) are developed. There are much easier to read and write, though they are still converted to machine code.

```
b = a + 2
```

Compilation

Converting human-legible source code to machine code is done by processes called compiling or interpreting. The result of this is an executable file that the computer's operating system can run in order to follow the instructions in the code.

The process is quite extensive, and involves various types of analyses of your code (primarily based on formal language theory), optimisations, generation of assembly code, etc., before an executable is created.



Interpretation

It is also possible to obtain machine code by interpretation. This is a far more linear approach - the code is translated into machine code one line at a time, usually at runtime.

Typically, interpreted code runs more slowly than compiled code, but compiling can take a long time. *Edit-interpret-debug* can potentially be much faster than *edit-compile-run-debug*.

Certain languages are compiled, others are interpreted. Higher-level languages (such as Python) *tend* to be interpreted more often than compiled (like C).

An Analogy

Programming	**Cooking**
Data	Ingredients
Source Code	Recipe
Compiler	Cook

C

- Low-level programming language
- Compiled
- Very fast, but mostly manually-implemented
- Extremely popular for scientific use. C++ is an 'improved' version of C

MATLAB

- High-level "technical computing" language
- Interpreted
- Optimised for matrix operations
- Slower computationally, but great for certain applications.
- Extended userbase and lots of implemented functionality via *toolboxes*

Programming in C and MATLAB - Environments

C is an open programming language. There are many Integrated Development Environments (IDEs) for it, each with different functionality. These can offer project management, syntax highlighting, debugging and easy compilation. We will use CodeBlocks when coding under Windows, and g++ directly in Linux during the HPC session.

MATLAB has its own environment and is a proprietary language (although there are now open source alternatives).

MATLAB Basics

Variables : simply declare using the variable name. Must start with a letter, can contain alphanumerics and underscores.

Arithmetic : `+`, `-`, `*`, `/`, `^`, `( )`. Very intuitive use.

Suppressing output : follow the command by a semi-colon.

Built-in functions : self-explanatory and very logical : `sin(0.5)`, `exp(myVar)`, `log(x)`.

Colon notation : `3:8` gives `3 4 5 6 7 8`. `0.1:-0.1:-0.3` gives `0.1 0 -0.1 -0.2 -0.3`.

Comments : `% This is a comment`.

doc : `doc sum` will tell you more about the `sum` function, including how to use it and its overloads.

FEx : MATLAB's File Exchange, on its website, has a large number of user-contributed files and functions.

Vectors

```
x = [1 3 -3]
      x =       1     3     -3

length(x)
      ans =    3

 x * x
      ???  Error using ==> *
      Matrix dimensions must agree

 x * x' % The single quote is a transpose
      ans =    19

 x .^3 % This is an element-wise operation
      ans =    1     27    -27
```

Vectors and Arrays

```
y = [x x; x x] % Semi-colon denotes a new row
     y =      1     3     -3     1     3     -3
              1     3     -3     1     3     -3

size(y)
     ans =    2     6

length(y) (= max(size(y)))
     ans =    6

x(2)
     ans =    3

y(2, :)  = 5:5:30
     y =      1     3     -3     1     3     -3
              5     10    15     20    25    30
```

Exercise - System of Linear Equations

Solve this linear system in MATLAB.

$$2\alpha + 8\beta + \gamma + 3\delta = 100$$
$$\alpha + \beta + 9\gamma + 7\delta = 143$$
$$4\alpha + 9\beta + \gamma + 5\delta = 111$$
$$4\alpha + 8\beta + 8\gamma + 2\delta = 264$$

This can be solved using linear algebra :

$$\mathbf{A}\,\mathbf{x} = \mathbf{y} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\,\mathbf{y}$$

Use MATLAB's `inv(myMatrix)` function to find the inverse of **A**.

Exercise - System of Linear Equations

Solve this linear system in MATLAB.

$$2\alpha + 8\beta + \gamma + 3\delta = 100$$
$$\alpha + \beta + 9\gamma + 7\delta = 143$$
$$4\alpha + 9\beta + \gamma + 5\delta = 111$$
$$4\alpha + 8\beta + 8\gamma + 2\delta = 264$$

This can be solved using linear algebra :

$$\mathbf{A}\,\mathbf{x} = \mathbf{y} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\,\mathbf{y}$$

Use MATLAB's `inv(myMatrix)` function to find the inverse of $\mathbf{A}$.

A better way to solve this is using MATLAB's `mldivide` function to solve the linear system $\mathbf{A} \backslash \mathbf{b}$.

Exercise - $\sin(x)$

Plot one cycle of $\sin(x)$:

- Declare a vector **x** between 0 and 2π. Use either colon notation, or the `linspace` function.
- Declare another vector **y** to be equal to the sin of **x**, using `sin`.
- Use `plot(x, y)` to plot **x** as a function of **y**.
- Set the axes labels using `xlabel` and `ylabel`, and a legend using `legend`.

Plotting Syntax

You can plot different colours and line styles by adding an argument to `plot(x, y, 'r-')`, for example. See MATLAB's LineSpec help online for the full list.

`hold` is a very useful function. `hold on` allows you to plot several functions, one after the other, on the same figure. It will automatically select a different line colour for each. `hold on` does not change the colour of subsequent plots, though you can do this yourself. You can use `hold off` to turn either of these off.

Plotting Syntax

You can plot several functions with the same command :

```
plot(x, sin(x), 'k-', x, cos(x), 'r.');
legend('Sin curve', 'Cos curve');
```

The `axis` command takes a vector as argument, defining x_{min}, x_{max}, y_{min}, y_{max} :

```
axis([0 2*pi -1 1])
```

Other Plotting Commands

You can create a plot with logarithmic axes using:

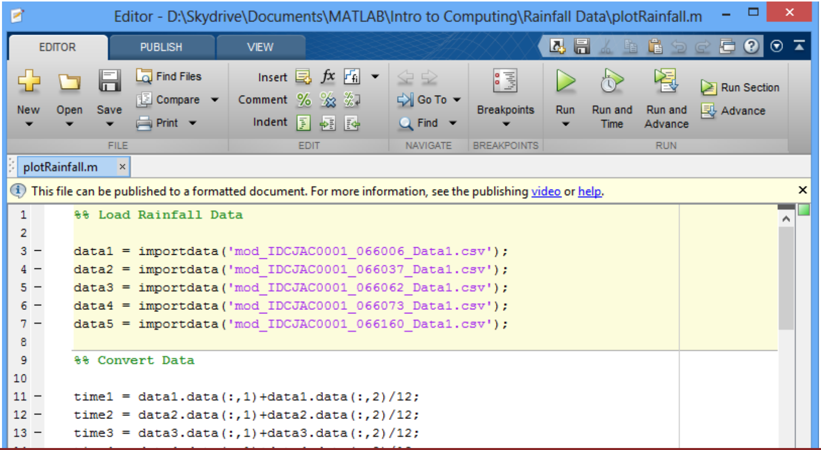
```
loglog( x, x, 'k-');

semilogy( x, exp(x), 'r-');

semilogx( x, log(x), 'k-');
```

.m Files

.m files allow you to save pieces of code as macros. This allows you to easily rerun several lines of code as many times you wish.



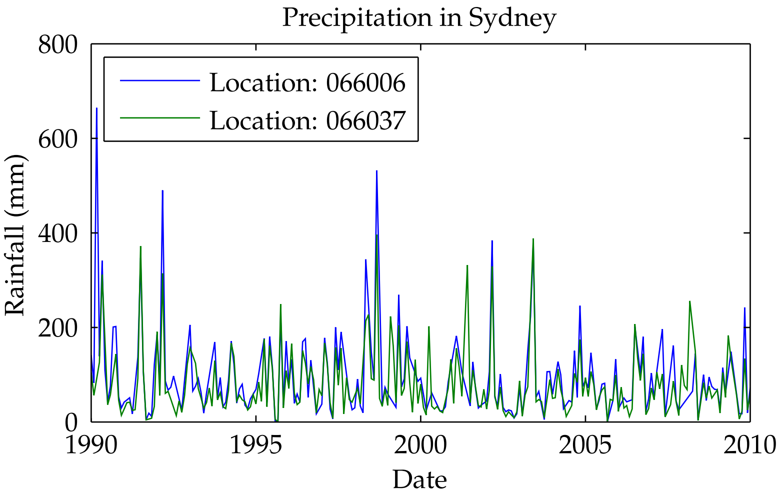
Rain in Sydney

We will look at precipitation data, freely accessible from the Australian Government Bureau of Meteorology.

Data is available from 1937 to early 2010.

Download CSV-formatted data from the module website.
`go.warwick.ac.uk/jonathanmascietaylor`

What does the data look like ?



Data Structure

The dataset contains recordings from five different instruments, none of which are perfect.

Data Structure

The dataset contains recordings from five different instruments, none of which are perfect.

Real data is rarely clean and complete.

This particular dataset is missing certain data points. We will need to interpolate.

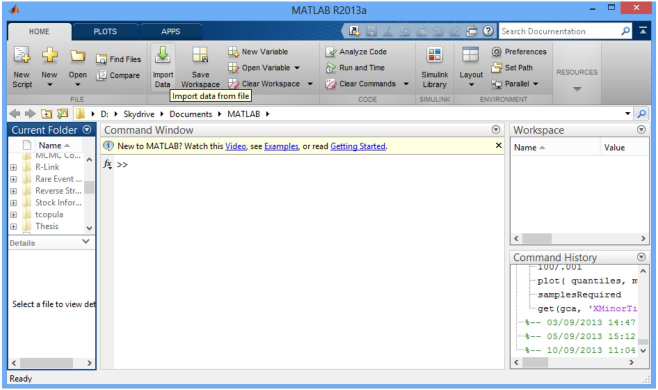
Measurements contain noise. We can calculate the standard error for each point in order to assess our confidence in the measurements.

Our Objectives

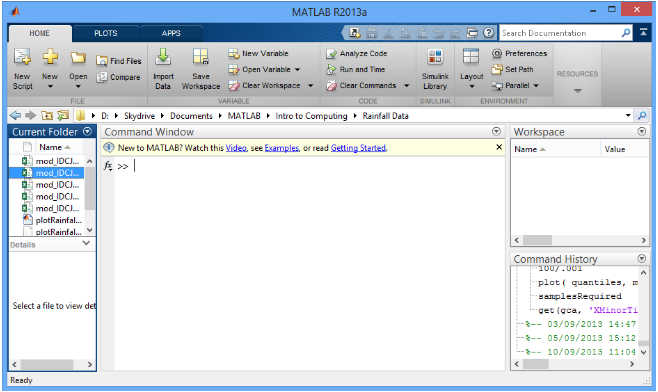
We aim to find a mean values for the monthly precipitation in Sydney over the range of time for which we have data.

We will plot these values with error bars representing the standard error.

Importing Data: Using the Toolbar



Importing Data: Drag and Drop



Importing Data: Selecting Columns

The `.csv` is a comma-separated value file. It looks like this :

Year,Month,Monthly Precipitation Total (millilitres)
1937,1,54.2
1937,2,31.9
1937,3,254.5
1937,4,157
1937,7,81.4

Importing Data: Programatically

We can also import the data using a simple command:

```
>> data1 = importdata('mod_IDCJAC0001_066006_Data1.csv');
>> data1.colheaders
ans =
 'Year' 'Month' [1x41 char]
>> data1.data
ans =
 1.0e+003 *
 1.9370 0.0010 0.0542
 1.9370 0.0020 0.0319
 1.9370 0.0030 0.2545
```

Cleaning Variables

The data for each run is stored in its own variable. The variable contains the time of measurement and the measurement value.

What is the `size` of a particular variable ? What does this represent ?

Cleaning Variables

The data for each run is stored in its own variable. The variable contains the time of measurement and the measurement value.

What is the `size` of a particular variable ? What does this represent ?

Let's clean things up. Create a series of variables `time1`, ..., `time5` and `rainfall1`, ..., `rainfall5` for the time and the measurement values respectively. Time should be in years.

Cleaning Variables

The data for each run is stored in its own variable. The variable contains the time of measurement and the measurement value.

What is the `size` of a particular variable ? What does this represent ?

Let's clean things up. Create a series of variables `time1`,...,`time5` and `rainfall1`,...,`rainfall5` for the time and the measurement values respectively. Time should be in years.

```
rainfall1 = data1(:,3);
```

Cleaning Variables

The data for each run is stored in its own variable. The variable contains the time of measurement and the measurement value.

What is the `size` of a particular variable ? What does this represent ?

Let's clean things up. Create a series of variables `time1`, ..., `time5` and `rainfall1`, ..., `rainfall5` for the time and the measurement values respectively. Time should be in years.

```
rainfall1 = data1(:,3);
time1 = data1(:,1) + data1(:,2) / 12;
```

Interpolating over Missing Points

We have five runs of data, each one missing monthly recordings (not because the meteorologists were lazy, but because the dataset was decimated randomly).

Assuming we want regular monthly measurements, we can interpolate the missing points by looking at the points to either side. We will use Matlab's interpolation function in linear mode (`interp1`) to obtain an estimate of the missing data.

Interpolating over Missing Points

We have five runs of data, each one missing monthly recordings (not because the meteorologists were lazy, but because the dataset was decimated randomly).

Assuming we want regular monthly measurements, we can interpolate the missing points by looking at the points to either side. We will use Matlab's interpolation function in linear mode (`interp1`) to obtain an estimate of the missing data.

Start by creating a time vector `x` beginning with the earliest date you have, incrementing in monthly intervals until the final date you have available. The time vector should also have units of years.

Interpolating over Missing Points

We have five runs of data, each one missing monthly recordings (not because the meteorologists were lazy, but because the dataset was decimated randomly).

Assuming we want regular monthly measurements, we can interpolate the missing points by looking at the points to either side. We will use Matlab's interpolation function in linear mode (`interp1`) to obtain an estimate of the missing data.

Start by creating a time vector `x` beginning with the earliest date you have, incrementing in monthly intervals until the final date you have available. The time vector should also have units of years.

```
time = time(1) :  1 / 12 :  time(end);
```

Interpolation

Matlab's interpolation function can take multiple combinations of arguments (it is *overloaded*).

Interpolation

Matlab's interpolation function can take multiple combinations of arguments (it is *overloaded*). We will use it in the following form :

```
my_interpolated_rainfall = interp1(my_original_time,
my_original_rainfall, desired_time);
```

Run this command for each data run.

Interpolation

Matlab's interpolation function can take multiple combinations of arguments (it is *overloaded*). We will use it in the following form :

```
my_interpolated_rainfall = interp1(my_original_time,
my_original_rainfall, desired_time);
```

Run this command for each data run.

```
interpRainfall1 = interp1(time1, rainfall1, time);
```

Interpolation

Matlab's interpolation function can take multiple combinations of arguments (it is *overloaded*). We will use it in the following form :

```
my_interpolated_rainfall = interp1(my_original_time,
my_original_rainfall, desired_time);
```

Run this command for each data run.

```
interpRainfall1 = interp1(time1, rainfall1, time);
```

How many points do you have ? Does each run have the same number of data points ?

Calculating the Mean : the Naïve Way

Create a vector `Rainfallmean` by summing over vectors `Rainfall1` to `Rainfall5` and dividing by five.

Calculating the Mean : the Naïve Way

Create a vector `Rainfallmean` by summing over vectors `Rainfall1` to `Rainfall5` and dividing by five.

```
meanRainfall = (interpRainfall1 + interpRainfall2 +
interpRainfall3 + interpRainfall4 + interpRainfall5) / 5;
```

Calculating the Mean : the Matrix Way

Delete your `Rainfallmean` variable : `clear Ymean`.

Create a matrix `Rainfall` by collating vectors `Rainfall1`,...,`Rainfall5` so that you obtain a $5 \times n$ matrix.

Calculating the Mean : the Matrix Way

Delete your `Rainfallmean` variable : `clear Ymean`.

Create a matrix `Rainfall` by collating vectors `Rainfall1`,...,`Rainfall5` so that you obtain a $5 \times n$ matrix.

```
interpRainfall = [interpRainfall1; interpRainfall2;
interpRainfall3; interpRainfall4; interpRainfall5];
```

Calculating the Mean : the Matrix Way

Delete your `Rainfallmean` variable : `clear Ymean`.

Create a matrix `Rainfall` by collating vectors `Rainfall1`, ..., `Rainfall5` so that you obtain a $5 \times n$ matrix.

```
interpRainfall = [interpRainfall1; interpRainfall2;
interpRainfall3; interpRainfall4; interpRainfall5];
```

Use Matlab's `mean` function to create a vector `Rainfallmean` containing the mean precipitation, from your matrix `Y`. Before you do this, read the `doc mean` documentation so that you know what to expect when you give `mean` a matrix.

Calculating the Mean : the Matrix Way

Delete your `Rainfallmean` variable : `clear Ymean`.

Create a matrix `Rainfall` by collating vectors `Rainfall1`, ..., `Rainfall5` so that you obtain a $5 \times n$ matrix.

```
interpRainfall = [interpRainfall1; interpRainfall2;
interpRainfall3; interpRainfall4; interpRainfall5];
```

Use Matlab's `mean` function to create a vector `Rainfallmean` containing the mean precipitation, from your matrix `Y`. Before you do this, read the `doc mean` documentation so that you know what to expect when you give `mean` a matrix.

```
meanRainfall = mean(Rainfall);
```

Calculating the Standard Deviation

Using your matrix `Rainfall`, use Matlab's `std` function to create a new vector `Rainfallstd` of the standard deviation of each point in the sampled data. Once again, read the `doc std` documentation to see if `std` can take a matrix argument, and what it will return.

Calculating the Standard Deviation

Using your matrix `Rainfall`, use Matlab's `std` function to create a new vector `Rainfallstd` of the standard deviation of each point in the sampled data. Once again, read the `doc std` documentation to see if `std` can take a matrix argument, and what it will return.

```
stdRainfall = std(Rainfall);
```

Calculating the Standard Deviation

Using your matrix `Rainfall`, use Matlab's `std` function to create a new vector `Rainfallstd` of the standard deviation of each point in the sampled data. Once again, read the `doc std` documentation to see if `std` can take a matrix argument, and what it will return.

```
stdRainfall = std(Rainfall);
```

If you transpose your matrix `Rainfall` before calling `std`, what would you expect to obtain ? What could this represent ?

Calculating the Standard Deviation

Using your matrix `Rainfall`, use Matlab's `std` function to create a new vector `Rainfallstd` of the standard deviation of each point in the sampled data. Once again, read the `doc std` documentation to see if `std` can take a matrix argument, and what it will return.

```
stdRainfall = std(Rainfall);
```

If you transpose your matrix `Rainfall` before calling `std`, what would you expect to obtain ? What could this represent ?

```
std(Rainfall');
```

The Standard Error

The Standard Error of the Mean (sampling error) is defined simply as

$$E = \frac{\sigma_{\texttt{sample}}}{\sqrt{n_{\texttt{sample}}}}$$

We have already calculated the standard deviation of the sample $\sigma_{\texttt{sample}}$ in our variable, $\texttt{Y}_{\texttt{std}}$. We also know how many samples we have for each point in the function or process.

Plotting the Mean with Errors

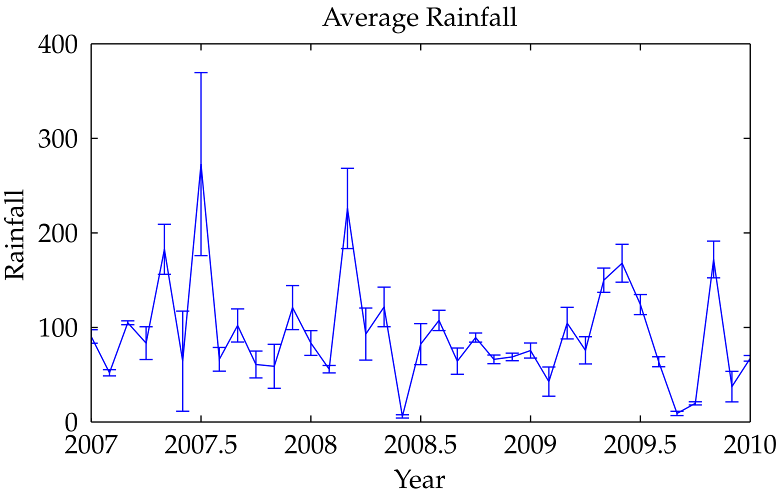
Matlab has a function `errorbar`. Use the `help` to see what arguments this expects, and then plot the mean precipitation against time, with symmetrical standard errors, between 2007 and 2010. Use `find` to locate the relevant index for the start date.

Plotting the Mean with Errors

Matlab has a function `errorbar`. Use the `help` to see what arguments this expects, and then plot the mean precipitation against time, with symmetrical standard errors, between 2007 and 2010. Use `find` to locate the relevant index for the start date.

```
i = find(time == 2007);
j = find(time == 2010);
errorbar(time(i:j), meanRainfall(i:j), (stdRainfall(i:j) /
sqrt(5)));
axis([2007 2010 0 350]);
xlabel( 'Year' );
ylabel( 'Rainfall (mm)' );
```

Plotting the Mean with Errors



If Statements

If statements allow you to execute some code, if a specific condition tests out as *true*. It works like this :

```
if a == b
  do_something;
end
```

If Statements

If statements allow you to execute some code, if a specific condition tests out as *true*. It works like this :

```
if a == b
  do_something;
end
```

These can be expanded to do other things depending on different conditions :

```
if a > b
  c = a;
elseif a < b
  c = b;
else
  c = 0;
end
```

For Loops

For loops are a general programming concept present in just about every language out there. They are used to iterate over objects, or do things multiple times.

For Loops

For loops are a general programming concept present in just about every language out there. They are used to iterate over objects, or do things multiple times.

```
for i = 1:10
  do_something;
end
```

For Loops

For loops are a general programming concept present in just about every language out there. They are used to iterate over objects, or do things multiple times.

```
for i = 1:10
  do_something;
end
```

Note the colon notation in the `for` statement. You can use any standard Matlab colon notation here :

```
for i = 0:0.1:0.6
  7 - i * 2
end
```

For Loops

For loops are a general programming concept present in just about every language out there. They are used to iterate over objects, or do things multiple times.

```
for i = 1:10
  do_something;
end
```

Note the colon notation in the `for` statement. You can use any standard Matlab colon notation here :

```
for i = 0:0.1:0.6
  7 - i * 2
end
```

The result is $7, 6.8, 6.6, 6.4, 6.2, 6, 5.8$.

Generating Random Numbers

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

- John von Neumann

Generating Random Numbers

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*
>
> \- John von Neumann

`rand` generates uniformly-distributed pseudorandom numbers between 0 and 1.

It can be used on its own, or take arguments to generate a vector or matrix of random numbers.