Introduction to MATLAB

Welcome! This minicourse will get you familiar with the basics of MATLAB, and doesn't require previous programming experience.

Interface

You will frequently work in MATLAB's *Command Window*: here you can give MATLAB commands and see the results of computations. You may also open the *Editor* by going to *File*, then *New*, then *Blank M-File*. (We will return to this later.)

The *Workspace* lists the variables and data you have loaded into MATLAB, while the *Command History* lists previous commands you have entered. Finally, the *Current Folder* displays the details of the current working directory (which is important when you import and export datasets).

Language Basics

Enter this into your Command Window and press Enter:

```
>> 5
ans =
    5
>> 5;
```

MATLAB takes what we enter at the command prompt, evaluates it, and prints the results. However, ending an expression with a semicolon supresses the output.

For a complex example:

```
>> 5 + 5
ans =
    10
```

MATLAB also works like a calculator: addition (+), subtraction (-), multiplication (*), and division (/) work as expected, although they sometimes have special roles which you will see later.

You're rarely work with just literals: you need variables to hold values. In this example, we assign the value `5` to the variable `x`:

```
>> x = 5
x =
    5
>> x
x =
    5
```

Whenever we write `x` in an expression, MATLAB will now replace `x` with `5`:

```
>> x + 1
ans =
    6
```

Matrices

MATLAB is well suited for working with matrices. We declare a matrix using square brackets ([ ]), separating column elements with commans and row elements with semicolons:

```
>> [1,2,3]
ans =
    1    2    3
>> [1;2;3]
ans =
    1
    2
    3
>> [1,2;3,4]
ans =
    1    2
    3    4
```

Assign matrices to variables identically to normal values:

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

Extract array elements using parentheses. For a one-dimensional matrix, we write

```
>> b(1)
ans =
    1
>> b(2)
ans =
    2
```

For a two-dimensional matrix, we write the *row* and the *column*:

```
>> a(1,1)
ans =
    1
>> a(1,2)
ans =
    2
>> a(2,1)
ans =
    3
```

Remember that MATLAB starts counting indices from 1, not 0 (unlike many other programming languages).

Colons generate number ranges: useful for both generating matrices and extracting elements from matrices.

```
>> c = 1:5
c =
  1     2       3       4       5
>> c = 1:2:10
c =
  1     3       5       7       9
>> c(1:3)
ans =
    1    3    5
>> c(2:4)
ans =
    3    5    7
```

Using just a colon selects entire columns and entire rows:

```
>> a(:,1)
ans =
    1
    3
>> a(1,:)
ans =
    1   2
```

Logical Indexing can also be used to extract elements from a matrix:

```
>> m = [1 2 3 ; 4 5 6]
m =
    1       2       3
    4       5       6
>> m(:, [false false true])
ans =
    3
    6
```

The colon indicates *all rows*, and the second clause (`[false false true]`) selects the last column only.

We can do arithmetic on matrices: operations between matrices and singular values result in apply the operation with the value to each element of the matrix, while operations between matrices perform the corresponding matrix operation (e.g. matrix multiplication).

```
>> a + 1
ans =
    2   3
    4   5
>> a * 5
ans =
    5   10
    15  20
>> a + a
ans =
```

```
    2   4
    6   8
>> a * a
ans =
    7   10
    15  22
>> a * b
ans =
    5
    11
```

To perform an element wise operation between two matrices, use a period before the operation:

```
>> b .* b
ans =
    1
    4
>> b * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Here are more common matrix functions:

Transpose:

```
>> a'
ans =
    1   3
    2   4
```

Diagonal (both access and generation):

```
>> diag(a)
ans =
    1
    4
>> diag(diag(a))
ans =
    1   0
    0   4
```

Sum (across rows, unless you specify the dimension):

```
>> sum(a)
ans =
    4   6
>> sum(a,2)
ans =
    3
    7
>> sum(sum(a))
ans =
    10
```

Zero, generate an *n* by *n* or *n* by *m* matrix of zeros:

```
>> zeros(2)
ans =
    0   0
    0   0
>> zeros(1,2)
ans =
    0   0
```

Ones, generate an *n* by *n* or *n* by *m* matrix of ones:

```
>> ones(2)
ans =
    1   1
    1   1
>> ones(2,1)
ans =
    1
    1
```

Identity:

```
>> eye(2)
ans =
    1   0
    0   1
```

Random, random numbers between 0 and 1:

```
>> rand(2)
ans =
    0.8947      0.1270
    0.9058      0.9243
```

Conditionals and Loops

Conditional statements and loops control how MATLAB executes code: we can optionally run statements (conditionals) and we can run statements multiple times (loops).

MATLAB uses `1` to represent *True Expressions* and `0` to represent *False Expressions*. == tests for equality:

```
>> 1 == 1
ans =
    1
```

~= tests for inequality (not `!=`, common in other languages):

```
>> 1 ~= 1
ans =
    0
```

We also have *less than* (<), *less than or equal to* (<=), *greater than* (>), and *greater than or equal to* (>=).

```
>> 5 > 3
ans =
    1
```

We can also use *and* (`&&`) and *or* (`||`) to compose expressions:

```
>> (5 > 3) && (5 < 7)
ans =
    1

>> (3 >= 5) || (10 ~= 11)
ans =
    1
```

Applying these operations to matrices applies them to each element:

```
>> a < 2
ans =
    1   0
    0   0
```

We can also use *if statements*:

```
if (boolean value)
   do something
end
```

With runs `do something` if `boolean value` is *true*. E.g.

```
>> if 5 > 2
n = 5
end

n =
  5

>> if 2 > 5
n = 0
end
>>
```

We can also use *else statements* to supply alternative actions to take if the boolean value is *false*.

```
>> if 2 > 5
n = 0
else
n = 1
end

n =
  1
```

We can even concatenate *if statements* and *else statements* for more options:

```
>> if n == 0
n = 1
else if n == 1
n = 0
else
n = 5
end
```

For Loops repeat a statement a certain number of times:

```
for loopvariable = array
    action
end
```

For each element in `array`, MATLAB will run `action` once. The value of `loopvariable` ranges over the elements of `array`. On the first time the loop runs, `loopvariable` has the value of `array(1)`, and on the second time the loop runs, `loopvariable` has the value of `array(2)`, and so on.

Let's calculate the factorial of 5:

```
>> fact = 1;
>> fact = fact * 2;
>> fact = fact * 3;
>> fact = fact * 4;
>> fact = fact * 5
ans =
    120
```

More verbosely:

```
>> fact = 1;
>> i = 2;
>> fact = fact * i;
>> i = 3;
>> fact = fact * i;
>> i = 4;
>> fact = fact * i;
>> i = 5;
>> fact = fact * i
ans =
    120
```

And more succinctly in a *for loop*:

```
>> fact = 1;
>> for i = [2:5]
fact = fact * i
end

fact =
     2
fact =
     6
```

```
fact =
    24
fact =
   120
```

Exercise

Write MATLAB code that computes all of the prime numbers less than 100. You'll find it helpful to think of two components: testing for primality and testing for less than 100.

If you get stuck, try using the *modulo* (`mod`) function: `mod(6,3)` is 0, and `mod(6,5)` is 1.