Matlab Reference Guide

Arithmetic

Basic arithmetic operators work like you'd expect; parentheses can be used to group terms:

Basic operators: `+ - * / ^`
Grouping: `( )`
 	
The ^ operator does exponentiation, so `2^3` computes $2^3 = 2 * 2 * 2 = 8$.
	`>> 2*(3+10)`
	`ans = 26`
	`>> 2^3`
	`ans = 8`

Variables

Variables are just places where you can store values. Variable names can be almost anything, except they can't start with a number or contain spaces or operators.

Assignment operator: `=`
 	
A useful way to think about this operator is to read it as "gets" - that is the first example on the right would be "a gets 10".
 	
*You can stop Matlab from printing the value after any line of code by adding a **semicolon** (;) to the end of the line.*
	`>> a = 10`
	`a = 10`
	`>> a + 2`
	`ans = 12`
	
	`>> a_long_variable_name = 100`
	`a_long_variable_name = 100`
	
	`>> a_string = 'alpaca';`

Expressions and Values

A **value** is simply a concrete piece of data - it could be a number, a string, or a matrix (more on these later). An **expression** is a piece of code that can be **evaluated** to produce a value. So 10 is a value, and `5+5`, `10`, and `20 / 2` are all expressions whose value is 10. If we type `a = 10`, then `a` is also an expression whose value is 10.

Comparison Operators and Boolean Expressions

Boolean expressions evaluate to either **true** or **false**, represented in Matlab as 1 and 0. Comparison operators are the simplest kind of boolean expressions:

Comparison operators:
`<` (less than)
`>` (greater than)
`<=` (greater than or equal to)
`>=` (less than or equal to)
`==` (equal to)
`~=` (not equal to)
A common mistake is to mix up = and ==. Remember that = is the assignment operator, and means "gets", whereas == means "is equal to".
	`>> 5 < 10  % (true)`
	`ans = 1`
	`>> 5 >= 10  % (false)`
	`ans = 0`
	
	`>> a = 10`
	`>> a == 10  % (true)`
	`ans = 1`

Control Flow: if statements and for loops

We can make decisions about what to do based on the value of a boolean expression using an **if statement**, and we can repeat some lines of code many times using a **for loop**. You can specify the range of values a `for` loop's variable takes on using brackets (`[a,b,c]`) or using a range (`1:10`).

```
if <condition>
  <do this if condition is true>
else
  <do this if condition is false>
end
```

If you don't want to do anything if the condition is false, you can leave out the `else` clause.

Range operator:
`a:b` gives the range of values from a to b inclusive.
`a:x:b` gives the range from a to b counting by x

```
for <loop_var> = <range of values>
  <do this for each value of
loop_var>
end
```

You can also simply give a list of values you want loop_var to take on, e.g., `[1, 2, 3, 4, 5]` is equivalent to `1:5`.

```
>> if 5 < 10
     a = 100;
   else
     a = 200;
   end
>> a
a = 200


>> 1:5
ans = 1    2    3    4    5
>> 1:3:10
ans = 1    4    7    10

total = 0;
for i = 1:5
  total = total + i;
end
>> total
total = 15
```

Calling Functions

Matlab has lots of extremely useful built-in functions. Typically, there are three things involved in calling a function: the function name, its **arguments**, and its **return value**. Arguments are passed into the function, and the return value is the result.

Anatomy of a function call:

```
  return_value =
function(arguments);
```

If you leave out `return_value =`, Matlab will assign the return value to `ans` as with any unassigned value.

Find out how to use any built-in function:

```
  help <functionname>
```

Some functions take more than one argument, separated by commas:

```
  function(arg1, arg2, ...)
```

```
>> sqrt(9) % square root
ans = 3
>> abs(-9) % absolute value
ans = 9
>> big_llama = upper('llama')
big_llama = 'LLAMA'

>> help upper
(documentation prints)

% are these strings equal?
>> strcmp('llama', 'alpaca')
ans = 0 % false - not equal
```

Writing New Functions

Remember that a function call has a name, arguments, and a return value. To define a function, we create a file that specifies the name, arguments, and some code that calculates the return value. The filename should match the name of the function, with a .m extension.

Anatomy of a function definition:

```
function [ output ] = function_name( input_args )
  <write some code here that assigns a value to the output variable>
end
```

This code should go in a file called function_name.m. Then you can call the function as we saw above:

```
out = function_name( iargs );
```

It's possible to write a function that has no return value and/or no input args. In this case, you simply leave the brackets and parens empty, respectively, i.e., `[] = function_name()`*. However, usually a function isn't very useful unless it computes an output based on some input variables.*

To create a new function, click the New button in the Matlab toolbar and select Function. Matlab creates a file and fills in the default structure similar to the above example.

Example function definitions:

```
function [area] = square_area(side_length)
  area = side_length * side_length;
end
```

Functions can take multiple arguments and return multiple outputs, as in the following example:

```
function [area, perim] = rect_info(side1, side2)
  area = side1 * side2;
  perim = 2*(side1 + side2);
end
```

Note that if you don't assign the outputs to variables, matlab will only print the first output (`area`*, in this case).*

Example usage:

```
>> sq_area(10)
ans = 100
>> b = 5;
>> bsq = sq_area(b)
bsq = 25

>> [a,p] = rect_info(5,10)
a=50
p=30

>> rect_info(10,10)
ans = 100
```

Reading and Writing Images

Basic image functions:

`imread` - read an image from a file

`imshow` - display an image in a figure window

`imwrite` - save an image to a file

Remember a semicolon when you work with images, or Matlab will print out the entire image as a matrix (that's a lot of numbers!)

```
>> img = imread('llama.jpg');
>> img = img - 50;
>> imshow(img);
>> imwrite(img,'darkllama.jpg');
```

Matrices

A matrix is just a grid of numbers. These are really easy to work with in Matlab - in fact, that's what the "Mat" stands for. Because images are made up of a grid of pixels, it's natural to think of them as matrices.

Creating new matrices:
`zeros` - create a matrix full of zeros
`ones` - create a matrix full of ones
 	
*When specifying the size of a matrix of the index of an element, you always specify **rows then columns**. So a 2-by-3 matrix has 2 rows and 3 columns, and element (1, 2) of a matrix is element in the first row and the second column.*
	`>> M = zeros(2,3)`
	`M =`
	`  0  0  0`
	`  0  0  0`
	`>> N = ones(3,2)`
	`N =`
	`  1  1`
	`  1  1`
	`  1  1`

Accessing individual elements:
`M(i,j)` - the (i, j)th element of M
 	
*This can be used either to **get** the value or to **set** it.*
	`>> M(2,1)`
	`ans = 0`
	`>> M(2,1) = 100`
	`M =`
	`  100  0  0`
	`  0  0  0`

Element-wise arithmetic: If you use any normal arithmetic operator with a matrix, the operation will be done to all of its elements..	`>> M + 10`
	`ans =`
	`  110  10  10`
	`  10  10  10`

Color images are represented as 3-dimensional matrices - this means each element has 3 indices, a row, column, and a *color channel*. A color image has three channels to store the red, green, and blue color values of a pixel.
 	
Slicing allows you to get a submatrix of a larger matrix. Use `:` instead of an index to look up all indices along that dimension.
 	
Getting the **size** of a matrix:
`size` - return the dimensions of a matrix
	`>> img = imread('color_llama.jpg');`
	`>> img(1,1,1)`
	`ans = 45  % (red channel value)`
	`>> img(1,1,3)`
	`ans = 221  % (blue channel value)`
	
	`>> img(1,1,:) % (all channels)`
	`ans = 45  52  221`
	
	`>> green_image = img(:,:,2);`
	`% green_image now contains the`
	`% entire (2D) green channel image`
	
	`>> size(img)`
	`ans = 400  600  3`
	`>> size(green_image)`
	`ans = 400  600`

Functions

** Functions marked with an asterisk are not built-in so you must be in the same folder as the code (functionname.m) to use them.*

Math: `abs, sqrt, min, max, sum, mod`
Strings: `upper`
Turtles*: `Turtle, turtleForward, turtleTurn, turtleDown, turtleUp, turtleSquare`
Matrices: `size, circshift, cat`
Images: `imread, imshow, imwrite, imresize`

Logical operations: `or, and, not`
Figure management: `figure, close, clf`
Workspace management: `clear, clc`