MATLAB – One Page Summary

Rule 1: All matrices follow matrix arithmetic rules. (e.g. `A*B` only works if `A` and `B` are the right dimensions, etc.)
Rule 2: If you're just working with ordinary numbers, e.g. 25, don't worry about matrix arithmetic rules.
Rule 3: Ending a line with a semicolon (;) suppresses the display of the results for that line.

Useful commands

`doc` *fn*	- get explanation for function *fn*, with usage examples
`edit`	- starts the MATLAB editor
`clc`	- clears the screen
`clear`	- clears the variables from the workspace (and from memory)
`plot(x,y)`	- plot y vs. x.
`xlabel(),ylabel()`	- add labels to the x and y axes

Common functions

`sin(x), cos(x), tan(x), abs(x), sqrt(x), exp(x)`

Matrix-related

`A = [1 2; 3 4]`	- specifies a matrix
`A(m,n)`	- gets value in row *m*, col *n* of A
`A(:,n)`	- gets column *n* of A
`A(m,:)`	- gets row *m* of A
`zeros(row,col)`	- creates a zero-filled matrix of size row-by-col
`ones(row,col)`	- creates a one-filled matrix of size row-by-col
`[A B]`	- joins A and B side-by-side (horizontal concatenation)
`[A ; B]`	- joins A and B top and bottom (vertical concatenation)

Structure of a function

```
function [output] = functionname(input1, input2, ...)
      ...
end
```

Common structures

```
if condition
  ...
elseif condition
  ...
else
  ...
end
```

```
for var = 1:N
      ...
end
```

```
while condition
      ...
end
```

Normal Operators

`A + B`	- addition
`A - B`	- subtraction
`A * B`	- matrix multiplication (using matrix arithmetic rules)
`A .* B`	- element-by-element multiplication
`A ^ n`	- matrix raised to the power n, (A * A * A ....)
`A .^ n`	- each element of the matrix A raised to the power n
`inv(A)`	- matrix inverse
`A'`	- matrix transpose

Conditional operators

`A == B`	Is it true that A equals B?
`A =~ B`	Is it true that A is not equal to B?
`A > B`	Is it true that A is greater than B?

`A >= B`	Is it true that A is greater than or equal to B?
`A <  B`	Is it true that A is less than B?
`A <= B`	Is it true that A is less than or equal to B?

Quick and Dirty® Primer to MATLAB

Before we start...

... there is one handy MATLAB command that you absolutely have to remember:

```
doc functionname
```

Whenever you are unsure about how to use a particular MATLAB function, just type the `doc` command followed by the name of the function. For instance, to get the lowdown on the `cos` function (cosine), type:

```
doc cos
```

A screen will appear describing the cosine function with examples of how to use it in MATLAB.

Starting MATLAB

When you start MATLAB, the first thing you will see is the "Command Window".

```
                    < M A T L A B >
          Copyright 1984-2004 The MathWorks, Inc.
          Version 7.0.1.24704 (R14) Service Pack 1
                    September 13, 2004

     To get started, select MATLAB Help or Demos from the Help menu.

     >>
```

All MATLAB commands issued in this window will be processed instantly. This manner of issuing instructions is called "interactive mode", where you "interact" directly with MATLAB. You can also write programs in MATLAB called **M-files** (which are thus named because of their **.m** extension).

The rudiments

In engineering, we are primarily interested in numbers[1]. It is therefore no coincidence that numerical computation is the bread-and-butter of the engineering sciences, and therein lies also the strength of MATLAB. MATLAB is currently the industry standard[2] for numerical computation, and the McMaster standard[2] for upper-year chemical engineering (3P3, 4E3, etc.) courses, so it is worthwhile investing the time to learn how to use it properly.

In numerical computation, we customarily represent and manipulate large sets of numbers by means of *matrices* and *matrix operations*. (If you have forgotten your Linear Algebra, it is probably a good idea to review the basic rules of matrix arithmetic now.)

In MATLAB, almost every piece of data is a matrix, hence the name MATLAB (which stands for MATrix LABoratory). Even a single number is a matrix! (It's a 1-by-1 matrix.) It is therefore quite important to know how to enter a matrix. Consider this example:

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

[1] Although one needs to bear in mind that "the [ultimate] purpose of computing is insight, not numbers." (Richard W. Hamming, 1915-1998)
[2] Like it or not.

It will spit out the following:

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

If you study the above example, you can easily deduce the rules for entering matrices: (1) enclose the matrix in square brackets [ ]; (2) put a space between each element; and, (3) put a semicolon (;) at the end of each matrix row. (Tip: the semicolon may be omitted for the last row of the matrix.)

Getting values of matrix elements. To retrieve the value of an element in a matrix, use the `MATRIX(row,column)` notation[3]. For instance, if you want to retrieve the value of the element in row 1, column 3 of matrix A above, you would type something like:

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

Potential trap for programmers with backgrounds in other languages - Index 0 not allowed: Unlike other programming languages where you access the 1st element of an array using a construct like A(0) or A(0,0), MATLAB does not allow this because there is no index 0 in MATLAB. MATLAB always starts matrix indices from 1.

This is primarily due to MATLAB's design which uses the matrix as a primitive datatype. You see, by mathematical convention, there is no such thing as a row 0 or a column 0 of a matrix, so it makes sense that you can't specify such a thing in MATLAB either.

Specifying ranges. You can also slice and dice a matrix by using a colon (:) to specify ranges. For instance, let's say you wanted to extract a sub-matrix from the matrix A above, comprising the 2nd-3rd rows and the 1st-2nd columns of A:

```
>> A(2:3,1:2)
ans =
     5    10
     9     6
```

Case sensitivity. Note that all variable names are case sensitive, so an "A" is not the same thing as an "a".

Zero, ones, range matrices. At some point, you may want to create zero-filled matrices. Say you want to create a zero-filled column matrix (i.e. "vector") called `guybrush` of size 3-by-1. You would do it as follows:

```
>> guybrush = zeros(3,1)
guybrush =
     0
     0
     0
```

Similarly, for a 2-by-2 matrix filled with ones called `elaine`, you would type:

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

[3] Hint: Since you are going to encounter this notation a lot (especially in mathematics), it may help to memorize the mnemonic "rowcol", which stands for: specify *row* first, *column* second. Note that this is exactly the opposite of the Cartesian (x, y) notation that we're used to seeing in geometry.

```
     1     1
```

If you want to create a vector of numbers from 1 to 10, you can do this:

```
>> threepwood = 1:10
threepwood =
     1     2     3     4     5     6     7     8     9    10
```

If you want a vector from 1 to 20, in increments of 2, just shove the size of the increment in between the two ends of the range, like this:

```
>> wooster = 1:2:20
wooster =
     1     3     5     7     9    11    13    15    17    19
```

.

Wildcards. The colon (:) has another meaning: it behaves like a "grab everything" operator when used **on its own**. In computer parlance, this is known as a "wildcard". It is easier to understand this through an example.

For instance, let's say you want the entire 2nd **column** of the matrix A above. Recalling that the notation for retrieving values from a matrix is MATRIX(row,column), you can type this:

```
>> A(:,2)
ans =
     3
    10
     6
    15
```

MATLAB then grabs every row of column 2 (and thus, you end up with 2nd column of the matrix, naturally).

If, on the other hand, you want the 2nd **row**, you would type:

```
>> A(2,:)
ans =
     5    10    11     8
```

This time, MATLAB grabs every column of row 2, which gives you the entire 2nd row.

> **Tip**: An easy way to remember how this wildcard thing works is to substitute the colon (:) with the word EVERYTHING in your mind. For example, `A(:,2)` reads like:
>
> "A(rows = EVERYTHING, columns = only column 2)"

Matrix arithmetic - Multiplication (matrix, element-by-element). Since almost everything in MATLAB is a matrix, all arithmetic operations follow matrix arithmetic rules[4].

Consider two matrices, *a* and *b*:

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

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

[4] If you recall your Linear Algebra, this means among other things that: (1) A*B does not generally equal B*A; (2) you cannot multiple two matrices A*B if the dimensions of A and B are not conformable, that is, if the no. of columns in A is not equal to the no. of rows in B.

```
     1     2
     3     4

>> a*b
ans =
     7    10
    15    22
```

Notice that the matrices *a* and *b* were multiplied using *matrix multiplication* rules.

However, in some special cases, you may want to multiply the matrices using "**element-by-element**" multiplication (i.e. row 1, column 1 of *a* with row 1, column 1 of *b*). MATLAB allows you to do this using the .* (dot-star) operator, as follows:

```
>> a.*b
ans =
     1     4
     9    16
```

Tip: When you add a dot (.) in front of an arithmetic operator, you are telling MATLAB that you want to do an **element-by-element** operation. More examples of this follow below.

Raising a matrix to a power. In this example, `a^3` is equivalent to `a*a*a` (following the rules of matrix multiplication).

```
>> a^3
ans =
    37    54
    81   118
```

Now, if you were to add a dot (.) in front of the exponentiation operator (^), the matrix's elements are raised to the specified power, in situ (an **element-by-element** operation).

```
>> a.^3
ans =
     1     8
    27    64
```

Matrix arithmetic - Inverse, transpose. To find the inverse of a matrix (a^{-1}), simply use the `inv` function:

```
>> inv(a)
ans =
   -2.0000    1.0000
    1.5000   -0.5000
```

To transpose a matrix (a^T), just add a single right quote (a.k.a. prime, apostrophe – on a U.S./Canadian standard keyboard, it's the key next to the semicolon (;) key):

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

Joining matrices. You will often need to join two matrices (you know, sort of glue them together) to form a bigger matrix. There are two ways to join matrices: **side-by-side** (horizontal), or **top-and-bottom** (vertical). Consider these examples:

```
>> [a b]
ans =
     1     2     1     2
```

```
     3     4     3     4

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

Ordinary Numbers. We've been talking about matrices thus far. But what if you want to work with ordinary numbers? Well, single valued numbers (which are really just 1-by-1 matrices) are entered like in any other programming language:

```
rho = 24
```

Single valued numbers *behave like* normal numbers under the usual arithmetic operations (because matrix arithmetic rules *reduce to* ordinary arithmetic rules for single values). So, `2*3 = 6` and `4/2 = 2`, and so on. When you're working with just numbers, you can once again rely on the mathematics you learned in kindergarten.

> **Big Tip**: In short, if you're working with single valued numbers, don't worry about matrix arithmetic rules. (Phew!) Only worry about matrix rules when you're working with matrices.

End of line marker – the semicolon (;). If you end your lines with a semicolon (;), the result of your line is NOT displayed. Consider this:

No semicolon:

```
>> x = 1:5
x =
     1     2     3     4     5
>>
```

With semicolon:

```
>> x = 1:5;
>>
```

So how do you decide when to end your line with a semicolon? Well, generally...

1) **Don't**: When you are issuing commands in interactive mode, you normally want to see the result of your command, so you normally don't want to use a (;) after your command.
2) **Do**: On the other hand, when you are writing a program, you will usually be performing lots and lots of intermediate calculations which you don't want to display, so it is generally a good idea to end your lines with semicolons (;) in your programs.

Other stuff. Some useful commands:

`clc` - clears screen
`clear` - clears variables from workspace

Some predefined constants in Matlab:

```
Pi  = 3.14159, Inf = infinity
```

Some common built-in functions:

```
sin(), cos(), tan(), sqrt(), abs(), exp()
```

Writing programs in M-files: The Hows and Whys

To write M-files, you will need to use the editor. Type `edit` to bring up the editor.

There are two kinds of programs in MATLAB: *scripts* and *functions*. They both end with the extension .m. Let's take a look at what the similarities and differences are:

M-file (scripts)	**M-file (functions)**
1) Filename ends with .m 2) You run them by typing their name, e.g. to run hello.m, just type: `>> hello` 3) Scripts are just files containing a series of MATLAB commands.	1) Filename ends with .m 2) You run them by typing their name, followed by the required inputs (and possibly outputs): `>> hello(2.0, 3.0)` 3) Functions are like functions in mathematics. They accept inputs, process them, and then spit out results.

For now, let us just learn how to write functions. Let's create an extremely simple function called `grog.m`:

```
function [y] = grog(p,q)
      % The Pirate LeChuck says: Har har har and a bottle of rum!
      y = p^2 + q^2;
end
```

Explanation:

1. `y` is the *output variable*, that is, the variable containing the result(s) we want to produce. At the end of your program, all you need to do is to assign the result of your calculations to the y variable, as shown above.[5]
2. The *function name* in this example is `grog`.
3. `p` and `q` are *input variables*.
4. The `%` sign denotes the start of a *comment*. Comments are ignored by MATLAB.
5. It's not required, but it is good practice to end your function with the keyword `end`.

Save it as `grog.m`. To run this M-file, call `grog` with some input values. Let's try giving it `p = 3` and `q = 2`, like this:

```
>> grog(3,2)
ans =
      13.00
```

Incidentally, when you don't specify an output variable, MATLAB automatically uses the variable `ans`, short for *answer*, to store the results of a calculation. If you want to refer to your result later, assign it a variable, like this:

```
>> u = grog(3,2)
u =
      13.00
```

[5] Neat MATLAB tip for advanced users: You can actually have more than one output variable if you want to (most other programming languages don't generally let you do this). For instance, let's say you want to output two results from your function. You can write something like this:

```
function [x,y] = grog(p, q)
      x = p + q
      y = p^2 + q^2
end
```

To call this function, you would type:

```
>> [x,y] = grog(3,2)
```

Conditionals: if, elseif, else

When you need your code to make a decision based on a set of criteria you specify, then you will need what are called conditional statements (`if, elseif, else`). The general structure is as follows:

```
if condition
      ...
elseif condition
      ...
else
      ...
end
```

This example illustrates the use of `if, elseif` and `else` statements.

```
if I == J
  disp('I and J are equal');
elseif I > J
  disp('I is greater than J');
else
  disp('I is less than J');
end
```

This block of code is more or less self-explanatory. It expresses three ideas:

1) If I is equal to J is true, then display "I and J are equal".
2) If I is greater than J is true, then display "I is greater than J".
3) If the above two cases are both false, then display "I is less than J".

There are a few things to note:

1) You must always end your **`if`** statements with an **`end`** statement.
2) The **`elseif`** and **`else`** blocks are optional. You don't have to specify them.

Potential trap – Difference between = and ==

You may be wondering, what's the deal with that double equals sign (==)?
How does it differ from the single equal sign (=)?

Well, simply put:

1) the **single equal sign** (=) is used ONLY for **assigning values (setting a value to a variable)**.
2) the **double equals sign** (==) is used to **check if a condition is true or false**.

Generally, you would use = anywhere in your code *except* when you need to check a condition (typically with IF, WHILE statements). Likewise you must only use == when you need to check a condition.

For instance, if you were to write this:

```
if I = J
 disp('I and J are equal');
```

MATLAB will give you an error, because the code is trying to assign the value of J to the variable I, when it should really be checking the truth of the assertion of whether I == J. For a list of conditional operators, see the One Page Summary.

Looping: for, while

When you want to get your program to do something iterative/repetitive, you need to write a loop. To "loop" is computerese for "to perform a set of actions over and over again".

For-loops. The general structure is as follows:

```
for var = 1:N
      ...
end
```

This is an example of a for-loop (with another for-loop inside it):

```
N = 2
for I = 1:N
  for J = 1:N
    B(I,J) = 1/(I+J-1);
  end
end
```

The above for-loop starts counting from I = 1 all the way to I = N. Inside the first loop, the second loop counts from J = 1 to J = N.

For N = 2, the result of the above loop is:

```
A =
    1.0000    0.5000
    0.5000    0.3333
```

For-loops are often used when you need repeat a set of actions N times (where N is a number that is known beforehand).

While-loops. While-loops are slightly different. Unlike for-loops, you don't have to know in advance the number of times you need to repeat a set of actions. While-loops will just repeat forever until the given condition becomes false.

The general structure is:

```
while condition
      ...
end
```

Consider this piece of code.

```
G = 5
while G >= 0
  G = G - 1
end
```

G starts off at 5 and shrinks by 1 at every iteration. The loop is supposed to stop as soon as G turns negative. The results are:

```
G =     5
G =     4
G =     3
G =     2
G =     1
G =     0
G =    -1
```

Notice that when G finally becomes negative, the condition `G >= 0` becomes untrue, so the loop ceases to continue.

Plotting graphs

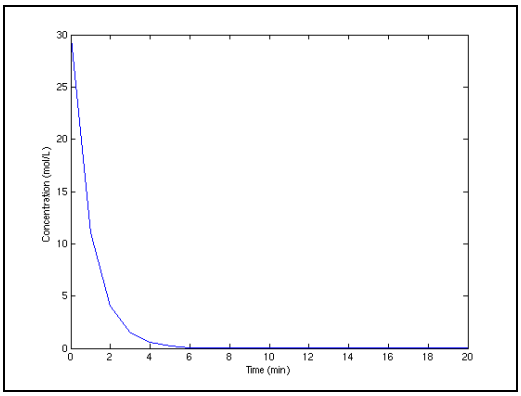
Let's say you want to plot Concentration of A (`Ca`) vs. time (`t`), given the relationship $Ca = 40e^{-t}$, for time 0 to 20.

```
>> t = 0:20;
>> Ca = 40*exp(-t);
```

To plot the values, just type:

```
>> plot(t,Ca)
```

Notice that in the `plot` command, the independent (x-axis) variable comes first. That is to say, you type `plot(t,Ca)` instead of `plot(Ca,t)`. This may seem a trivial thing, but it has tripped some folks in the past. Finally, you will get a graph that looks something this:



You can add labels to x and y axes like this:

```
>> xlabel('Time (min)')
>> ylabel('Concentration (mol/L)')
```

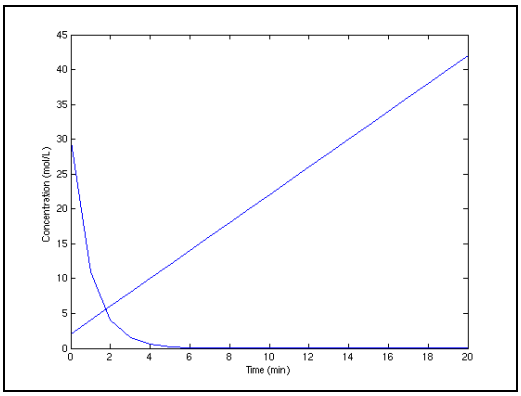
Multiple curves on the same plot. If you want plot another graph one *on top of the current one*, type:

```
>> hold on
```

`hold on` holds the current plot and all axis properties so that subsequent graphing commands simply add to the existing graph. Then you can proceed to plot another graph using the `plot` command.

```
>> z = 2*t + 2;
>> plot(t,z)
```

Et voila!



Saving data

Occasionally, you may want to save the matrices that you generate. To save matrices (say x, y and z) to a file called mydata, just type:

```
>> save mydata x y z
```

To load these matrices, type:

```
>> load mydata
```

Further References

If you want to learn more, there is a plethora of MATLAB tutorials on the web.

Here's a good starting point:
`http://www-h.eng.cam.ac.uk/help/tpl/programs/matlab.html`

Here's a terse one:
`http://www.math.tamu.edu/~mpilant/math308H/Matlab/QuickStart.html`

For others, just enter this search string in Google™:
`matlab tutorials | guides | introduction | "quick start"`

CHE3E04 - MATLAB Hands-on session

1) Start MATLAB.
2) **Plotting a graph.** Let's plot a simple graph: $y = \exp(x)$ in the interval [-3, 3], spaced in 0.5 unit intervals
First, set up the x-axis (which spans [-3, 3] in 0.5 unit intervals):

```
>> x = -3:0.5:3
x =
  Columns 1 through 12
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000
  Column 13
    3.0000
```

Now write the function:

```
>> y = exp(x)
y =
  Columns 1 through 12
    0.0498    0.0821    0.1353    0.2231    0.3679    0.6065    1.0000
1.6487    2.7183    4.4817    7.3891   12.1825
  Column 13
   20.0855
```

3) To plot this function, type:

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

Close the figure.

4) **Writing an M-file (script).** To automate this process, you can write an M-file. Start the MATLAB editor by typing:

```
>> edit
```

5) Type this code into the editor: (Notice the semicolons (;))

```
x = -3:0.5:3;
y = exp(x);
plot(x,y);
```

6) Save the file as `myplot.m`. You just created a MATLAB M-file (script).
7) Close the editor and go back to the MATLAB Command Window.
8) **Running an M-file.** To run the file, just type:

```
>> myplot
```

9) **Writing an M-file (function).** Let's create another type of M-file (a function). A function is a type of M-file that can take inputs and produce outputs. Start the MATLAB editor again by typing `edit`.We'll create a function that calculates the square of a real number. Type this code into the editor:

```
function out = square(x)
      out = x^2;
end
```

10) Save the file as square.m. Close the editor and go back to the MATLAB Command Window.
11) Let's try it out:

```
>> square(5)
```

```
ans =
    25
```