Introduction to Matlab
by Dr. Adam Larios

It is important to type all of the code as you go along. You will learn it better this way: your brain will make connections based on what your hands are physically typing.

1. Matlab As A Calculator

Let's first learn the basic operations in Matlab. We will type them into the command window at the prompt which looks like "`>>`". For example, to compute `3+5`, we would do:

```
>> 3+5
```

(don't type the "`>>`") and hit `[Enter]`. Try these as well:

```
>> sqrt(2)
>> 3^2
>> 2*5
>> sin(pi)
```

Notice the last one gives you a result of `1.2246e-16`. This is computer short-hand for scientific notation, with the "`e`" meaning "times 10 to the power of". So,

$$\texttt{1.2246e-16} = 1.2246 \times 10^{-16} = 0.00000000000000012246$$

But, shouldn't $\sin(\pi) = 0$? What is going on here is that computers typically only have about **16 digits of accuracy**, so to a computer, `0` is the same thing as `0.00000000000000012246`, and all smaller positive numbers. This usually doesn't cause trouble, but it is important to be aware of because it can sometimes cause unexpected bugs in your code.

1.1. Comments.

"Comments" are parts of a program that the computer doesn't read. They are intended for human use only, and can make a program much easier to read. **Good code is well-commented code.** In Matlab, comments are made by the "`%`" sign like this:

```
>> 2 + 3 % The text after the '%' sign does nothing.
>> % This line does nothing at all.
```

2. Variables

In mathematics, an equals sign "`=`" means "is equal to", but in Matlab (and most computer languages), it is used in a different way; namely, it is used for assignment. Try:

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

The first line assigns `x` the value of 5. The semi-colon "`;`" suppresses output (try it without the semi-colon too). The second line tells Matlab to print the value of `x`. This should be fairly straight-forward. However, consider the following code:

```
>> x = 5;
>> x = x + 2;
>> x
```

Each line must be entered *sequentially*, i.e., in order starting from the top line. Here, the second line does not make much sense in mathematics, because in an equation like "$x = x + 2$", we could cancel the x to get $0 = 2$, which is false. However, remember that the

"=" sign to a computer means assignment, so it computes the right-hand side `x + 2` first. Since `x` is equal to `5`, the result is `5 + 2` or `7`. The computer then assigns this value to the left-hand side, which is `x`, so now is equal to `7`. Try these as well:

```
>> x = 10;
>> myVariable = 2.5 + 1e1; % Remember: 1e1 means 10^1 or 10
>> myVariable + x^2
```

3. Matrices and Vectors

Matlab is very good at handling matrices and vectors. In fact, Matlab stands for "**Mat**rix **Lab**oratory." In Matlab, everything is a matrix (also called an "array"). For example, numbers (i.e., scalars) are really 1×1 matrices. There are a few different ways to input matrices and vectors. Try the following examples:

Time-Saving Hint: Use the up-arrow ↑ on your keyboard to cycle through old commands.

```
>> u = [4 5 6]
>> v = [4,5,6]
>> x = [4;5;6]
>> A = [1 2 3; 4 5 6; 7 8 9]
>> M = [1,2,3; 4,5,6; 7,8,9]
>> A*x  % Matrix time vector.
>> u*x  % Row matrix times column vector.
>> A'   % The transpose of A. It switches rows and columns.
```

3.1. Matrix and Vector Operations.

As you can see form the last two examples, matrix multiplication works like it usually does in linear algebra. However, sometimes we will want to treat matrices and vectors as just tables of numbers, and we want to do element-wise operations on them. This is done my putting a dot before the operation. Try this:

```
>> [4 5 6].^2
>> [1 2 3; 4 5 6; 7 8 9].^2
>> u = [1 2 3];
>> v = [4 5 6];
>> u.*v
>> 2.^u
```

Matlab is smart enough to apply many common operations element-wise. For example:

```
>> t = [-pi, -pi/2, 0, pi/2, pi];
>> cos(t)
>> sin(t)
>> exp([1 2 3])
>> log([1 exp(1) 10 exp(2)])
```

Note: In Matlab "`log`" is base $e \approx 2.71828 =$ `exp`(1).

Matlab also makes many common operations easy. For example:

```
>> x = [-3.7 0 5.5 -9];
>> max(x)
>> min(x)
>> abs(x)
>> max(abs(x))
>> min(abs(x))
>> sum(x)
>> norm(x) % The squart-root of the sum of the squares of x.
>> sqrt(sum(x.^2)) % Another way to compute the norm of x.
```

3.2. **Building Matrices and Vectors.** Matlab has many built-in features to generate vectors easily. Let's make a list of numbers from 3 to 10. We can do it like this:

```
>> x = 3:10
```

If we want to "step" or "skip" by a certain amount, we put the skip-size in the middle:

```
>> 3:2:10
>> 3:0.5:10
```

It is very common to want a certain number of points between two fixed numbers. To get a vector with exactly 16 points between (and including) 3 and 8, use `linspace` like this:

```
>> x = linspace(3,8,16)
```

This saves time, since you don't have to figure out what the step-size will be, but you could also do it like this:

```
>> delta_t = (8-3)/16;
>> x = 3:delta_t:16
```

(Check out `logspace` as well.) Here are some shortcuts to generate special, useful matrices:

```
>> rand(2,3)  % A random 2 by 3 matrix.
>> eye(3,3)   % The 3x3 identity matrix.
>> zeros(2,3) % A 2 by 3 matrix of all zeros.
>> ones(2,3)  % A 2 by 3 matrix of all ones.
```

3.3. **Accessing Matrices and Vectors.** Often, we want to read or edit the entries of a matrix or vector. We can do this by passing the "index" of the component we want. Matlab indices start at 1. (Some languages, like C and C++, start at 0. Fortran starts at 1, like Matlab.) Try these examples:

```
>> v = [4 5 6];
>> v(2)
>> v(2) = 3*v(2);
>> v
```

... and continue with these examples:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(2,3)
>> A(2,:)     % This ':' tells Matlab to use all columns
>> A(:,2)     % This ':' tells Matlab to use all rows
>> A(2:3,2:3) % This gives a 'submatrix' of A
>> A(:,3) = [0; 0; 0]
>> A(:,3) = ones(3,1)
>> A(:,2) = v' % Remember: v' turns the row vector v into
               % a column vector
```

4. Getting Help

Matlab has a built-in help feature for almost every function. You may have to scroll up to read everything it outputs. For a more graphical (but sometimes slower) version `help`, type `doc` instead of `help`. Try these:

```
>> help max
>> help abs
>> help logspace
>> doc sin
```

Note: Professional programmers use Google all the time! Google is your friend. Men and women who gain skill at Googling their programming issues are men and women who get better at programming. One tip is to Google the exact or nearly exact error output from an error message, preferably using copy/paste.

5. Plotting

In Matlab, plotting works differently than in the symbolic-based math programs you may be used to. Here is the key point to keep in mind: **In Matlab, you have to tell it every point in the plot!** First, build a vector of the independent variables, and call it `x` (for example). Then, we build another vectors of `y`-values. These points are then plotted one-by-one. Below is an example. We type "`close all`" first to get rid of any other possibly open plot windows.

```
>> close all;
>> x = linspace(-2,2,10);
>> y = x.^2;
>> plot(x,y)
```

You can see that Matlab is just connecting the dots. To get better resolution, try `x = linspace(-2,2,100);` or `x = linspace(-2,2,1000);`. Next, you can plot in different colors and different symbols. Try the commands below, after inputting an `x` and `y` as above. **Remember to type `close all` each time!**

```
>> close all;
>> plot(x,y,'g'); % plot in green
>> close all;
>> plot(x,y,'*'); % plot with stars
>> close all;
>> plot(x,y,'-*'); % plot with stars connected by lines
>> close all;
>> plot(x,y,'b-*'); % plot with stars connected by lines in blue
>> help plot % find out about other plotting options
```

5.1. **Multiple plots.** Matlab will wipe out all previous plots each time it plots something new. To avoid this, we use “`hold on`” like this:

```
>> close all;
>> x = linspace(-2,2,100);
>> plot(x,x.^2,'r');
>> hold on;
>> plot(x,x.^3,'b');
>> axis([-1, 1, -2, 4]);  % [x_min, x_max, y_min, y_max]
```

5.2. **Labeling and Legends.** Labeling your plots and plot axes is very good practice as a scientist, mathematician, engineer, etc. Let’s see how to do this in Matlab.

```
>> close all;
>> t = linspace(-pi,pi,100);
>> plot(t,cos(t),'r');
>> hold on;
>> plot(t,sin(t),'b');
>> title('Input and Response');
>> xlabel('Time');
>> ylabel('Amplitude');
>> legend('Input signal','Response Signal');
```

The plot may have gone behind the main Matlab window, so try minimizing some windows to find it.

5.3. **Subplots.** Matlab has a wonderful ability to make different plot windows in the same figure. Try the following.

```
>> close all;
>> x = linspace(0,2*pi):
>> subplot(1,2,1);
>> plot(x, sin(x));
>> subplot(1,2,2);
>> plot(x, cos(x));
```

The first two arguments of `subplot` tell the number of rows and columns for the different plots. The last argument tells Maplab which plot num ber is the current plot. For more information, type `help subplot`.

6. Saving work in m-files

As can be seen from the example above, we often have many lines of code, and we don't want to have to enter them into the command line each time. Let's put these in a file. Matlab files end with ".m". Open the editor, and save a new file as "myTest.m". Type this in it (the numbers on the left are just line numbers; don't type them).

```
1  x = 5;
2  x = x + 2;
3  x
```

You can now run this script (m-file) by pushing the green "▷"(run) button.

Tip: If you ever write some code, and then want to change it, you don't have to delete it! A common technique is to just "comment it out" by highlighting the text and pressing the "Comment" button in the menu. Uncomment it with the "Uncomment" button.

7. Loops

Loops are one of the most useful constructs in programming. They often come up when you have to do something repetitive many times, making only small changes each time. Let's look at a basic "`for` loop" to get started in an m-file (call it `loopTest.m` or something).

```
1  x = 0;
2  for i = 1:10
3      x = x + i
4  end
```

Let's break down the parts of this loop to get the idea. We start by setting `x = 0`, since we are going to use it later. Then, lines 2 and 4 tell Matlab that it is going to start and end a loop. Note that the Matlab keywords `for` and `end` get highlighted differently when you type them. This is called "syntax highlighting." It makes code easier to read.

This loop will repeat the instructions on line 3 each time it is run. The number of times it will run is determined by the "`i = 1:10`" after the "`for`" in line 2. On the first pass through the loop, i will equal 1. On the second pass, it will equal 2, and so on, until it counts all the way up to 10. (Recall that in Matlab `1:10` means the vector `[1 2 3 4 5 6 7 8 9 10]`, so Matlab is just running over all of those values in order).

Let's analyze what will happen to `x` as we go through the loop. Before the loop, `x` is set to zero, so on the first pass, `x = 0` and `i = 1`, so `x+i` will equal `0+1=1`, and line 3 sets `x` to this value. Matlab then goes back up to line 2, and updates `i` to the next value. So, as we start this next pass, `x = 1` and `i = 2` (remember, `i` just counts up by one each time). Then, line 3 will execute *again* so `x+i` will equal `1+2=3`. Continuing in this way, the outputs will be 1, 3, 6, 10, 15, 21, 28, 36, 45, and 55. Try it!

What if we didn't start with the line `x = 0`? This can cause many problems. If `x` has not been defined yet, the right-hand side of line 3 would not make any sense (since there would be no `x` to add to `i`), and Matlab will raise an error. Even worse would be if no error arises, because `x` was somehow defined somewhere else, and now Matlab will use whatever value `x` happens to be, likely yielding a wrong result but no error to tell you something is wrong. This kind of bug can be hard to catch! We must be very careful before we trust the output of a program.

7.1. **Nested loops.** A loop within a loop is called a nested loop. These arise very often. Here is an example which defines a certain matrix.

```
for i = 1:3
    for j = 1:3
        A(i,j) = i - j;
    end
end
display(A);
```

The "`display(A);`" line is a more formal way to output the matrix, but we could have just written "`A`" without a semi-colon to output it as well.

7.2. **Initialization.** The above construction is inefficient, because the matrix keeps growing as the loop runs. First, it is a 1×1 matrix, then when `A(2,1)` is assigned, it becomes a 2×1 matrix, and Matlab does this by *copying* the old 1×1 matrix into a 2×1 matrix, which is an unnecessary step. If the matrix is really big, this can really slow down your program. However, if, we just tell Matlab beforehand what size we want the matrix `A` to be, it doesn't need to grow the matrix, and will run much faster. You can do this by first making `A` be a zero-matrix of the correct size, like this:

```
1 A = zeros(3,3);
2 for j = 1:3
3     for i = 1:3
4         A(i,j) = i - j;
5     end
6 end
7 display(A);
```

> **Tip:** In Matlab, the most efficient way to access a matrix is first by the rows, then by the columns (similarly in Fortran). If we switched lines 2 and 3 above, the result would be the same, but slightly slower. In C/C++, the order is reversed.

7.3. **Tracking loop iterations.** Sometimes, it is convenient to index by one set of values, and loop over another. For example, we can plot several polygons like this:

```
% A program to plot several polygons.
i = 1;  % Initialize a counter to index the plots.
for N = 3:10
    theta = linspace(0,2*pi,N+1);
    subplot(2,4,i);
    plot(cos(theta),sin(theta));
    axis('square');
    title(sprintf('N = %g',N));
    i = i + 1; % Update the counter.
end
```

Note that the counter `i=1,2,3,...`, but `N=3,4,5,...`. Therefore, we don't always need to use the loop iteration as an index, and sometimes is it convient to keep them separate.

The `sprintf` ("string print format") command is useful for the `%g` command tells Matlab, "put the following number here."

8. First basic program: Fibonacci Numbers

Let's compute the first few Fibonacci numbers. The first two are 1 and 1. After that, the next number is always the sum of the previous two, so the third number is 1+1=2, the fourth is 1+2=3, and the fifth is 2+3=5, then 8, 13, 21, 34, and so on. To compute these, we need to store the old values. We can do this in a vector (called "`fib`", and then loop over the first few numbers, like this:

```
fib(1) = 1;
fib(2) = 1;
for i = 3:10
    fib(i) = fib(i-1) + fib(i-2);
end
display(fib);
```

This will do the job, but we can make it even better with these fixes:

- The loop is inefficient because we did not pre-declare the vector. We can fix this by putting "`fib = zeros(1,10);`" at the top of the program.
- We should also comment the code to tell the user what it does. We shouldn't comment every line, but just enough so that we can see what is going on.
- Another fix is that we have "hard coded" the number 10 in the program. What if we want more numbers? We would have to dig through the code and change each place where the 10 occurs (after the above fix, it now occurs in two places). If we make a mistake, the code will have a bug. It is easier to give it a name (say, "`N`"), and then use it.

Tip: In general, you should try to **avoid hard-coding** as much as possible. Even if you are only going to use a number twice, give it a variable. Often, you should name something even if you only use it once. This can help make it more clear for the user of the code (which may be your future self!).

The above fixes now give us a better program:

```
% A program to compute the first N Fibonacci numbers.

N = 10; % The number of Fibonacci number to compute.

fib = zeros(1,N);

fib(1) = 1;
fib(2) = 1;
for i = 3:N
    fib(i) = fib(i-1) + fib(i-2);
end
display(fib);
```

Finally, let's look at a different approach to this problem. Maybe we only want the N^{th} Fibonacci number, and we don't want to store all the previous values. A little thought let's us see that we can do this by using two auxiliary variables to hold the previous two values, call them "`fibLast1`" and "`fibLast2`".

```
1  % A program to compute the Nth Fibonacci number.
2
3  fibLast1 = 1; % initialize
4  fibLast2 = 1;
5  fib = 1;       % in case N == 1 or N == 2
6  for i = 3:N
7      fib = fibLast1 + fibLast2; % Compute next fib number
8      fibLast2 = fibLast1; % Update fibLast2 to fibLast1
9      fibLast1 = fib;      % Update fibLast2 to fib
10 end
11 display(fib);
```

Question: What would happen in we switched lines 8 and 9? Would the code be correct still? Remember, Matlab process each line *sequentially*; that is, one after the other.

Tip: The overwriting done in the update steps in lines 8 and 9 are very common in programming! You will often see a mathematical statement written with indices, such as $f_i = f_{i-1} + f_{i-2}$. This does *not* mean your code must contain indices! It is often far better to overwrite previous data whenever it is not longer needed, as in the code above. This can save memory, especially when problems become large.

9. Conditionals (if/then statements)

Another major programming construct involves "`if`" statements, which often have an `else` statement with them. Try this example to get the idea:

```
1  x = 3;
2  if x == 3
3      display('I love Matlab!');
4  else
5      display('I love Corn Flakes!');
6  end
```

Try running this, then change the first line to something else, like `x = 5`, and run again.

Note the double equals sign, "`==`" on line 2. This is more like the traditional mathematical equality. It does not set `x` equal to 3, it *checks* if `x` is equal to 3. For example, Matlab will return a value of "true" for `3 == 3`, and a value of "false" for `3 == 5`. (Matlab uses the number 1 for true, and the number 0 for false.) You can also use inequalities. Try these:

```
>> 3 == 3 % Is 3 equal to 3?
>> 3 == 5 % Is 3 equal to 5?
>> 3 < 5  % Is 3 less than to 5?
>> 3 < 3  % Is 3 less than to 3?
>> 3 <= 5 % Is 3 less than or equal to 5?
>> 3 >= 5 % Is 3 greater than or equal to 5?
>> 3 >= 3 % Is 3 greater than or equal to 3?
>> 3 ~= 3 % Is 3 not equal to 3?
>> 3 ~= 5 % Is 3 not equal to 5?
```

9.1. Breaks and While Loops.

Sometimes, we don't know how long a loop should run. Suppose we want to find the biggest Factorial number less than 200. We could do it like this;

```
% Find the largest factorial less than N.
N = 200;
fac = 1;
for i = 1:N
    fac = fac*i;
    if fac > N
        break;
    end
end
display(sprintf('%g! = %g',i,fac/i));
```

This can also be achieved with a `while` loop, like this:

```
% Find the largest factorial less than N.
N = 200;
fac = 1;
i = 1;
while fac < N
    i = i + 1;
    fac = fac*i;
end
display(sprintf('%g! = %g',i-1,fac/i));
```

A `while` loop will run so long as its conditional is true; in this case, so long as `fac` is less than `N`. Note that `for` loops and `while` loops are similar, but `while` loops have a greater danger of becoming infinite loops that never stop. If this happens, press `[CTRL + C]`.

10. Functions

Functions are very useful in Matlab. A function is like a mini-program with an input and an output. Sometimes, you want to do a complicated operation many times. It can often be useful to put this into a function.

10.1. **Programming the Factorial Function.**

Let's build the factorial function. To do this, open a new m-file, and call it "`factorial.m`" (In Matlab, the function name has to match the file name.)

```
function value = factorial(N);
% A function to compute the factorial of N.
% N! = 1*2*3*...*(N-1)*N

value = 1;
for i = 1:N
    value = value*i;
end

end % End of the function.
```

Note how the input is `N`, so `N` does not need to be defined (the user will pass it into the function). Also, the output is whatever the final value of "`value`" is.

To call this function, make sure you are in the same directory as the file. Then type in the command window:

```
>> factorial(5)
```

and you should get an answer of 120 (= 5!). This function isn't bad, but we could make it better. What if the user passes in a zero? Recall that $0! = 1$. Also, we can throw an error if the user tries to input a negative number. This can all be handled using conditionals:

```
function value = factorial(N);
% A function to compute the factorial of N.
% N!=1*2*3*...*(N-1)*N

if N < 0
    error('Input must be a postive integer!');
elseif N == 0
    value = 1;
else % The code below computes the factorial if N>0.
    value = 1;
    for i = 1:N
        value = value*i;
    end
end

end % End of function factorial.
```

Note the use of the "`elseif`" keyword.

10.2. **Two-variable functions: Programming the choose function.** Recall the choose function from probability:

$$_nC_m = \binom{n}{m} = \frac{n!}{m!(n-m)!}$$

We can use our factorial function to code this! Both programs need to be saved in the same folder and run from that folder, so that "`choose.m`" can call "`factorial.m`". The choose function also requires two inputs. We can make a choose function like this:

```
function value = choose(n,m);
% Compute "n choose m", i.e, n!/(m!(n-m)!)

if n >= m
  value = factorial(n)/(factorial(m)*factorial(n-m));
else
  value = 0;
end

end % End of function choose.
```

Test this program to compute $_7C_4 = 35$.

10.3. **Multiple Outputs.** Multiple outputs are also common:

```
function [v,p] = freeFall(a0,v0,p0,t);
% Compute velocity and position of an object in free fall.

v = a0*t + v0;
p = (a0/2)*t^2 + v0*t + p0;

end % End of function freeFall.
```

You can call this code from the command line or another program (in the same folder as the freeFall function), like this:

```
a0 = 9.81;
v0 = 0.0;
p0 = 0.0;
t = 0.2;
[v,p] = freeFall(a0,v0,p0,t);
display(sprintf('At t = %g, pos = %g, and vel = %g.',t,p,v));
```

10.4. **Symbolic functions.** Sometimes, it is convenient to have a simple calculus-type function defined on the fly. Matlab can do this using "function handles," provided by the @ symbol, like this:

```
>> f = @(x,y) x^2+y^2; % Define f as a function of two variables
>> f(2,3);              % Evaluate f at the point (2,3).
```

Here is an example of a vector-valued function:

```
>> f = @(t,y) [ -0.5.*y(1)+y(2)+1 ; -y(1)-0.5.*y(2)+2 ];
```

In general, using functions like this is fairly slow (function calls are expensive), so it is best to avoid them if possible, especially if it is in a loop. For example, rather than using function handles to take a symbolic derivative in Matlab, instead, compute the derivative by hand (or compute it once using Matlab or Mathematica or Maple or something), but then use the result directly.

11. Miscellaneous Tips

(1) **Always properly indent your code! Otherwise, it can be very hard to read. To automatically indent it, push: `[CTRL+A]` then `[CTRL+I]`. Do this every few minutes, and every time before you show anybody else your code!**
(2) Don't use spaces in file names. Use underscores, such as `my_cool_file.m` or "camel case," such as (`myCoolFile.m`).
(3) Don't use filenames like `project.m`, `project_new.m`, `project_new2.m`, `project_new2_old.m`, and so on, for obvious reasons. Instead, name files by date in `YYYYMMDD` format. They will self-organize this way. For example, for a file made on Sept 3, 2015, call it something like `myFile20150903.m`. You could also do something like `my_file_2015_09_03.m`. (If you want to get really fancy, you can look up "Unix time" and use it to name files.)
(4) Use short-but-descriptive names for variables. If you name everything `a, b, c, aa, bb, cc, x, xx, xy, xx2, xx3`, and so on, it can be very hard to keep track of what you are doing. Better names might be something like: `delta_t`, `y_old`, `y_new`, `max_error`, `norm_error`, and so on. You could also use camelCase here too.
(5) Save a copy of your file every time you work on it! The best way to do this is with a "versioning system" like git or svn (Google "git basics" or "svn basics", etc., if you want to learn more), but you should try to keep copies of all your old versions somehow, at least organized by date.
(6) Matlab uses unusual keys for copy/paste. You can use the more standard `[CTRL+C]`, `[CTRL+V]`, etc. by going in the menu to:

$$\text{Home} \longrightarrow \text{Preferences} \longrightarrow \text{Keyboard} \longrightarrow \text{Shortcuts}$$

and changing "Active settings" from "Emacs Default Set" to "Windows Default Set".
(7) Time things with `tic` and `toc`:

```
>> tic;
>> inv(rand(20,20));
>> toc
```

This finds how long it takes Matlab to generate a random 20×20 matrix and compute its inverse.
(8) **Matlab stuck?** Overheating your computer? Caught in an infinite loop? Just hit:

`CTRL + C`

while in the command-line window to kill Matlab's current operation.