From Ione Fine & Geoff Boynton's MATLAB for the Behavioral Sciences
Adapted for Mehtalab by David Groppe

Chapter 1
Topic 1 – What is programming?

Programming is telling a computer what to do. There are only 2 tricky things

1) Computers are very stupid – you have to tell them exactly what to do

2) Computers don't speak English. Any programming language is a compromise between the computer's native language (0010001) and your native language (English). High level programming languages are closer to English, low level languages are closer to computer-ese. The closer a language is to English, the easier it tends to be to program, but the slower it is for the computer to interpret it and the more constrained it is on what it can do. Low level languages tend to be hard to program but very fast to run, and less constrained. Matlab is a mid- to high level language.

Like any language, programming languages have grammar. Like real languages some things are easier to say in one language rather than another (Italian is the language of love etc. etc.). Some languages are better for computations, others for graphics. Unlike people who speak real languages (with the exception of the French) computers are very fussy about grammatical errors. Like learning a real language, at first it will be hard to say the simplest thing, but it will get easier, fast.

Hardware
Hardware is the physical presence of the computer. The monitor, the hard drive, the CPU (central processing unit, i.e. the "brain"). Hardware is anything you can damage by poking it with a screwdriver.

Software
Software = programs. Programs are instructions to your computer to behave in a particular way. So a software program like Microsoft Office gets all machines to behave in a particular way – the instructions are slightly different for different computers (different hardware), but the program makes all computers behave (almost) the same.

All software is written in a programming language. Some programs, (like Matlab) are there to help you write new programs. Matlab will run on Macs, PC, and UNIX. But some of the commands only run on some computers.

NOTE – these days it is almost impossible to damage your computer by writing a program. The computer usually makes it very hard for you to do anything that will damage it. In this book you won't be using any commands that can do any permanent damage. So crash your computer hard. Have fun!

Getting Started

You need:

1. A PC running Windows or a Mac computer running OSX.

WARNING: This book is not designed to be compatible with Macs running Mac OS9 or earlier. Most of the book will work with Mac OS9 anyway, but if you are using Mac OS9 then where we differentiate in the code between PCs and Mac, pretend you are a PC.

2. You need to install Matlab on your computer. The student version is fine.
You may run into license manager problems installing Matlab Mac OSX since it's really confusing and badly designed – just call Mathworks' technical support number and they will sort you out.

To start Matlab from a Linux or Max terminal window, type:

```
> matlab &
```

A window will appear that's divided into a number of sub-windows. For the time being, ignore (or close) all the sub-windows except the one that called the "command window" which has a little prompt >>. Later you may find these other windows helpful. By the way, the "&" after "matlab" tells Linux to run Matlab in the background of the terminal window, which enables you to use the terminal window for other things while it's running Matlab.

[**Side note:** It is possible to run Matlab in a terminal window without the Matlab windows. To do this enter the following in a terminal window:

```
> matlab -nojvm
```

This can be useful when running Matlab remotely through an ssh connection as the Matlab windows can be very slow and clunky]

OK – the line with >> is the Matlab command line. It is analogous to the Unix command line. When you type things into the command line (e.g., function calls, script names, simple commands), Matlab tries to execute them right away. For example if you enter:

```
>>str1='I have no clue what I am doing'
```

You just told the computer to create a list of letters *'I have no idea what I'm doing'* and to name that list of letters *str1*. *str1* is a ***variable*** - The single quotes tell the computer that str1 is a list of letters (not numbers, more on that later). Any list of letters is called a ***string*** or a ***character array.*** After you executed the command, Matlab output the result of that command in the command window. In this case:

```
str1 =

I have no clue what I am doing
```

To suppress such output, end a command with a semicolon. To demonstrate, press the **up arrow key** to get back to the previous line.

```
>>str1='I have no clue what I am doing';
```

Now type:

```
>>who
```

This command asks your computer to give you a list of all the variables you have in your Matlab workspace. At the moment the only variable you have is *str1*. You can get a bit more information about the variables in your workspace by typing:

```
>>whos
```

whos tells you not only the name of all the variables in your workspace, it also tells you what kind of variables they are (e.g., characters or numbers), their dimensions (e.g., *str1* has 30 characters in it, and how much memory they occupy (e.g., 60 bytes--a drop in the bucket).

Typing the name of a variable asks your computer to tell you what is contained within that variable.

```
>>str1
```

The computer should show you what's in str1

```
str1 =

I have no clue what I am doing
```

Using the function *disp* also displays what a variable is, but only shows the contents, instead of repeating the name of the variable:

```
>>disp(str1)
```

So all the computer does is display the contents of the variable as follows:

```
I have no clue what I am doing
```

Let's create a new variable:

```
>>str2='Is it all going to be boring?';
```

Now type:

```
>>who
```

Now you have both str1 and str2.

You can change what's stored in a variable:

```
>>strl='I still have no clue what I am doing';
```

Now type:

```
>>strl
```

See - the list of letters contained within *str1* has changed.

You can also access select parts of a variable.

```
>>strl(3)
```

You've asked the computer to display the third letter in *str1*. This is called ***indexing*** or ***subscripting***. 3 is an index (or subscript) into the third character in *str1*. Now try:

>>str(6)

You can see from this that the computer is counting spaces

```
>>mixstr=strl;
```

Now you've created a new variable called *mixstr*. You've told the computer to make *mixstr* the same as *str1*.

```
>>mixstr
```

See, they are exactly the same.

```
>>mixstr(3)=strl(l)
```

Now you are telling the computer to make the 3rd letter in *mixstr* the same as the 1st letter in *str1*.

```
>>mixstr(l)=strl(3)
```
Now make the 1st letter in *mixstr* the same as the 3rd letter in *str1*.

```
>>mixstr
```

You should now have:
'h Iave no idea what I am doing'

You can also create lists of numbers. These are called ***arrays*** or ***vectors***. Here are four different ways of creating a vector list that goes from 2 to 9 in steps of 1.

```
>>arrayl=[2 3 4 5 6 7 8 9]
```
Here the brackets tell Matlab that you want everything in between them to be

concatenated together into a single list.

```
>>array1=linspace(2, 9, 8)
```

Here you are saying you want a list of 8 numbers that are evenly spaced between 2 and 9. You can imagine that this command would be useful if you had collected 8 pieces of data evenly spaced between 2 and 9 seconds.

```
>>array1=2:1:9
```

Here you are saying that you want a list of numbers that goes from 2 to 9 with a step-size between each number of 1. You can imagine that this command would be useful if you collected data that went from 2 seconds, to 9 seconds, and had collected data every second. Actually, Matlab assumes a default step-size of 1, so you can simply skip it for this particular way of creating vectors.

```
>>array1=2:9
```

Here are the three ways of creating a list of numbers that goes from 1 to 17 in steps of two.

```
>>array2=[1 3 5 7 9 11 13 15 17]
>>array2=linspace(1,17,9)
>>array2=1:2:17
```

You can also index vectors. 2 indexes the second integer in array2:

```
>>array2(2)
```

You can also index more than one number in an array or string.

```
>>array2(2:4)
```

Gives you the 2nd through 4th elements of *array2*. In contrast, the following gives you the 2nd and 5th elements of *array2*:

```
>>array2([2 5])
```

The *disp* function we used earlier can also be used to display numbers

```
>>disp(array2)
```

By now you should be getting a little irritated with having to type in every command one at a time. We are therefore going to create a ***program*** – a program is simply a document containing a sequence of commands.

In Matlab programs are written in documents called ***m-files***. So now we are going to put

the commands you just did in a m-file.

Before going on though, one handy pointer. If you ever get stuck on the MATLAB command line or if some program is running in MATLAB and you want it to stop, the all purpose panic button is *Ctrl-c*. Pressing the *Ctrl* and *c* keys will break out of most anything MATLAB is doing and return to control of the command line to you.

Creating an m-file

Make sure the command window is at the front. Now go to the menu bar and choose *File->New->Blank M-file.*
You'll get a blank document in a new editor window.

Every program begins with a few lines of documentation. This is called a ***header***. Good headers contain the following information
1) The name of the program
2) A description of what it does
3) Who wrote it, and when

OK, type the following header into your m-file. Make sure every new line begins with a %. The % tells the computer to ignore that line as it's a ***comment*** – these comments aren't commands for the computer, they're English to help users understand how to use the program. Commented text will probably show up as being green.

```
% MixStrings.m
%
% Replaces 'this is gibberish' with a
% string of xxxxxx's
%
% Written by IF & GMB 4/50025


% Note, because there is an empty line with no % above it,
% this line is not part of the header
```

Headers are important. You may think that you will remember the programs you write – but trust me, you won't! Getting in the habit of having good up-to-date headers is like flossing – it's boring but it will save you a lot of pain in the long run.

Now we need to save the file. Make a folder called *MYMATLAB* somewhere. Create a subfolder called *MISC*. **Don't** put these folders inside the Matlab application folder or you will lose everything if you reinstall Matlab. Note, when running MATLAB on Linux, you might have to add the extension *.m* at the end of your m-file's filename. On some platforms (e.g., Macs) this is automatically done for you.

Save the file as MixStrings.m (the same as the header) in the *MISC* folder. Note, that every time you edit a file in the MATLAB editor, you need to save the file for the changes to take effect. Until you save the file, the changes exist only in a temporary copy

of the file in the MATLAB editor and this copy can't be executed.

Now go back to the command window and type

```
>>help MixStrings
```

You will almost certainly get an error message:

```
MixStrings.m not found.
```

When the computer says that a file is "not found" that means the computer can't find an m-file that has that particular name. The reason the computer can't find the file even though you saved it in the folder 'MISC' is because the computer is only allowed to look for files in certain places.

One place that the computer always looks for files is the ***current directory*** or ***working directory***. In fact this is the first place that the computer looks. When Matlab opens it automatically links to a particular folder (the default setting is made by Matlab). If you don't tell it otherwise it will save files to that folder. You can see what folder Matlab thinks is the current directory using the ***print working directory*** command:

```
>>pwd
```

The other places that the computer can find files are in folders that are in Matlab's ***search path***. This ***path*** is a simply a list of folders that the computer is allowed to look in whenever it is trying to find a file. Again Matlab comes with a default set of paths. You can get a list of the current folders in the path very easily:

```
>>path
```

To tell the computer where to look, you ***set the path*** …

Setting the path via the menu bar

Make sure the command window is at the front, and go to:
File->Set Path
in the menu bar. A pop-up window will appear Choose *Add With Subfolders,* then choose the *MISC* folder, and click *OK*. Then choose *Save* and *Close*.

Now type

```
>> help MixStrings
```

You should see the following information.

```
MixStrings.m

Replaces 'this is gibberish' with a
```

```
 string of xxxxx's

written by IF & GMB 4/2005
```

Hopefully this will look a little familiar? It's the header you wrote.

The command *help* tells the computer to display the header you wrote. That's why you always need to write headers for every m-file that describe clearly what the program does and how to use it. The header will also help you find the right program when you can't remember its name – I'll show you how that works in just a second.

Try typing the following and read the headers.

```
>>help who
>>help length
>>help path
>>help addpath
>>help round
>>doc round
```

doc is like help but provides more information. Note, you can also search for help on functions via the Matlab *help* window. The help window should automatically open when you use the *doc* command and you access it from the *Desktop* menu on the top of the command window.

However, to use help or doc you need to know the name of the command you are looking for. The solution to this problem is the command *lookfor*. *lookfor* finds all the m-files that contain a particular word in their headers. For example, try:

```
>>lookfor anova
```

An important thing to remember about setting your path is that if there are two files with the same name, and your path allows the computer to see both of them you won't get a warning, **the computer will just use the first one it finds!** You can find out which file the computer is using via the *which* command:

```
>>which MixStrings
```

Matlab should spit out the location of the files MixStrings.m in its path. You only have one version of MixStrings.m. If you had multiple versions Matlab would print out the path for each version that was within the path. The one at the top of the list is the version that Matlab will use by default at that current moment.

But be careful – which version that is used depends on what directory is the current working directory for Matlab. If the current directory changes it is possible that the version of MixStrings that is used will also change. This can lead to some really weird ***bugs*** (a bug is any time a program doesn't do what you want it to do and you have no idea why). One really confusing thing that can happen if more than one file of the same name is in the path is that you can make changes to a file, but the changes don't affect

what the computer does. What's happening is that the computer is not actually using the file that you are changing – it is using a different file of the same name somewhere else in the path.

So you need to be careful about two things:
1) Don't be sloppy about having multiple m-files with the same name sitting in different directories
2) Be careful about your path.

Make sure that old directories with out-of-date files in them aren't still in your path. One good technique for path management is not to simply put folders in your default path so they are permanently in the Matlab path. Instead, when you write a program you simply add the paths you will need for that program at the beginning of the program using Matlab commands (instead of the Menu bar). This technique lets you have different paths depending on which programs you are running, instead of having one mega-path. Using this technique to manage your path minimizes the probability of having out-of-date folders in your path. We illustrate this in the next section.

Changing the path within Matlab

Play with the following commands and use them to move to your MYMATLAB/MISC directory.

```
>>pwd
```

This tells you the current directory

```
>>cd MYMATLAB/MISC
```

Moves you to the desired directory (*cd* in Matlab functions just like *cd* in Linux)

```
>>ls
```

Lists the files in that directory.

Then you can add Misc to your path as follows:

```
>>addpath(pwd)
```

What you are doing here is telling the computer to add the directory you are currently in (pwd) to the path. Now we can use the m-files in MYMATLAB/MISC no matter what the current working directory is.

As mentioned above, it can be useful to put an *addpath* command at the beginning of m-files so that when the program is run, Matlab can use related programs that are in that same directory. Although this isn't much use to us now (since we only have one

program), we'll illustrate by adding something like following line after the header in your program:

```
addpath('/home/myaccount/MYMATLAB/MISC');
```

Ok, now to make *MixStrings.m* do something. Go back to your *MixStrings* m-file in the editor window and add the following lines of code below the addpath line. Remember to save *MixStrings* when you've finished changing it.

```
clear all;

str='this is gibberish'

for i=1:17
    str(i)='x';
    disp(str)
    pause
end;

disp('all done!')
```

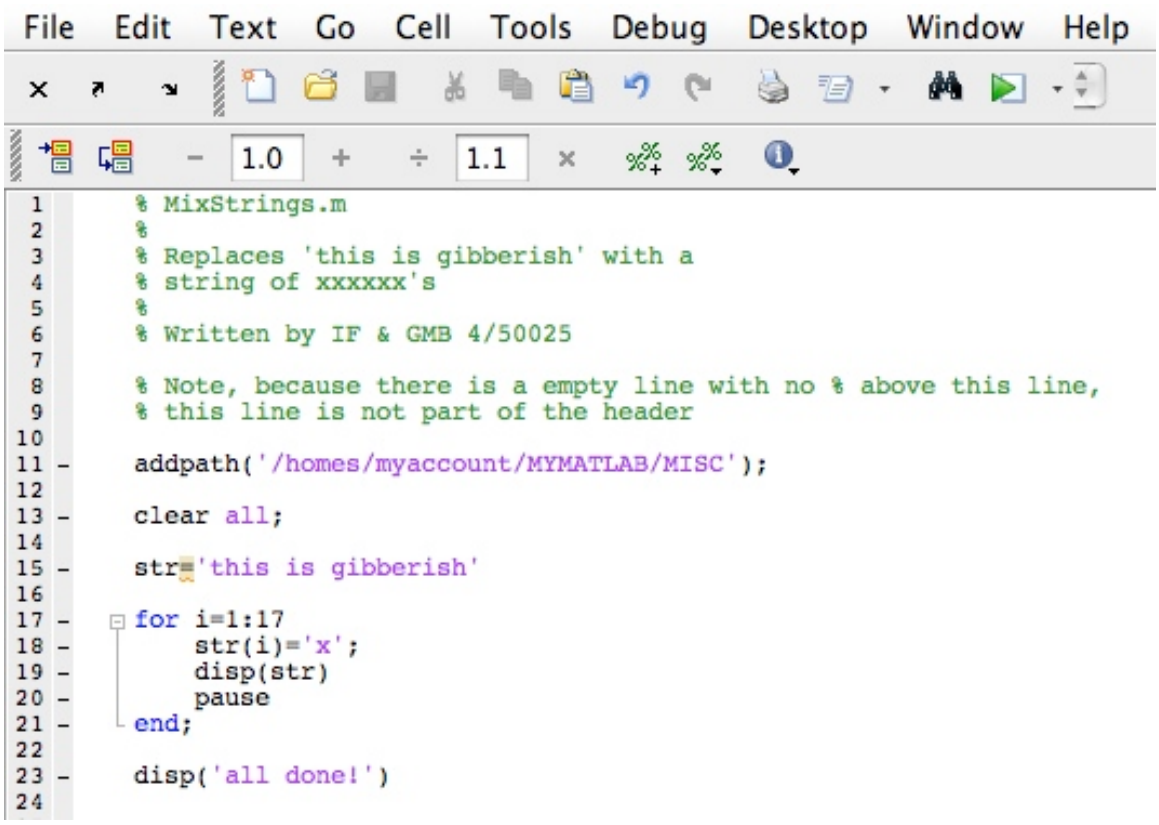
Please note a few things about the Matlab Editor: [1] green text are comments, strings are purple, and the *for* loop (a "control structure"--more on this in a second) are in blue; [2] there's an orange square at the top right of the Matlab editor which indicates that there may be a problem with your program; [3] `str='this is gibberish'` is highlighted in orange--this means that the potential problem with the program is here (if you mouse over the highlighted region, a textbox will appear suggesting an improvement).

Now go back to your command window and type:

```
>>MixStrings
```

This tells Matlab to run the program *MixStrings.m*. Then keep hitting the space bar (or any other key) until "all done!" appears.

If you have questions about how your program did what it just did, here's a line-by-line rundown (refer the image for line numbers):



Line 13: Erase all the variables currently in Matlab's workspace (not necessary here, but generally a good idea when you transition for one project to another)
Line 15: Define a string that contains the letters 'this is gibberish'
Lines 17-21: This is called a ***for*** loop. The variable *i* won't simply create an array that goes from 1 to 17. Instead it ***steps*** from 1 to 17 in steps of 1. So the first time around the loop *i*=1, the second time round the loop i=2, and so on up to *i*=17. Each time *i* increases in value Matlab will perform the commands that are inside the loop (these should be indented (tabbed) to make your code easier to read). Each of the 17 times the program repeats the commands inside the loop, *i* will be a different number.
Line 18: Gradually replace *str* with 'x' as *i* goes from 1 to 17
Line 19: Display what *str* is ('*this is gibberish'* will gradually be replaced by *'xxxxx …'*, as *i* increases from 1 to 17
Line 20: Wait for you to press any key.
Line 21: *end* tells the program that the ***loop*** is over
Line 23: Display *'all done!'* in the command window.

You can actually run the program one line at a time to see what it's doing with the Matlab debugger. To activate the debugger, you can type:

```
>>dbstop MixStrings 13
```

A red dot now appears to the left of Line 13 in the editor, which is a ***breakpoint*** (i.e., the program will pause here when you run it). You can click on the where the red dot is to

toggle it on or off--this is an even easier way to create breakpoints. Now run your function:

```
>>MixStrings
```

Matlab executes the function until you get the breakpoint, then it pauses and the command line looks like this:

```
K>>
```

The *K* indicates that you are using the debugger and are inside a program. To go to the next line of the program type:

K>> dbstep

If you type the following:

K>> str

you'll get an error message saying the variable *str* isn't defined yet. Advance one more line and repeat that command:

K>> dbstep
K>> str

Now you can see the contents of *str* because it was just defined. By using *dbstep* you can go through each line of the program and see what it does. To simply finish the program type:
K>> dbcont

Now Matlab will execute all the lines of the program until it finishes or hits another breakpoint. To remove all breakpoints, simply type:

>> dbclear MixStrings

Also note that if you ever want to **stop a program that's running**, simply type ***Ctrl-C.***

Customizing your MATLAB paths at startup

It can be a pain to manually add paths to MATLAB every time you start MATLAB. A way to do this automatically is to create an m-file called *startup.m* that contains *addpath* commands. If you start MATLAB in a directory that contains your *startup.m* file, MATLAB will automatically run that m-file and add those paths.

EXERCISE

Make a *startup.m* file that adds your /MYMATLAB/MISC directory to the set of MATLAB paths. Quit MATLAB, restart it from the directory where the *startup.m* file is, and make sure your *startup.m* file worked. Note, you'll probably want to keep your *startup.m* file in you home directory.

Variable Types

Let's talk a bit more about the two different types of variables we've created in this chapter. Type:

```
>> clear all
>> str='this is a string';
>> vect=[1 2 10 -4 5];
>> whos
```

You should see something like this:

Name	Size	Bytes	Class	Attributes
str	1x16	32	char	
vect	1x5	40	double	

Grand total is 21 elements using 72 bytes

What do the Bytes and Class columns represent?
The "Bytes" column tells you how much memory the variable is taking up. The "Class" column tells you that *str* is a list of characters (letters) and *vect* is a list of ***doubles***. (If you are used to programming in another language you will appreciate the fact Matlab does not require you to specify what a class a variable is.)

What on earth is a double?
To represent a variable the computer needs to store information about what that number is. The more precision you represent the number with, the more space it takes in the computer memory to store that number. Computers work in a binary code where a ***bit*** represents the ability to take 2 different values (usually taken to be 0 or 1). A double is a number that is represented using 264 different bits. 64 bits can represent up to 1.819 different values, so this allows for a big range of numbers and a lot of precision. In these days of powerful computers Matlab uses doubles as the default.

So if you look again at *str* and *vect* you will notice the somewhat curious fact that *vect* is using up more bytes than *str* even though *vect* only has 5 numbers in it, whereas *str* is a list of 16 letters. This is because it doesn't take a lot of memory to define a character (There are only 28 letters in the alphabet, plus you have capitals and punctuation. But even so, you only have just over 100 of alternatives to differentiate between). In contrast, when you represent numbers you are representing them with a lot of precision (1.819). Of course the particular numbers that are in *vect* could be represented easily with less precision, but by default Matlab uses high levels of precision for numbers unless you tell

it otherwise. We'll talk about that more later.

Now I'm going to show you something tricky:
>> clear all
>> a=4;
>> b='4';
>> whos

a represents the number 4, and *b* represented the *character* '4'. As far as Matlab is concerned they are not the same thing. Sometimes you may want to change a number into a character. For example, you'll commonly want to include a number in a string. Here's an example of doing that:

```
>>killednum=num2str(2500);
>>leftystr='the number of left-handed people killed
annually by right-handed products is ';
```

Note the space at the end of the string *leftrystr*

```
>>killedleftystr=[leftystr killednum]
```

We used the square brackets to tell Matlab to concatenate the strings separated by a space into a single string.

You can also skip the steps of creating variables along the way

```
>> killedleftystr=['the number of left-handed people killed
annually by right-handed products is ' num2str(2500)]
```

You can also go the other way, and change a string into a number

```
>>marriedstr='35% of people using personal ads for dating
are already married';
>>baddates=str2num(marriedstr(1:2))
```

Summary

The point of this chapter was to introduce you to the following concepts/tools:

- the Matlab command window and command line
- the Matlab editor and debugger
- Matlab file paths
- the Matlab workspace (what you see when you use the *who* command)
- variables (strings and arrays)
- using the up-arrow and down-arrow keys to access previously typed commands
- m-files and m-file headers
- for-loops

- the meaning of the percent symbol, semicolons, colons, single quotes, parenthesis, and brackets in Matlab syntax

The chapter also introduced you to the following functions/commands:

- who
- whos
- help
- doc
- lookfor
- disp
- linspace
- pwd
- path
- addpath
- cd
- ls
- clear
- pause
- for
- dbstop
- dbcont
- dbclear
- num2str
- str2num

Exercises:

1) A common thing you'll need to do in Matlab is to create strings that store the names of files. For example:

```
>> input_file1='oddball22.mat';
```

Instead of typing out the full string 'oddball22.mat' (as in the line above), store the string 'oddball22.mat' in a variable using the function *num2str*.

2) Create an m-file that will produce the following output in the command line:

```
oddball20.mat
oddball21.mat
oddball22.mat
oddball23.mat
oddball24.mat
oddball25.mat
oddball26.mat
```

by using a *for* loop and the function *num2str*

3) Imagine you can't use Participant 22's data for some reason. Edit your m-file to omit that participant. In other words your m-file should now produce:

```
oddball20.mat
oddball21.mat
oddball23.mat
oddball24.mat
oddball25.mat
oddball26.mat
```

4) First, type:

```
>> which who
```

Now, create an m-file called *who.m* that consists of the following line

```
disp('Haha! This is not the function you wanted');
```

and save it in your current working directory.

Now, type:

```
>> who
```

Explain what happened and how you fix it.