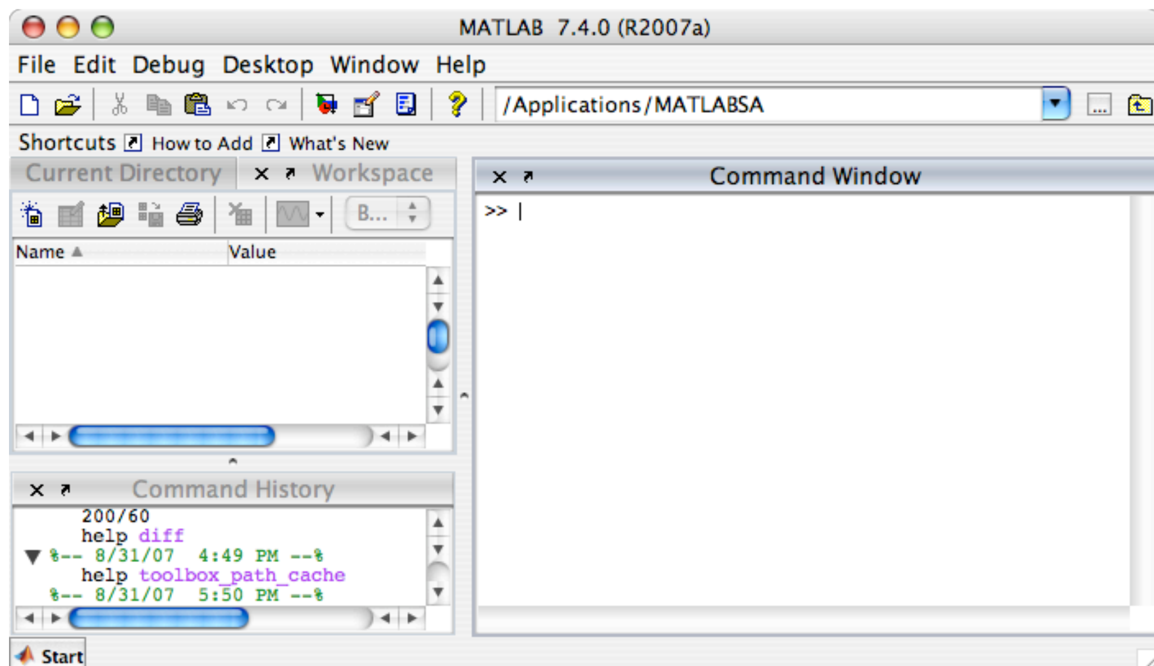
Chapter 1

A sample MATLAB session

This tutorial assumes that you have purchased and installed the student version of MATLAB on your computer. The student version is currently cheaper than most college textbooks, and is well worth the investment. It is available directly from the Mathworks and at most university computer or bookstores.

As we go through this tutorial, please try everything yourself (typing each command and seeing what it does). Remember, however, that we are mainly getting used to the look and feel of a MATLAB session, and seeing what kinds of things MATLAB can do. Some of the commands I'll ask you to input will seem a little mysterious at present; relax, don't worry, enjoy the test drive.

Start MATLAB on your computer. You should see a window something like this:



(Yours will look slightly different, but there should be a large "command window" on the right, and this is the window in which we will working in for this tutorial.)

Note that the flashing cursor is next to a ">>". This is called a "command prompt". In this document, I will precede all commands you should actually enter with this prompt. For example, let's start out by using MATLAB as a calculator:

```
>> 1+1
```

(always hit return or enter at the end of the line)

MATLAB should respond with the (hopefully) correct answer:

```
ans =

     2
```

An important aspect of programming languages is the ability to store values in variables, which are containers for data, similar to cells, rows, or columns in Excel, SPSS, etc. (The “ans” in the output above is actually a special variable that MATLAB creates when you don’t specify one).

Store something into a variable called x:

```
>> x = 1+1

x =

     2
```

Note that the resulted is output back to the console. You can suppress this by ending the line with a semicolon.

```
>> x = 1+1;
```

This is very handy when the output of a calculation is large or, later, when you write programs that do lots of calculations.

To see the value of a variable, just type the variable name:

```
>> x

x =

     2
```

Variable names can be more than just a single letter so they can be a little more descriptive.

```
>> nqp = 22/7
```

You can also make them really descriptive, but they get cumbersome to type:

```
>> not_quite_pi = 22/7
```

Notice I was able to use underscores to make the variable name more readable. You can use numbers too, but avoid using anything else. Also, variable names must begin with a letter, and they are case sensitive (as is everything else in MATLAB). So X is not the same variable as x.

We can test the equality (or inequality) of variables. The variable “pi” is built into MATLAB, so:

```
>> nqp == pi
```

Note the double-equals, which means “are these equal?”
Let’s see how bad the approximation is:

```
>> pi - nqp
```

Okay, then perhaps you can guess what this returns:

```
>> nqp > pi
```

(Note that command prompt is special and not part of the command per se, but the “>” you type is interpreted as “greater than”)

You can see what variables you have created by listing them:

```
>> who
```

or you can get a little more detail with “whos”:

```
>> whos
```

And you can remove a variable like this:

```
>> clear nqp
```

Whoops, maybe we didn’t want to do that. To re-do a command, you can hit the “up arrow” on your keyboard to go back through previously entered commands. When you see the one you want, just hit return. (If you overshoot it, hit the down arrow.)

Let’s clean up by removing all the variables:

```
>> clear
```

Don’t worry if this command, or others below, seem mysterious to you; we’re just taking MATLAB for a test drive.

Variables can hold more than one number (note: a variable that holds a single value is often called a *scalar*, and one that holds multiple values is often called a *vector*):

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

You can also enter numbers into a variable by hand:

```
>> y = [2, 3, 5, 8, 10]
```

which means "concatenate the numbers 2, 3, 5, 8, 10 and assign them to y".

When working with these kinds of variables, MATLAB behaves in a convenient fashion:

```
>> x + 5
```

No need to add 5 to each and every element of x; MATLAB automatically interprets the command this way.

If we add two vectors:

```
>> x + y
```

Again, the result is sensible; MATLAB adds each value of y to the corresponding value of x.

Comparisons work the same:

```
>> x > 3

>> x == y
```

Now we can do some fun stuff:

```
>> scatter(x, y)
```

Looks like x and y are pretty highly correlated. Let's see by how much:

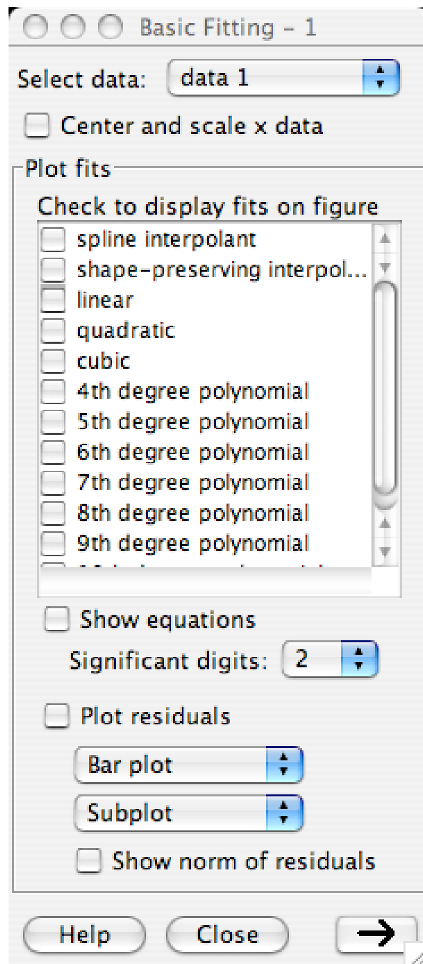
```
>> corrcoef(x, y)
```

Note that this returns the entire correlation matrix but only the upper (or lower) triangle – in this case a single point - has meaning in this case. Some connecting lines might make the form of the relationship a little clearer:

```
>> line(x, y)
```

This looks sigmoidal ("s-shaped") to me, but I'll bet a straight line still does a pretty good job of describing the data. So let's fit a linear model - y as a linear function of x - and plot it.

One nice thing about MATLAB is that, even though at its heart it is a programming language, you can also do a lot with plots using the GUI (graphical user interface). From the menu bar at the top of the figure, select Tools -> Basic Fitting. A new window similar to the one shown here should appear.



Now check the "linear" box, as well as "Show equations" and "Plot residuals" boxes to see the results of the fit on the plot. You can also press the large right-arrow button at the lower right of the page to see additional fitting information in this window.

Visually, the fit isn't bad, but it still looks like the data are departing from a straight line in a systematic fashion. If these were actual data, I'd want to explore the possibility that the relationship was really sigmoidal ("s" shaped), but this is a subject for later discussion.

Let's clean up again:

```
>> clear
```

One of the great things about MATLAB is that it has lots of random number generators and statistical distributions built in.

Let's get a sample of size 100 from a normally distributed population. Here:

```
>> x = randn(100, 1);
```

And look at them (What do you expect the mean and standard deviation to be?):

```
>> hist(x)
```

This looks like a standard normal distribution in that the mean looks to be about zero, the s.d. looks to be about one, and most or all of it looks to be between +/- 3. Let's check:

```
>> mean(x)

>> std(x)

>> summary(dataset(x))
```

So, everything looks consistent with a standard normal – mean and median both near zero, s.d. of about 1, etc. I realize that the "summary(dataset…" command looks a little awkward and mysterious right now, but don't worry about it, we're just taking MATLAB out for a test-drive.

Here's another useful way to look at the data:

```
>> boxplot(x)
```

This command creates a boxplot. The center line of a boxplot shows the median, the ends of the box show the interquartile range (i.e. the medians of the upper and lower halves of the data), the whiskers show the extreme values of the data (defined as either the minimum or maximum, or a point 1.5 times the interquartile range from the median, whichever is closer). Any values beyond the whiskers are plotted as individual points, and are candidates – only candidates – for outliers.

Why a boxplot instead of a histogram? With a histogram, you get a better look at the fine structure of the data (provided you or your software have chosen the bins wisely). With a boxplot, you lose some of this fine structure, but you can compare multiple distributions easily:

```
>> y = 0.5 + 2.*randn(100, 1);
```

This creates 100 normally distributed random numbers, doubles the standard deviation, adds 0.5 to all of them, and stores them in y.

```
>> z = lognrnd(0, 1, [100, 1]);
```

And this creates 100 numbers from a log-normal distribution (don't worry if you are not familiar with the log-normal distribution – for our purposes today, it is just a handy non-Gaussian distribution).

```
>> boxplot([x, y, z])
```

Now we can see at a glance that, relative to x, y has a small upward shift (because we shifted the mean) and a larger s.d., whereas z has an even bigger overall shift, and is highly skewed in the positive direction.

You can take a closer look at the log-normal distribution in z:

```
>> hist(z)
```

As the boxplot indicated, it is highly skewed.

Lastly, let's introduce the concept of "looping", which is how we get computers to do something over and over again. We are going to use looping a lot as we proceed, so we might as well get started. Type the following code in exactly as shown. Note that after you enter the second line, the prompt disappears and then then comes back again after you enter the final end.

```
>> mymeans = zeros(50,1);
>> for(i=1:50)
x = randn(100,1);
mymeans(i) = mean(x);
```

```
end
```

Now let's unpack this. The first line creates a vector of 50 zeros so we have a place to store the results of our calculation after each pass through the loop (we actually don't have to do this, MATLAB will create a vector for you "on the fly" but will run more slowly than if you set up a vector in advance). The second line starts our loop (which consists of the code between the "for" and the "end"), and defines a variable "i' that will start at one and then increment by one on each pass through the loop until it reaches 50, at which point the loop will terminate. The third line we already know about; it creates a variable x and stuffs it full of 100 random numbers from a normal distribution. The fourth line computes the mean of our new x and puts it in the ith element of mymeans. Since i changes with each pass through the loop, each new mean will get stored in a successive element of mymeans (like a postman going down a row of mailboxes, or an Excel user going down a column of cells.).

Let's look at this distribution of means:

```
>> hist(mymeans)
```

It looks roughly normal, but look closely at the x axis. Notice that everything is about a tenth of what it should be for a standard normal distribution.

Let's compare the distribution of means with our last sample, x.

```
>> grp = [zeros(100,1);ones(50,1)];
>> boxplot([x; mymeans], grp)
```

So, the distribution of the means from a normal distribution is quite a bit narrower than a representative sample from which those means came. As I pointed out earlier, the difference in width looks to be about a factor of 10:

```
>> std(x)./std(mymeans)
```

I get a value pretty close to 10, and 10 just so happens to be the square-root of our sample size, 100 - hmmmm. So our distribution of means is roughly normally distributed, and has a width equal to the width of one of the sample distributions divided by the square root of the sample size.

It almost seems like there should be a theorem about that… Actually, there is. Within a short amount of time and very little programming, we were able to get the computer to run a simulation that illustrated the central limit theorem, which is the most fundamental theorem of traditional statistics. We will cover this more thoroughly soon.

To end your MATLAB session, select "exit MATLAB" from the file menu, or type:

```
>> exit
```

You may be given the option to "save your workspace" – if you do, any variables you still have will be ready and waiting for you the next time you start MATLAB.