A Basic Guide to Using MATLAB in Econ 201FS

Matthew Rognlie

February 1, 2010

Contents

1 Finding Matlab

First, of course, you need to have MATLAB available. There are several options:

1. You can go to the Linux computer clusters in the basement of Teer or in Hudson 111 and use MATLAB on one of the computers there.

2. You can get a copy of MATLAB for your own computer. Student editions are available for $100; although I don't endorse it, many students download cracked versions intead.

3. You can use MATLAB remotely from your own computer by logging into a Duke Linux machine via SSH. (This will only give you text-based access, which may be a little difficult if you're used to the graphical interface.)

   If you use Windows, you can do this by downloading PuTTY (WinSCP might be useful as well for file transfer). Under Hostname, type in something like "teer14.oit.duke.edu" (14 here can be any number below 45 or so; it refers to the computer in the Teer lab you're logging into); select SSH, and hit Open. Use your NetID and password to log in. Once you've done this, you are at the Linux command line, and you can type in `matlab` to open an interactive text-based session. If you want to run an m-file, just type its name into the prompt. You can edit m-files either by using a Linux text editor like emacs or vim via SSH, or by writing them on your own computer and using WinSCP or another file transfer program to copy them over.

   Alternatively, if you're using a Mac, SSH should be built in, although I don't know the details. If you're using Linux, SSH almost certainly is built in, and by adding the "-X" option you can even use the MATLAB graphical interface.

4. You can use Octave, a free MATLAB clone, instead. Octave itself is generally not accessed using a graphical interface, although there is a free extension called "QtOctave" that is.

5. You can directly use the Linux desktop and graphical interface on a Teer or Hudson computer by remotely logging in via something called VNC. This can be a little complicated; talk to me if you really want to do it.

2 Getting Started

These instructions will be given assuming that you're using a Duke Linux computer, and I'll try to be as specific as possible for that setting. Still, this guide will inevitably leave a few details out. Please don't hesitate to ask me if there's anything ambiguous or confusing. (And please don't be insulted if these instructions are way more specific than you need!)

1. Go to the Applications menu in the upper left corner of the screen, and then under Accessories click on the "Terminal" icon. This will give you the Linux command line. Useful note: you can scroll to commands you entered earlier by pressing the up key.

2. You probably want to make a special directory for your work and files in Econ 201FS. Type

   ```
   mkdir econ201
   cd econ201
   ```

   to make a new directory called "econ201" and then move yourself to it.

3. Now you need to download your data from the course website, which you can see at `http://econ.duke.edu/~get/browse/courses/201/spr10/PROJECTS/SP100-1MIN/`. If I want to download the data for Morgan Stanley, I'll type:

   ```
   wget http://econ.duke.edu/~get/browse/courses/201/spr10/PROJECTS/SP100-1MIN/MS_078.dat.gz
   ```

The "wget" command downloads whatever is at the url after it and places it in your current directory.

4. Now you want to uncompress the data. To uncompress my Morgan Stanley data, I'll type:

   ```
   gzip -d MS_078.dat.gz
   ```

5. Now I have a file named `MS_078.dat` in my `econ201` directory. To open MATLAB, just type:

   ```
   matlab
   ```

   This will open the latest version of MATLAB installed on these computers, with the `econ201` directory as MATLAB's current directory. (For some reason, the Applications menu seems to have some problems correctly opening the more recent version.) If you open MATLAB elsewhere, you may need to change the current directory to `econ201` from within the program.

For everything that follows, note that we can either use MATLAB by typing commands at the interactive prompt or by saving a series of commands in an m-file. m-files can be edited using the built-in editor, which is reached by going to the "Desktop" menu and selecting "Editor". For simple exploratory analysis, the interactive prompt can be best, but for serious projects it is critical to save your work in m-files.

3 Basic Data Manipulation

Now that I'm in MATLAB, I'll load my Morgan Stanley data file and start manipulating it. To load, I type:

```
load MS_078.dat
```

This may take a few seconds. Once it's done, the full data file will be loaded as a matrix in Matlab named MS_078. Each row of this matrix refers to the observation of a stock's price at a particular minute. The columns of the matrix are described at `http://econ.duke.edu/~get/browse/courses/201/spr10/PROJECTS/SP100-1MIN/stock_data_documentation.txt`.

Suppose I want to see the dimensions of the data. I type:

```
size(MS_078)
```

I get `285670 10` back. This tells me that `MS_078` has 285670 rows and 10 columns. The number of rows is an important feature of the data, and I'll store it by typing:

```
n = 285670;
```

Note that I include a semicolon after typing `n = 285670`. If I don't write a semicolon, MATLAB echoes the result of a computation back to me; in this case, simply 285670, which would be extraneous. This isn't a big deal in this particular case, but it becomes a very big deal when we're dealing with massive matrices. Your screen will quickly be filled with data if you forget a semicolon.

Now suppose I want to extract the prices and do some work with them. I'll type:

```
prices = MS_078(:,6);
```

This creates a new vector (i.e. single-column matrix) called `prices`, which contains the sixth column of the `MS_078` matrix. The colon tells MATLAB to take *every row* of the matrix, and the 6 tells MATLAB to take the sixth column of the matrix.

This syntax can be generalized. For instance, if I wanted to take the first 100 price datapoints, I'd type:

```
otherprices = MS_078(1:100,6);
```

which tells MATLAB to take rows 1 through 100, and column 6. If I wanted to take every fifth price from the first 100, I'd type:

```
otherprices = MS_078(1:5:100,6);
```

This tells MATLAB to take rows 1, 6, and so on, up to 96; and column 6.

4 Plotting and Finding Returns

4.1 Basic Price Plotting

Suppose that I want to make a plot of the prices. This is very simple; I just type

```
plot(prices)
```

and all the price data will appear in a plot. Unfortunately, as you can see, the x-axis in this plot doesn't show dates; it just shows the *index numbers* of each datapoint. To better interpret the graph, we want to plot dates as well, which means that we'll need to do a little more work.

First, in our data we see that years, months, days, hours, and minutes are columns 1-5, respectively. We need to translate this information into Matlab's internal representation for dates, which is described at `http://www.mathworks.com/access/helpdesk/help/techdoc/ref/datenum.html`. I type:

```
dates = datenum(MS_078(:,1),MS_078(:,2),MS_078(:,3),MS_078(:,4),MS_078(:,5),zeros(n,1));
```

This calls the `datenum` function, which among other options accepts 6 columns of data and translates them into a MATLAB date: year, month, day, hour, minute, second. The first five of these are the first five columns in the data, and thus we take these columns and use them as the first five arguments to the `datenum` function. Since our data is minute-by-minute, seconds are irrelevant, and we make a column vector of n zeros by calling `zeros(n,1)` to fill this field.

Now we plot the dates on the x-axis against the prices on the y-axis:

```
plot(dates,prices)
```

Now, strange numbers are on the x-axis: they're actually the serial numbers that MATLAB gives to dates. To make these nicer, we tell MATLAB to apply the `datetick` option to the x-axis:

```
datetick('x')
```

Now the x-axis will display a date format that MATLAB selects to fit the series it's given. You can customize further using options documented at `http://www.mathworks.com/access/helpdesk/help/techdoc/ref/datetick.html`.

4.2 Returns

Normally in this class we work with *log-returns*. To find these, first we need to take the logarithm of our price data. Fortunately, this is simple:

```
log_prices = log(prices);
```

In general, if you ask MATLAB to evaluate a function like `log` that only makes sense on scalars on a matrix or vector, it will take that function element-by-element, which is exactly what we want.

Now, to find log-returns, we use a simple trick that will frequently be convenient in MATLAB:

```
log_returns = log_prices(2:n) - log_prices(1:n-1);
```

Here we take elements 2 through n of `log_prices` and subtract them by elements 1 through n-1 of `log_prices`. Since subtraction is done element-by-element of equal-sized vectors, element 2 in the first vector is matched with element 1 in the second, 3 in the first vector is matched with 2 in the second, and so on, giving us the minute-on-minute returns we want.

We can plot these returns against the start date of each return interval by typing:

```
plot(dates(1:n-1),log_returns)
datetick('x')
```

Note, however, that many of the returns are rather large. This is because when we take returns on the *entire* vector of log-prices, we are including returns from the interval between 4:00 PM and 9:35 AM, which is *much* larger than the other minute-long intervals.

To fix this, we need to somehow take only *intraday* returns. Remember that there are 385 price observations each day. Let's make a new variable for the number of days in the sample:

```
ndays = n/385;
```

Now we use a command called `reshape`, which copies the entries of one matrix into another matrix with different dimensions but the same number of elements:

```
log_prices_d = reshape(log_prices,385,ndays);
```

Our vector `log_prices` (which is really a one-column matrix) has $n = 385 * ndays$ elements, and we are making a new matrix `log_prices` with 385 rows and *ndays* columns — a column for each day in the sample. The `reshape` command takes elements column-by-column from the first matrix into the second, which means that it takes the first 385 elements in `log_prices` into the first column of `log_prices_d`, the second 385 elements of `log_prices` into the second column of `log_prices_d`, and so on, exactly as we want.

Now we take minute-on-minute returns for each day separately:

```
log_returns_d = log_prices_d(2:385,:) - log_prices_d(1:384,:);
```

This contains 384 minute-on-minute returns corresponding to each day. (This data format will be very useful when you have to compute daily statistics for the class!) We can analyze the days separately, or we can plot all the returns together again to see what they look like now that the 4:00-to-9:35 jumps have been removed. To do the latter, we need to reshape the returns back into a single column vector:

```
log_returns_dfull = reshape(log_returns_d,384*ndays,1)
plot(log_returns_dfull)
```

Note that the magnitude of the largest returns has decreased significantly. To match up the dates to these returns, we need to do a little more work—essentially paralleling our operations on the prices with the dates:

```
dates_d = reshape(dates,385,ndays);
dates_dr = dates_d(1:384,:);
dates_rfull = reshape(dates_dr,384,ndays);
plot(dates_rfull,log_returns_dfull)
datetick('x')
```

5 Computations

5.1 Basic Math in Matlab

As we've already shown, you can add and subtract matrices/vectors of the same dimensions simply by using the '+' and '-' operators. We can also do:

```
result = matrix1.*matrix2;   % multiplies matrix1 by matrix2 element-by-element
result = matrix1./matrix2;   % dividies matrix1 by matrix2 element-by-element
result = matrix1.^2;         % squares matrix1 element-by-element
result = matrix1^2;          % squares matrix1 using matrix multiplication
result = matrix1*matrix2;    % multiplies matrix1 by matrix2 using matrix multiplication
result = scalar*matrix1;     % multiplies every element of matrix1 by scalar
```

Note that for examples 1-2 above, the matrices must have the same dimensions. In example 3, the matrix must be a square matrix, and in example 4, the matrices must have compatible dimensions for matrix multiplication. Note also that there is a pattern here: matrix operations use the arithmetic operators alone, while element-by-element operations add a dot.

5.2 Example

Suppose we wanted to find the correlation between consecutive log-returns in our sample. First, let's store the returns from 1 to n-1 in one vector and the returns from 2 to n in another, for simplicity:

```
ret1 = log_returns(1:n-2);
ret2 = log_returns(2:n-1);
```

We could do this using built-in statistical operations that you can look up on the MATLAB website, but the goal is to illustrate matrix arithmetic. To get the covariance of the two series, we write:

```
covar = mean(ret1.*ret2) - mean(ret1)*mean(ret2);
```

Now, to get the correlation coefficient:

```
var1 = mean(ret1.^2) - mean(ret1)^2;
var2 = mean(ret2.^2) - mean(ret2)^2;
r = covar/sqrt(var1*var2)
```

On my data, this gave $r = -0.0226$, which is exactly as we'd expect: very close to zero, but slightly negative since we're looking at such high-frequency data, where microstructure effects tend to produce a small negative autocorrelation.

6 Other Matlab Syntax

If you're familiar with programming, you almost certainly have experience with `for` and `while` loops. MATLAB has these as well:

```
for i = 1:100
   (do commands in here for each i from 1 to 100)
end

while [some logical condition goes here]
   (commands go here; they will repeat until the condition is no longer satisfied)
end
```

For most languages, these are fundamental programming constructs. The situation in MATLAB is slightly different. MATLAB is an *interpreted* language, which essentially means that the code you write is not run at a level as close to the hardware as in other, *compiled* languages. This makes basic constructs like `for` loops in MATLAB substantially slower than they are in a language like C.

When you call a built-in numerical operation, however, MATLAB is *not* slow; it runs relatively fast under-the-hood compiled code. For instance, it's *much* faster—maybe 50 times faster—for you to multiply two vectors using the `.*` command

```
result = vector1.*vector2;
```

than it is for you to write a MATLAB for loop doing the same thing:

```
n = length(vector1);
result = zeros(n,1);
for i = 1:n
  result(i) = vector1(i)*vector2(i);
end
```

This doesn't mean that you can never use `for` or `while` loops. There are a few cases in which they're acceptable and possibly useful:

1. You can't figure out how to do something with built-in MATLAB operations, but it's simple to program using a loop and you have no alternative but to pay the speed penalty. (The speed penalty can be eliminated while retaining flexibility by linking MATLAB to C or FORTRAN.)

2. A loop takes up so little time with respect to the rest of your program that optimizing it with matrix operations is unnecessary.

One special case of (2) is very common: you place an "outer" `for` or `while` loop around a much more computationally intensive "inner" loop. For instance, suppose we're squaring a 1,000,000-by-10 matrix. We can either just square it:

```
result = matrix1.^2;
```

or loop through the 10 columns and square each:

```
result = zeros(1000000,10);
for i = 1:10
  result(:,i) = matrix1(:,i).^2;
end
```

In this case, of course, it's simpler to do the former. The `for` loop doesn't make the latter much *less* efficient, however, because `Matlab` only has to read it 10 times, which is almost nothing compared to the million operations done on the inside. In some more complicated cases, it will actually be simpler to do something more like the second example—sometimes it's even the only way.

MATLAB also has `if` statements, although they aren't likely to be frequent in the code that we're writing:

```
if [logical statement goes here]
  [some command]
else
  [some command (optional)]
end
```

Finally, one useful trick is that if you try to evaluate a logical condition on one or more vectors/matrices, you'll get a vector/matrix of 1s and 0s corresponding to the truth or falsity of the condition. For instance, if `log_returns` is our vector of returns, we might write:

```
absret = abs(log_returns); %this gives absolute log returns
mask = (absret > 0.01);    %this contains 1s where abs log returns are greater than 0.01
indices = nonzeros((1:n-1)'.*mask); %indices of abs log returns greater than 0.01
```

With this list of indices, we can examine the return sequence more closely wherever there was an unusually large return.

7 Other Resources

This guide, of course, does not even come close to covering everything you'll want to know to be a happy MATLAB programmer. You should use other resources, including:

1. Google. (I learned everything I know about MATLAB via aggressive googling.)

2. The official MATLAB documentation: `http://www.mathworks.com/access/helpdesk/help/techdoc/`.

3. This summary of MATLAB commands: `http://www.rpi.edu/dept/acs/rpinfo/common/Computing/Consulting/Software/MATLAB/Hints/commands.html`.

4. Any library book with MATLAB in the title.

5. My office hours!