MATLAB Introduction Workshop
15 February 2017
William Armstrong

1 Introduction

In today's workshop we will cover the basics of using MATLAB scripts to import, process, analyze, and plot data. MATLAB is powerful data analysis software that is one of the standard tools for industry as well as academic work. Other common packages you may have heard of are R and Python. These have the advantage of being free, but are generally a bit clunkier. As a CU student, you have free access to MATLAB, so that's what we'll work with today. We will have to rush through some topics today; I've tried to make this handout detailed enough that you can walk yourself through the topics we skim.

One important thing to note: in MATLAB you can always phone a friend by typing `help commandName` where `commandName` is the command you're having trouble with. This will bring up MATLAB's built-in command documentation that can help you find where you are going wrong.

2 Getting data into MATLAB

Before we can start analyzing data, we need to bring it into our MATLAB workspace. There are many ways to do this, but we today we will introduce four different methods. Before doing anything, we will need to open a **script editor window**. You can do this by pressing `command + n` or by clicking the "home" tab and then selecting "new script" or selecting "new" and then "new script". Once you have a script editor window open, we can start bringing in data.

We also need to download example datasets to use today. I have uploaded data to GitHub, which you can find at: github.com/armstrwa/presentations. The files to download are: 'exampleScript.m', 'gglaDataAll.csv', 'gglaDataTemp.csv', 'glacierVelocityData.mat', and 'tempAndSnow.mat'. You can also download this handout at that link.

2.1 Manual entry

This is by far the most tedious method, but works well if you have small datasets or if you only plan on running your analysis once. To do this, simply create a MATLAB variable by typing, for example, `y1 = [10 2 9 100 3 0 19 28 90 50];`. This will create a variable called `y1` that contains the variables you entered. The semi-colon at the end tells MATLAB to suppress its output; otherwise it will print the value of `y1` to the screen. This isn't so bad when a variable only contains 11 values, but if it's 11,000 then it becomes problematic.

You can make MATLAB do some of the work for you if your data are regularly spaced. For example, typing `x = [0:10:100];` will produce a vector called `x` that begins with 0 and increases by 10 until reaching 100.

Another quick-and-dirty way to bring data into MATLAB is to copy & paste. To do this, start to create a vector by typing `x = [` then copy in your data, and then be sure to close the vector with a `];` after you have copied in.

It is important to note that a variable can be named anything, and MATLAB doesn't care what you call something (for the most part, the exception being certain reserved words like `for` and `end` that MATLAB uses for other things). It is good practice to name variables things that make sense (e.g., `temp` if it's temperature data) or you will eventually confuse yourself as to what is what.

2.2 A tangent into matrix operations

One of the most powerful things about MATLAB is its ability to easily deal with matrix operations. For example, let's return to our variable `x = [0:10:100];`. Let's also make a new variable `y = [0:1:10]';`. The `'` is MATLAB's simple way to *transpose* a matrix (i.e., flip columns to rows). You can see `size(x) = (1,11)`, indicating it is a *row vector*, with 1 row and 11 columns. `size(y)` gives the opposite answer because we transposed the vector.

We can now perform mathematical operations these vectors. What does `y+2` output? How about `y/2`? How about `x*y`? Is this answer what you were expecting?

This unexpected (I'm guessing) outcome highlights MATLAB's handling of *matrix* operations versus *element-wise* operations. Recall that multiplying two vectors $\vec{x}$ and $\vec{y}$ to produce answer z is calculated as $z = x_1y_1 + x_2y_2 + ... + x_ny_n$ where n is the length of $\vec{x}$ or $\vec{y}$. So `x*y` in MATLAB is the notation for the *dot product* of two vectors, whose answer is a scalar (i.e., just a number, not a vector). This may be confusing at first, and give you an answer you didn't want, but is really powerful and saves you a lot of time. Just imagine how much more complicated solving `x*y` would be if you had to do it by hand, or figure it out in Excel.

What you may more commonly want to do, however, is *element-wise* operations. Imagine that `x` represented the prices of various items in an imaginary store, and `y` were the amount of each item sold. If you, as the store owner, wanted to quickly tally the revenue generated from item, you could type `x.*y`, where the `.` indicates an *element-wise* operation. Now, you are calculating $\vec{z} = [x_1y_1, x_2y_2, ..., x_ny_n]$, so your answer is a list, or vector, instead of just a number (which, in this case, the dot product would be your total income on all items sold).

An even more powerful function of MATLAB is its ability to perform *matrix inversions.* Consider the equation $\overline{\overline{G}}\vec{m} = \vec{d}$ where $\overline{\overline{G}}$ is a tensor/matrix of size $i \times j$, and $\vec{m}$ and $\vec{d}$ are vectors of size $j \times 1$. Physically, the matrix $\overline{\overline{G}}$ could represent your system's governing equations, $\vec{d}$ could represent your measured field data, and $\vec{m}$ could be unknown model parameters for which you wish to solve. For example's sake, let's create a matrix `G = [1 2 3; 4 5 6; 7 8 9];` and `d = [10 11 12]'`. The `;` within the brackets tells MATLAB to start a new row, such that `size(G)=(3,3)`. To solve for $\vec{m}$, you would be performing a *matrix inversion*, which, in MATLAB is accomplished by typing `m=G\d`. With this one simple backslash, MATLAB finds the matrix inverse $\overline{\overline{G}}^{-1}$ (where $\overline{\overline{GG}}^{-1} = \overline{\overline{I}}$) and solves $\vec{m} = \overline{\overline{G}}^{-1}\vec{d}$. You can check your answer by typing `G*m`, which should equal your original values in `d`.

2.3 Delimited text files - dlmread()

Delimited text files are convenient ways to store data. A delimited text file is a very simple format that contains data where each new column of data is separated by a special character called the delimiter. The most common forms of delimiters are tabs, commas, or spaces, but in theory any character (or combination of characters) can act as a delimiter. Reading a delimited text file into MATLAB is pretty straightforward as long as all your data are of the same type (i.e., all numbers or all letters, but not a combination of the two). If your data are of different types, we'll have to use more complicated tools that are addressed in the next section.

The general form for using the `dlmread` function is

```
variableName = dlmread('filename','delimiterCharacter',startRow,StartCol);
```

where `variableName` is what you want the variable to be called, `filename` is the filename of the file (including file ending, e.g., .txt or .csv), `delimiterCharacter` is the character that denotes new columns (e.g., ‘,’ or ‘;’), and `startRow` and `startCol` are the numbers of the first row/column in the text file that contains data. If the text file has a header, you will need to tell MATLAB to skip it. The data we are using today are comma delimited and have one header row, so we open the data by typing

```
tempData = dlmread('gglaTempData.csv',',',2,1);
```

You should see a new variable `tempData` in your workspace. It contains 8679 observations and two columns. The first column contains the date of the observation and the second tells the air temperature at that time. You can double-click on `tempData` in the ‘workspace’ window to open up the ‘variable editor’ and inspect the data.

2.4 Delimited text files - textscan()

`dlmread` works well if your data are all numeric, but does not like mixed data types. `textscan` is your go-to method for importing data if your data contain some numeric fields and some string (i.e., letters) fields. `textscan` is a little more complicated in that we have to assign a file handle using the `fopen` command, and then tell `textscan` what kind of data to expect in each column. The lines below will open our data file for use today.

```
fid = fopen('gglaDataAll.csv');
moreTempData = textscan(fid,'%s %s %f %f %f %f %f %f %f','HeaderLines',6,'delimiter'
    ,',');
fclose(fid)
```

The `%s` tells MATLAB that the column contains string data. The `%f` indicates float data, which is a type of numeric data. `HeaderLines` tells MATLAB to skip the first 6 lines, which only contain header data. You will see you now have a variable `moreTempData` in the workspace. `textscan` is funny in that it outputs a cell-type variable. To access the first columns in this variable, type `moreTempData{1}`. I usually rename these columns for easier use, e.g., `daysSince = moreTempData{5}`.

2.5 Loading .mat files

MATLAB stores data as .mat files. To bring in data stored in such a way, simply type `load filename.mat`. For the data we are using today, this is `load tempAndSnow.mat` You will see the data in the .mat file now appears in your workspace. To save a .mat file, simply type `save filename.mat`.

3 Data processing and analysis

3.1 Removing bad data

Getting rid of bad data is often the first step in data processing. Making a quick plot is often the easiest way to assess data quality. We will cover plotting more in depth in the next section, but for now, make a plot of snow depth (variable `snow`) as a function of time (variable `date`) by typing

```
        figure(1) % create a new figure window
        clf % clear the window
        plot(date,snow,'b') % plot date vs. snow using a blue line
        datetick() % make the x-axis dates instead of date numbers
        hold on
```

Do you see any areas that have data quality issues? What month has the most issues? What aspect are the data are you using to single it out as bad data?

Bad data takes the form of unrealistic values, such as the large jumps in snow depth on your plot. To get rid of these data, we first need to introduce the concept of **indexing** in MATLAB. Indexing is a way to select data from in a MATLAB vector; you can think of it as highlighting text in a Word document. Going back to our vector `x` before, we can write `x(1)` to display the first value in `x`. We write `x(1:5)` to show the first five values of `x`. We can also perform **logical indexing**, where we use a true/false statement select data. For example `x(x>50)` will show `ans = 60 70 80 90 100` – all the values of `x` that are greater than 50. We will use logical indexing to cull bad data from `snow`. To do this, we set a threshold which defines the largest value contained within the good data. Using your plot, what is this value? We then type

```
        thresholdValue = 175; % name the threshold value
        snowClean = snow; % make a copy of snow to work with
        snowClean(snow>thresholdValue) = NaN; % turn all data above the threshold to nan's
```

This will turn all values of `snow` greater than 175 into 'NaN', which is MATLAB's no data value. Now type `plot(date,snowClean,'r')`. Did using this threshold approach work the remove all bad data? There are more sophisticated approaches we could use to clean data by employing a similar method (e.g., filtering based on how large the jump is between data points, or filtering if there is a large time gap between observations), but we will not cover these today.

3.2 Smoothing, fitting, and statistical tests

We will have to breeze over this today, but MATLAB has many powerful tools for data processing and analysis. I will only have time to introduce the `smooth` command, but `polyfit`, `polyval`, and statistical tests such as `ttest` are a small sample of the many useful analyis commands.

The syntax for smoothing data is `ySmooth = smooth(x,y,span,method)` where `x` and `y` are the independent and dependent variables you want to smooth, respectively. `span` is the number of datapoints you want to use for smoothing and `method` is the smoothing method you want to employ. `moving` for a moving average and `rloess` for a robust, locally weighted fit are the options I use most. Add a smoothed line to the snow depth plot you created by typing

```
snowSmooth = smooth(date,snowClean,100,'rloess'); % smooth cleaned snow depths using 100
    points (50 points on either side) and robust locally weighted smoothing
plot(date,snowSmooth,'k','linewidth',2) % plot the line
```

4 Plotting

MATLAB can make really nice figures, but it requires a little finesse, practice, and knowing how MATLAB refers to things. This means it is generally slower (at first) to make a figure in MATLAB compared to Excel, but you have much more control and will be able to quickly make nicer figures once you get the hang of it. In addition, a healthy dose of Googling answers on stackoverflow.com and mathworks.com will go a long way.

4.1 Line plots

The basic steps for MATLAB plotting are to create a figure window, populate it with data, and touch up the axes and figure labels. To plot a line, we simply type `plot(variable1,variable2)`, followed by an optional string of parameters that specify the details of how we want MATLAB to draw things. In addition, we can create plots that have multiple subplots using the self-explanatory `subplot` command. To use this, before calling the `plot` command, we type `subplot(rowNumber,colNumber,whichPlot)` where `rowNumber` and `colNumber` indicate how many rows and columns of subplots we want, respectively. `whichPlot` is a 'pointer' number that tells MATLAB which subplot you want to work on. This number increases like you read a book: lowest value in the top left, highest value in the bottom right. I have included a sample script for plotting the meteorological data you imported using `textscan` in the lines below.

```
% Plot meteorological data
figure(2) % create a new figure
clf % clear figure
orient landscape % make the plot with a landscape aspect ratio (wider than tall). The
    opposite of this is 'portrait'
subplot(2,1,1) % create a subplot. There will be 2 rows of figures, 1 column, and we are
    highlighting the first plot
plot(daysSince,temp2,'linewidth',2,'color','r','linestyle','-','marker','o') % plot
    temperature as a function of days since the start of the record. Plot a red solid line
    of width 2, using circles as data markers
text(1,12,'Station = GGLA2','fontsize',16) % annotate the plot at (x,y) = (1,12) with text
    and set the fontsize

subplot(2,1,2) % create a subplot. There will be 2 rows of figures, 1 column, and we are
    highlighting the second plot
plot(daysSince,relh,'linewidth',2,'color','b','linestyle','--','marker','None') % plot
    relative humidity as a function of days since the start of the record. Plot a blue
    dashed line of width 2, using no data markers

```

```
subplot(2,1,1) % highlight the first subplot
ylabel('Air temperature [^{\circ}C]','fontsize',16) % label the y-axis and set the fontsize
xlabel('Days since start of record [d]','fontsize',16) % label the x-axis and set the
    fontsize
axis([0 60 -20 15]) % set the axis limits. The first two values are the min/max of the x-
    axis. The second two values are the min/max for the y-axis
title('Air temperature and relative humidity record, beginning April 01, 2014','fontsize'
    ,20) % add a title to the plot and set the fontsize
grid on % turn grid lines on

subplot(2,1,2) % highlight the second subplot
ylabel('Relative humidity [%]','fontsize',16) % label the y-axis and set the fontsize
xlabel('Days since start of record [d]','fontsize',16) % label the x-axis and set the
    fontsize
axis([0 60 0 105]) % set the axis limits. The first two values are the min/max of the x-axis
    . The second two values are the min/max for the y-axis
grid on % turn grid lines on
```

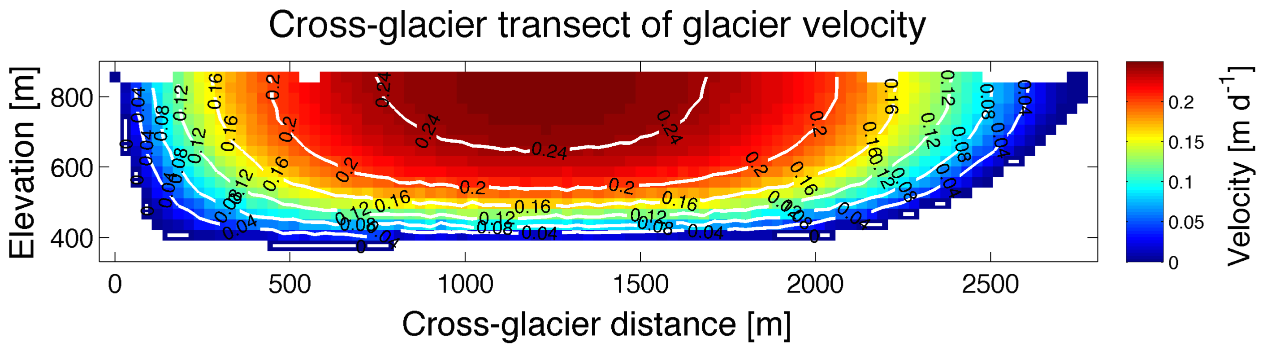
4.2 Multi-dimensional plots

I mainly use `imagesc` for plotting data involving more than two variables. I will not be able to cover this, but I have included a sample script below for plotting a cross-section of glacier velocity and drawing contour lines of equal velocity.

```
load glacierVelocityData.mat % load data
figure(3) % create new figure
clf % clear figure
orient landscape % make the plot with a landscape aspect ratio (wider than tall). The
    opposite of this is 'portrait'
h = imagesc(y,z,U);  % create a plot with the handle 'h' (this allows editing later), and
    plot U as a function of y and z
hold on
caxis([0,0.25]) % set the limits of the colorscale -- 0 at the bottom, 0.25 at the top
v = 0:0.04:0.30; % create vector marking the values at which you want contours drawn
[c i] = contour(y,z,U,v); % draw contours, c and i are handles for future editing
clabel(c,i,v,'LabelSpacing',100) % label the contour lines and space labels out (I don't
    actually know what the 100 does, I just iteratively modify until it looks good).
set(i,'color',[1 1 1],'linewidth',2) % make the contours white and draw the lines with width
     2 ([1,1,1] is the rgb value for white -- like in a rainbow, white actually has all the
    colors in it).
set(gca,'YDir','normal','fontsize',14) % 'gca' stands for 'get current axis'. This says, "
    get the current axis and make the y-axis direction normal (increasing upwards), and make
     the tick labels fontsize 14
```

```
set(h,'AlphaData',~isnan(U)) % where there is no velocity data, make the plot transparent (
    it defaults to the color of the zero value otherwise).
f = colorbar(); % draw a colorbar to show what the colors correspond to
set(get(f,'YLabel'),'string','Velocity [m d^{-1}]','fontsize',16) % give the colorbar a
    label and set its fontsize
xlabel('Cross-glacier distance [m]','fontsize',18) % label the x-axis and set the fontsize
ylabel('Elevation [m]','fontsize',18) % label the y-axis and set the fontsize
title('Cross-glacier transect of glacier velocity','fontsize',20) % add a title to the plot
    and set the fontsize
```

The figure produced will look like this:


Cross-glacier transect of glacier velocity
Elevation [m]
Cross-glacier distance [m]
Velocity [m d^-1]