COSC3000 Visualization, Computer Graphics & Data Analysis

Tutorial 1 2022

1 Aims and overview

In this practical, we have a lot of stuff. You might not get through everything in the lab; that's OK. What do we have here for you?

- Matlab overview/refresher
- More on plots and subplots in Matlab
- Data types in Matlab
- Getting data
- Univariate visualisation

You should see two Matlab m-files on Blackboard for this practical. The first, `wk1.m`, is what you will script your answers in for each of the questions in sections 3 and 6.1. The second, `wk1example.m`, shows some plotting examples concerned with the *hold* and *subplot* functions. If you are having trouble using any of the functions required by the questions, you can use the *help* or *doc* functions to get information on how to use each of them.

If you're comfortable with Matlab already, feel free to move onto one of the later sections!

2 Matlab overview/refresher

Why Matlab?

- Good combo of data analysis and visualisation
- Quick versatile visualisation
- UQ licence

Other software

- Freeware Matlab equivalents: Octave, SciLab, FreeMat
- Other visualisation software: VTK; Paraview; VisIt; Processing

New to Matlab?

- This Matlab overview/refresher
- Introduction to Matlab from Mathews & Fink 2004
- Introduction to Matlab from Sauer 2012
- Type "demo" in Matlab and look at the graphics/visualisation demos

2.1 Starting Matlab

To get going, start Matlab (e.g. from the Start-Programs menu in Windows). You'll see various windows, which can be configured to your liking. Click *Layout* in the *Home* menu tab for more available windows. The important or convenient windows for our purposes are:

- *Command window*: where you enter direct commands (at the command prompt ">>")
- *Figure*: where figures are displayed and can be edited
- *Help*: for navigating around Matlab's extensive help
- *Workspace*: showing what variables you have available and allowing quick plotting
- *Variable Editor*: allows direct, spreadsheet-like editing of data
- *Command History*: where you see previous commands (including from previous sessions)
- *Editor*: where you can write, debug, execute scripts and functions
- *Current Folder*: showing your current directory and its contents

Need help? If you're new to Matlab and feel lost at any stage, check out the examples in Matlab's help (see under *Help* in the *Home* menu tab), especially "Getting Started". An old but detailed Matlab Primer which was written by Kermit Sigmon (1936-1997) at the University of Florida is available online: `http://www.math.ucsd.edu/~bdriver/21d-s99/matlab-primer.html`. Newer editions of this are available in the library. We also have introduction to Matlab sections from two textbooks available on Blackboard (see tute 1).

There's massive help, support and file-sharing from the Matlab producers and users, at `http://www.mathworks.com.au/products/matlab/` (the User Community tab leads to "Matlab Central" including a useful File Exchange; see also the Support tab for various types of help). And there's plenty of other useful Matlab stuff available on www.

If you know what function you want to use, and just need to look details of the usage, try the command line help. In Matlab, just type "help <function name>" at the prompt. line. For example, the function `diary` records the text of what happens in your command window (including most output, which isn't stored in the Command History). Find out more by typing:

```
help diary
```

Turn on the diary function to start saving your session (i.e. this prac) as "vizdiary1.txt", by typing

```
diary vizdiary1.txt
```

(if you've read the help you'll know that typing `diary('vizdiary1.txt')` does the same thing).

Next, use the help command, and/or the help window, to find out about some or all of the functions below. (We'll get into the graphing functions in more detail in the weeks to come.) Find out what your current working directory is by typing *pwd*. Your diary file will eventually be saved to here. Try out some of the numerical functions with any data you choose to make up (e.g. with a random vector you create with *rand*).

General: who; whos; clear; load; save; what; path; pwd ;cd; mkdir; beep; help
Maths: sin; cos; tan; exp; log; sqrt; abs; floor; ceil; fix; round; mod; rem
Data exploration: max; min; mean; std; sum; cumsum; prod; find; size; numel
Operations: zeros; ones; eye; rand; randn; magic
Manipulations: sort; sortrows; cat; repmat; shiftdim; reshape; flipud; fliplr;

Next we'll look at getting some data into Matlab. Load some of Matlab's own data, by typing:

```
    load count.dat
```

The 24-by-3 array count contains hourly traffic counts (the rows) at three intersections (the columns) for a single day.

You can check that it's loaded, and its size and other properties by typing

```
    whos
```

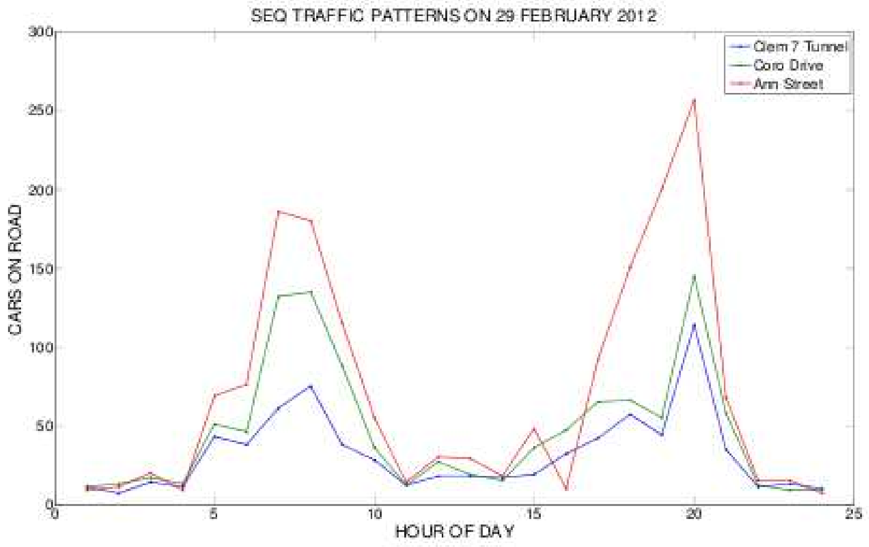
which gives details of all the variables in the workspace, and you can show the count dataset in the command window by just typing

```
    count
```

Another way of keeping track of what variables are in your workspace is to use the *Workspace Window*. If it's not showing already, check the *Show workspace* option in the *Layout* in the *Home* menu tab. You should see *count* listed in the workspace, along with basic information that it's 24×3, double-precision, and with a range of $[7, 257]$. You can display more variable properties by right-clicking in the column headers (e.g. beside where "name" appears).

You can also do quick plots by right-clicking a variable name (*count*) and selecting *plot(count)* from the menu that appears. This makes a plot in the *Figure* window: you should see peaks in traffic at rush hours and early evening. There seems to be a dip at the 19th entry (19:00 = 7pm = dinner time). For illustrative purposes, let's edit the data here: back in the *Workspace* window, double-click on *count*, which will open it in the *Variable Editor*[1]. Change the value in the 19th row, 3rd column from 90 to 200; just type 200 in the cell that says 90, and press Enter[2]. You may notice the Figure window won't have automatically updated, so click *plot(count)* in the Workspace window again to re-plot the graph. (You may also notice the command window has recorded executing the graph-plotting function plot()—we'll return to this in the near future.)

You can edit graphs a lot in the *Figure* window. We'll get very familiar with this but get a taste of what you can do by using the *Insert* menu to add an *X Label*, *Y Label*, and *Title* of your choosing. Similarly insert a *Legend*; a box showing "`count(:,1) count(:,2) count(:,3)`" appears (this reflects how Matlab's indexed the data by columns to generate the graph) but you can double-click on these and edit them to make them more informative. You might end up with something like this:



Feel free explore more of Matlab using the Help browser—the examples are particularly

[1]Alternative ways of opening variables are from the "open variable" menu in the command window's Home tab, or you can just type "open count" in the command window.

[2]You could also edit that value at the command prompt by just typing "count(19,3) = 200".

useful (see under Help in the Home menu tab, or just type demos in the command window).

Finally, quit Matlab by typing *quit*—your diary file "vizdiary1.txt" will be saved automatically in your working directory.

2.2 Programming in Matlab

Matlab is not just an interactive calculator and plotter—it's also an interpreter for a programming language. These programs can be either *scripts* or *functions*

1. What is the difference between Matlab *scripts* and *functions*?
2. Write a Matlab function that takes a vector or matrix of values as the input, converts this to a column vector using the colon operator (`a = b(:);`), and prints the numbers and whether they are odd or even to the screen; the output for `my_odd_even([16 3 5 10 1])` might be:

   ```
   16 even
   3 odd
   5 odd
   10 even
   1 odd
   ```

   You might find the `size()` function useful.

   (a) Use a `for` loop to cycle through the input values.
   (b) Use a `while` loop to cycle through the input values.
3. As for the previous exercise, but intead of even or odd, report whether they are divisible by 2, 3, 4, or 5.

   (a) Use a `for` loop to cycle through the input values.
   (b) Use a `while` loop to cycle through the input values.

3 Plots and subplots

Here we will look at some basic plotting, culminating in the use of *for* loops with functions such as *hold* and finally, *subplot*. The code discussed below can be seen in the `wk1example.m` file. If you aren't already familiar with the basic use of functions like *figure*, *plot*, *axis*, and *title*, have a brief read of their help documentation before continuing.

You have a few different options for working through the code demonstrated here: you can run the code all at once by typing "`wk1example`" on the command line; type each line in turn; copy and paste blocks of code from the editor; or use the (very handy) Debugger facility to step through the code line-by-line as demonstrated in the Appendix below.

The following snippet of code is used to generate a matrix of data that we wish to plot. Each column in the matrix is a single data series; thus, there are three different data series that we'll need to plot.

```
% clear all our current variables and figures:
clear all;
close all;
% generate our base data:
```

```
x = -pi : 0.01 : pi;
% pre-allocate space for our matrix:
data = zeros ( [ length(x) 3] );
% assign values to each column:
data (: ,1) = sin( x );
data (: ,2) = cos( x );
data (: ,3) = log( abs(x) );
```

With the data matrix populated, we now have something to plot. The following code demonstrates a simple way of plotting the first series. Notice the use of the *axis*, *xlabel*, *ylabel* and *title* functions.

```
% create a new figure with a white background:
figure ( 'color' , [1 1 1] );
% plot the first column of data:
plot ( data(: ,1) , '-b' );
% set the min and max values for the x- and y- axes:
axis ( [0 length(x) -3 3] );
% label the axes and give the plot a title:
xlabel ( 'x' );
ylabel ( 'y' );
title ( 'Data' );
% plot remaining series on same axes:
hold on;
plot ( data (: ,2) , '-r' );
plot ( data (: ,3) , '-g' );
plot ( zeros (1, length (x)), ':k' );
% turn hold back off once we're done:
hold off ;
```

Plotting data on the same set of axes is good, but sometimes we want to have each series on its own axes. The code below demonstrates the use of the *subplot* function. It results in three separate axes being drawn on the one figure. As you can see though, there is a substantial amount of code duplication, and is already cumbersome with only three different series.

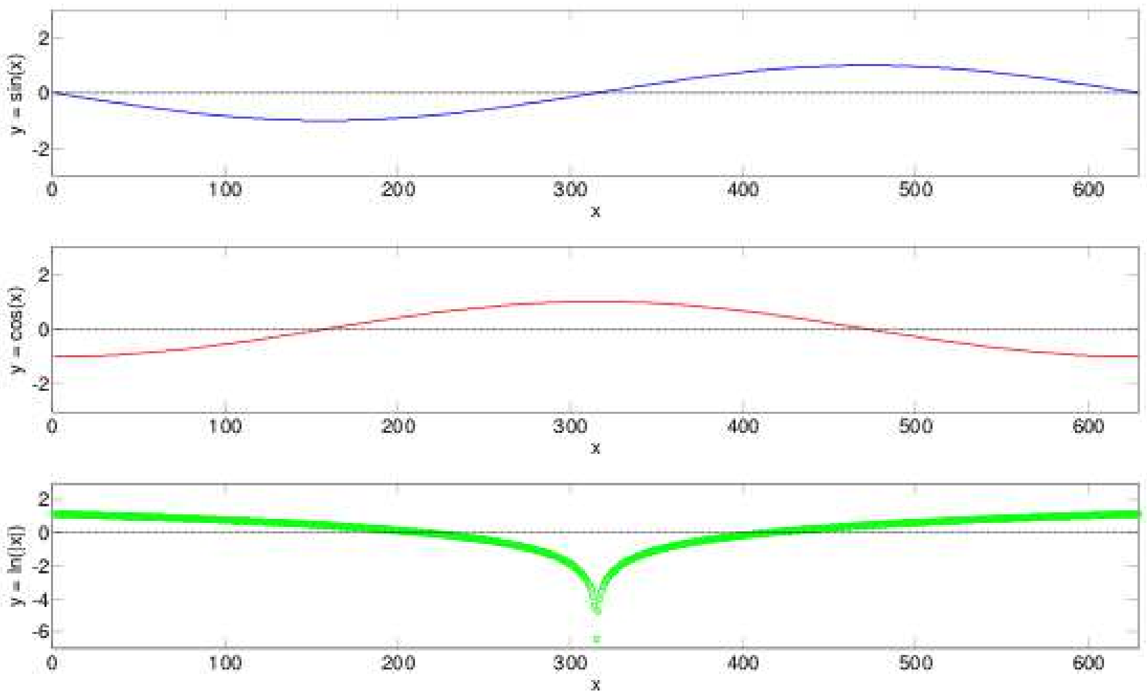
```
figure ( 'color' , [1 1 1] );
% create subplot -- we will have 3x1 set of axes in our figure,
% and the next plot drawn will be drawn in the first subplot:
subplot ( 3, 1, 1 );
plot ( data(: ,1) , '-b' );
hold on;
plot ( zeros(1, length (x)), ':k' );
axis ( [0 length(x) -3 3] );
xlabel ( 'x' );
ylabel ( 'y' );
title ( 'sin(x)' );
% change which subplot to draw on:
subplot ( 3, 1, 2 );
plot ( data(: ,2) , '-r' );
hold on;
plot ( zeros (1, length(x)), ':k' );
axis ( [0 length(x) -3 3] );
```

```
xlabel ( 'x' );
ylabel ( 'y' );
title ( 'cos(x)' );
% change again:
subplot ( 3, 1, 3 );
plot ( data(: ,3) , '-g' );
hold on;
plot ( zeros(1, length (x)), ':k' );
axis ( [0 length(x) -3 3] );
xlabel ( 'x' );
ylabel ( 'y' );
title ( 'log |x|' );
```

A *for* loop is one way to relieve code duplication. In Matlab a *for* loop has the structure

```
for i = 1:n
        % code to be repeated goes here;
        % you may wish to make use of the variable i here)
end
```

This will repeat the code n times, each time increasing the value of variable `i` by 1. Try and modify the above code using a for loop to reduce the amount of duplication. You might also wish to modify the code in other ways to gain a better understanding of the Matlab functions used in the examples above, e.g. use green circles as plot points in the bottom graph instead of a green line, and change the y-axis to be from -7 to $+3$, to get something like:



4 Mixed data types in Matlab

The basic Matlab data structure is the array, an ordered set of real or complex elements retrieved via numerical indexing; that is, the element in row *i* and column *j* of an array, `A`, is denoted by `A(i,j)`. The issue with basic arrays however, is that they can only store one type of data and you can quickly find yourself wishing for a true programming language. Fortunately though, there are other array types that can contain any type of data. The examples below show different ways of storing mixed data. Pay particular attention to how different formats use different reference methods: e.g. numerical indexing (`temperature(5,:)`); or dot referencing (`hospital.Smoker`), which'll be handy in section 5. Obviously, the discussion of these types below is limited, so if you want to know more, consult Matlab's help and online documentation.

4.1 Cell arrays

The Matlab documentation describes cell arrays as "a data type with indexed data containers called cells. Each cell can contain any type of data. Cell arrays commonly contain lists of text strings, combinations of text and numbers from spreadsheets or text files, or numeric arrays of different sizes". For example, store temperature data for three cities over time in a cell array, then plot the temperatures for each city by date:

```
temperature(1,:) = {'01-Jan-2010', [45, 49, 0]};
temperature(2,:) = {'03-Apr-2010', [54, 68, 21]};
temperature(3,:) = {'20-Jun-2010', [72, 85, 53]};
temperature(4,:) = {'15-Sep-2010', [63, 81, 56]};
temperature(5,:) = {'31-Dec-2010', [38, 54, 18]};

allTemps = cell2mat(temperature(:,2));
dates = datenum(temperature(:,1), 'dd-mmm-yyyy');

plot(dates, allTemps)
datetick('x', 'mmm')
ylabel('deg F')
legend(['city 1';'city 2';'city 3'])
```

4.2 Structure arrays

Similarly, the Matlab documentation describes structure arrays as "a data type that groups related data using data containers called fields. Each field can contain data of any type or size". For example, store patient records in a structure array, then create a bar graph of the test results for each patient:

```
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];

numPatients = numel(patient);
for p = 1:numPatients
    figure
    bar(patient(p).test)
    title(patient(p).name)
```

end

4.3 Datasets

The Matlab documentation for datasets states the following:

Dataset arrays are used to collect heterogeneous data and metadata, including variable and observation names, into a single container. They can be thought of as tables of values, with rows representing different observations and columns representing different measured variables.

Dataset arrays can contain different kinds of variables, including numeric, logical, character, categorical, and cell. However, a dataset array is a different class than the variables that it contains. For example, even a dataset array that contains only variables that are double arrays cannot be operated on as if it were itself a double array. However, using dot subscripting, you can operate on variable in a dataset array as if it were a workspace variable.

An example of using a dataset is shown below. Notice that referencing a variable in a dataset is the same as referencing a field in a structure array.

```
% Load a dataset array from a mat file and create some simple subsets:
load hospital
h1 = hospital(1:10 ,:)
h2 = hospital(:, { 'LastName' 'Age' 'Sex' 'Smoker' })

% Create a new dataset variable from an existing one :
hospital.AtRisk = hospital.Smoker | ( hospital.Age > 40)

% Use individual variables to explore the data
boxplot ( hospital.Age , hospital.Sex )
h3 = hospital( hospital.Age <30 ,{ 'LastName' 'Age' 'Sex' 'Smoker' })

% Sort the observations based on two variables
h4 = sortrows( hospital ,{ 'Sex' 'Age' })

% Draw a bar chart for a number of variables
vars = { 'Age' , 'Weight' }
for i = 1: length( vars )
    % field/variable names for structures/datasets must be char arrays
    variable = char( vars (i) );
    % dynamically reference dataset variable
    data = hospital.( variable );
    figure;
    bar( data );
    title( vars(i) );
end
```

5 Getting and importing data

See file “week1example.pdf”. Also try “`image(temp2)`”.

When plotting geographical data like this, it might be nice to include coastlines, national or state borders, etc. Can you find a Matlab function online that will plot coastlines and/or borders?

6 Univariate data

6.1 Univariate visualisation

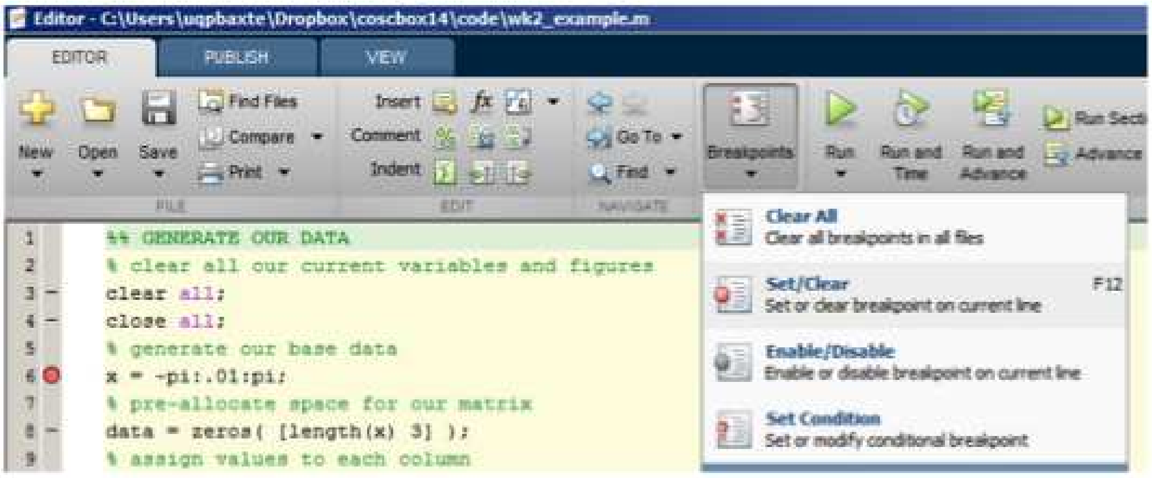
Let's finally get down to looking at a variety of approaches to univariate plotting. Some you will have already seen in lectures, while others are nicely described at following blog site by Gramener (a data visualisation company):
`http://blog.gramener.com/54/charting-one-dimensional-data-linearly`.

Load Matlab's inbuilt *flu* dataset (by typing *load flu*) then produce the following plots. Think about some of the pros and cons of each and how it might be useful to describe some of the data for your project.

1. Using the hist function, show one of the flu data regions. Use help hist in the command window to see usage information for the hist function.
2. Draw a quantile plot for one of the flu data regions.
3. Draw a Q-Q plot using the MidAtl and SAtl flu data. Make sure to include a 45° reference line and set the axis limits and plot title appropriately. You will need to use the hold function.
4. Using the quantiles calculated for the Q-Q plot above, draw a Tukey Mean-Difference plot. Again, be sure to include a reference line (this time horizontal) and set the axis limits and plot title appropriately.
5. Use the HeatMap function to draw a heat-maps for some of the flu data regions. Some heat maps are really boring, for example, the mid-Atlantic region (MidAtl). Can you explain why? Look at the heat maps for the Mtn or ESCentral regions for comparison.
6. Draw vertical and horizontal bar charts in the same figure for one of the flu data regions. You will need to use the subplot function.
7. Draw stairstep and stem plots of the same data using the stairs and stem functions.
8. Draw sparklines for all the regions in the flu dataset in the same figure. Again, you will need to use the subplot function. You should also use a for loop to avoid code duplication. See above for an example of using a for loop to draw multiple plots and some advantages of dealing with dataset objects like flu over simple matrices. Make sure the axes for each sparkline are hidden.
9. Draw streamgraphs in a similar same way to how you drew the sparklines. Remember to halve the input data and plot both the halved values and their inverses.
10. Use both the plot and scatter functions to draw jitter plots of one of the flu data regions.
11. Use the boxplot function to draw box plots for all the regions in the flu data. If you read the help documentation for boxplot you should notice that you won't need to use subplot. Make sure the box plots are oriented horizontally. As a slight challenge, make the NE data appear at the top, with the other regions in order below.

Appendix: Interactive debugging in Matlab

You can have a step-by-step look at how code executes directly from the Editor, using its debug mode, as follows. Put the cursor at the any line of code (actual code, not comment lines) and make a breakpoint by clicking on “Set/clear breakpoint” in the breakpoints menu. You’ll see the breakpoint appear as a red dot at the start of the line (e.g. line 6 here):


Editor - C:\Users\uqpbaxte\Dropbox\coscbox14\code\wk2_example.m
EDITOR
PUBLISH
VIEW
New
Open
Save
Find Files
Compare
Print
Insert
Comment
Indent
Go To
Find
Breakpoints
Run
Run and Time
Run and Advance
Run Secti
Advance
FILE
EDIT
NAVIGATE
Clear All
Clear all breakpoints in all files
Set/Clear
Set or clear breakpoint on current line
F12
Enable/Disable
Enable or disable breakpoint on current line
Set Condition
Set or modify conditional breakpoint

```
1     %% GENERATE OUR DATA
2     % clear all our current variables and figures
3  -  clear all;
4  -  close all;
5     % generate our base data
6  ●  x = -pi:.01:pi;
7     % pre-allocate space for our matrix
8  -  data = zeros( [length(x) 3] );
9     % assign values to each column
```

You can also make a breakpoint by just clicking on the “–” in the margin beside the line-number, where the red dot has appeared above. You can use multiple breakpoints. The script will now pause before implementing this line. To see this, start running the script by clicking the “Run” button ▶. Now you can get information (values or sizes) on any variables that have been created by hovering the mouse over the variable names in the code. You can also check their values in the command window or variable editor window. Then step through the code line by line by clicking the Step button. If you want to quickly proceed to the next breakpoint you entered (or finish off the code without stopping) click on the Continue icon. At any time you can exit debugging mode by pressing the red “quit debugging” button ■. To return to normal and run the code without stopping you can clear all the breakpoints by clicking “clear all” in the breakpoints menu (or clicking individually on any red dots beside the code lines to turn them off).