ELEC6021 Research Methods
Matlab Session 2

In the first session, we looked at how to write our own Matlab functions, among other things. This session poses a number of problems and asks you to write Matlab functions to solve them. However, before we attempt those exercises, let's remind ourselves about writing and running Matlab functions.

Writing and running functions

Let's start by writing a Matlab function to perform the mathematical function $f(x) = x^2$. Use the Matlab text editor to create a new M-file and type the following commands into it.

```
function y=my_function(x)
y=x^2;
```

Save your M-file as `my_function.m`. You can run your function by typing the following command in the main command window.

```
my_function(3)
```

Matlab should display a value of 9, which is $f(3) = 3^2$.

Function handles

The name of your function can be assigned to a string variable using the command

```
f='my_function';
```

This string variable is referred to as a *function handle*. The built-in Matlab function `feval` can be used to evaluate the function specified by a function handle. You can try this using the command

```
feval(f,4)
```

This time, Matlab should display a value of 16, which is $f(4) = 4^2$.

The `feval` function is usually used inside functions that take function handles as arguments. An example of such a function is `fplot`. You can get this to plot your function using the following command.

```
fplot(f,[0 4])
```

Here, the argument `[0 4]` tells `fplot` to plot the function for inputs in the range 0 to 4.

Note that using a function handle and the `fplot` function has saved us a lot of effort. The commands required to generate the same plot without using a function handle are as follows.

```
x=0:0.08:4;
for index=1:length(x)
   y(index)=my_function(x(index));
end
plot(x,y);
```

For further information on `feval` and `fplot`, type the following commands at the main command window.

```
help feval
help fplot
```

Exercise 1 - Plotting elevation data

Southampton is surrounded by many great places to visit. These include Bournemouth, the New Forest, Salisbury, Winchester, Portsmouth and the Isle of Wight. Some of these are shown in the satellite imagery of Figure 1. In this exercise, we will use Matlab to plot the elevation of the area surrounding Southampton.

The elevation data for the area surrounding Southampton is stored in the file `elevation.dat`, which can be downloaded from

```
http://www.ecs.soton.ac.uk/notes/elec6021/matlab/elevation
```

You should save this file into the directory that is shown as 'Current directory' at the top of your Matlab window. You can then load the data into Matlab using the command

```
load elevation.dat
```

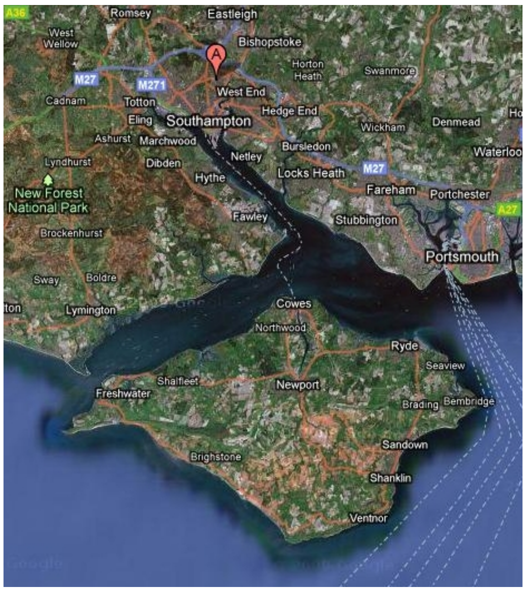

Figure 1: Satellite imagery of Southampton, Portsmouth, the New Forest, the Solent and the Isle of Wight. The label 'A' indicates the Highfield Campus at the University of Southampton. ©Google 2009.

The resultant matrix `elevation` contains the elevations (in metres above sea-level) of positions arranged in a grid at intervals of 75 m, over an area of 50 km by 50 km.

We need two vectors to describe the co-ordinates of the data points: one for the East-West co-ordinates and one for the North-South co-ordinates. The length of the East-West vector should be equal to the number of columns in the `elevation` matrix. You can therefore obtain this vector using the command

```
x = (0:size(elevation,2)-1)*75;
```

where `75` is used because the elevations are recorded for positions that are spaced 75 m apart. Use Matlab's `help` command if you're not sure what the extra argument for the `size` function is for. Similarly, the length of the North-South vector should be equal to the number of *rows* in the `elevation` matrix. Adjust the argument of the `size` function in the command above to obtain the North-South vector `y`.

You can generate a 3D surface plot of the elevation data using the command

```
surf(x,y,elevation,'LineStyle','none');
```

Here, `'LineStyle','none'` is used to remove the grid that would otherwise be drawn over the surface plot, obscuring the colours from view.

You can give appropriate lengths to the three axes of the plot by using the commands

```
xlim([0,max(x)])
ylim([0,max(y)])
zlim([-1000,1000])
axis square
```

You can use the 'Rotate 3D' tool in the plot window to view it from different angles. Note that in order to make the plot more exciting, the commands I provided above make the hills look like mountains! You can obtain a more realistic view by typing the command

```
axis equal
```

I think you'll agree though that this makes the plot look too boring! You can undo this by typing the following command again.

```
axis square
```

You can find out more about the axis command by typing

```
help axis
```

Remember that it is always important to annotate the axes of your plots and to include the units. You can do this using the commands

```
xlabel('Easting (m)')
ylabel('Northing (m)')
zlabel('Elevation (m)')
```

If you like, you can also include a key to explain the various colours by using the `colorbar` command.

Next, let's draw a contour plot for the elevation data. You can do this using the command

```
contourf(x,y,elevation)
```

Remember to set the lengths of the axes and to annotate them, as described above.

Next, we can merge the elevation data with the satellite imagery of Figure 1 to generate a 3D representation of the area surrounding Southampton. Start by downloading the JPEG `satellite.jpg` from the URL provided above. You can load this into Matlab using the command

```
satellite = imread('satellite.jpg');
```

You can display the image by typing the command

```
imshow(satellite)
```

Notice that the image shows the exact same area as the elevation data; this took quite a lot of trial and error to get right!

You can merge the satellite imagery and the elevation data by using the command

```
warp(x,y,flipud(elevation),satellite)
```

This command maps an image onto a surface plot. Here, the `flipud` function is required because the rows of an image are numbered from top to bottom, not from bottom to top like the y axis of a plot. You can find out more about the `flipud` command using Matlab's `help` function. Again. remember to set the lengths of the axes and to annotate them, as described above.

Try using the 'Zoom In' and 'Rotate 3D' tools in the plot window to see what the Isle of Wight looks like when you're stood at the top of the tallest building on the Highfield Campus at the University of Southampton. Alternatively, you can change the viewing angle using the command

```
view(az,el)
```

where `az` is the azimuth in the range 0 to 360 degrees and `el` is the elevation in the range 0 to 90 degrees. Also, you can zoom in using the command

```
zoom(2)
```

and zoom out using the command

```
zoom(0.5)
```

Finally, you can draw the contours into the same plot by using the commands

```
hold on
contour3(x,y,flipud(elevation))
```

Again, you may like to include a key by typing the `colorbar` command. Pretty neat, huh?

Exercise 2 Random integer generators

In this exercise we shall consider some random integer generators. Different random integer generators may output different integers with different probabilities. For example, a six-sided dice is a random integer generator that outputs the integers 1 to 6, each with a probability of 1/6. The probability of a six-sided dice outputting any integer outside the range 1 to 6 is zero. By contrast, a ten-sided dice is a different random integer generator that outputs the integers 1 to 10, each with a probability of 1/10.

Some random integer generators may output different integers with different probabilities. For example, the act of rolling two six-sided dice, dividing the results by two, rounding up to the nearest integers and summing the two resultant integers together can be considered to be a random integer generator. This outputs the integers 2 and 6, each with a probability of 1/9, the integers 3 and 5, each with a probability of 2/9 and the integer 4 with a probability of 3/9.

You can write a Matlab function that will output the probability $P(s, k)$ of an s-sided dice outputting the integer k as follows

```
% Function that determines the probability p of an s-sided dice
% outputting the integer k
function p=uniform_dist(s,k)
% Check that the inputs are integers and in the correct range
if round(k) ~= k
    error('k should be an integer!');
end
if round(s) ~= s
    error('s should be an integer!');
end
if s < 1
    error('s should be positive!');
end
% Calculate the integer probability
if k >= 1 && k <= s
    p=1/s;
else
    p=0;
end
```

Note that this function will generate an error if either s or k is not an integer or if s has a value less than 1. It is always a good idea to include some error checking in your functions so that they only ever return meaningful values. This function also includes some comments. In general, comments are invaluable when you want to share your code with a friend; without any comments, your friend would have no idea what your Matlab code does!

The function listed above is called `uniform_dist`, since it describes a discrete uniform probability distribution, in which

$$P(s,k) = \begin{cases} 1/s & \text{if } 1 \leq k \leq s \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $s \geq 1$ and k are integers.

Your first task is to write similar functions for the discrete geometric and zeta probability distributions, remembering to include some error checking. In the discrete geometric distribution

$$P(s,k) = \begin{cases} (s-1)s^{-k} & \text{if } k \geq 1 \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

where k is an integer and $s > 1$ is a real-valued parameter of the distribution, similar to in the discrete uniform distribution described above. Meanwhile, the discrete zeta distribution is defined by

$$P(s,k) = \begin{cases} \frac{k^{-s}}{\zeta(s)} & \text{if } k \geq 1 \\ 0 & \text{otherwise} \end{cases}, \tag{3}$$

where the zeta function ζ is implemented by Matlab's built-in function `zeta`, k is an integer and $s > 1$ is real-valued.

The following function determines the sum of the probabilities of the integers in the vector `k_values`, when using the probability distribution that is specified using the function handle `P` and a particular value of s.

```
% Function that determines the sum of the probabilities of the
% integers in the vector k_values, when using the probability
% distribution that is specified using the function handle P
% and the parameter s.
function sum = sum_P(P,s,k_values)
% Initialise the sum to zero
sum = 0;
% Accumulate the probabilities of the various integers
for index = 1:length(k_values)
    sum = sum + feval(P,s,k_values(index));
end
```

For each of your probability distributions and for a range of values for $s > 1$, use the sum_P function to verify that no matter what vector of k_values you use, the sum of the probabilities is never greater than 1. This demonstrates that our probability distributions are valid.

The following function generates random integers using the probability distribution that is specified using the function handle P and a particular value of s.

```
% Function that generates random integers using the
% probability distribution that is specified using the
% function handle P and a the parameter s.
function k = random(P,s)
% Generate a random number in the range 0 to 1, taken from a
% continuous uniform distribution
a = rand;
% Initialise the integer
k=1;
% Find the integer value having the probability that matches
% the random number
while a > feval(P,s,k)
    a = a - feval(P,s,k);
    k = k + 1;
end
```

Your next task is to figure out how this works. To help you out with this, Figure 2 provides a flow chart for the random function. Try generating a bunch of random integers using each of your probability distribution functions. Note that to start with, $s = 2$ is a good choice for the parameter values of the discrete geometric and zeta distributions. See what happens to the random integers generated by these distributions as you use larger and larger values for s.

It is easier to understand what is happening to the random integers as s changes by plotting $P(s, k)$ versus k. Your final task in this exercise is to write a Matlab function that has a similar functionality to fplot, but which is specifically designed to plot $P(s, k)$ versus k for a specified probability distribution and value of s. The first line of your function should therefore be

```
function plot_distribution(P,s,k_values)
```

where P is a function handle and k_values is a vector that specifies which values of k to use. You may find that drawing a flow diagram makes it easier

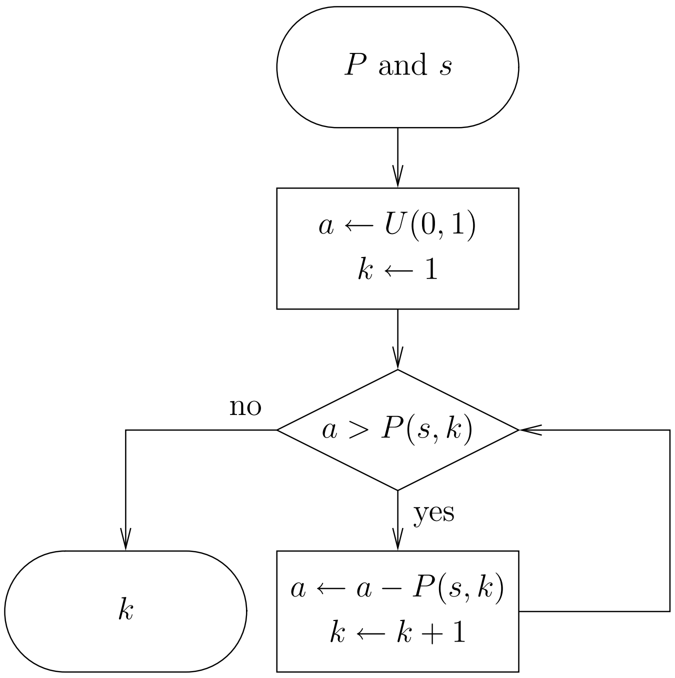

Figure 2: Flow diagram for the `random` function. Here, $U(0,1)$ generates a random real value from a *continuous* uniform distribution over the range 0 to 1.

to work out what loops are required. To help you out with this exercise, the output you should get for `plot_distribution('geometric_dist',2,1:10)` is exemplified in Figure 3. Try to get your function to automatically generate plots that look like Figure 3, without having to manually insert titles and axis labels in the plot window.

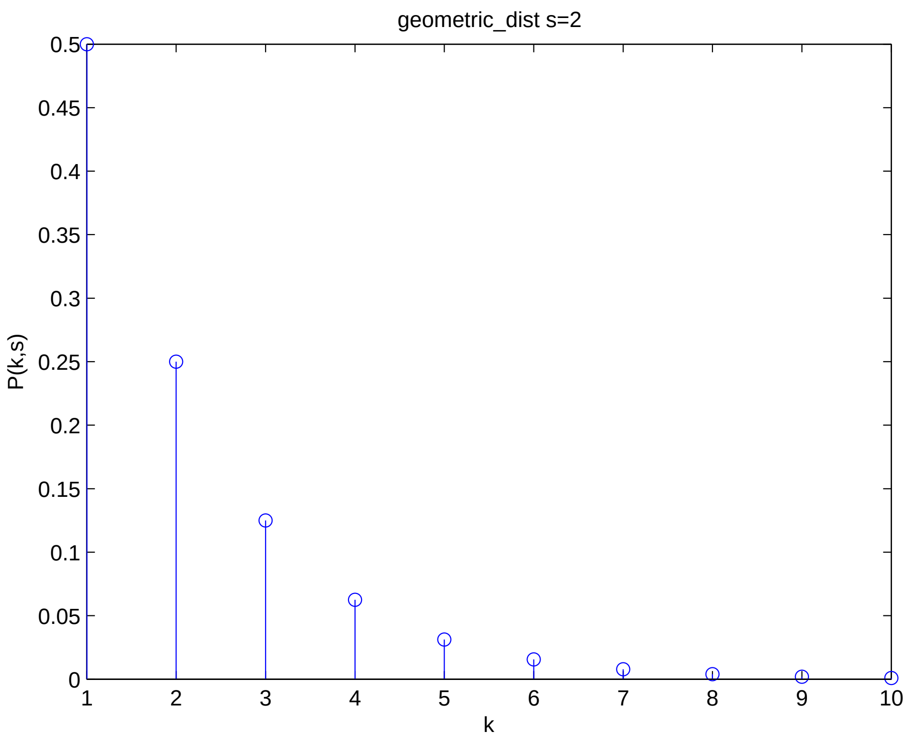

Figure 3: Discrete geometric probability distribution for $s = 2$.

Use your `plot_distribution` function to see how the probability distributions change as you vary the value of s. You should find that this explains the observations you made about the random integers generated by the discrete geometric and zeta distributions, when you used larger and larger values of s.

Finally, you may like to write a Matlab script that repeatedly calls the `random` function and collects statistics about how many times each integer value is generated. You can then compare these statistics with the probabilities shown in the plots generated using your `plot_distribution` function. In this way, you can verify that the `random` function generates each integer value with the correct probability.

Exercise 3 Sudoku checker

A sudoku puzzle consists of a 9×9 grid of numbers, most of which are initially unknown to the player. The objective is to fill in the missing numbers by exploiting the following three rules:

- each of the integers in the range 1 to 9 must appear in each *row* of the grid exactly once;
- each of the integers in the range 1 to 9 must appear in each *column* of the grid exactly once;
- each of the integers in the range 1 to 9 must appear in each 3 × 3 *box* in the grid exactly once.

An example of a sudoku puzzle is shown in Figure 4.

(a)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(b)

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4: (a) A sudoku puzzle and (b) its solution.

Your task in this exercise is to write a Matlab function that checks whether or not a sudoku solution adheres to the three rules listed above. The first line of your function should be

```
function valid = check_sudoku(solution)
```

where `solution` is expected to be a 9 × 9 matrix of integers and `valid` is given a value of 1 when the solution adheres to the above-listed rules and a value of 0 otherwise.

Remember to include comments and some error checking in your function. This should check that `solution` is a 9 × 9 matrix of integers in the range 1 to 9. You should also try to make your function as *elegant* and *efficient* as possible. You may find that Matlab's built-in function `unique` is useful in this exercise. You can find out how to use this by using Matlab's `help` facility.

Once you have written your function, test its functionality using the data files that can be downloaded from

```
http://www.ecs.soton.ac.uk/notes/elec6021/matlab/sudoku
```

You can perform the tests by saving the data files to your Matlab working directory and using commands like the following ones.

```
load solution1.dat
check_sudoku(solution1)
```

Note that some of the data files will test the error handling of your function.

Exercise 4 - Sorting

In this exercise, your task is to write a Matlab function that will sort the elements of a vector into ascending order. There are lots of different algorithms that will achieve this and many are described on Wikipedia at

```
http://en.wikipedia.org/wiki/Sorting_algorithm
```

Feel free to pick whichever algorithm you like, but notice that different algorithms have different speeds, memory requirements and algorithmic complexities. You should draw a flowchart for your algorithm to help you visualise its operation.

The first line of your function should be

```
function out = my_sort(in)
```

where `in` is expected to be a vector and `out` should contain the same elements as `in`, but sorted into ascending order. Try to make your function as elegant and efficient as possible and remember to include some useful comments.

You can check that your algorithm works by using the Matlab command

```
issorted(my_sort(rand(1,10000)))
```

which will use your algorithm to sort a vector of 10000 random numbers and return a 1 if it is successful or a 0 if not.

You can test the efficiency of your algorithm using the command

```
tic; my_sort(rand(1,10000)); toc
```

This will tell you how long it takes for your algorithm to sort a vector of 10000 random numbers. You may like to compare this to the efficiency of Matlab's built-in sort function by using the command

```
tic; sort(rand(1,10000)); toc
```

You can find out about the `tic` and `toc` commands by using the commands

```
help tic
help toc
```

Rob Maunder
rm@ecs.soton.ac.uk
September 2010