Homework 4

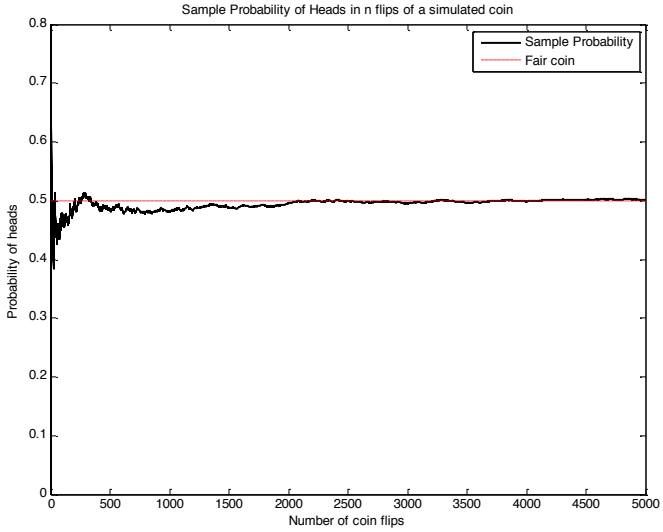
This homework is designed to give you practice with more advanced and specific Matlab functionality, like advanced data structures, images, and animation. As before, the names of helpful functions are provided in **bold** where needed. **Homework must be submitted before the start of the next class.**

What to turn in: Copy the text from your scripts and paste it into a document. If a question asks you to plot or display something to the screen, also include the plot and screen output your code generates. Submit either a *.doc or *.pdf file.

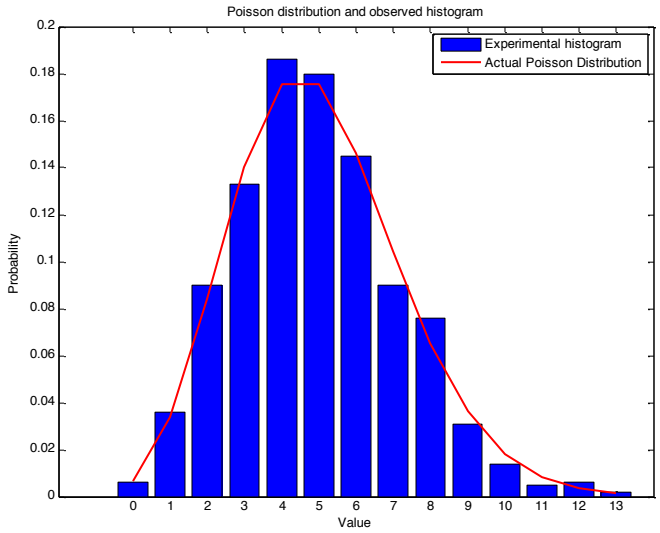
Keep all your code in scripts/functions. If a specific name is not mentioned in the problem statement, you can choose your own script names.

1. **Random variables.** Make a vector of 500 random numbers from a Normal distribution with mean 2 and standard deviation 5 (**randn**). After you generate the vector, verify that the sample mean and standard deviation of the vector are close to 2 and 5 respectively (**mean**, **std**).

2. **Flipping a coin.** Write a script called `coinTest.m` to simulate sequentially flipping a coin 5000 times. Keep track of every time you get 'heads' and plot the running estimate of the probability of getting 'heads' with this coin. Plot this running estimate along with a horizontal line at the expected value of 0.5, as below. This is most easily done without a loop (useful functions: **rand**, **round**, **cumsum**).



3. **Histogram.** Generate 1000 Poisson distributed random numbers with parameter $\lambda = 5$ (**poissrnd**). Get the histogram of the data and normalize the counts so that the histogram sums to 1 (**hist** – the version that returns 2 outputs N and X, **sum**). Plot the normalized histogram (which is now a probability mass function) as a bar graph (**bar**). Hold on and also plot the actual Poisson probability mass function with $\lambda = 5$ as a line (**poisspdf**). You can try doing this with more than 1000 samples from the Poisson distribution to get better agreement between the two. **Hint:** By default **hist** gives 10 equally-spaced bins; we want one bar for each non-negative integer.



4. **Practice with cells.** Usually, cells are most useful for storing strings, because the length of each string can be unique.
    a. Make a 3x3 cell where the first column contains the names: 'Joe', 'Sarah', and 'Pat', the second column contains their last names: 'Smith', 'Brown', 'Jackson', and the third column contains their salaries: $30,000, $150,000, and $120,000. Display the cell using **disp**.
    b. Sarah gets married and changes her last name to 'Meyers'. Make this change in the cell you made in a. Display the cell using **disp**.
    c. Pat gets promoted and gets a raise of $50,000. Change his salary by adding this amount to his current salary. Display the cell using **disp**.

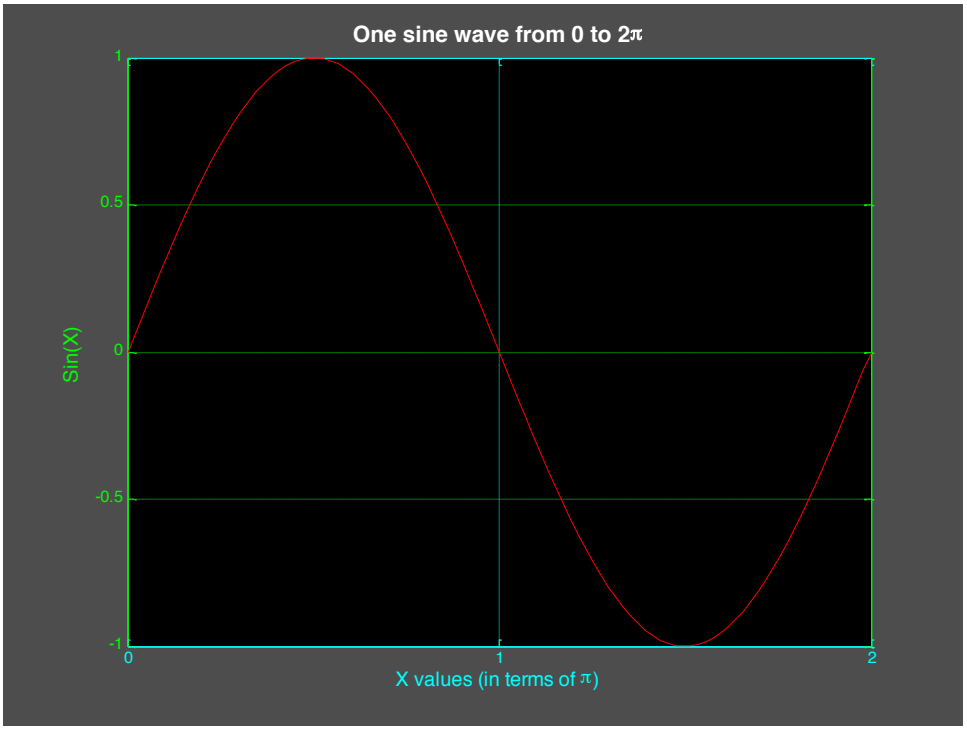
    The output to parts a-c should look like this:

```
>> cellProblem
    'Joe'       'Smith'       [ 30000]
    'Sarah'     'Brown'       [150000]
    'Pat'       'Jackson'     [120000]

    'Joe'       'Smith'       [ 30000]
    'Sarah'     'Meyers'      [150000]
    'Pat'       'Jackson'     [120000]

    'Joe'       'Smith'       [ 30000]
    'Sarah'     'Meyers'      [150000]
    'Pat'       'Jackson'     [170000]
```

5. **Using Structs**. Structs are useful in many situations when dealing with diverse data. For example, get the contents of your current directory by typing `a=dir;`
    a. `a` is a struct array. What is its size? What are the names of the fields in `a`?
    b. Write a loop to go through all the elements of a, and if the element is not a directory, display the following sentence 'File *filename* contains X bytes', where *filename* is the name of the file and X is the number of bytes.
    c. Write a function called `displayDir.m`, which will display the sizes of the files in the current directory when run, as below:

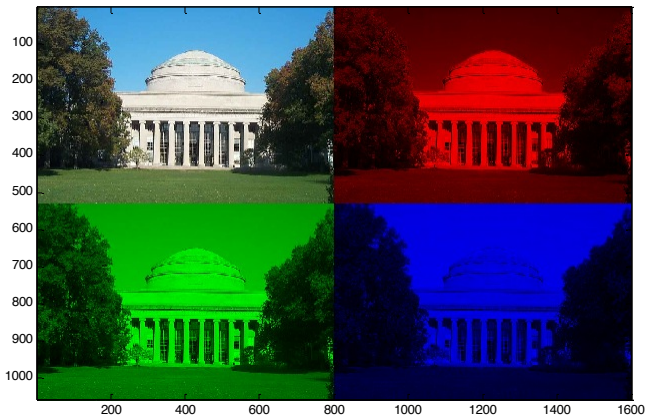
```
>> displayDir
File brown2D.m contains 417 bytes.
File coinTest.m contains 524 bytes.
File displayDir.m contains 304 bytes.
File plotPoisson.m contains 543 bytes.
File someData.txt contains 66 bytes.
```

6. **Handles.** We'll use handles to set various properties of a figure in order to make it look like this:



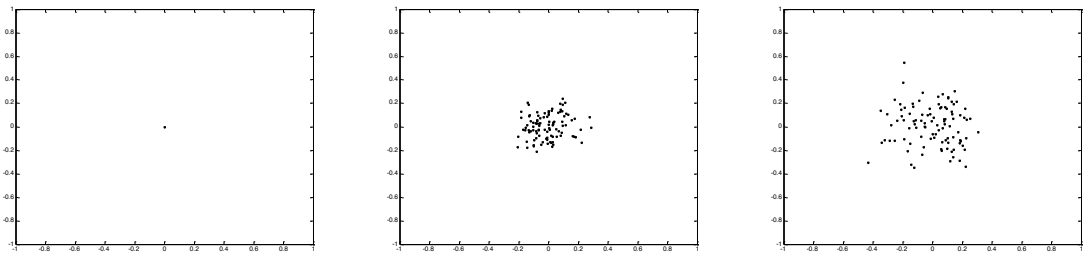
a. Do all the following in a script named `handlesPractice.m`
b. First, make a variable `x` that goes from 0 to 2π, and then make `y=sin(x)`.
c. Make a new `figure` and do `plot(x,y,'r')`
d. Set the x limit to go from 0 to 2π (**xlim**)
e. Set the `xtick` property of the axis to be just the values `[0 pi 2*pi]`, and set `xticklabel` to be `{'0','1','2'}`. Use **set** and **gca**
f. Set the `ytick` property of the axis to be just the values `-1:.5:1`. Use **set** and **gca**
g. Turn on the grid by doing `grid on`.
h. Set the `ycolor` property of the axis to green, the `xcolor` property to cyan, and the `color` property to black (use **set** and **gca**)
i. Set the `color` property of the figure to a dark gray (I used [.3 .3 .3]). Use **set** and **gcf**
j. Add a title that says 'One sine wave from 0 to 2π' with `fontsize` 14, `fontweight` bold, and `color` white. Hint: to get the π to display properly, use \pi in your string. Matlab uses a Tex or Latex interpreter in xlabel, ylabel, and title. You can do all this just by using **title**, no need for handles.
k. Add the appropriate x and y labels (make sure the π shows up that way in the x label) using a `fontsize` of 12 and `color` cyan for x and green for y. Use **xlabel** and **ylabel**
l. Before you copy the figure to paste it into word, look at copy options (in the figure's Edit menu) and under 'figure background color' select 'use figure color'.

7. **Image processing**. Write a function to display a color image, as well as its red, green, and blue layers separately. The function declaration should be `im=displayRGB(filename)`. `filename` should be the name of the image (make the function work for *.jpg images only). `im` should be the final image returned as a matrix. To test the function, you should put a jpg file into the same directory as the function and run it with the filename (include the extension, for example `im=displayRGB('testImage.jpg')`). You can use any picture you like, from your files or off the internet. Useful functions: **imread**, **meshgrid**, **interp2**, **uint8**, **image**, **axis equal**, **axis tight**.
    a. To make the program work efficiently with all image sizes, first interpolate each color layer of the original image so that the larger dimension ends up with 800 pixels. The smaller dimension should be appropriately scaled so that the length:width ratio stays the same. Use **interp2** with cubic interpolation to resample the image. **Hint:** The image is an MxNx3 matrix; you need to interpolate each of the 3 color layers separately. **Note:** If you have difficulty doing this section, try part (b) first. (But if you get part (b) working, try to do this and use the interpolated image for part (b).)
    b. Create a composite image that is 2 times as tall as the original, and 2 times as wide. Place the original image in the top left, the red layer in the top right, the green layer in the bottom left, and the blue layer in the bottom right parts of this composite image. The function should return the composite image matrix in case you want to save it as a jpg again (before displaying or returning, convert the values to unsigned 8-bit integers using **uint8**). **Hint:** To get just a single color layer, all you have to do is set the other two layers to zero. For example if `X` is an MxNx3 image, then `X(:,:,2)=0; X(:,:,3)=0;` will retain just the red layer. Include your code and the final image in your homework writeup. It should look something like this:



8. **Animation: Brownian motion**. Write a function with the following declaration: `brown2D(N)`. The function takes in a single input `N`, which is an integer specifying the number of points in the simulation. All the points should initially start at the origin (0,0). Plot all the points on a figure using '.' markers, and set the axis to be square and have limits from -1 to 1 in both the x and y direction. To simulate Brownian motion of the points, write a 1000-iteration loop which will calculate a new x and y position for each point and will display these new positions as an animation. The new position of each point is obtained by adding a normally distributed random variable with a standard deviation of 0.005 to each x and y value (use **randn**; if you have 100 points, you need to add 100 distinct random values to the x values and 100 distinct random values to the y values). Each time that the new positions of all the points are calculated, plot them on the figure and **pause** for 0.01 seconds (it's best to use **set** and the line object handle in order to update the **xdata** and **ydata** properties of the points, as mentioned in lecture).

   What you will see is a simulation of diffusion, wherein the particles randomly move away from the center of the figure. For example, 100 points look like the figures below at the start, middle, and end of the simulation:



Optional Problem

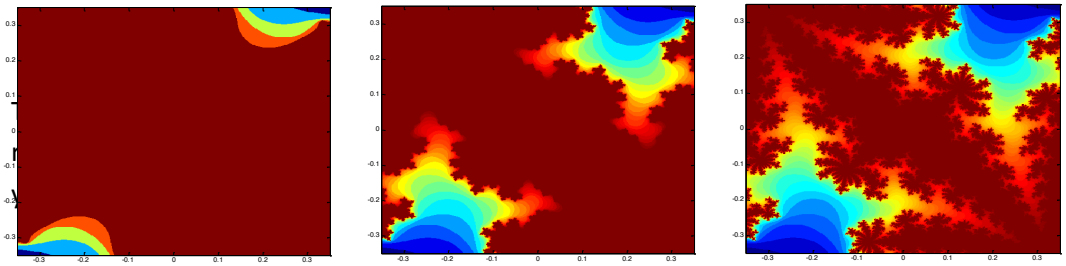
9. **Optional: Julia Set Animation.** In this problem, we will animate the Julia Set problem from Homework 3. If you haven't done the Julia Set problem in HW3, you can still do this one because the `julia` program has to be rewritten anyway. To review: two complex numbers z_0 and c define a recursive relationship for subsequent values of z_n with $n = 1, 2, 3...$:

$$z_n = z_{n-1}^2 + c$$

The escape velocity of a particular z_0 is equal to the time step n at which $|z_n| > 2$. You will write a program to compute the escape velocities of an entire grid of z_0 values. To make the animation look cool, we'll have to do things a bit differently than they were done in HW3: in HW3, you computed the escape velocity of a particular z_0, then moved on to the next z_0 and computed its escape velocity, etc. In this version of the program. We'll compute z_{n+1} for the entire grid of z_0's, and for each n, find and mark all the z_0's that 'escape' for that n and display them. The following steps will lead you to this end

   a. Write the function `juliaAnimation(zMax,c,N)`, which takes three inputs: `zMax` is a real number that describes the section of the complex plane that we'll be looking at. We're going to compute escape velocities for numbers with real part between `-zMax` and `zMax`, and imaginary part between `-zMax` and `zMax`. `c` is the c parameter, and `N` is the maximum number of iterations (and since we're displaying each iteration as a frame in our animation, it's also the number of frames we'll see).
   b. In this function, first make a vector `temp`, which has 500 values between `-zMax` and `zMax`. Then, use `temp` within **meshgrid** to make `R` and `I`, which are two real matrices containing the real and imaginary parts of a complex matrix `Z`. The convention in the complex plane is to have the real part vary across the x dimension, and the imaginary part vary across the y dimension. To make `Z`, simply add up `R` and `i*I` (you have to multiply `I` by the imaginary number `i` in order to make it imaginary). Now Z contains 250,000 complex numbers, and we'll compute escape velocities for all of them.
   c. Initialize a matrix `M` to be the same size as `Z`, and whose elements all have the value `N`.
   d. Now, write a loop that will iterate for values of `n` from 1 to `N`. In each iteration of the loop, compute the new `Z` using the equation $z_n = z_{n-1}^2 + c$. Then, **find** the indices of all the numbers in `Z` which have 'escaped' ($|z_n| > 2$), and set those indices in `M` to have the value of the current loop iteration `n`. Because these numbers have 'escaped', we don't care about them anymore, so set all those indices in `Z` to be `NaN`. Display the `M` matrix by using `imagesc(temp,temp,atan(0.1*M)); axis xy; drawnow;` . Because the computation takes a while to do on its own, there is no need to do an explicit `pause`, the `drawnow` command tells the plot to update immediately rather than waiting for the loop to finish.

e. When you run the function with something like : `juliaAnimation(.35,-.297491+i*.641051,100)`, you will see the Julia Set come to life. The figures below are from the beginning, middle, and end of the animation:



f. Try your animation with various values of `zMax` and `c` to see various fractals. See the Wikipedia entry http://en.wikipedia.org/wiki/Julia_set under 'Quadratic polynomials' for suggested values of `c` to make cool fractals, for example `c=-0.8+i*0.156` gives the image below.