Homework 1

This homework is designed to teach you to think in terms of matrices and vectors because this is how Matlab organizes data. You will find that complicated operations can often be done with one or two lines of code if you use appropriate functions and have the data stored in an appropriate structure. The other purpose of this homework is to make you comfortable with using **help** to learn about new functions. The names of the functions you'll need to look up are provided in **bold** where needed. **Homework must be submitted on the stellar website 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.

For problems 1-7, write a script called `shortProblems.m` and put all the commands in it. Separate and label different problems using comments.

1. **Scalar variables.** Make the following variables
   a. $a = 10$
   b. $b = 2.5 \times 10^{23}$
   c. $c = 2 + 3i$, where i is the imaginary number
   d. $d = e^{j2\pi/3}$, where j is the imaginary number and e is Euler's number (use **exp**, **pi**)

2. **Vector variables.** Make the following variables
   a. $aVec = \begin{bmatrix} 3.14 & 15 & 9 & 26 \end{bmatrix}$
   b. $bVec = \begin{bmatrix} 2.71 \\ 8 \\ 28 \\ 182 \end{bmatrix}$
   c. $cVec = \begin{bmatrix} 5 & 4.8 & \cdots & -4.8 & -5 \end{bmatrix}$ (all the numbers from 5 to -5 in increments of -0.2)
   d. $dVec = \begin{bmatrix} 10^0 & 10^{0.01} & \cdots & 10^{0.99} & 10^1 \end{bmatrix}$ (logarithmically spaced numbers between 1 and 10, use **logspace**, make sure you get the length right!)
   e. $eVec = Hello$ ($eVec$ is a string, which is a vector of characters)

3. **Matrix variables.** Make the following variables
   a. $aMat = \begin{bmatrix} 2 & \cdots & 2 \\ \vdots & \ddots & \vdots \\ 2 & \cdots & 2 \end{bmatrix}$ a 9x9 matrix full of 2's (use **ones** or **zeros**)

b. $bMat = \begin{bmatrix} 1 & 0 & \cdots & & 0 \\ 0 & \ddots & 0 & \ddots & \\ \vdots & 0 & 5 & 0 & \vdots \\ & \ddots & 0 & \ddots & 0 \\ 0 & & \cdots & 0 & 1 \end{bmatrix}$ a 9x9 matrix of all zeros, but with the values $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$ on the main diagonal (use **zeros**, **diag**).

c. $cMat = \begin{bmatrix} 1 & 11 & \cdots & 91 \\ 2 & 12 & \ddots & 92 \\ \vdots & \vdots & \ddots & \vdots \\ 10 & 20 & \cdots & 100 \end{bmatrix}$ a 10x10 matrix where the vector 1:100 runs down the columns (use **reshape**).

d. $dMat = \begin{bmatrix} NaN & NaN & NaN & NaN \\ NaN & NaN & NaN & NaN \\ NaN & NaN & NaN & NaN \end{bmatrix}$ a 3x4 NaN matrix (use **nan**)

e. $eMat = \begin{bmatrix} 13 & -1 & 5 \\ -22 & 10 & -87 \end{bmatrix}$

f. Make $fMat$ be a 5x3 matrix of random integers with values on the range -3 to 3 (use **rand** and **floor** or **ceil**)

4. **Scalar equations.** Using the variables created in 1, calculate x, y, and z.

a. $x = \dfrac{1}{1+e^{(-(a-15)/6)}}$

b. $y = \left(\sqrt{a} + \sqrt[21]{b}\right)^{\pi}$, recall that $\sqrt[g]{h} = h^{1/g}$, and use **sqrt**

c. $z = \dfrac{\log\left(\Re\left[(c+d)(c-d)\right]\sin(a\pi/3)\right)}{c\bar{c}}$ where $\Re$ indicates the real part of the complex number in brackets, $\bar{c}$ is the complex conjugate of c, and $\log$ is the *natural* log (use **real**, **conj**, **log**).

5. **Vector equations.** Using the variables created in 2, solve the equations below, elementwise. For example, in part a, the first element of $xVec$ should just be the function evaluated at the value of the first element of $cVec$: $xVec_1 = \dfrac{1}{\sqrt{2\pi 2.5^2}} e^{-cVec_1^2/(2\cdot 2.5^2)}$, and similarly for all the other elements so that $xVec$ and $cVec$ have the same size. Use the elementwise operators **.***, **./**, **.^**.

a. $xVec = \frac{1}{\sqrt{2\pi 2.5^2}} e^{-cVec^2/(2\cdot 2.5^2)}$

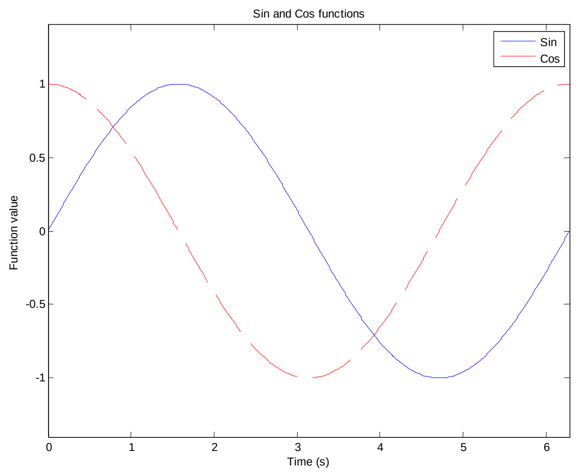
b. $yVec = \sqrt{\left(aVec^T\right)^2 + bVec^2}$, $aVec^T$ indicates the transpose of $aVec$

c. $zVec = \log_{10}(1/dVec)$, remember that $\log_{10}$ is the log base 10, so use **log10**

6. **Matrix equations.** Using the variables created in 2 and 3, solve the equations below. Use matrix operators.
   a. $xMat = (aVec \bullet bVec) \bullet aMat^2$
   b. $yMat = bVec \bullet aVec$, note that this is *not* the same as $aVec \bullet bVec$
   c. $zMat = |cMat|(aMat \bullet bMat)^T$, where $|cMat|$ is the determinant of $cMat$, and T again indicates the transpose (use **det**).

7. **Common functions and indexing.**
   a. Make $cSum$ the column-wise sum of $cMat$. The answer should be a row vector (use **sum**).
   b. Make $eMean$ the mean across the rows of $eMat$. The answer should be a column (use **mean**).
   c. Replace the top row of $eMat$ with $[1 \quad 1 \quad 1]$.
   d. Make $cSub$ the submatrix of $cMat$ that only contains rows 2 through 9 and columns 2 through 9.
   e. Make the vector $lin = [1 \quad 2 \quad \cdots \quad 20]$ (the integers from 1 to 20), and then make every other value in it negative to get $lin = [1 \quad -2 \quad 3 \quad -4 \quad \cdots \quad -20]$.
   f. Make r a 1x5 vector using **rand**. Find the elements that have values <0.5 and set those values to 0 (use **find**).

8. **Plotting multiple lines and colors.** In class we covered how to plot a single line in the default blue color on a plot. You may have noticed that subsequent plot commands simply replace the existing line. Here, we'll write a script to plot two lines on the same axes.
   a. Open a script and name it `twoLinePlot.m`. Write the following commands in this script.
   b. Make a new figure using **figure**
   c. We'll plot a sine wave and a cosine wave over one period
      i. Make a time vector t from 0 to 2π with enough samples to get smooth lines
      ii. Plot $\sin(t)$
      iii. Type **hold on** to turn on the 'hold' property of the figure. This tells the figure not to discard lines that are already plotted when plotting new ones. Similarly, you can use **hold off** to turn off the hold property.
      iv. Plot $\cos(t)$ using a red dashed line. To specify line color and style, simply add a third argument to your plot command (see the third paragraph of the **plot** help). This argument is a string specifying the line properties as described in the help file. For example, the string 'k:' specifies a black dotted line.
   d. Now, we'll add labels to the plot
      i. Label the x axis using **xlabel**
      ii. Label the y axis using **ylabel**
      iii. Give the figure a title using **title**
      iv. Create a legend to describe the two lines you have plotted by using **legend** and passing to it the two strings 'Sin' and 'Cos'.
   e. If you run the script now, you'll see that the x axis goes from 0 to 7 and y goes from -1 to 1. To make this look nicer, we'll manually specify the x and y limits. Use **xlim** to set the x axis to be from 0 to 2π and use **ylim** to set the y axis to be from -1.4 to 1.4.
   f. Run the script to verify that everything runs right. You should see something like this:



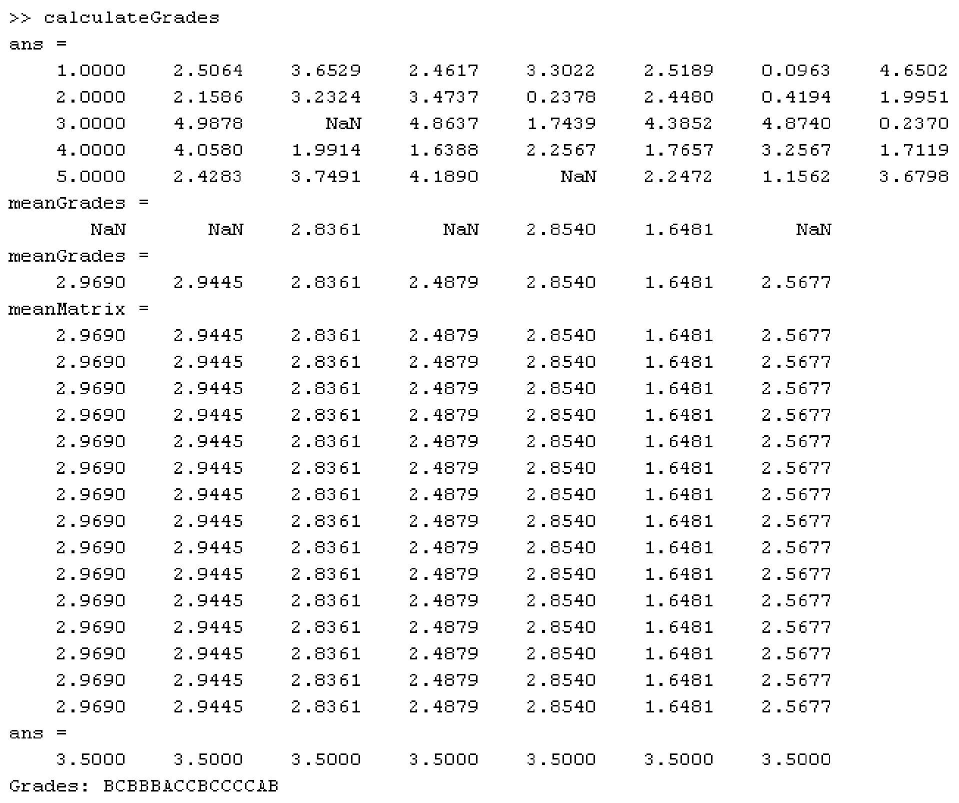
Optional Problems

9. **Optional: Manipulating variables.** Write a script to read in some grades, curve them, and display the overall grade. To do this, you'll need to download the file `classGrades.mat` off the class website and put it in the same folder as your script.
    a. Open a script and name it `calculateGrades.m`. Write all the following commands in this script.
    b. Load the `classGrades` file using **load**. This file contains a single variable called $namesAndGrades$
    c. To see how $namesAndGrades$ is structured, display the first 5 rows on your screen. The first column contains the students 'names', they're just the integers from 1 to 15. The remaining 7 columns contain each student's score (on a scale from 0 to 5) on each of 7 assignments. There are also some NaNs which indicate that a particular student was absent on that day and didn't do the assignment.
    d. We only care about the grades, so extract the submatrix containing all the rows but only columns 2 through 8 and name this matrix $grades$ (to make this work on any size matrix, don't hard-code the 8, but rather use **end** or **size(namesAndGrades,2)**).
    e. Calculate the mean score on each assignment. The result should be a 1x7 vector containing the mean grade on each assignment.
        i. First, do this using **mean**. Display the mean grades calculated this way. Notice that the NaNs that were in the grades matrix cause some of the mean grades to be NaN as well.
        ii. To fix this problem, do this again using **nanmean**. This function does exactly what you want it to do, it computes the mean using only the numbers that are not NaN. This means that the absent students are not considered in the calculation, which is what we want. Name this mean vector $meanGrades$ and display it on the screen to verify that it has no NaNs
    f. Normalize each assignment so that the mean grade is 3.5 (this is a B- on our 5 point scale). You'll want to divide each column of $grades$ by the correct element of $meanGrades$.
        i. Make a matrix called $meanMatrix$ such that it is the same size as $grades$, and each row has the values $meanGrades$. Do this by taking the outer product of a 15x1 vector of ones and the vector $meanGrades$, which is a row (use **ones**, *****). Display $meanMatrix$ to verify that it looks the way you want.
        ii. To calculate the curved grades, do the following: $curvedGrades = 3.5\left(grades / meanMatrix\right)$. Keep in mind that you want to do the division elementwise.
        iii. Compute and display the mean of $curvedGrades$ to verify that they're all 3.5 (**nanmean**).

    iv. Because we divided by the mean and multiplied by 3.5, it's possible that some grades that were initially close to 5 are now larger than 5. To fix this, find all the elements in *curvedGrades* that are greater than 5 and set them to 5. Use **find**

g. Calculate the total grade for each student and assign letter grades
    i. To calculate the *totalGrade* vector, which will contain the numerical grade for each student, you want to take the mean of *curvedGrades* across the columns (use **nanmean**, see help for how to specify the dimension). Also, we only want to end up with numbers from 1 to 5, so calculate the ceiling of the *totalGrade* vector (use **ceil**).
    ii. Make a string called *letters* that contains the letter grades in increasing order: FDCBA
    iii. Make the final letter grades vector *letterGrades* by using *totalGrade* (which after the ceil operation should only contain values between 1 and 5) to index into *letters* .
    iv. Finally, display the following using **disp**: Grades: *letterGrades*

h. Run the script to verify that it works. You should get an output like this

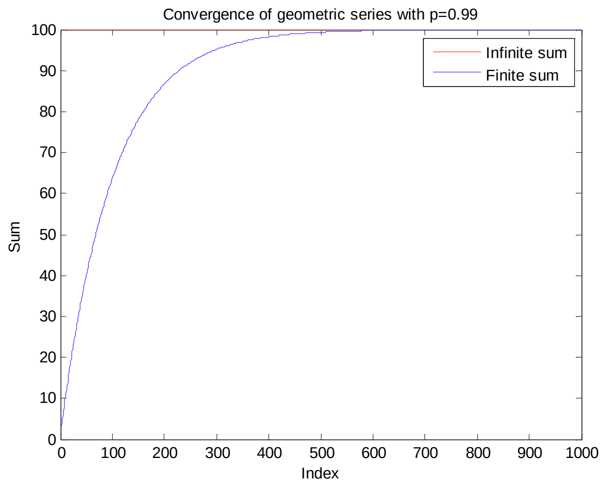


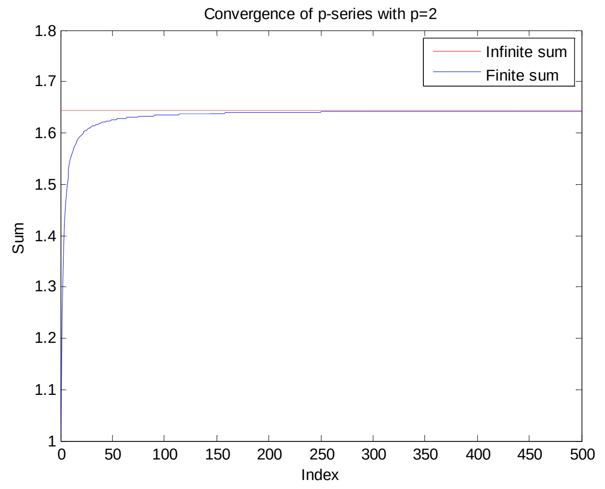
```
>> calculateGrades
ans =
    1.0000    2.5064    3.6529    2.4617    3.3022    2.5189    0.0963    4.6502
    2.0000    2.1586    3.2324    3.4737    0.2378    2.4480    0.4194    1.9951
    3.0000    4.9878       NaN    4.8637    1.7439    4.3852    4.8740    0.2370
    4.0000    4.0580    1.9914    1.6388    2.2567    1.7657    3.2567    1.7119
    5.0000    2.4283    3.7491    4.1890       NaN    2.2472    1.1562    3.6798
meanGrades =
       NaN       NaN    2.8361       NaN    2.8540    1.6481       NaN
meanGrades =
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
meanMatrix =
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
    2.9690    2.9445    2.8361    2.4879    2.8540    1.6481    2.5677
ans =
    3.5000    3.5000    3.5000    3.5000    3.5000    3.5000    3.5000
Grades: BCBBBACCBCCCCAB
```

10. **Optional: Convergence of infinite series.** We'll look at two series that converge to a finite value when they are summed.
    a. Open a new script in the Matlab Editor and save it as `seriesConvergence.m`
    b. First, we'll deal with a geometric series $G=\sum_{k=0}^{\infty} p^k$. We need to define the value of p and the values of k
        i. p =0.99.
        ii. k is a vector containing the integers from 0 to 1000, inclusive.
    c. Calculate each term in the series (before summation)
        i. $geomSeries = p^k$ (this should be done elementwise)
    d. Calculate the value of the infinite series
        i. We know that $G=\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$
    e. Plot the value of the infinite series
        i. Plot a horizontal red line that has x values 0 and the maximum value in k (use **max**), and the y value is constant at G.
    f. On the same plot, plot the value of the finite series for all values of k
        i. Plot the cumulative sum of $geomSeries$ versus k. The cumulative sum of a vector is a vector of the same size, where the value of each element is equal to the sum of all the elements to the left of it in the original vector. (use **cumsum**, and try cumsum([1 1 1 1 1]) to understand what it's doing.) Use a blue line when plotting.
    g. Label the x and y axes, and give the figure a title (**xlabel**, **ylabel**, **title**). Also create a legend and label the first line 'Infinite sum', and the second line 'Finite Sum' (**legend**).
    h. Run the script and note that the finite sum of 1000 elements comes very close to the value of the infinite sum.
    i. Next, we will do a similar thing for another series, the p-series: $P=\sum_{n=1}^{\infty}\frac{1}{n^p}$
    j. At the bottom of the same script, initialize new variables
        i. $p=2$
        ii. n is a vector containing all the integers from 1 to 500, inclusive.
    k. Calculate the value of each term in the series
        i. $pSeries = \frac{1}{n^p}$
    l. Calculate the value of the infinite p-series. The infinite p-series with $p=2$ has been proven to converge to $P=\sum_{n=1}^{\infty}\frac{1}{n^p}=\frac{\pi^2}{6}$.

m. Make a new figure and plot the infinite sum as well as the finite sum, as we did for the geometric series
    i. Make a new figure
    ii. Plot the infinite series value as a horizontal red line with x values 0 and the maximum value in n, and the y value is constant at P.
    iii. Hold on to the figure, and plot the cumulative sum of $pSeries$ versus n (use **hold on**, **cumsum**).
    iv. Label the x and y axes, give the figure a title, and make a legend to label the lines as 'Infinite sum', and 'Finite sum' (use **xlabel**, **ylabel**, **title**, **legend**)

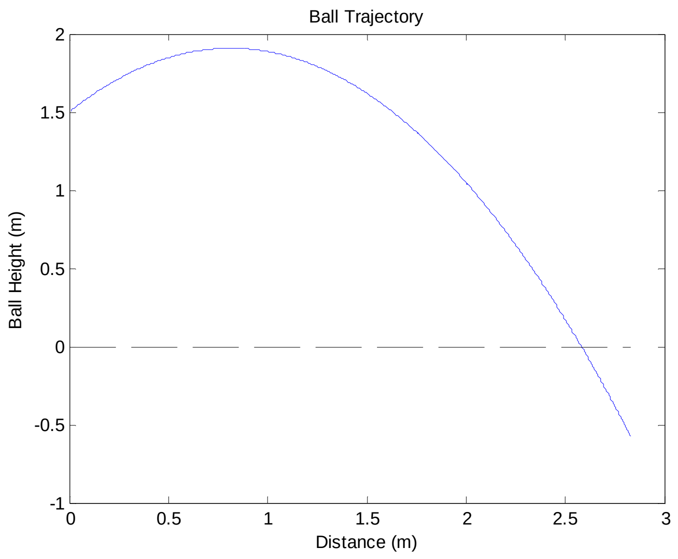
n. Run the script to verify that it produces the expected output. It should look something like this:





11. **Optional: Throwing a ball.** Below are all the steps you need to follow, but you should also add your own meaningful comments to the code as you write it.
    a. Start a new file in the Matlab Editor and save it as `throwBall.m`
    b. At the top of the file, define some constants (you can pick your own variable names)
        i. Initial height of ball at release = 1.5 m
        ii. Gravitational acceleration = 9.8 m/s^2
        iii. Velocity of ball at release = 4 m/s
        iv. Angle of the velocity vector at time of release = 45 degrees
    c. Next, make a time vector that has 1000 linearly spaced values between 0 and 1, inclusive.
    d. If x is distance and y is height, the equations below describe their dependence on time and all the other parameters (initial height h, gravitational acceleration g, initial ball velocity v, angle of velocity vector in degrees θ). Solve for x and y
        i. $x(t) = v\cos\left(\theta\frac{\pi}{180}\right)t$. We multiply θ by $\frac{\pi}{180}$ to convert degrees to radians.
        ii. $y(t) = h + v\sin\left(\theta\frac{\pi}{180}\right)t - \frac{1}{2}gt^2$
    e. Approximate when the ball hits the ground.
        i. Find the index when the height first becomes negative (use **find**).
        ii. The distance at which the ball hits the ground is value of x at that index
        iii. Display the words: *The ball hits the ground at a distance of X m*eters. (where X is the distance you found in part ii above)
    f. Plot the ball's trajectory
        i. Open a new figure (use **figure**)
        ii. Plot the ball's height on the y axis and the distance on the x axis (**plot**)
        iii. Label the axes meaningfully and give the figure a title (use **xlabel**, **ylabel**, and **title**)
        iv. Hold on to the figure (use **hold on**)
        v. Plot the ground as a dashed black line. This should be a horizontal line going from 0 to the maximum value of x (use **max**). The height of this line should be 0. (see **help plot** for line colors and styles)
    g. Run the script from the command window and verify that the ball indeed hits the ground around the distance you estimated in e,ii. You should get something like this:

```
>> throwBall
The ball hits the ground at a distance of 2.5821 meters
```



12. **Optional:** Write a simple shuffling 'encryption' algorithm.
    a. Open a new script and save it as `encrypt.m`
    b. At the top of the script, define the *original* string to be: This is my top secret message!
    c. Next, let's shuffle the indices of the letters. To do this, we need to make a string of encoding indices
        i. Make a vector that has the indices from 1 to the length of the original string in a randomly permuted order. Use **randperm** and **length**
        ii. Encode the original string by using your encoding vector as indices into *original*. Name the encoded message *encoded*.
    d. Now, we need to figure out the decoding key to match the encoding key we just made.
        i. Assemble a temporary matrix where the first column is the encoding vector you made in the previous part and the second column are the integers from 1 to the length of the original string in order. Use **length**, and you may need to transpose some vectors to make them columns using **'**.
        ii. Next, we want to sort the rows of this temporary matrix according to the values in the first column. Use **sortrows**.
        iii. After it's been sorted, extract the second column of the temporary matrix. This is your decoding vector.
        iv. To make the *decoded* message, use the decoding vector as indices into *encoded*.
    e. Display the original, encoded, and decoded messages
        i. Display the following three strings, where : *original*, *encoded*, and *decoded* are the strings you made above. Use **disp**

            Original: *original*

            Encoded: *encoded*

            Decoded: *decoded*
    f. Compare the original and decoded strings to make sure they're identical and display the result
        i. Use **strcmp** to compare the *original* and *decoded* strings. Name the output of this operation *correct*. *correct* will have the value 1 if the strings match and the value 0 if they don't
        ii. Display the following string: Decoded correctly (1 true, 0 false): *correct* use **disp** and **num2str**
    g. Run the script a few times to verify that it works well. You should see an output like this:

```
>> encrypt
Original: This is my top secret message!
Encoded : sisrpigct tessaoheem m  yTse!
Decoded : This is my top secret message!
Decoded correctly (1 true, 0 false): 1
```

Homework 2

This homework is designed to give you practice with writing functions and visualizing data. This assignment will give you more freedom than Homework 1 to choose how you implement your functions. You will just be graded on whether your functions produce the correct output, but not necessarily on how efficiently they're written. As before, the names of helpful functions are provided in **bold** where needed. **Homework must be submitted before the start of the next class.**

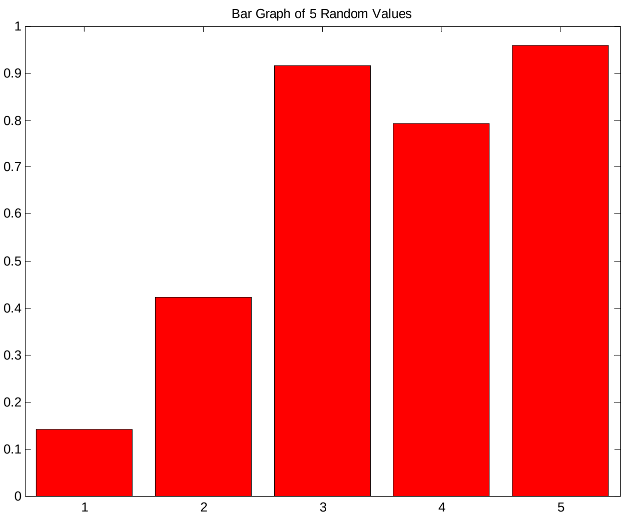
What to turn in: Copy the text from your scripts and paste it into a document. If a question asks you to plot or display something to the screen, also include the plot and screen output your code generates. Submit either a *.doc or *.pdf file.

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

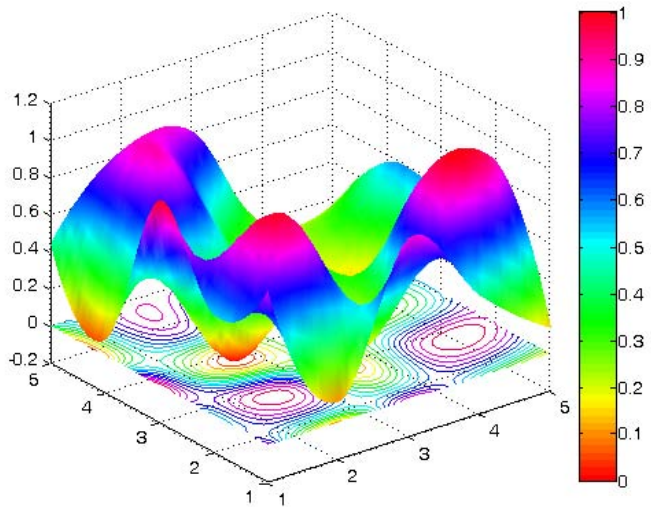
1. **Semilog plot**. Over the past 5 years, the number of students in 6.094 has been 15, 25, 55, 115, 144. Class size seems like it's growing exponentially. To verify this, plot these values on a plot with a log y scale and label it (**semilogy**, **xlabel**, **ylabel**, **title**). Use magenta square symbols of marker size 10 and line width 4, and no line connecting them. You may have to change the x limits to see all 5 symbols (**xlim**). If the relationship really is exponential, it will look linear on a log plot.

Problem 2 removed due to copyright restrictions.

3. **Bar graph**. Make a vector of 5 random values and plot them on a bar graph using red bars, something like the figure below.



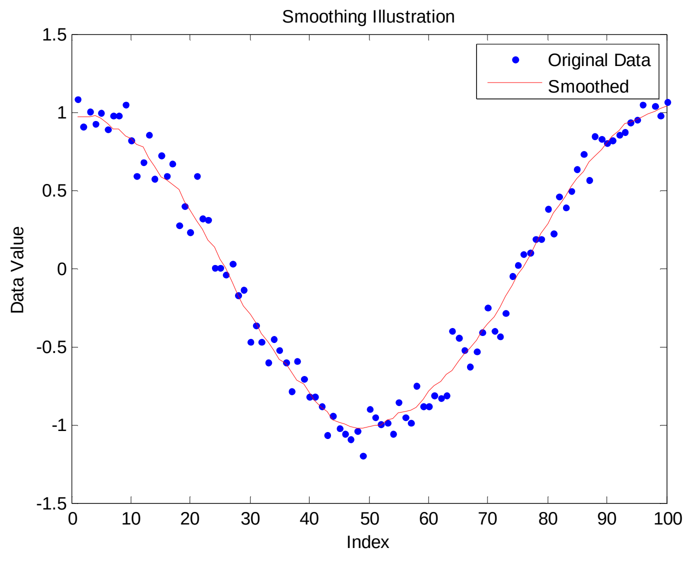
4. **Interpolation and surface plots**. Write a script called `randomSurface.m` to do the following
   a. To make a random surface, make Z0 a 5x5 matrix of random values on the range [0,1] (**rand**).
   b. Make an X0 and Y0 using **meshgrid** and the vector 1:5 (use the same vector for both inputs into meshgrid). Now, X0, Y0, and Z0 define 25 points on a surface.
   c. We are going to interpolate intermediate values to make the surface seem smooth. Make X1 and Y1 using **meshgrid** and the vector 1:.1:5 (again use the same vector for both inputs into meshgrid).
   d. Make Z1 by interpolating X0, Y0, and Z0 at the positions in X1 and Y1 using cubic interpolation (**interp2**, specify cubic as the interpolation method).
   e. Plot a surface plot of Z1. Set the colormap to hsv and the shading property to interp (**surf**, **colormap**, **shading**).
   f. Hold on to the axes and plot the 15-line contour on the same axes (**contour**).
   g. Add a colorbar (**colorbar**).
   h. Set the color axis to be from 0 to 1 (**caxis**). The final figure should look something like this (if the figure isn't copy/pasting into your document appropriately, try changing the figure copy options to use a bitmap):



5. **Fun with find**. Write a function to return the index of the value that is nearest to a desired value. The function declaration should be: `ind=findNearest(x, desiredVal)`. `x` is a vector or matrix of values, and `desiredVal` is the scalar value you want to find. Do not assume that `desiredVal` exists in `x`, rather find the value that is closest to `desiredVal`. If multiple values are the same distance from `desiredVal`, return all of their indices. Test your function to make sure it works on a few vectors and matrices. Useful functions are **abs**, **min**, and **find**. **Hint:** You may have some trouble using `min` when `x` is a matrix. To convert a matrix Q into a vector you can do something like `y=Q(:)`. Then, doing `m=min(y)` will give you the minimum value in Q. To find where this minimum occurs in Q, do `inds=find(Q==m);`.

6. **Loops and flow control**. Make function called `loopTest(N)` that loops through the values 1 through N and for each number n it should display 'n is divisible by 2', 'n is divisible by 3', 'n is divisible by 2 AND 3' or 'n is NOT divisible by 2 or 3'. Use a **for** loop, the function **mod** or **rem** to figure out if a number is divisible by 2 or 3, and **num2str** to convert each number to a string for displaying. You can use any combination of **if**, **else**, and **elseif**.

7. **Smoothing filter.** Although it is a really useful function, Matlab does not contain an easy to use smoothing filter. Write a function with the declaration: `smoothed=rectFilt(x,width).` The filter should take a vector of noisy data (`x`) and smooth it by doing a symmetric moving average with a window of the specified `width`. For example if `width=5`, then `smoothed(n)` should equal `mean(x(n-2:n+2))`. Note that you may have problems around the edges: when `n<3` and `n>length(x)-2`.
   a. The lengths of `x` and `smoothed` should be equal.
   b. For symmetry to work, make sure that `width` is odd. If it isn't, increase it by 1 to make it odd and display a warning, but still do the smoothing.
   c. Make sure you correct edge effects so that the smoothed function doesn't deviate from the original at the start or the end. Also make sure you don't have any horizontal offset between the smoothed function and the original (since we're using a symmetric moving average, the smoothed values should lie on top of the original data).
   d. You can do this using a loop and **mean** (which should be easy but may be slow), or more efficiently by using **conv** (if you are familiar with convolution).
   e. Load the mat file called noisyData.mat. It contains a variable `x` which is a noisy line. Plot the noisy data as well as your smoothed version, like below (a width of 11 was used):

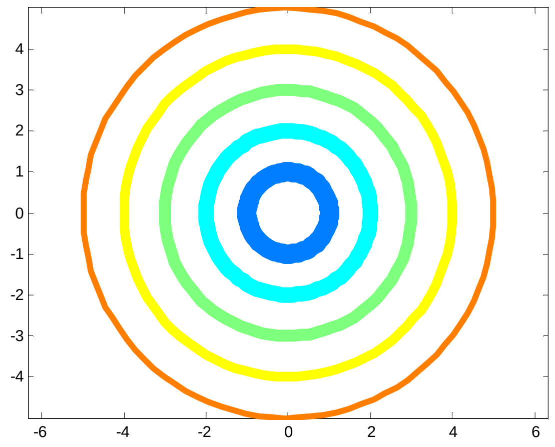


Optional Problems

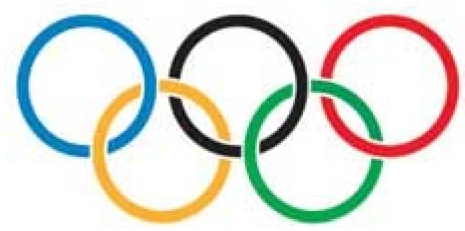
8. **Optional: Plot a circle.** It's not immediately obvious how to plot a circle in Matlab. Write the function `[x,y]=getCircle(center,r)` to get the x and y coordinates of a circle. The circle should be centered at `center` (2-element vector containing the x and y values of the center) and have radius `r`. Return `x` and `y` such that `plot(x,y)` will plot the circle.

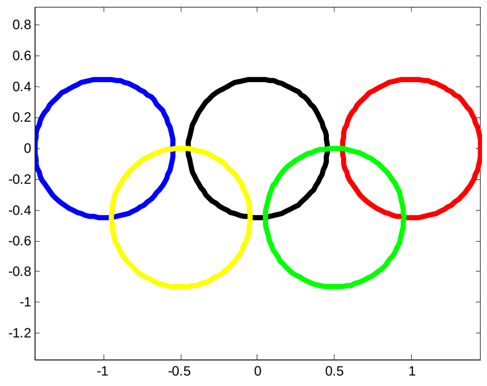
   a. Recall that for a circle at the origin (0,0), the following is true: $\begin{matrix} x(t)=\cos(t) \\ y(t)=\sin(t) \end{matrix}$, for t on the range $[0,2\pi]$. Now, you just have to figure out how to scale and translate it.

   b. Write a script called `concentric.m`. In this script, open a new figure and plot five circles, all centered at the origin and with increasing radii. Set the line width for each circle to something thick (at least 2 points), and use the colors from a 5-color `jet` colormap (**jet**). The property names for line width and color are `'LineWidth'` and `'Color'`, respectively. Other useful function calls are **hold on** and **axis equal**. It should look something like this

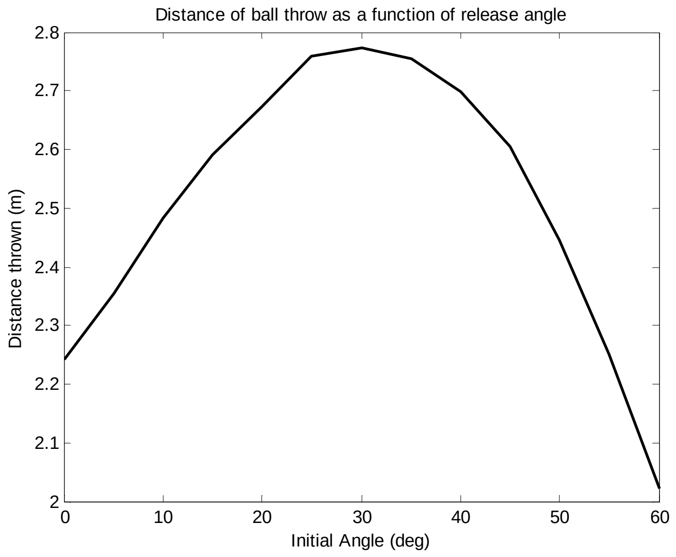


   c. Make a script called `olympic.m`. This script should use your `getCircle` function to draw the Olympic logo, as shown below. Don't worry about making the circles overlap in the same way as the official logo (it's possible but is too complicated for now). Also, when specifying colors, you can use the same color codes as when plotting lines ('b' for blue, ‘k’ for black, etc.) instead of providing an RGB vector.

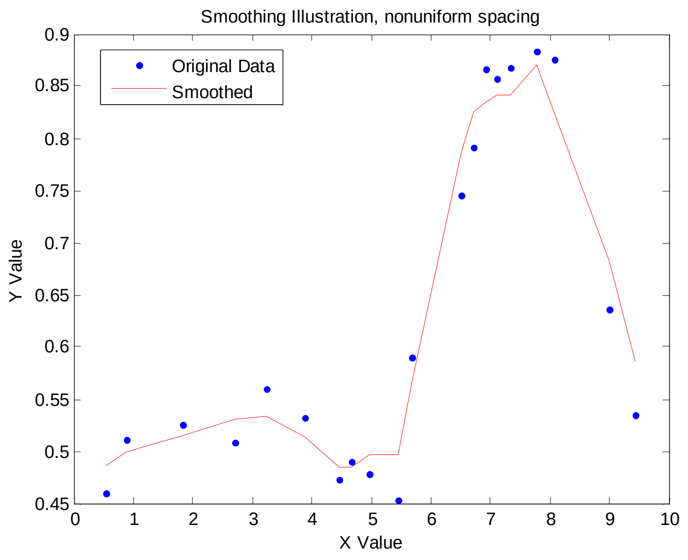




9. **Optional: Functions**. If you wrote the ball throwing script for Homework 1, turn it into a function. Add a function declaration that takes `v` and `theta` as inputs and returns the `distance` at which the ball hits the ground: `distance=throwBall(v,theta)`. We generally don't want functions to plot things every time they run, so remove the figure command and any plotting commands from the function. To be able to simulate a wide range of v and theta, make the time go until 10sec and add an `if` statement that will display the warning 'The ball does not hit the ground in 10 seconds' if that turns out to be the case. Also, if the ball doesn't hit the ground in 10 seconds, you should return `NaN` as the `distance`. To test your function, write a script `testBall.m` to throw the ball with the same velocity `v` but different angles and plot the distance as a function of angle `theta`. The plot should look something like the figure below. You will need to run `throwBall` within a loop in order to calculate the distances for various `thetas`.



10. **Optional: Smoothing nonuniformly sampled data.** Modify your smoothing filter above to also works on data that isn't sampled at a constant rate. Modify the program to also accept `x` if it is an `Nx2` matrix, where the first column is the x values and the second column is the y values of the data points (such that doing `plot(x(:,1),x(:,2),'.')` would plot the data). In this case, `width` should be on the same scale as the scale of the x values, so for example if the x(:,1) values are on the range [0,1], a `width` of 0.2 is acceptable. Also, in this case the `width` doesn't have to be odd as before. Assume that the x values are in increasing order. The output should also be an `Nx2` matrix, where the first column is the same as the first column in `x`, and the second column contains the smoothed values. The most efficient way to do this is unclear; you may find the interpolating function **interp1** helpful, but you can definitely do this without using it. The file optionalData.mat contains a matrix `x` that you can test your function on. When smoothed with a width of 2, you should get a result like:



11. **Optional: Buy and sell a stock**. Write the following function:
`endValue=tradeStock(initialInvestment, price, buy, sell)`
The weekly `price` of Google stock from 8/23/2004 until 1/11/2010 is saved in the file googlePrices.mat. This file also contains two other vectors: `peaks` and `lows`. The `peaks` vector contains indices into the `price` vector when the stock peaked, and the `lows` vector contains indices into the `price` vector when the stock hit a low. Run your program so that it buys stock when it's low, and sell it when it's high. Below is a list of specifications.
    a. The inputs are: `initialInvestment` – the amount of money you have to invest at the beginning, in dollars; `price` – a vector of stock prices over time; `buy` – a vector of times when to buy (should just be integers that index into the price vector); `sell` – a vector of times when to sell (similar to the buy vector). `endValue` is the end value of your investment. If all of your stock isn't sold at the end, use the last price of the stock to calculate how much it's worth and add it to your available cash. Make the function general so that it will work with any given `price`, `buy` and `sell` vectors.
    b. You can only buy integer numbers of shares of stock (you can't buy 3.23 shares). You also can't buy more stock than you can afford (if the current price is $100 and you have $599 in cash, you can only buy 5 shares). When deciding how much you can buy, factor in the transaction cost (see next section) so that you don't go negative. When buying, always buy as many shares as you can, and when selling, sell all your shares.
    c. Each buy or sell transaction costs you $12.95, make sure you include this in your program. If your initial investment is small, you may not be able to carry out all the buy and sell orders. It may also not make sense to sell at each specified time: for example if you bought 10 shares of stock and the price increases by $1, it doesn't make sense to sell it since the transaction cost would eat up all your profit and cost you $2.95 extra. However you can't make the decision of whether to buy or not since you don't know where the price is going to go (you can't look forward in time). You don't have to be super clever about deciding whether to sell or not, but you can if you're so inclined.
    d. After it's complete, run your program with various initial investments. Load googlePrices.mat into your workspace and then run the function using the `price`, `peaks`, and `lows` vectors. To check that you did it right, try an initial investment of $100. This should give you an end value of $100. Also try an initial investment of $100000, which should result in a total value of about $61,231,407 (with a $100,000 initial investment, it turns out that you don't even have to decide whether it's a good idea to sell because the transaction cost becomes negligible compared to the number of shares you have).

Homework 3

This homework is designed to give you practice writing functions to solve problems. The problems in this homework are very common and you will surely encounter similar ones in your research or future classes. 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.

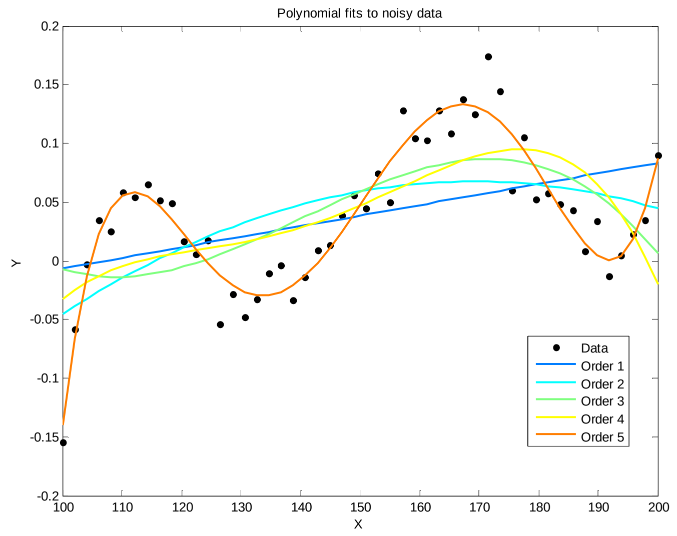
1. **Linear system of equations.** Solve the following system of equations using \. Compute and display the error vector
$$\begin{aligned} 3a+6b+4c&=1\\ a+5b&=2\\ 7b+7c&=3 \end{aligned}$$

2. **Numerical integration.** What is the value of: $\int_0^5 xe^{-x/3}dx$? Use **trapz** or **quad**. Compute and display the difference between your numerical answer and the analytical answer: $-24e^{-5/3}+9$.

3. **Computing the inverse.** Calculate the inverse of $\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$ and verify that when you multiply the original matrix by the inverse, you get the identity matrix (**inv**). Display the inverse matrix as well as the result of the multiplication of the original matrix by its inverse.

4. **Fitting polynomials.** Write a script to load the data file randomData.mat (which contains variables x and y) and fit first, second, third, fourth, and fifth degree polynomials to it. Plot the data as blue dots on a figure, and plot all five polynomial fits using lines of different colors on the same axes. Label the figure appropriately. To get good fits, you'll have to use the centering and scaling version of **polyfit** (the one that returns three arguments, see **help**) and its counterpart in **polyval** (the one that accepts the centering and scaling parameters). It should

look like this:



5. **Hodgkin-Huxley model of the neuron.** You will write an ODE file to describe the spiking of a neuron, based on the equations developed by Hodgkin and Huxley in 1952 (they received a Nobel Prize for this work). The main idea behind the model is that ion channels in the neuron's membrane have voltage-sensitive gates that open or close as the transmembrane voltage changes. Once the gates are open, charged ions can flow through them, affecting the transmembrane voltage. The equations are nonlinear and coupled, so they must be solved numerically.
    a. Download the HH.zip file from the class website and unzip its contents into your homework folder. This zip folder contains 6 m-files: `alphah.m`, `alpham.m`, `alphan.m`, `betah.m`, `betam.m`, `betan.m`. These functions return the voltage-dependent opening $\alpha(V)$) and closing ($\beta(V)$) rate constants for the h, m, and n gates: $C \underset{\beta(V)}{\overset{\alpha(V)}{\rightleftarrows}} O$
    b. Write an ODE file to return the following derivatives

$$\frac{dn}{dt} = (1-n)\alpha_n(V) - n\beta_n(V)$$
$$\frac{dm}{dt} = (1-m)\alpha_m(V) - m\beta_m(V)$$
$$\frac{dh}{dt} = (1-h)\alpha_h(V) - h\beta_h(V)$$
$$\frac{dV}{dt} = -\frac{1}{C}\left(G_K n^4 (V-E_K) + G_{Na} m^3 h (V-E_{Na}) + G_L (V-E_L)\right)$$

    and the following constants (C is membrane capacitance, G are the conductances and E are the reversal potentials of the potassium (K), sodium (Na), and leak (L) channels):

$C = 1$

$G_K = 36$

$G_{Na} = 120$

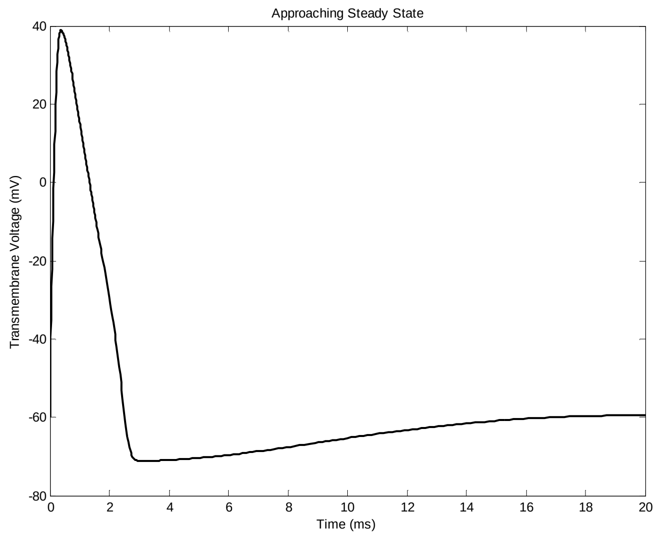
$G_L = 0.3$

$E_K = -72$

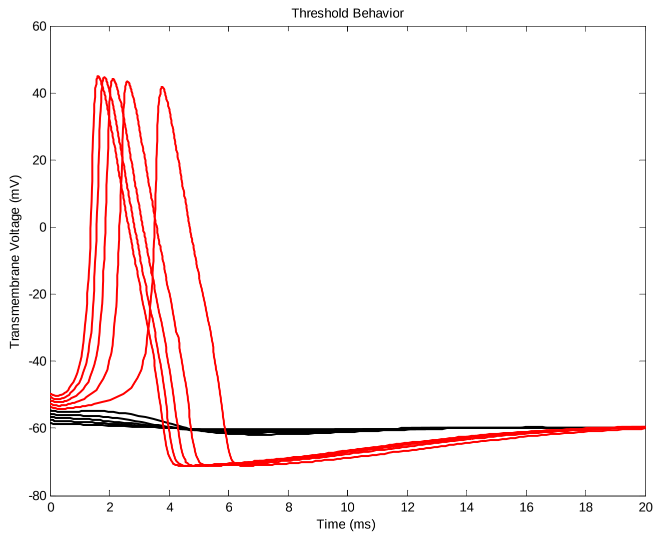
$E_{Na} = 55$

$E_L = -49.4$

    c. Write a script called `HH.m` which will solve this system of ODEs and plot the transmembrane voltage. First, we'll run the system to steady state. Run the simulation for 20ms (the timescale of the equations is ms) with initial values: $n = 0.5; m = 0.5; h = 0.5; V = -60$ (**ode45**). Store the steady-state value of all 4 parameters in a vector called ySS. Make a new figure and plot the timecourse of $V(t)$ to verify that it reaches steady state by the end of the simulation. It should look something like this:



d. Next, we'll explore the trademark feature of the system: the all-or-none action potential. Neurons are known to 'fire' only when their membrane surpasses a certain voltage threshold. To find the threshold of the system, solve the system 10 times, each time using ySS as the initial condition but increasing the initial value of V by 1, 2, … 10 mV from its steady state value. After each simulation, check whether the peak voltage surpassed 0mV, and if it did, plot the voltage with a red line, but if it didn't, plot it with a black line. Plot all the membrane voltage trajectories on the same figure, like below. We see that if the voltage threshold is surpassed, then the neuron 'fires' an action potential; otherwise it just returns to the steady state value. You can zoom in on your figure to see the threshold value (the voltage that separates the red lines from the black lines).



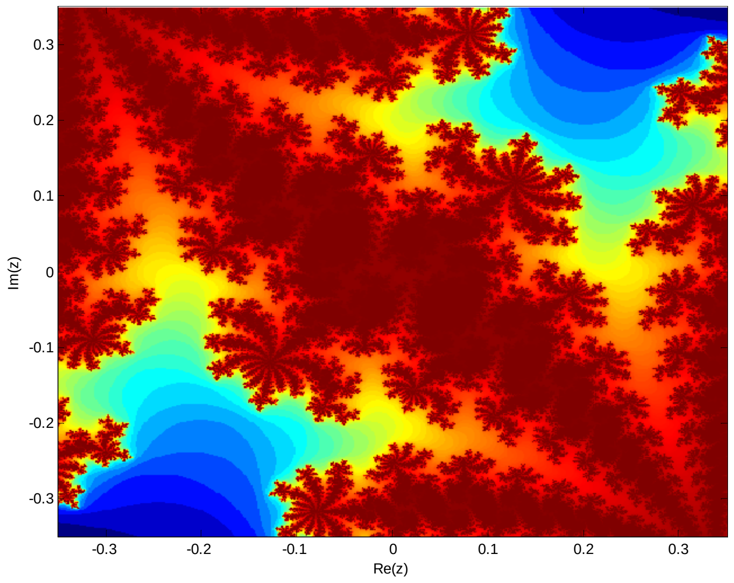
Optional Problem

6. **Optional, but highly recommended: Julia Sets.** In this problem you will generate quadratic Julia Sets. The following description is adapted from Wikipedia at http://en.wikipedia.org/wiki/Julia. For more information about Julia Sets please read the entire article there.

Given two complex numbers, c and z_0, we define the following recursion:

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

This is a dynamical system known as a quadratic map. Given a specific choice of c and z_0, the above recursion leads to a sequence of complex numbers $z_1, z_2, z_3 \ldots$ called the orbit of z_0. Depending on the exact choice of c and z_0, a large range of orbit patterns are possible. For a given fixed c, most choices of z_0 yield orbits that tend towards infinity. (That is, the modulus $|z_n|$ grows without limit as n increases.) For some values of c certain choices of z_0 yield orbits that eventually go into a periodic loop. Finally, some starting values yield orbits that appear to dance around the complex plane, apparently at random. (This is an example of chaos.) These starting values, z_0, make up the Julia set of the map, denoted J_c. In this problem, you will write a MATLAB script that visualizes a slightly different set, called the filled-in Julia set (or Prisoner Set), denoted K_c, which is the set of all z_0 with orbits which do not tend towards infinity. The "normal" Julia set J_c is the edge of the filled-in Julia set. The figure below illustrates a Julia Set for one particular value of c. You will write MATLAB code that can generate such fractals in this problem.



a. It has been shown that if the modulus of z_n becomes larger than 2 for some n then it is guaranteed that the orbit will tend to infinity. The value of n for which this becomes true is called the 'escape velocity' of a particular z_0. Write a function that returns the escape velocity of a given z_0 and c. The function declaration should be: `n=escapeVelocity(z0,c,N)` where `N` is the maximum allowed escape velocity (basically, if the modulus of z_n does not exceed 2 for `n`<`N`, return `N` as the escape velocity. This will prevent infinite loops). Use **abs** to calculate the modulus of a complex number

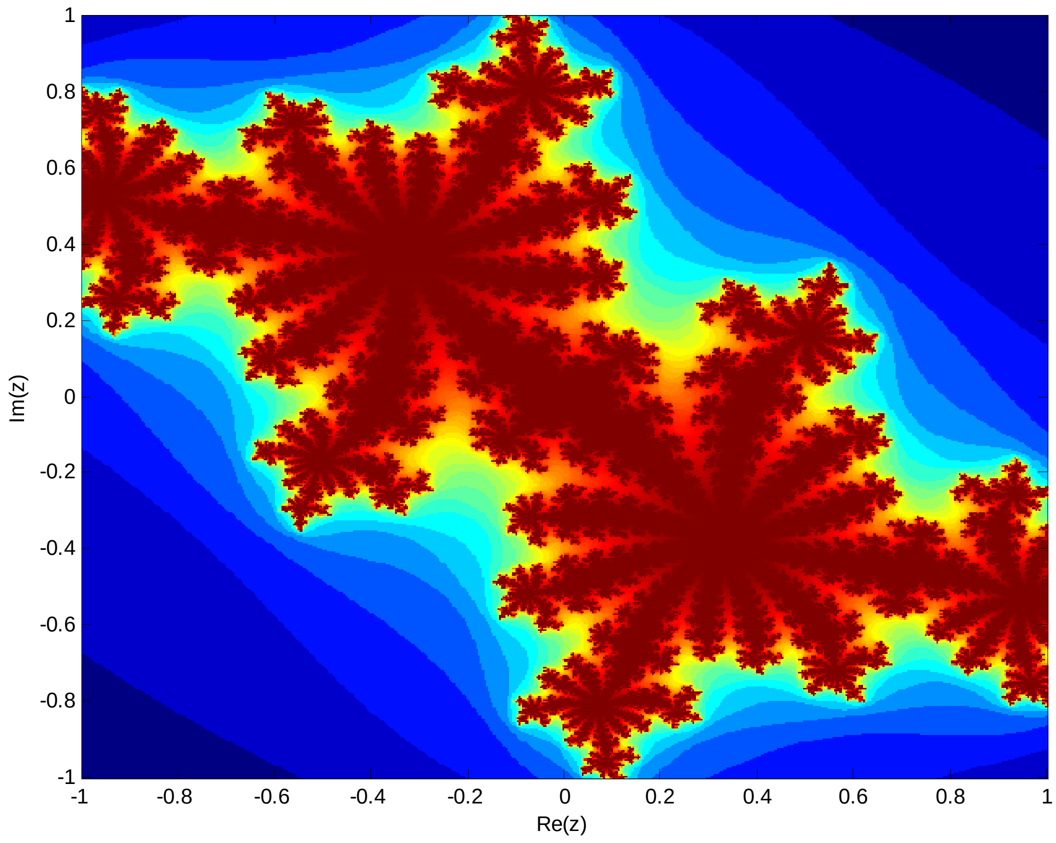
b. To generate the filled in Julia Set, write the following function `M=julia(zMax,c,N)`. `zMax` will be the maximum of the imaginary and complex parts of the various z_0's for which we will compute escape velocities. `c` and `N` are the same as defined above, and `M` is the matrix that contains the escape velocity of various z_0 's.

   i. In this function, you first want to make a 500x500 matrix that contains complex numbers with real part between `–zMax` and `zMax`, and imaginary part between `–zMax` and `zMax`. Call this matrix `Z`. Make the imaginary part vary along the y axis of this matrix. You can most easily do this by using **linspace** and **meshgrid**, but you can also do it with a loop.

   ii. For each element of `Z`, compute the escape velocity (by calling your `escapeVelocity`) and store it in the same location in a matrix `M`. When done, the matrix `M` should be the same size as `Z` and contain escape velocities with values between 1 and `N`.

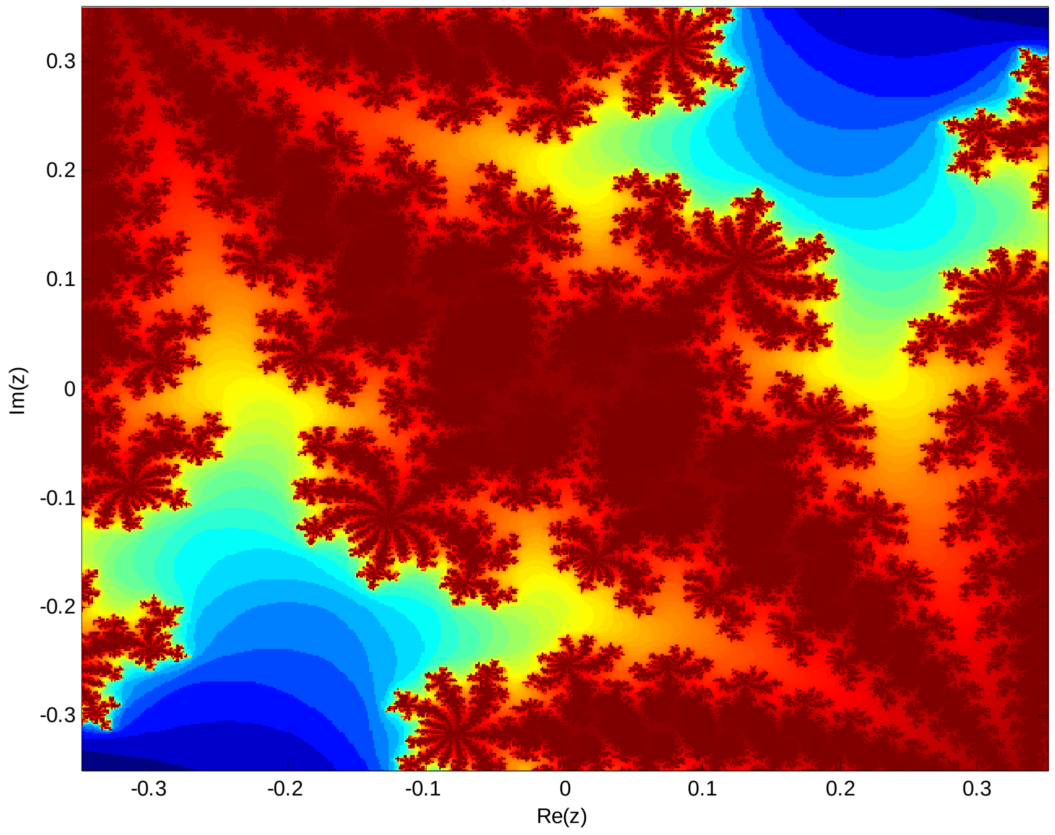
   iii. Run your `julia` function with various `zMax`, `c`, and `N` values to generate various fractals. To display the fractal nicely, use **imagesc** to visualize `atan(0.1*M)`, (taking the arctangent of `M` makes the image look nicer; you may also want to use **axis xy** so the y values aren't flipped). WARNING: this function may take a while to run.

The figure below was created by running:
`M=julia(1,-.297491+i*0.641051,100);` and visualizing it as described above.



The figure below was generated by running the same `c` parameter as above, but on a smaller range of `z` values and with a larger `N`:

```
M=julia(.35,-.297491+i*0.641051,250);
```



Homework 4

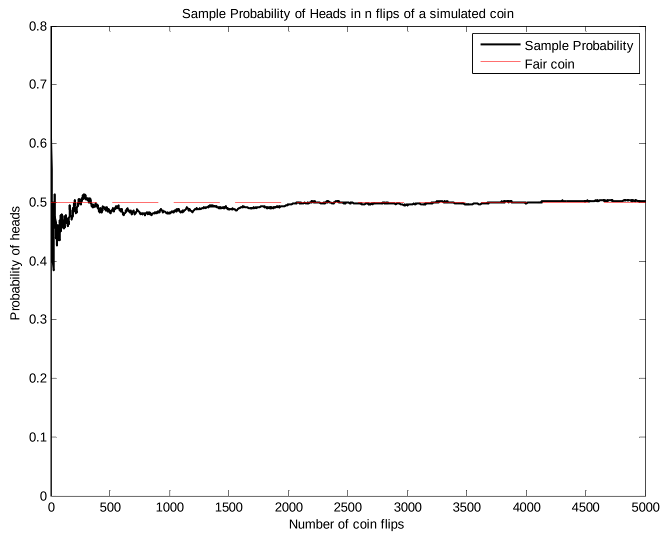
This homework is designed to give you practice with more advanced and specific Matlab functionality, like advanced data structures, images, and animation. As before, the names of helpful functions are provided in **bold** where needed. **Homework must be submitted before the start of the next class.**

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.

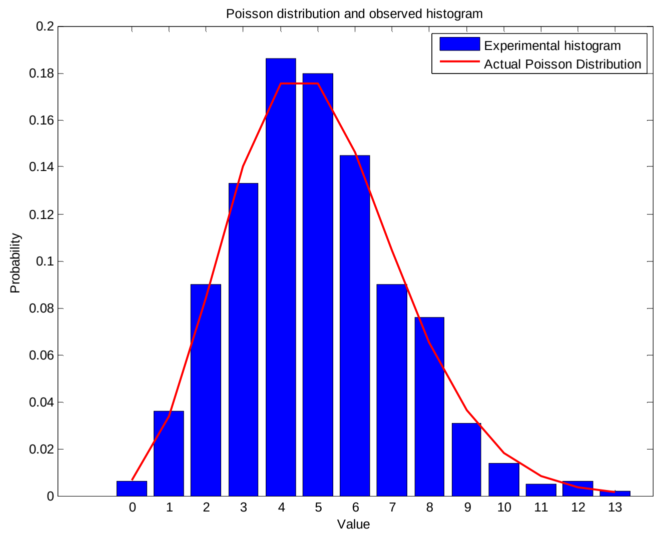
Keep all your code in scripts/functions. If a specific name is not mentioned in the problem statement, you can choose your own script names.

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.



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**.

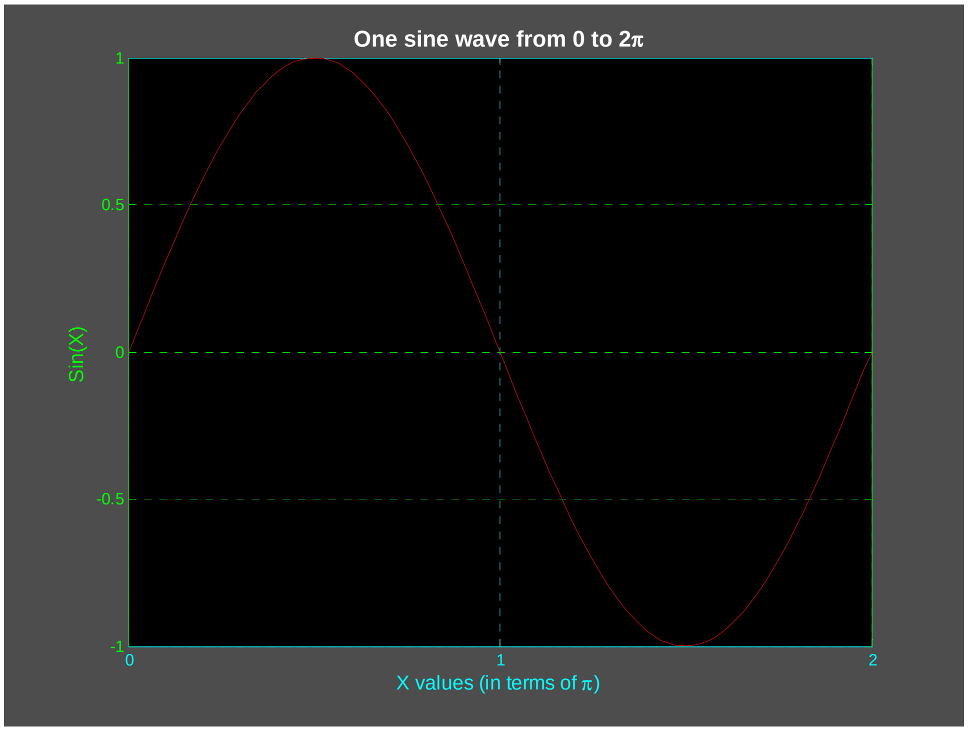
   The output to parts a-c should look like this:

```
>> 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:

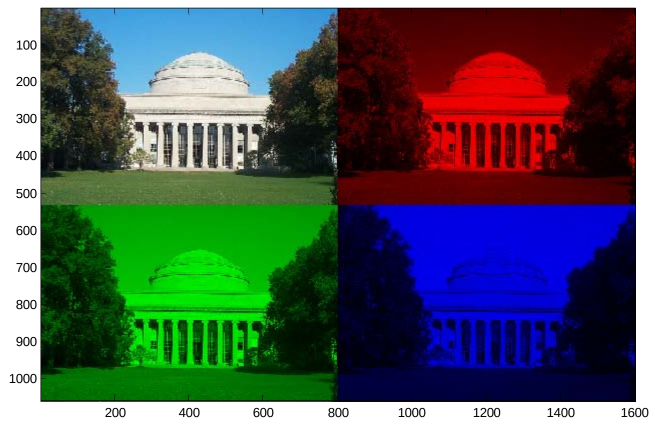
```
>> displayDir
File brown2D.m contains 417 bytes.
File coinTest.m contains 524 bytes.
File displayDir.m contains 304 bytes.
File plotPoisson.m contains 543 bytes.
File someData.txt contains 66 bytes.
```

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



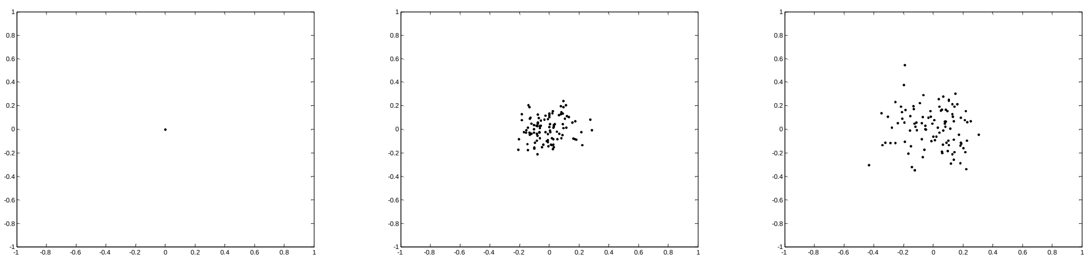
a. Do all the following in a script named `handlesPractice.m`
b. First, make a variable `x` that goes from 0 to 2π, and then make `y=sin(x)`.
c. Make a new `figure` and do `plot(x,y,'r')`
d. Set the x limit to go from 0 to 2π (**xlim**)
e. Set the `xtick` property of the axis to be just the values `[0 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.
   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

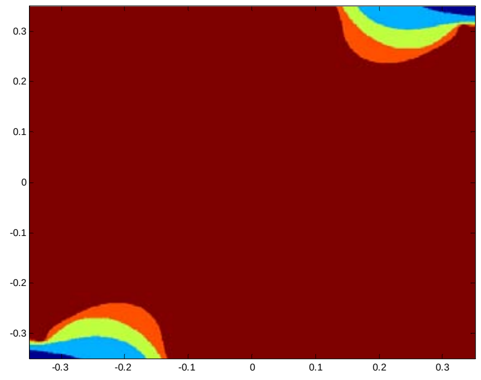
a. Write the function `juliaAnimation(zMax,c,N)`, which takes three inputs: `zMax` is a real number that describes the section of the complex plane that we'll be looking at. We're going to compute escape velocities for numbers with real part between `–zMax` and `zMax`, and imaginary part between `–zMax` 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).

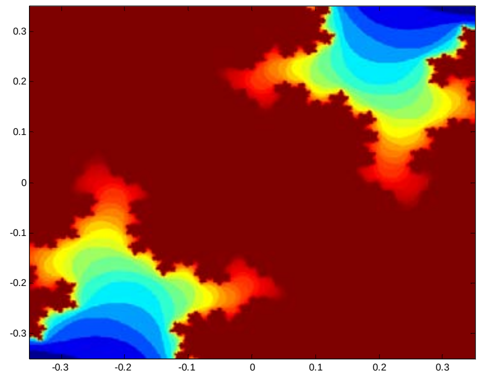
b. In this function, first make a vector `temp`, which has 500 values between `–zMax` and `zMax`. Then, use `temp` within **meshgrid** to make `R` and `I`, which are two real matrices containing the real and imaginary parts of a complex matrix Z. The convention in the complex plane is to have the real 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.

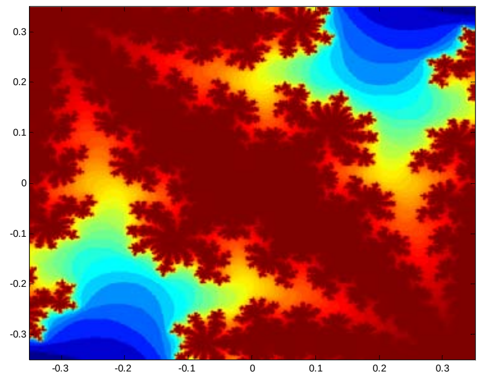
c. Initialize a matrix `M` to be the same size as Z, and whose elements all have the value `N`.

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.

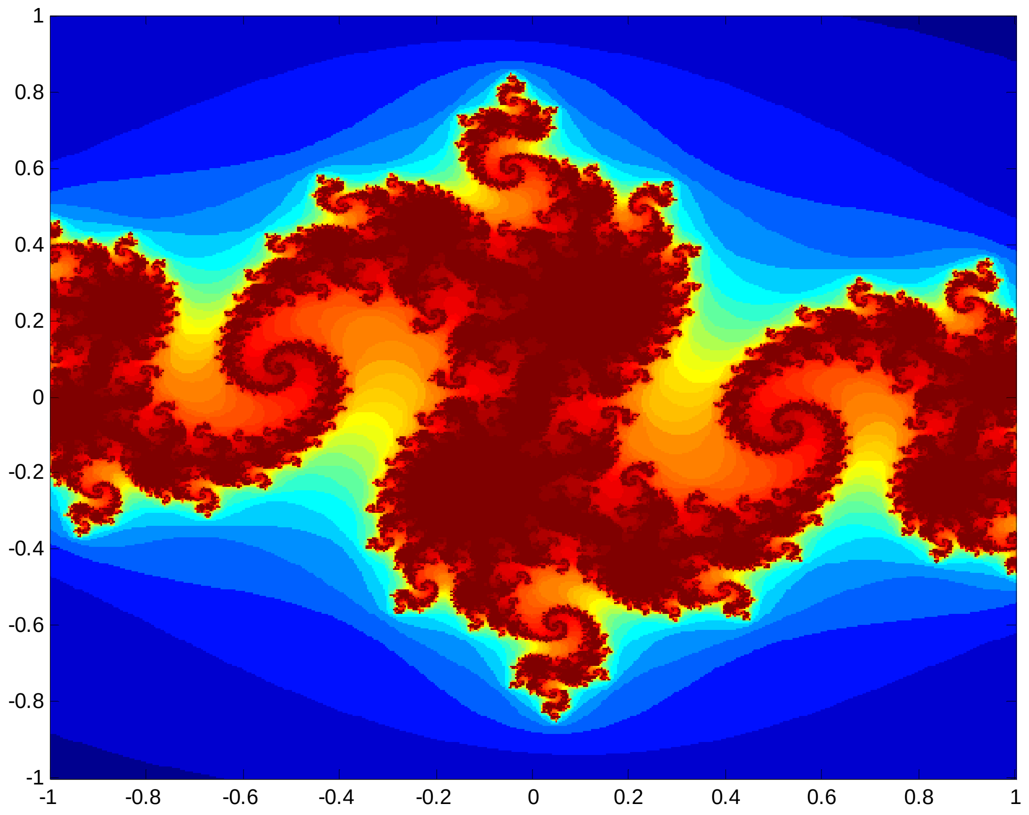
e. When you run the function with something like : `juliaAnimation(.35,-.297491+i*.641051,100)`, you will see the Julia Set come to life. The figures below are from the beginning, middle, and end of the animation:







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 on the next page.


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1