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.**

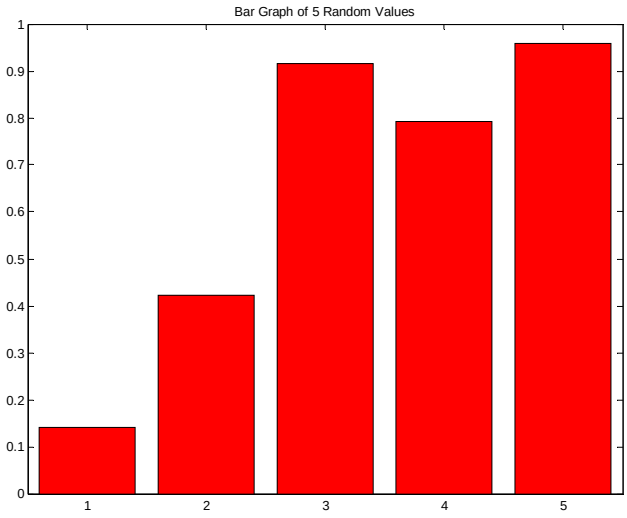
What to turn in: Copy the text from your scripts and paste it into a document. If a question asks you to plot or display something to the screen, also include the plot and screen output your code generates. Submit either a *.doc or *.pdf file.

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

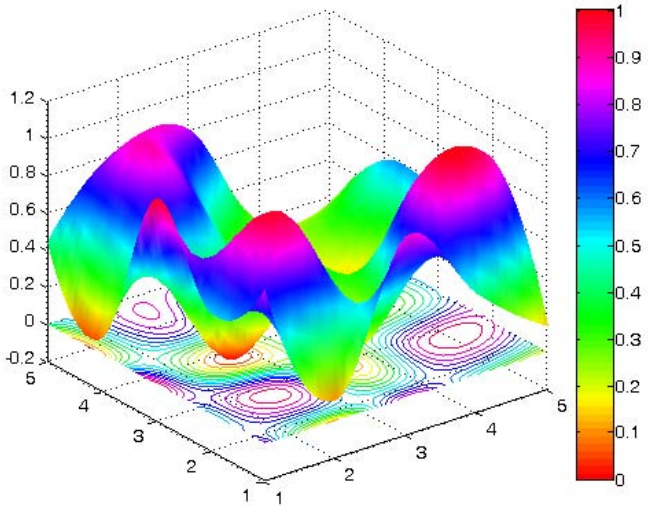
1. **Semilog plot**. Over the past 5 years, the number of students in 6.094 has been 15, 25, 55, 115, 144. Class size seems like it's growing exponentially. To verify this, plot these values on a plot with a log y scale and label it (**semilogy**, **xlabel**, **ylabel**, **title**). Use magenta square 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.



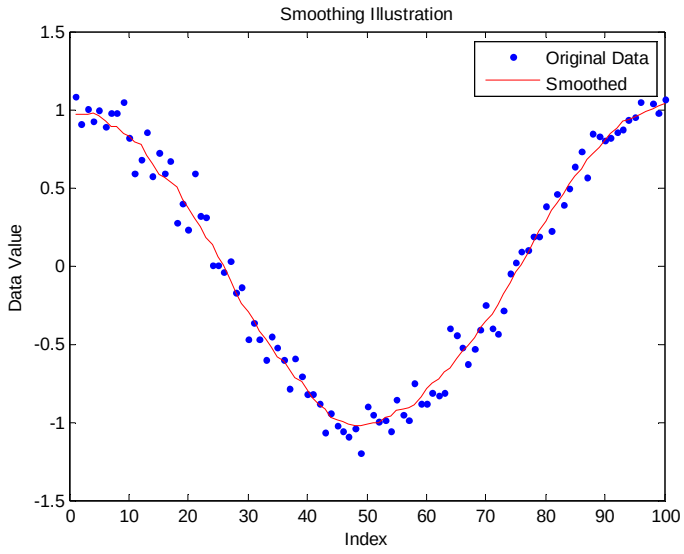
4. **Interpolation and surface plots**. Write a script called `randomSurface.m` to do the following
   a. To make a random surface, make Z0 a 5x5 matrix of random values on the range [0,1] (**rand**).
   b. Make an X0 and Y0 using **meshgrid** and the vector 1:5 (use the same vector for both inputs into 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):

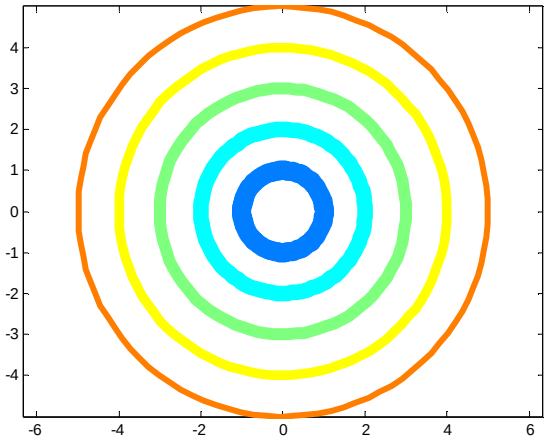


Optional Problems

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.

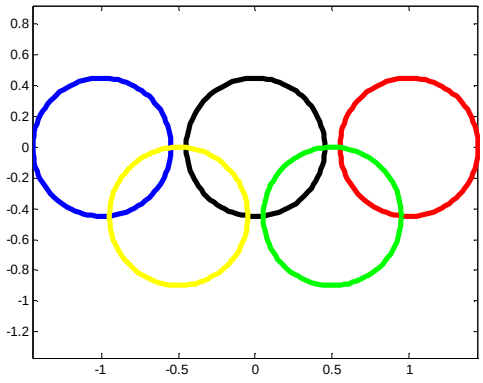
   a. Recall that for a circle at the origin (0,0), the following is true: $\begin{aligned} x(t) &= \cos(t) \\ y(t) &= \sin(t) \end{aligned}$, for t on the range $[0, 2\pi]$. Now, you just have to figure out how to scale and translate it.

   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

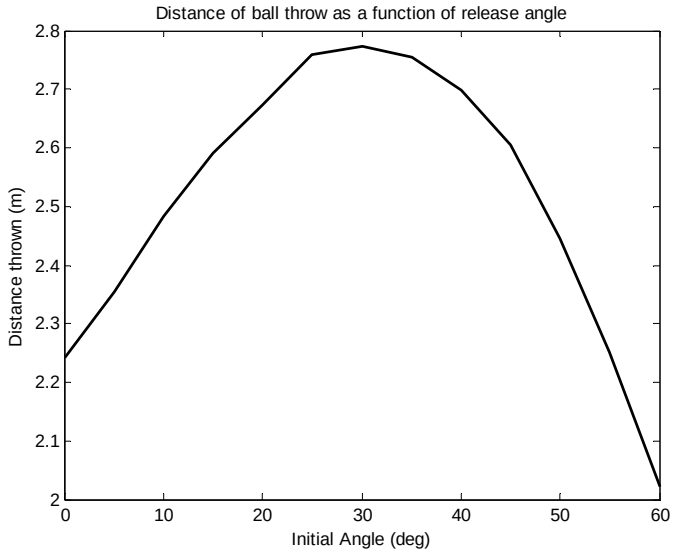


   c. Make a script called `olympic.m`. This script should use your `getCircle` function to draw the Olympic logo, as shown below. Don't worry about making the circles overlap in the same way as the official logo (it's possible but is too complicated for now). Also, when specifying colors, you can use the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

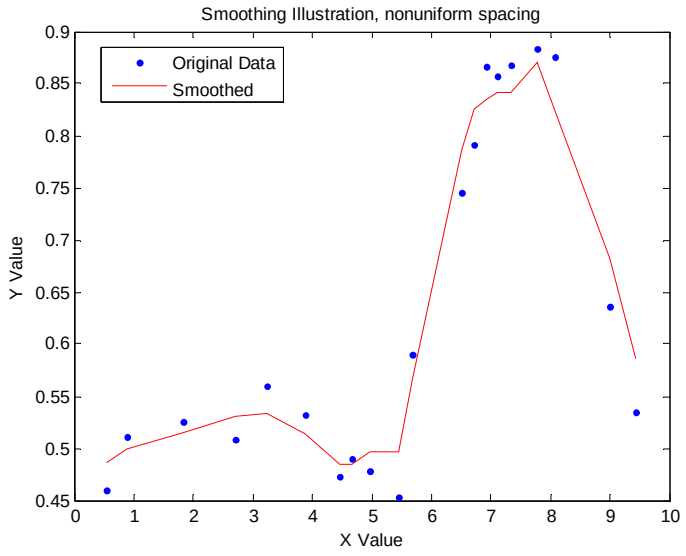
blue, ‘k’ for black, etc.) instead of providing an RGB vector.



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).

6.094 Introduction to MATLAB®

January (IAP) 2010