Notice - Testing plot outputs:

For this homework, some of your outputs will be plots. We have provided you with the function `checkPlots()` to help you test the plot outputs of your functions. Use

```
help checkPlots
```

for a full description of how the function works, including what inputs to call with it. Below is the example from the documentation explaining how to run the function.

If you have a function called `testFunc` and the following test case:

```
testFunc(30, true, {'cats', 'dogs'})
```

To check the plot produced by `testFunc` against the solution function `testFunc_soln`, for this test case you would run:

```
checkPlots('testFunc', 30, true, {'cats', 'dogs'})
```

Happy Coding!
~Homework Team

Function Name: `physics`

Inputs:

1. (*double*) 1xM vector of time values
2. (*double*) 1xM vector of velocity values
3. (*double*) Degree that best fits velocity vs time data

Outputs:

None

Plot Outputs:

1. A plot with subplots of Velocity vs. Time, Numerical Derivative vs. Time, and Analytical Derivative vs. Time

Background:

You are in PHYS 2211, and you are tired of plotting velocity and acceleration graphs by hand. Luckily, you just learned how to plot and to find derivatives numerically and analytically in MATLAB.

Function Description:

Write a function that first plots the velocity values vs. time and then estimates and plots the acceleration in two different ways. Your function should do the following:

1. Create a 2x2 subplot.
2. Plot the velocity vs time graph in black in the top left position.
3. Approximate the acceleration data by finding the numerical derivative of the velocity. Plot the numerical acceleration vs time graph in red in the top right position.
4. Now find the best-fit polynomial of the velocity data using the order given by the third input. Then, find a second approximation for the acceleration data by taking the analytical derivative of this polynomial and evaluating the derivative at the original time values. Plot your derivative in green in the bottom right position.

Notes:

- The subplot in the bottom left position should not have a plot.
- Make the axes for all the subplots square.
- When plotting the numerical derivative, leave off the last time value, so the vector lengths are the same.

Function Name: `stockExchange`

Inputs:

1. *(double)* A 2xN array of data from the New York Stock Exchange

Outputs:

1. *(double)* A 2xN array of updated data

Background:

Due to years of poor record keeping, the New York Stock Exchange is missing valuable historical information about the prices of stocks and needs your help! The previous record keeper missed work often, and for each day of absence a stock price is missing in the NYSE's data. Impressed with your MATLAB knowledge, the Head of Data Management and Analytics hires you to avert the impending data crisis.

Function Description:

You are given an array containing two rows, the first corresponding to a day of the month and the second corresponding to the price of a particular stock on that day. On a day when the record keeper was absent, there will be an NaN in the index of the stock price for that day. Your function should replace each NaN with the correct stock price, which you will determine by spline interpolation.

Example:

```
>> arr = [1       2       3     4       5        6       7      8
         200.10   203.20  NaN  197.41   201.30   200.00  NaN   206.4]
>> out = stockExchange(arr)
out = [1       2       3       4       5       6       7       8
       200.10  203.20  199.34  197.41  201.30  200.00  198.35  206.4]
```

Notes:

- Do **not** use iteration to solve this problem.
- If an `NaN` occurs in the first or last index, you should extrapolate the data to find the correct stock price.
- All computed stock prices should be rounded to the nearest penny.
- Use of the `spline` function with inputs that contain `NaN` is not required to solve this problem. If you choose to solve it this way you can ignore the warning produced.

Hints:

- What happens when you call `isnan()` with a vector input?

Function Name: `crimeStats`

Inputs:

1. (*double)* 1xN vector of years to consider
2. (*char*) Category from which to plot the data
3. (*char*) Style description

Outputs:

None

Plot Outputs:

1. Plot of data from the chosen category vs years

Background:

You have just landed your dream job as a crime analyst at The Federal Bureau of Investigation. Your team has to present crime stats to the Director next week. To make your presentation super snazzy, you decide to plot the some of your data in MATLAB.

Function Description:

Write a function that takes in the years and the category you have considered to plot. You will need to read the file '`FBIDataSet.xlsx`' to get the data to plot. The first column of the spreadsheet contains the years, and the first row contains the headers. Find the column that contains the given category (second input), and plot each data point from this column for each corresponding year (first input).

The plot should be labelled on both axes. The x-axis will always have the label '`Years`'. The y-axis should be labelled with the category name. The title of the graph have a title in the following format:

'`Plot of <Category> vs Years`'

The third input tells you the line type, plot symbol and color for the graph. If style is '`Type 1`', the graph should have stars connected by a blue dotted line. If style is '`Type 2`', the graph should have circles connected with a black solid line. If style if '`Type 3`', the graph should have diamonds connected by a red dashed line.

Notes:

- The years in the input could be of any order. You will need to sort this before you get the data in the category.
- Your function should work for any number of years.
- The unedited data set of the file can be found here.
- Except for the header row, the `'FBIDataSet.xlsx'` sheet is guaranteed to be all numbers.

Function Name: `roots`

Inputs:

1. (*double*) A vector of x values
2. (*double*) A vector of y values
3. (*double*) A lower bound for the initial interval
4. (*double*) An upper bound for the initial interval

Outputs:

1. *(double*) The approximate root of the function

Background:

To find the root of a function means to find the value x of a function *f* where *f(x)* = 0. Root finding algorithms provide a method of approximating the root of a function where there is no analytical or closed form expression that can be used to determine the exact solution. Root finding algorithms have a variety of practical applications, especially in engineering.

Function Description:

The bisection method is a root finding algorithm that, given an interval [a,b] that contains only one root, iteratively cuts that interval in half, keeping the half that contains the root.

1. Given a set of x and y values, determine the coefficients of the highest order unique polynomial that passes through all the points.
2. Calculate the midpoint of the interval using the following formula:
$$x_{midpoint} = \frac{lowerBound + upperBound}{2}$$
3. Calculate the function values at the lower bound, the upper bound, and the midpoint using the coefficients determined in step 1.
4. If the sign of the function value at the midpoint is opposite of the sign of the function value at the lower bound, replace the upper bound with the midpoint.
5. Otherwise, if the sign of the function value at the midpoint is opposite of the sign of the function value at the upper bound, replace the lower bound with the midpoint.
6. Repeat steps 2 through 5 until the absolute value of $\frac{upperBound - lowerBound}{upperBound + lowerBound}$ (the approximate error) is less than 0.0001. Your final guess for the root will be the midpoint of the final interval.

Notes:

- The initial interval is guaranteed to contain exactly one root.
- Round your **final** answer to two decimal places.

Hints:

- If the product of two numbers is negative, then they have the opposite sign.

Function Name: `integrals`

Inputs:

1. (*double*) A 2xN double array representing (x,y) coordinates, one point per column
2. (*double*) A 2xN double array representing (x,y) coordinates, one point per column

Outputs:

1. *(double*) The area between the input functions

Plot Outputs:

1. A plot of the two functions with filled in area

Background:

You have been collecting data on two runners, keeping track of their speed by taking measurements at discrete points in time. In order to confirm the speed readings, you need to integrate and compare the final difference between the two to the distance between them when they stopped.

Function Description:

Given two sets of (x,y) values ordered by their x values, compute the positive area between the two curves. One set of points is guaranteed to lie entirely above the other, but it may be either the first or second input. In other words, the y-values of one input will be greater than the corresponding y-values in the other input. Use the trapezoidal approximation of the integral to find the area under each, and take the positive difference.

We need a visualization for the math we just did! First, plot the greater of the two inputs with a green line, and plot the lesser of the two with a red line. Next, 'fill in' the area between the two curves you just plotted with a vertical line of magenta stars at every integer along the x-axis (`'m*'`) using the following steps:

1. For every integer in between the minimum x value to the maximum x value, linearly interpolate the corresponding y values on the lower and upper curves.
2. At each x value, plot the stars from the interpolated y value on the lower line to the interpolated y value on the upper line so that each star will lie at an integer point.

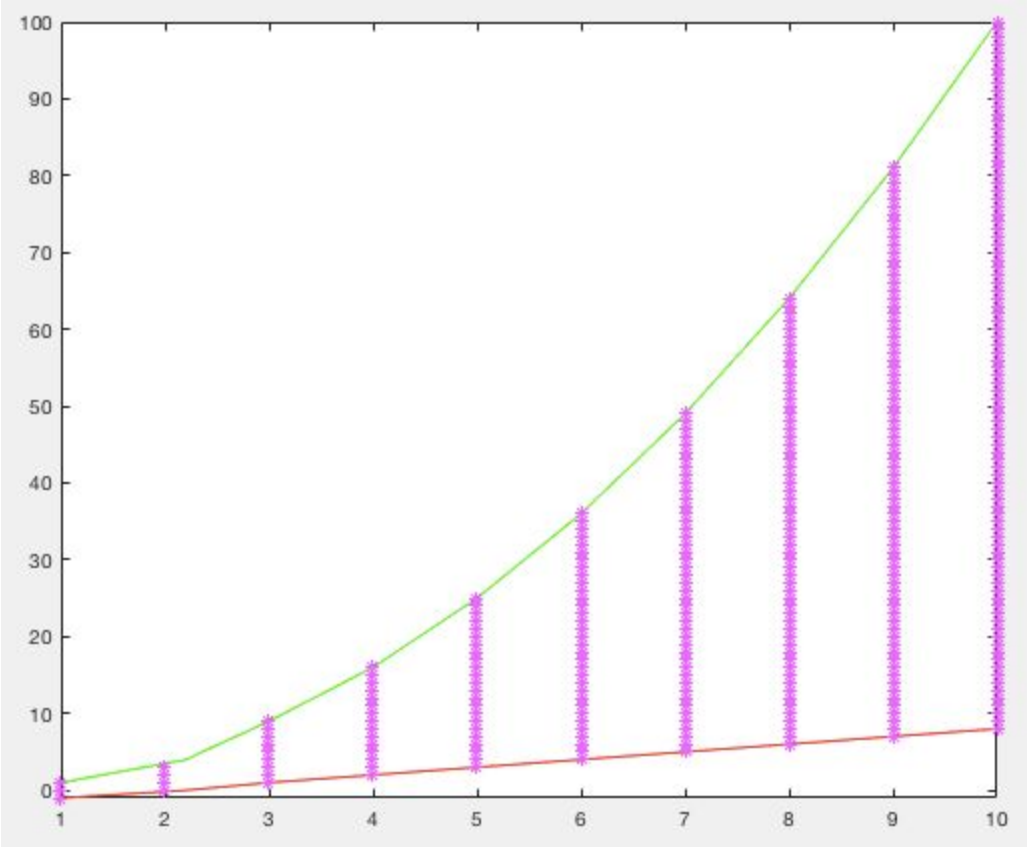
Example:

```
>> curve1 = [1   2.2   3   4   5   6   7   8   9   10;
             1    4    9   16  25  36  49  64  81  100]

>> curve2 = [1  2.2  3  4  5  6  7  8  9  10;
            -1   0   1  2  3  4  5  6  7   8]

>> area = integrals(curve1, curve2)
area = 302.4
```

Continued...



Notes:

- *After* you finish plotting, adjust the axis limits to be as small as possible to contain all data points.
- Round the **final** area output to two decimal places.
- Use `ceil()` and `floor()` when determining where to plot the lowest and highest stars in each vertical line, respectively.
- You are guaranteed that the two inputs will have the same domain; one set of points will not extend past the other.
- The smallest and largest x-values will be integers, so you do not need to worry about rounding these values when plotting the magenta stars.

Hints:

- When finding which y values to plot at each x value, think about how can you use the colon operator?