Notice #1 - Testing variable outputs:

Previously, the variable outputs for the test cases in the `hw##.m` cases have been displayed underneath the test case. From now on, variable outputs will not be displayed here. In order to see what the expected output is, please run the solution function with the same inputs and compare the outputs of the solution function with the outputs of your function using `isequal()`.

Notice #2 - 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'})
```

Then 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: `illuminati`

Inputs:

1. (*double*) a positive value representing the length of the hypotenuse
2. (*double*) a positive value representing the angle between the hypotenuse and the x-axis

Outputs:

none

Plot Outputs:

1. Plotted triangle

Background:

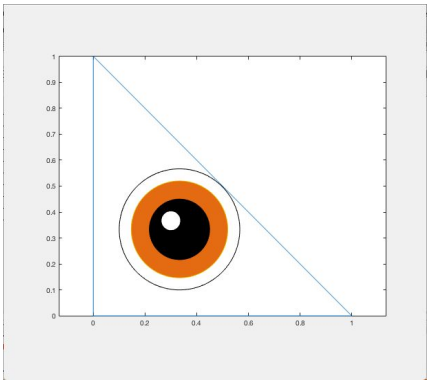
All your life you've heard about the hidden Illuminati, the world's most powerful, secret organization. You never believed in the conspiracy theory but to your surprise, one day in the mail you get an invitation to join. The only requirement is knowing how to create different sized illuminati right triangles!

Function Description:

Given an angle value and the length of the hypotenuse, plot a blue right triangle. The angle given will always be measured from the x-axis. After plotting the right triangle, use the helper function given, called `plotEye()`, to plot the illuminati eye in the middle of the triangle.

Example:

```
>> illuminati(sqrt(2), 45)
```



Notes:

- The magnitude of the angle will always be between 0 and 90.
- `plotEye()` has the same inputs as `illuminati`.
- Set the axes equal.

Hints:

- Remember to `hold on`

Function Name: `flatEarth`

Inputs:

1. *(double)* 1xN vector representing velocity data
2. *(double)* 1xN vector representing time data

Outputs:

1. *(char*) Sentence describing the true shape of the Earth

Background:

All your life, you have been told to accept that the Earth is spherical, without any solid evidence to establish that the Earth's surface is curved. And given NASA's success in faking the moon landings, how can we know that the images we see of the Earth from space are not altered? Also, your recent digging into the issue in the deep web has yielded this image. Being the creature of science that you are, you set out on your own to prove whether the Earth is flat or not. Using MATLAB, of course.

Function Description:

Using the data you have collected for the velocity of the Earth at various times, determine the highest order unique polynomial that passes through all the points, v(t). Take the integral of this function to obtain the position function with respect to time, or x(t). You know from your EAS course at Georgia Tech that the sum of the coefficients of the position function for a spherical planet is always greater than or equal to zero, while a flat planet would have a negative sum of the coefficients of the position function.

Using this information, if the Earth is flat you should output the following:

```
'The Earth is flat! I was right all along!'
```

If the Earth is round, you should output the following:

```
'Hmm... It seems the Earth is actually round.'
```

Notes:

- The `polyfit()` function will be useful.
- The highest order unique polynomial that passes through n points has an order of (n-1).
- When you take the analytical integral of the coefficients for the velocity with respect to time to find the coefficients of the position with respect to time, assume an integration constant of zero.

Function Name: `timeTravel`

Inputs:

1. (*char*) A word
2. (*double*) A 2xN array of data pertaining to popularity of a word throughout time.
3. *(double*) A year to which you plan on travelling

Outputs:

1. *(char*) A sentence describing how successful the use of the word will be

Background:

Imagine you are a time traveller that is often sent back to the past on covert operations, mostly to hide government secrets. However, you have a large vocabulary that could potentially give away your cover, and reveal that you are a time traveller. Thankfully, you have MATLAB to help you double check some of your favorite words for a certain year before you use them so your operations remain undetected!

Function Description:

Write a function that takes in an array of data corresponding to the use of a word for each decade, and determine whether use of the word will keep your identity secret. The first row of the array contains the years, with 1 representing the year 1901, and the second row contains all the corresponding rankings of a word's popularity for that year.

To determine the output, you must first calculate the numerical derivative of the data. Then, use linear interpolation to determine both the popularity value of the word for the given year, and if the value is increasing or decreasing at the time. The values are scaled such that if the value is anything above 30, you will be able to blend into the past. However, you must be careful. If the use of the word is declining, give yourself a warning so you don't stay too long and give away your identity.

There are 3 different scenarios that will determine your output string:

- If the value is 30 or below, the output string should read:

  `'<word> is only a <popularity value>, you'll never blend in!'`

- If the value is above 30 and decreasing, the output string should read:

`'<word> is a <popularity value>, you'll fit right in! But don't stay too long, it's starting to decline.'`

- If the value is above 30 and increasing, the output string should read:

`'You're in the clear! <word> is a <popularity value> and it's on the rise.'`

Notes:

- The x-values can be irregularly incremented.
- The 3rd input will be the actual year, i.e. the double `1901` (not 1).
- Round the interpolated values to the second decimal place.

Function Name: `area51`

Inputs:

1. *(double)* 2xN array of data representing time (s) and velocity (m/s)
2. *(double)* 1x2 vector representing range of possible distances (km) of the alien planet
3. *(double)* Number representing how far away in days the alien planet is

Outputs:

1. *(char*) A sentence describing the where the craft came from

Plot Outputs:

1. 3x1 Subplot with Position vs. Time, Velocity vs. Time, and Acceleration vs. Time plots

Background:

You finally land a job at the top secret location called Area 51, and on your first day, you and your team uncover a spacecraft! You find the spacecraft's navigation systems are still intact and are able to recover information about the craft's velocity. Upon looking at the data, you notice that the craft started on Earth and tried to leave, but then the navigation systems broke and the craft crashed back on Earth. Your team needs the data about position and acceleration to determine whether or not this craft was on the path to arrive at a previously identified alien planet, and what better way to do this than through MATLAB?

Function Description:

Given an array of data points with time data in the first row and the corresponding velocity data in the second row, determine the position and acceleration data based on the inputs. Then, plot these on separate subplots with position vs time first, then velocity vs time, and lastly acceleration vs time. This should be a 3x1 subplot, so they will be stacked vertically. Since the acceleration vector will be one element shorter than the others, you should add the point (0,0), representing an acceleration of 0 m/s^2 at a time of 0 sec. Follow these style rules for plotting:

- Use blue asterisks for the position subplot, green plus signs for the velocity subplot, and red diamonds for the acceleration subplot.
- The points of all three subplots should be connected with a solid line.
- The axes should be set from 0 to the max value of that axis data set.
- Each subplot should be titled `'Position vs. Time'`, `'Velocity vs. Time'` and `'Acceleration vs. Time'` respectively.
- The x and y axes should be labeled accordingly with their units in parenthesis.
    - E.g. for the Acceleration vs. Time graph, the x label should be `'Time (s)'` and the y label should be `'Acceleration (m/s^2)'`.

Continued...

Next you need to check if the craft could have come from a previously identified planet. This known planet is located somewhere in a range of distances, given in km, away from Earth (the second input) and located a certain amount of days away from Earth (third input). Use extrapolation to determine whether or not this spacecraft would be within the range if it traveled for the length of time given in the third input. If it is, output the string-

```
'You've found the alien planet! It is located <distance> km away.'
```

If the location of the craft at that time is not within the range, output the string-

```
'This craft did not come from the known alien planet. Better luck next time!'
```

Notes:

- You are guaranteed to need to use extrapolation (the time value will be outside the range of times in the first input).
- Use linear extrapolation.
- Round the distance to the second decimal place only when formatting it into the output string, using **%.2f**.
- At a time of zero, the craft is on the Earth, and as the time increases the distance away from Earth increases.
- Make sure to account for unit conversions!
- The array of time and velocity will always have data for at least 2 time points.
- The first column of the data vector will always be [0; 0], representing a velocity of 0 m/s at a time of 0 s.

Function Name: `cropPolygon`

Inputs:

1. (*double*) A 1xN vector containing the lengths of each consecutive line segment of the crop polygon
2. (*double)* A 1xN vector containing the counterclockwise angles in degrees between each consecutive line segment and its previous line segment

Plot Output:

1. The plotted crop polygon

Backround:

You have taken an interest in drawing crop circles and you are wondering if MATLAB can do it for you. You want to be able to draw many different crop patterns, not just circles so you decide to make the images out of straight lines. (You know that you can make something that looks like a curve by taking many small line segments and attaching them end to end at small angles from one another.) You will be able to use this function to draw all different kinds of shapes and patterns!

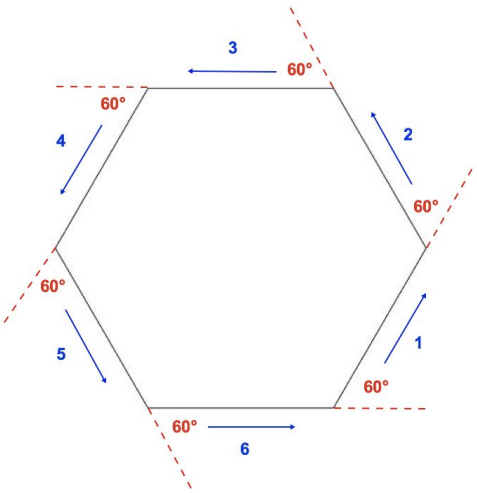
Function Description:

This function will take a vector of line segment lengths and angles and plot a crop polygon. The crop polygon will start at (0,0) and each line segment will be drawn from where the last one ends. Your first input will be a vector specifying the lengths of each consecutive line segment in the crop polygon. Your second input will be a vector specifying the *counterclockwise* angles between consecutive line segments.

Example:

The following input to `cropPolygon` will result in the hexagon below:

```
>> cropPolygon([5, 5, 5, 5, 5, 5], [60, 60, 60, 60, 60, 60]);
```



Continued...

The function will create the black hexagon. The blue and red lines and numbers are for reference. Each line segment has a length of 5. The blue arrows show the order in which the lines are plotted. The angles specified by the second input are also shown as the counterclockwise angles between consecutive line segments.

To rotate a set of points counterclockwise around the origin, multiply the coordinates by the rotation matrix as follows:

$$\begin{bmatrix} x_{rotated} \\ y_{rotated} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x_{original} \\ y_{original} \end{bmatrix}$$

This is matrix multiplication, not element-wise multiplication!

You do not have to use rotation matrices to solve this problem, but it is given here if you would like to use it.

Notes:

- Line segments should be drawn in black.
- The first angle is measured from the x-axis.
- Use `axis square` and `axis off` for you plot output.
- Use `sind()` and `cosd()` for sin and cos with degree inputs.
- You could theoretically draw anything you want with this function; there will be brownie points awarded for the coolest test cases.
- Notice that an n sided shape with equal side lengths and equal angles would have a second input of n angles that are each given by the equation: angle = 360/n. The larger you make n the closers your shape looks like a circle.

Hints:

- Since the rotation matrix rotates points counterclockwise relative to the positive x axis, and the angle of each line segment is given counterclockwise relative to its previous line segment, try keeping track of each line segment's angle counterclockwise relative to the positive x axis. Then, the rotation matrix can be used to rotate and create any line segment.

Extra Credit

Function Name: isSeahavenReal

Inputs:

1. *(double)* A 2xN array of Truman's collected data
2. *(char)* A 1xN list of colors

Outputs:

1. *(double)* 1xN The coefficients of the perfectly fitted polynomial

Plot Outputs:

1. A 1x2 subplot, where the first subplot displays the successively approximated polynomials, and the second contains the sum of the absolute errors.

Background:

Truman Burbank is starting to suspect that his reality is a lie. With stage lighting fixtures falling from the sky, and his wife and best friend insisting that he should not attempt to move to Fiji, Truman decides to record some data from his daily life. He makes some measurements of the times his repetitive life events occur throughout the day. If he can show that his life events follow a predefined polynomial pattern, he will have confirmed that, in fact, his life is a television show, constructed by some unknown entity. He needs your MATLAB expertise.

Function Description:

You will be provided with a set of Truman's measurements, where the x values are the top row, and the y values are the bottom row. Starting from order = 0, attempt to find the (coefficients of the) lowest order polynomial that perfectly fits the data Truman provided. Continue increasing the order until this condition is met:

> **If f(x) is the nth-order polynomial in question, and (x, y) are the data points provided by Truman, the sum(abs(y - f(x))), over every x, is less than 0.00001. This is commonly called the "sum of absolute errors".**

The "diameter" of a set of data is the distance between the lowest x value and highest x value. Create a plot where the x axis contains the data with a buffer on each side of 5% of the diameter. For each nth order polynomial you try, plot the polynomial as a line of 1000 linearly spaced points over this buffered domain. Cycle through the colors provided in the second input for each of these polynomial's plots.

Plot the final, condition-meeting polynomial in the same manner, except colored black. Then, plot Truman's provided data as black plus signs. Name the x-axis `'x'`, the y-axis `'y'`, and title the plot `'Successive Approximations'`. For the axis on this first subplot, use `axis tight`.

Continued…

For the second subplot, plot the degree of each polynomial you try vs. the sum of absolute errors as a black line. Use the default MATLAB axes on this plot. In other words, do not manually change the axes. Name the x-axis `'degree'`, the y-axis `'sae'`, and the title `'Sum of Absolute Errors'`.

The output should be the coefficients of the final polynomial that you find.

Notes:

- Seahaven is a TV set, of course, so there will always be an answer for the test cases we provide. Depending on how you make your own test cases, there may not be a feasible solution.