Several problems on this homework require you to create plot outputs which cannot be checked for visual equality using any built-in tool in MATLAB. To that end, we have provided you with a function called `plotCheck` that will check the plot output from your functions against the solution. You can type `help plotCheck` in the Command Window for a full explanation of how to use the function.

This is the first semester using `plotCheck` in its current form and it will probably not be perfect. The final test of whether or not your plots will get credit is this: if you can identify ANY visual differences between your plot and the solution plot, you will not get credit. **If you cannot see any differences at all, then you will get credit for the problem, even if `plotCheck` tells you otherwise.** However, you should only result to a visual comparison if the plot checker consistently returns a false negative and you are sure you plots are identical.

Function Name: `plotSnake`

Inputs:

1. *(double)* An array representing a game of snake

Outputs:

(none)

Plot Outputs:

1. A plot of the game of snake

Function Description:

The time has finally come for you to turn your snake game into something that actually looks (sort of) like snake! To do this, you will treat the indices of the snake on the board as Cartesian coordinates. Wherever a snake segment exists on the board, you will plot a corresponding square. Additionally, you will plot a square where the candy is located.

First, it's important that the player know where the bounds of the board are. To accomplish that, you should plot a black rectangle that has its upper left corner at `(1, -1)` and bottom right corner at `(C, -R)` where `C` is the number of columns in the input board and `R` is the number of rows in the input board.

When plotting the actual snake, to make sure the board indices translate properly to coordinates, you should use the column index as the x coordinate and the negative row index as the y coordinate.

Here's an example:

```
board = [ 0, 0, 0, 0, 0, 0; ...
          0, 1, 2, 3, 0, 0; ...
          0, 0, 0, 4, 0, 0; ...
          0, 0, 0, 5, 0, 0; ...
         -1, 0, 0, 0, 0, 0; ...
          0, 0, 0, 0, 0, 0]
```

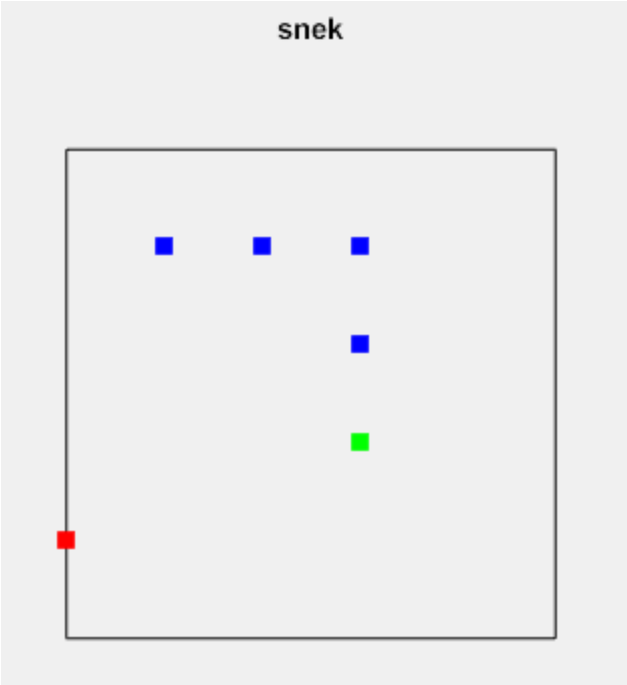
The row/column indices for the snake in this array, in order from head to tail are: `(4, 4), (3, 4), (2, 4), (2, 3), (2, 2)`. This would translate to the following Cartesian points: `(4, -4), (4, -3), (4, -2), (3, -2), (2, -2)`. The cookie is located at `(5, 1)` which translates to `(1, -5)`.

All of this will get the job done, but it won't look very good. So to fix that, here are the formatting settings you should use:

- The head of the snake should be a solid green square.
- The rest of the snake body should be solid blue squares.
- The candy should be a solid red square.

- All plotted squares should be 8 pixels in width and height.
- The board border should be a black line.
- The plot axis ranges should be set to: `(x min, x max) = (0, C + 1)` and `(y min, y max) = (-R - 1, 0)` where `R` and `C` are defined the same as above.
- Set `axis equal` and `axis off`
- The title of your plot should be `'snek'`

After all of these settings are applied, the example board above will correspond to the following plot:



Notes:

- `help axis` and `help plot` will get you started with the formatting.
- To plot solid shapes, you can use `'MarkerFaceColor'` and `<color char>` as two additional inputs to the `plot()` function. `<color char>` can be any of the options for plot colors.
- To set the plot shape size, use `'MarkerSize'` and `<num pixels>` as two additional inputs to the `plot()` function. `<num pixels>` is an integer that will be used to determine the marker size.
- The previous two notes can be used at the same time (you can specify `MarkerFaceColor` and `MarkerSize` in the same plot call).

Hints:

- Experiment with `find()` on arrays.

Function Name: `metaData`

Inputs:

1. *(char)* The filename of an Excel file of data to be plotted

Outputs: (none)

Plot Outputs:

1. A plot containing a 2xN array of subplots according to the following description

Function Description:

Imagine you are a student in a MATLAB class at Georgia Tech and you must complete a homework problem about graphing data. The problem requires you to write a function called `metaData` that reads in an Excel file and plots the data it contains in a series of subplots. The Excel file will contain multiple columns of numerical data, each with a header. The data will be plotted in multiple subplots. The first column of data are the x values for all of the subplots. Each of the other columns of data will be the y values for a corresponding subplot. Subplot 1 should contain the first column of y data, subplot 2 should contain the second column of y data, and so on. Every axis should be labeled with the column header from the data plotted along that axis. The dimensions of the subplot array should be `2xN` where:

```
N = ceil(number of subplots / 2)
```

All of the original data should be plotted as a black line. In addition to plotting the given data, you should also plot the numerical derivative (as a blue line) and the numerical integral (as a green line) of the data using trapezoidal approximation in each subplot. When plotting the numerical derivative, do not include the last x data value.

Notes:

- The Excel file will be formatted with the headers in the first row and the data in columns.
- There will be no non-numerical data or extraneous columns of data.
- You may have one less subplot on the second row than the first row.

Hints:

- The `subplot()` function will be helpful.

Function Name: `newtonsMethod`

Inputs:

1. *(double)* A vector of x values
2. *(double)* A vector of y values
3. *(double)* A starting x value

Outputs:

1. *(double)* A root (where the function equals zero) found using Newton's method
2. *(double)* The number of new x values found after the start value, including the root

Function Description:

Newton's method is a solution to finding the roots of a mathematical function. The roots of a function are where the function has a value of zero. You can learn more about Newton's method here. In short, it finds a root of a mathematical function by successively finding the roots of the tangent lines to that function. At the x value of each new tangent line root, another tangent is found until the algorithm converges on a root. Each successive x value can be found with the following formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Write a function that takes in a set of x and y values, and first determine the coefficients of the highest order unique polynomial that passes through all the points. Take the derivative of the function with respect to x, and use the derivative and the original function to find new x values until the absolute difference between your new and previous x is less than 0.0001. Return this as your root and return the total number of iterations it took to find that root.

Notes:

- You should calculate the derivative from the polynomial coefficients, not the raw data.
- The input x and y vectors are guaranteed to be the same length.
- Round your final root to 4 decimal places.
- You should only be rounding your final answer, not during each calculation.
- Inputs will not be given such that you calculate a slope of zero.

Hints:

- `polyval()` and `polyfit()` will be useful.

Function Name: `dragRace`

Inputs:

1. *(double)* An NxM array of sampled points in time
2. *(double)* An NxM array of sampled velocities
3. *(cell)* A 1xN array of car names
4. *(double)* The specified race distance

Outputs:

1. *(char)* A string describing the race results

Function Description:

You always dreamed of being a racecar driver, but with your numerous other MATLAB projects, you haven't had the time to practice. Fortunately, the most prestigious auto race in the world is taking place this week: The MATLAB Grand Prix!

Using the given set of times and velocities, determine which car wins the race of given length and which car has the fastest acceleration. Each row corresponds to the data of one racer whose name is in the corresponding index in the cell array input.

Numerically integrate the given values to find the distance each car covers. Since each racecar will likely cross the finish line between two sampled points, linearly interpolate to find the times each car crosses the finish line. Then, numerically differentiate the velocities and determine which car has the fastest acceleration between two adjacent data points.

Output a string with the format:

```
'The <race winner> won the <race distance> meter race in <time> seconds! The <fastest acceleration car> had the fastest acceleration at <acceleration> m/s^2!'
```

Numbers in the output should be rounded to the nearest tenth. Be sure to check your answers against the solution file.

Notes:

- The times are given in seconds and the velocities are given in meters per second.
- The times and velocities array will have the same number of columns.
- The number of cars and the number of rows in the times and velocities arrays are the same.
- Remember to use linear interpolation between data points.
- You can use `'%0.1f'` as a format specifier inside of `sprintf()` to display a number to one decimal place.

Hints:

- `cumtrapz()` will be useful

Function Name: `clockFaces`

Inputs:

1. *(char)* A time in 12-hour format (`HH:MM`)
2. *(char)* The time difference in your destination

Outputs:

(none)

Plot Outputs:

1. A 1 x 2 subplot showing the time in your current location and your destination

Function Description:

As an avid world traveler, you often find yourself sitting around in airports with nothing to do. To fill the time, you decide to write a MATLAB function to help you tell the time in your destination. This function will take in a time of the form `'HH:MM'` and a time difference in the form `'+<time diff>'` or `'-<time diff>'` and convert it to two subplots representing analog clocks, where the subplot on the left represents your current time and the subplot on the right represents the time at your destination. The plots should be titled `'Current Time'` and `'Destination Time'`, respectively.

Each clock face should have a radius of `1` unit and be composed of 100 equally spaced points connected by a black line. There should be hour numbers equally spaced around an "invisible circle" of radius `0.9`. There should also be a data point at the origin plotted with a black circle. The hour hand of the clock should be of length `.5` units and be plotted in blue. The minute hand should be of length `.7` units and be plotted in red. To avoid rounding errors, please do all of your angle math in degrees, not radians!

You may want to use the 2D clockwise rotation matrix to do this problem, which is:

```
[cosd(θ), sind(θ); ...
-sind(θ), cosd(θ)]
```

To use the rotation matrix to rotate a set of points, you must arrange the points into a 2xN array where the first row contains x coordinates and the second row contains y coordinates. You must also use **matrix multiplication** (`*`) not scalar multiplication (`.*`).

You do not have to interpolate the hands between numbers. For example, if the given time was `'12:15'` the hour hand should be vertical and the minute hand should be horizontal.

Notes:

- The `sind()` and `cosd()` functions are the sine and cosine functions with degree inputs.
- You should use `axis square` and `axis off`.
- You are guaranteed to be given a valid 12-hour time.
- You can use the `text()` function to plot text.
- You can specify `'HorizontalAlignment'` and `'center'` as the 4th and 5th input to the `text()` function to center the text value at the specified coordinate.