Drill Problem: #1

Function Name: polyFun

Inputs:

1. (*double*) A 1xN vector of x data points
2. (*double*) A 1xN vector of y data points
3. (*double*) A number indicating the power of the polynomial

Outputs:

(*none*)

Output Plots:

1. A 1x3 plot of 3 subplots for original data, derivatives, and integrals of the input data

Function Description:

Before jumping into images just yet, here's one more numerical methods problem! With an input vector of x values, an input vector of y values, and a value indicating the polynomial order to use for a fit, create three subplots with the following characteristics.

First Subplot:

- Plot the original data in green.
- Plot a polynomial fitted to the data, with 200 data points between the minimum and maximum x values, in blue.
- Title the subplot `'Original Data'`.

Second Subplot:

- Plot the fitted analytic derivative (derivative calculated from the polynomial fit) with 200 data points in green.
- Plot the numeric derivative in blue. The x values of the numeric derivative plot should be the original x values **excluding the last**.
- Title the subplot `'Derivative Data'`.

Third Subplot:

- Plot the fitted analytic integral with 200 data points in green.
- Plot the numeric integral in blue; use `cumtrapz()`. The x values should be the same as the original data.
- Title the subplot `'Integral Data'`.

Each plot should have the x axis labeled `'x axis'` and the y axis labeled `'y axis'`, and the data should be plotted in the order given for each subplot.

Drill Problem: #2

Function Name: checkImage

Inputs:

1. (*char*) The name of an image file
2. (*char*) The name of a second image file

Outputs:

1. (*char*) A sentence comparing the two images

File Outputs (Potential):

1. A black and white image file of where the two input images differ, named: `'<image1>VS<image2>.png'`

Function Description:

Given two images, write a function called `checkImage` that determines if the two images are the same, or if and how they are different.

- If the two images are completely identical, the output should read `'The images are the same.'`
- If the two images do not have the same dimensions, the output should read `'The images have different dimensions.'`
- If the two images have the same dimensions, but have different color values at the same pixel, the output should read `'The RGB values are different.'` Additionally, if the two images have different colors, you should write a new image that is white everywhere the two images have the same RGB values and black everywhere they are different. This new image should be called `'<image1>VS<image2>.png'`. For example, if you were comparing `'lilacs.png'` and `'roses.png'`, you would call your new image `'lilacsVSroses.png'`.

Notes:

- The `imwrite()` function will be useful.
- You **should not use iteration** in this function.
- Use a 3-layer uint8 array to write the output image.
- This function will be useful for the rest of your homework.

Drill Problem: #3

Function Name: trumpYourCat

Inputs:

1. (*char*) The filename of a .png cat picture
2. (*char*) The filename of a .png green-screened toupee picture

Outputs:

(*none*)

File Outputs:

1. An image of the toupee picture overlaid onto the cat picture

Function Description:

Fear not if you missed the 24-hour internet sensation “Trump Your Cat,” because you will be writing a MATLAB function to relive those glorious moments! The function will take in the filename of a cat image and the filename of a green-screen toupee image and overlay them so it appears the cat has a toupee. If you look at the toupee image you will notice it is mostly green. This is called the key color. The key color is removed from the toupee image and the remaining pixels replace the corresponding pixel of the cat image (read: masking). The key color is pure green (`[R, G, B] = [0, 255, 0]`), but because the edge of the toupee is not well defined, the range of RGB values that you will key out from the toupee image is:

```
red <= 150
green >= 180
blue <= 120
```

A pixel must match **ALL** of these conditions to be keyed out. All remaining pixels are considered to be part of the toupee and should replace pixels in the cat image. Even after accounting for this range of pixels, there will still be a small green outline around the toupee. This is okay!

The test cases have been made such that the toupee already lines up with the cat's head and you don't need to do any scaling, rotating, or moving to get the toupee onto the cat.

Once you have successfully toupeed your cat, you will output that image to a .png file. The filename should be the original filename with '`trumped_`' appended to the front. For example, if the input image was '`myCat.png`' the output image would be '`trumped_myCat.png`'.

Notes:

- The images will always be .png images and both the cat and toupee images will always be the same size.

Drill Problem: #4

Function Name: hairParlour

Inputs:

1. (*char*) The filename of a .png image of a person
2. (*char*) The original hair color of the person in the image
3. (*char*) The new hair color of the person in the image

Outputs:

(*none*)

File Outputs:

1. A .png image of the “Before” and “After” shots of each patron

Function Description:

In order to combat broke-college-student-syndrome, you have picked up a part-time job at a hair salon in Midtown. You are going to put your MATLAB skills to use making a function called `hairParlour` that will show customers what their hair coloring requests will look like. The filename of a customer's image will be given as the first input to the function. The second input to the function is their current hair color and the third input is the color you are going to dye their hair.

You only offer three dye choices, and you can only color three hair colors: Red, Blonde, and Brunette. The 'Original Color Range' column of the table below shows the color range that corresponds to each original hair color. The 'Dye Color' column contains the RGB values you should change the hair to if the color in the ‘Hair Color’ column is the new hair color desired.

Your output should be a horizontal concatenation of the original image on the left and the edited image on the right. The output image file should have the same name as the original image with '_beforeAfter.png' appended to the end.

Hair Color	Original Color Range	Dye Color

`'Blonde'`	`205 <= red <= 255`
	`180 <= green <= 230`
	`0 <= blue <= 165`
		`red = 255`
		`green = 219`
		`blue = 74`

`'Brunette'`	`100 <= red <= 155`
	`30 <= green <= 80`
	`15 <= blue <= 60`
		`red = 141`
		`green = 70`
		`blue = 50`

`'Red'`	`210 <= red <= 255`
	`80 <= green <= 115`
	`25 <= blue <= 30`
		`red = 212`
		`green = 80`
		`blue = 24`

Notes:

- You can assume that the specified hair colors will not appear anywhere else in the test images.
- If the second and third inputs to the function are identical, the 'after' image should be unchanged from the original.

Homework 13: Images

Drill Problem: #5

Function Name: ascii2image

Inputs:

1. (*char*) The filename of a .txt ASCII image
2. (*char*) A string of 'key' characters
3. (*uint8*) A vector of grayscale intensity values

Outputs:

(*none*)

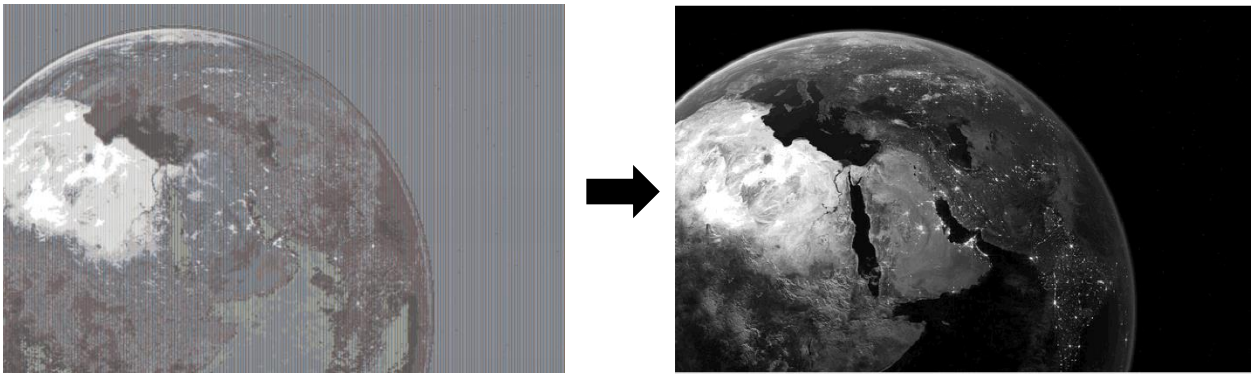
File Outputs:

1. The ASCII image recreated as a .png greyscale image

Function Description:

ASCII art, the process of turning an image into text characters, has been around for years. You decide to go against the grain and convert those ASCII images back into their original form. You'll create a function called `ascii2image`, which takes in an ASCII image as a .txt file and converts it into a .png image. Because characters do not contain any color information, the best you can do is convert the ASCII art into a grayscale image. The second and third inputs to the function provide the information for making this conversion. The second input contains all of the characters that could be found in the ASCII art and the third input contains the grayscale intensity corresponding to each of these characters. The intensity for a given character should be used as the red, green, and blue layer value anywhere that character appears in the ASCII art.

After you have converted your image, output it to a .png file. The filename should remain the same. For example, if the input file was '`earth.txt`' the output image should be '`earth.png`'.



Drill Problem: #6

Function Name: snapchat

Inputs:

1. (*char*) The filename of a given snap as an RGB image
2. (*double*) The number of swipes corresponding to the desired filter
3. (*uint8 or double*) A Snapchat Geo-filter, a saturation factor, or empty

Outputs:

(*none*)

File Outputs:

1. An image of the original snap with the proper filter applied

Function Description:

Have you ever thought: Snap! I should have taken that picture using snapchat so I can get those awesome filters! Well look no further, because you can use MATLAB to apply those filters on a picture that's already been taken. Here's how:

Based on the number of swipes, the filter is different (just like on your phone).

0 Swipes (no filter):

The output image is identical to the input image.

1 Swipe (negative filter):

The output image is the negative of the input image.

2 Swipes (grayscale filter):

The output image is the grayscale of the input image. You may not use the `rgb2gray()` function for this filter.

3 Swipes (saturation filter):

The output image is the input image with saturation adjusted by the third input to the function. To do this, you must first convert the input image from RGB color-space to HSL color-space (Hue, Saturation, Lightness). This conversion can be done with the provided helper function `rgb_hsl()`. Documentation on how to use this function can be found in the comments at the top of the `rgb_hsl.m` file. Once you have converted your function to HSL, multiply the saturation layer (the 2nd layer of the HSL array--analogous to the green layer of an RGB array) by the factor provided in the third input. Then convert the image back to RGB color-space and output the image.

Drill Problem: #6

4 Swipes (geo-filter):

Apply the "Atlanta Life" geo-filter to the input image. The RGB array of the geo-filter will be provided in the third input to the function as an MxNx3 uint8 array. You must resize the Geo-filter so that it matches the size of the input image. You **must** use the `linspace()` method of resizing the geo-filter. Do not worry about the potential for small rounding errors using this method. Then, overlay the original image with the Geo-filter. The pure green pixels (RGB = [0, 255, 0]) in the geo-filter image should be keyed out and replaced with the pixels of the original image. This process is the same as the process for the `trumpYourCat` problem, however, there is no tolerance for colors close to pure green.

The output image filename should have a "_filtered" appended. For example: `snap1.png` would become `snap1_filtered.png`.

Notes:

- The input swipes will always be an integer greater than or equal to 0. However, they may be beyond 4 swipes. In that case, swiping 5 times is the same as swiping 0 times, swiping 8 times is the same as swiping 3 times, etc.
- The 3rd input given will never be mismatched with the corresponding filter. For example, if the first input specifies the saturation filter, the third input will never be geo-filter data.
- If rounding is required, round normally **just before** casting back to uint8 (and not sooner or later).
- The function `rgb_hsl()` is provided for your use, please **read the instructions** at the top as to how to use it.