Many of the functions in this homework include image outputs. You can check your images against those produced by the solution function by using the provided `checkImage` function. You can type `help checkImage` to see how to use the function.

Also, the solution function will append '_soln' to the filename of any file it produces. The filenames your functions produce SHOULD NOT have this suffix.

Happy coding,
~Homework Team

Function Name: `str2img`

Inputs:

1. *(char)* A phrase to convert into an image
2. *(cell)* A font library
3. *(double)* A 1x3 RGB vector

Outputs:

(none)

Image Outputs:

1. The phrase written as an image

Function Description:

This function will write any string to an image. The process you will use is similar to the process used in the early days of computer displays. You will be given a font library in the form of a cell array. The array will be formatted such that calling:

```
fontCellArray{'<char>'} => MxN uint8 array that represents the letter
```

For example: `fontCellArray{'A'} =>`

```
[0, 0, 1, 1, 1, 0;
 0, 1, 0, 0, 0, 1;
 0, 1, 0, 0, 0, 1;
 0, 1, 0, 0, 0, 1;
 0, 1, 1, 1, 1, 1;
 0, 1, 0, 0, 0, 1;
 0, 1, 0, 0, 0, 1;
 0, 0, 0, 0, 0, 0]
```

The font will always be monospaced, meaning that each character will be the same width and height. You should horizontally concatenate all the characters in the string together, then assign red, green, and blue values based on the color given in the third input. This input will always be a 1x3 vector and each entry will be an integer between 0 and 255. The color order will always be `[<red>, <green>, <blue>]`. Additionally, the background color (the parts of the image that are not part of a letter) should be white (`[R, G, B] = [255, 255, 255]`).

Your output image should be saved as `<camelCaseInputString>.png` where `<camelCaseInputString>` is the input string in camelcase (you should use your camelCase function from homework 5 for this).

Function Name: `image2excel`

Inputs:

1. *(char)* Filename of image to convert to an Excel file

Outputs:

1. (*cell*) A cell array representing the data to write to the Excel file

Function Description:

First of all, if you want to nerd out and see the YouTube video that inspired this problem (and maybe get a laugh along the way), watch this video. If you don't care what inspired this problem, carry on.

You will be given an image file, which you must convert to a cell array. Each column of the original image will correspond to the same column in the cell array. However, the red, green, and blue layers of each pixel will be placed in separate, adjacent rows in the cell array.

Given an 2x3 image containing the following RGB data:

`Red: 200`
`Green: 10`
`Blue: 65`
	`Red: 40`
	`Green: 75`
	`Blue: 215`
		`Red: 230`
		`Green: 40`
		`Blue: 20`

`Red: 15`
`Green: 150`
`Blue: 180`
	`Red: 25`
	`Green: 255`
	`Blue: 10`
		`Red: 85`
		`Green: 55`
		`Blue: 210`

Convert it to a 6x3 cell array containing the following RGB data:

		Column 1	Column 2	Column 3
Row 1	**Red**	200	40	230
Row 2	**Green**	10	75	40
Row 3	**Blue**	65	215	20
Row 4	**Red**	15	25	85
Row 5	**Green**	150	255	55
Row 6	**Blue**	180	10	210

This cell array represents the data that will be written to an Excel file that contains color formatting, and the Excel file can be zoomed out to display the original image! You will be given a helper function called `excelDisplay` to write to and display the Excel file for the cell array you created. After finishing your `image2excel` function, be sure to test the output using `excelDisplay`. Note that if you are on a PC, you are welcome to just use `xlswrite`. This helper function is mainly for Mac users so they can view the result.

Notes:

- The `excelDisplay` helper function relies on the specially formatted Excel file called `excelImage.xlsx` to display the cell array. Therefore, it's vital that you do not modify the `excelImage.xlsx` in any way! If this file does get messed up, just download it from T-Square again. Open `excelImage.xlsx` to view the resulting Excel image.
- The `poi_library` folder included with this homework contains Java files that allow Excel files to be written using a Mac. DO NOT delete this folder (I guess you can delete it if you're on a PC, but it's not hurting anything).

Function Name: `carnival`

Inputs:

4. *(char)* The filename of an image
5. *(double)* A vector describing mirror waves

Outputs:

(none)

Image Outputs:

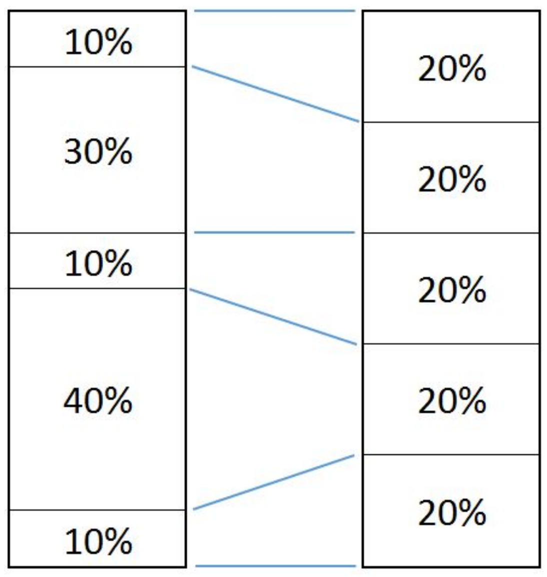
2. Edited image with `'_warped'` appended to the filename

Function Description:

Summer is coming and with it are all the traveling fairs and carnivals. Ferris wheels and and funnel cakes are great, but you're really interested in those wavy fun house mirrors! In anticipation of all the summer fun you'll have in a few weeks, you will make a function called `carnival` that takes the filename of an image and a vector describing the wavyness of the mirror. Each element of the vector is a number between 0 and 1 (non inclusive) and the sum of the vector will always be 1. The length of the vector determines how many equal "segments" the resulting image should be broken into. The number at each index determines what percentage of the original image goes in that section. For example, consider the input:

`[0.1, 0.3, 0.1, 0.4, 0.1]`

This vector would represent the following mapping from the original image to the output image:



The fact that the vector is of length 5 determines that the output image should be split into 5 segments, each comprising 20% of the rows and all columns. Each element in the vector determines how much of the original image goes to that segment. The first number is .1, so the first 10% of the rows of the original image get resized to fill 20% of the rows of the new image. The second number is .3, so the next 30% of the rows get resized to fill the next 20% of the rows of the new image, and so on.

This function only rescales the rows of the image; the columns are not scaled.

Notes:

- You should use `imresize()` for all image resizing.
- The number of rows in the image will always be divisible by the vector length.
- The percentages will never produce a fractional number of rows.
- The output image is not actually broken into segments, this is just a helpful way to visualize the problem.

Function Name: `goat`

Inputs:

1. *(char)* Filename of the original image to be edited
2. *(double)* A 1x3 vector containing the original RGB values to search for
6. *(double or char)* A 1x3 vector or comma separated string indicating the range of RGB values
7. *(double)* A 1x3 vector containing the replacement RGB values
8. *(double)* A 1x2 vector for the range of the brightness gradient

Outputs:

(none)

Image Outputs:

1. Edited image with '_farewell' appended in the filename

Function Description:

Kobe Bryant, regarded as one of the Greatest of All Time (GOAT) basketball players, is scheduled to play his final NBA game next week. During his 20 illustrious seasons with the Los Angeles Lakers, many have wondered, what if Kobe played for another team? Well, wonder no more. Open MATLAB and see for yourself.

Given an image, you will search for a specific range of colours and replace them with the RGB values desired. If the third input is a double vector, the range should be ± of the range for each colour. If the third input is a string, it will be formatted as `'xx%, xx%, xx%'`. They are **not** guaranteed to be 2 digits each. You should add or subtract the given percentage from the original RGB values to determine the range. The range should be inclusive in either case.

For example, if the original (2nd input) is `[100 200 150]`, and the range (3rd input) is `[90 40 30]`, you should search in the image for where the RGB values are between `[10 190]`, `[160 240]` and `[120 180]`, respectively.

If the original (2nd input) is `[100 200 150]`, but the range (3rd input) is `'100%, 50%, 5%'`, you should search in the image for where the RGB values are between `[0 200]`, `[100 300]` and `[143, 158]`, respectively. You should round the range to the nearest integer.

Once you have determined the pixels from the original image where the RGB values fall in the range, replace them with the new RGB values (4th input). Mirror the edited image and concatenate it to the right of the original image.

After you place the original and edited images side-by-side, create a brightness gradient to simulate a curtain closing effect as an ultimate tribute to Kobe and his greatness. Use the `linspace()` function to create a gradient of the brightness factor, given in the last input. Round to the nearest integer and apply the gradient to the original image, then mirror the gradient and apply to the edited image (if you choose to brighten the edited image before mirroring, that is also okay). For example, if the given image is 5x12 pixels and the last input is [60 90], your gradient array should look like

```
60    63    65    68    71    74    76    79    82    85    87    90
60    63    65    68    71    74    76    79    82    85    87    90
60    63    65    68    71    74    76    79    82    85    87    90
60    63    65    68    71    74    76    79    82    85    87    90
60    63    65    68    71    74    76    79    82    85    87    90
```

The leftmost pixels will be at 60% brightness while the rightmost pixels will be at 90% brightness. A 100% brightness means unchanged RGB values while a 0% brightness corresponds to black (`[0 0 0]`). This would be the gradient you apply to the original image (left side of your output image); the gradient for the edited image will be mirrored.

Finally, save the concatenated image with `'_farewell'` concatenated to the filename. For example, if the input filename is `'kobe1.png'`, the output file should be named `'kobe1_farewell.png'`.

Notes:

- The inputs will always be in the order of RGB.
- You may **not** use the `repmat()` function.
- You may **not** use iteration to solve this problem.
- Round to the nearest integer in the following steps:
    - Round the calculated range if input is given as a string
    - Round the brightness gradient after using `linspace()`
    - Round the pixel values after the gradient has been applied
- The last input may contained negative values and values greater than 100.

- Correctly guessing the team colour of choice in the test cases will earn you an endorsement (via Piazza).

Hints:

- You can use [100 100] as the last input to test whether the image is edited properly before working on the brightness gradient.
- Keep all variables as type `double()`. Only convert to `uint8()` when outputting the image.

Function Name: `remNoise`

Inputs:

1. *(char)* The "base" image name
2. *(double)* The largest image index
3. *(char)* The method of frame blending to use

Outputs:

(none)

Image Outputs:

1. An image of the background of the video

Background Information:

Removing noise from a video is often a first step when doing automated video analysis. You may think that removing noise simply means getting rid of grainyness from a video, but it can actually do much more than that. You will use a couple noise reduction techniques to extract the background from a video of moving objects. One simple method for identifying the background is to average several frames together. This works well in some cases, but can cause "ghosting". This phenomenon is when moving objects can still be seen as blurry and semitransparent overlays on top of the background. One way around this is to take the mode of each pixel across several frames. This can produce problems of its own, but it solves the ghosting issue. There are much more advanced methods of extracting backgrounds from video, but mean and mode are the only two this function will implement.

Function Description:

All of the "video" test cases for this function will be sequences of images whose filenames follow the format: `'<base>##.png'`. `<base>` will be given as the first input to the function and will be how you identify which image sequence to read. The numbers will always start at `'00'` and will count up to the second input to the function. For single digit image indexes, there will always be a leading zero, and you will never have more than 99 images in a sequence. The last input will always be either `'mean'` or `'mode'`. If the last input is 'mean', the function should average pixels together. If it is 'mode' the function should take the mode of the pixels. This process is described in detail below. Once you have computed the background image, you should save it under the filename: `'<base>_<method>.png'` where `<base>` is the first input and `<method>` is the third.

Blending Methods:

Both the `'mode'` and `'mean'` blending methods will operate on each color independently. For any given pixel in the video, mean blending will average the red values at that pixel across all frames and use that as the background red value at that pixel. Mode will do the same thing, except take the mode across all pixels and use that as the background value. Here is an example on a 2x2 video of 3 frames:

`Frame 1:`

R: 100 G: 80 B: 50	R: 230 G: 40 B: 100
R: 120 G: 100 B: 70	R: 230 G: 50 B: 120

`Frame 2:`

R: 130 G: 70 B: 20	R: 200 G: 30 B: 90
R: 120 G: 140 B: 80	R: 230 G: 60 B: 120

`Frame 3:`

R: 100 G: 80 B: 50	R: 195 G: 45 B: 120
R: 130 G: 100 B: 75	R: 230 G: 70 B: 120

For the rest of the example, we will only consider the pixel at position `[2, 1]`, but the following procedure should be done for all pixels in the image sequence.

The first step is to put all of the red values together, all of the green values together and all of the blue values together. For pixel `[2, 1]`, this would be:

```
R: 120, 120, 130
G: 100, 140, 100
B:  70,  80,  75
```

Using `'mean'` as the blending method, the function would calculate the background red value at this pixel to be 123 (remember that it must be a `uint8` so it's not a fraction). Likewise, the green value would be 113 and the blue value would be 75.

Using `'mode'` as the blending method, the function would calculate the background red value at this pixel to be 120 (the mode of 120, 120, and 130). Likewise, the green value would be 100 and the blue value would be 70. Note that there is a tie for the mode of the blue values; just use whichever one the `mode()` function returns.

Writing Efficient Code:

We encourage creative solutions to problems in this class and do not normally care about efficiency (how much time your code takes to run). However, this problem deals with much more data than any problem you have previously seen, and therefore, you will need to give efficiency a little bit of thought when doing this problem.

While you could do this problem with iteration, the code would be unusably slow. Instead, you should take advantage of the fact that both the `mean()` and `mode()` functions have a second input that specifies the dimension to operate across. Think about how you can use 3 dimensional arrays to get this to work to your advantage.

You may also want to preallocate arrays for this problem. This simply means that if you are planning on appending elements to an array inside of a loop, but you know what size the array will be at the end, you initialize the array to be that size before the loop runs. You don't have to know why this is faster, just know that it is faster (and in the case of this problem, much faster). Here is an example:

Without preallocation:

```
vec = []; %<= initialization
for i = 1:5
    vec = [vec, randi(10,
1)];
end
```

With preallocation:

```
vec = zeros(1, 5); %<= initialization and
preallocation
for i = 1:5
    vec(i) = randi(10, 1);
end
```

Both snippets of code accomplish the same thing, but the code that uses preallocation is faster. This does not matter for the relatively small arrays that you have been dealing with before, but images are huge arrays, and in this problem you are dealing with many images all together!

Notes:

- You should only use iteration to iterate through all of the images in the sequence; **DO NOT** use iteration for calculating mean or mode.

- Preallocate arrays wherever possible.
- You can use `%02.0f` inside of `sprintf()` to add leading zeros to a number.
- Type `addpath('imgSeq')` to add the test images folder to your MATLAB path.
- The video 'testCaseVideo.mp4' shows you what these frame sequences actually look like.

Hints:

- In a normal image, the red, green, and blue, layers are "stacked" in the 3rd dimension of the image array. Try "stacking" all of the same color layers from each frame into one 3D array.