CSCI 4831/5722 Computer Vision, Spring 2021
Instructor: Fleming
Homework 1, Due Tuesday, January 26st, by 9:55pm

Early Vision – One Image

For Homework 1, you will create a simple image-processing program, with functions similar to those found in Adobe Photoshop or The Gimp. The functions you implement for this assignment will take an input image, process the image, and produce an output image.

Note: it would be possible to achieve some the functionality required in this assignment by using built-in Matlab functions, especially from a couple of specialized toolboxes. You are NOT allowed to use these image-specific built-in functions. The convolution function (**conv**) is also not allowed. You will need to code your own implementation of these functions. Other functions that are not allowed: cart2pol, pol2cart. If in doubt, please ask if a certain built-in function is allowed.

Provided files:
A shell script *vision_hwk1.m* is provided to get you started. Also, a collection of images is provided for testing. You can use your own images as well.

What You Have to Do
Implement a menu driven program, where each button should trigger a call to a function. If you run the provided shell program, you will notice the following three buttons:

1. Load Image – loads one image file. In order to select one image, the file needs to be in the same folder as the main script. The loaded image will become the current image and can be passed as an input to other functions.
2. Display Image – displays the current image.
3. Exit Program – closes the menu and terminates the script.

You have to add and test additional functionality for the program. For every button/functionality you add, you can use one of the images to test it.

Implement a solution for each of the following tasks and add a menu button for each one. The solution/code for each task should be written as a separate function. Note: in Matlab, every function is written in a separate file.

Task 1 (0.25 points) Brighten: individually modifies the RGB channels of an image, brightening or darkening it. **Use Loops!**

```
function [ outImg ] = makeBright_L( inImg, brightness )
```

This function brightens each pixel and writes out the new image to `outImg`. The value of the input parameter `brightness` is the amount by which the image should be brightened, so it can be any real number in the range [-255 255]. Positive values will brighten the image, while negative values will darken the image. You must use loops to modify each individual pixel value.

Task 2 (0.5 points) Brighten2: individually modifies the RGB channels of an image, brightening or darkening it. **NO Loops!**

```
function [ outImg ] = makeBright_NL( inImg, brightness )
```

This function will achieve the same thing as `makeBright_L`, but without the use of loops.

- Add menu buttons for the `makeBright_L` and `makeBright_NL` functionality. Choosing one of these two menu items should result in:

a. Asking the user to input a value between -255 and 255 for the `brightness` parameter, followed by a call to the *makeBright_L / NL* function, passing the current image and the value of `brightness` as inputs.
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Save the resulting image.

Task 3 (0.25 points) Invert: inverts the colors of an image. **Use Loops!**

```
function [ outImg ] = invert_L( inImg)
```

Color inversion, also known as the negative effect, is one of the easiest effects to achieve in image processing. Subtract each RGB color value from the maximum possible value (**255**) and the result will be an inverted image. You must use loops to modify each individual pixel values.

Choosing the *Invert_L* menu button should result in:
a. Calling the `invert_L` function with the current image as an input
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.

Task 4 (0.5 points) Invert: inverts the colors of an image. **NO Loops!**
`function [ outImg ] = invert_NL( inImg)`
This function will achieve the same thing as `invert_L`, without the use of loops.

Choosing the *Invert_NL* menu button should result in:
a. Calling the `invert_NL` function with the current image as an input
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.

Task 5 (0.5 points) AddRandomNoise: adds random noise to an image. **No Loops!**
`function [ outImg ] = addRandomNoise_L( inImg)`
The input image `inImg` has pixel values between 0 and 255 in all three channels R,G and B. This function adds random noise to each pixel and writes out the new image to `outImg`. The random noise added should be different for every pixel and it should be in the range [-255, 255].

Choosing the *AddRandomNoise_L* menu button should result in:
a. Calling the `addRandomNoise_L` function, with the current image as input.
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.

Task 6 (0.5 points) Luminance: change a color image into a luminance (gray scale) image. **No Loops!**
`function [ outImg ] = luminance_L( inImg)`
This method transforms a color image into a gray image and writes out the new image to `outImg`. **Note:** here, the variable `outImg` will be a 2-dimensional matrix. You must use loops to modify each individual pixel values for `outImg`.

In order to convert a particular color into gray scale we need to figure out what the intensity or brightness of that color is. A quick way is to calculate the mean value of the red, green and blue channel components:

$$I = \frac{R + G + B}{3}$$

Although it can produce reasonable results, this method is not perfect. This is because the human eye does not perceive reds, greens and blues at the same intensity level. The color green, for example, at maximum intensity looks brighter than blue at maximum intensity. For this assignment we will use the following weighting system:

$$I = 0.299R + 0.587G + 0.114B$$

Given this image:

Converting to gray scale using the mean method:

… and using the weighted method:

Choosing the *Luminance_L* menu button should result in:
a. Calling the `luminance_L` function, with the current image as input.
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.

Task 7 (2 points) A *red filter* will make red things lighter and complementary colors darker. That's why a red filter, on black and white film, makes clouds stand out — it doesn't do a thing to the white clouds but it makes the blue sky darker, so the clouds stand out against the darker sky tones.

```
function [ outImg] = redFilter( inImg , redVal)
```

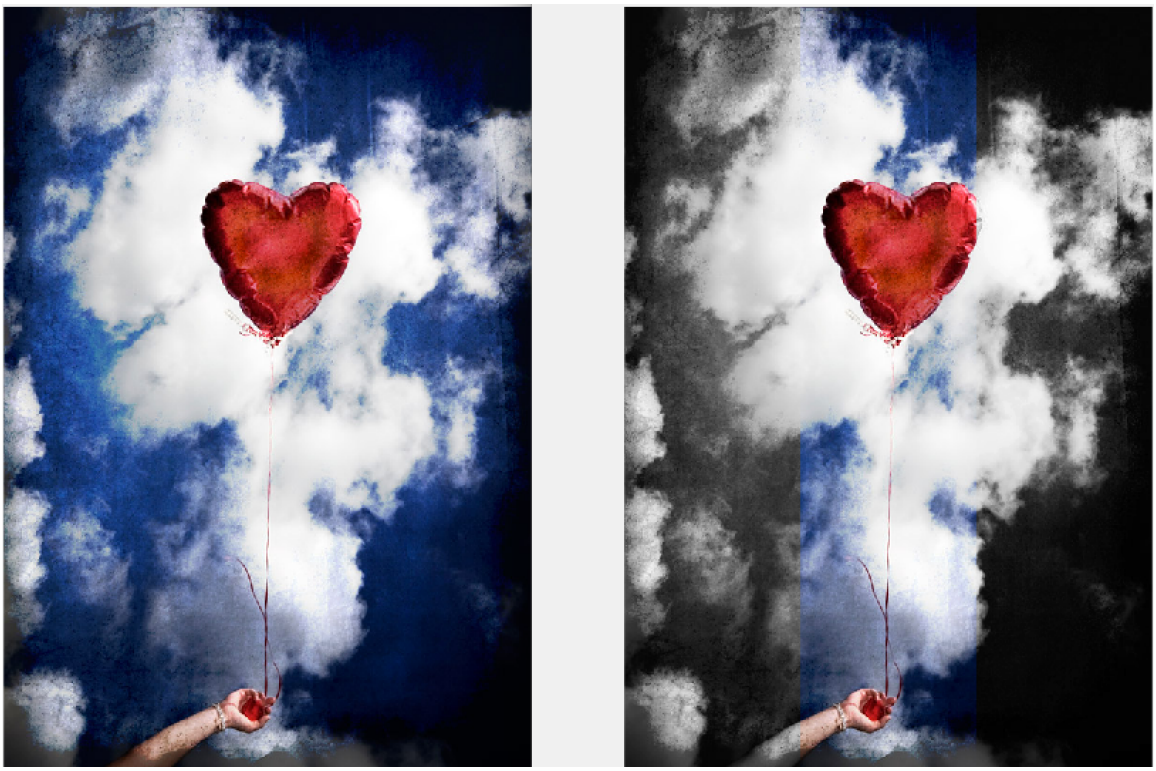
The `outImg` matrix has the same size as the input image matrix `inImg`. The middle 1/3 of the `outImg` image matrix is identical to the original, the left 1/3 is the gray scale version of the original image, and the right 1/3 has a *red filter* applied.

Just as the description above implies, a red filter will make the red components brighter and the blue and green components darker. The variable `redVal` can have any value between 0 and 1. Here is how the `redVal` value is used in applying the red filter. First, remember the weighting we used for each of the three channels in the *luminance* function:

$$I = 0.299R + 0.587G + 0.114B$$

When applying the *red filter*, the result is a gray scale image, where the weight value for the Red component will be `redVal`, and the Green and Blue components will equally "share" the remaining weight. The sum of the three weights for Red, Green and Blue should be equal to 1.

Choosing the *Red Filter* menu button should result in:
a. Asking the user to input a value for `redVal` between 0 and 1, followed by calling the `redFilter` function, passing the current image and the `redVal` value as inputs.
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.

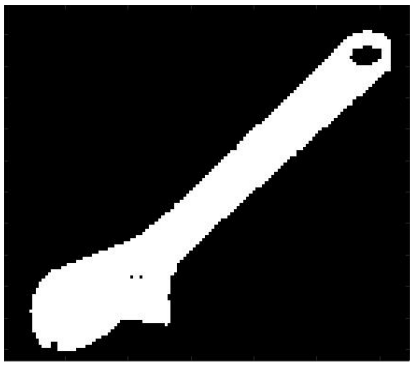


Task 8 (2.5 points) Binary image: similar to the blue-screening technique on TV and in movies

```
function [ outImg] = binaryMask(inImg)
```

Create one binary image `outImg` (also known as *the mask*), in which the pixels which belong to the "object of interest" will be white (1), and the background will be black (0). The `binaryMask` function takes a grayscale image as an input. You can call one of the *luminance* functions if the current image is a color one (*sully.bmp* is a good choice), or you can use the *wrench1.jpg* image file. Try to come up with an algorithm that finds the threshold value that separates the background from the "object of interest". (if they are part of the wrench, in this example).

Choosing the *Binary Image* menu button should result in:
a. Calling the `binaryMask` function, with the current image as input.
b. Displaying the original image and the resulting image, side by side (use subplots)
c. Saving the resulting image.



Task 9 (5 points) Mean Filter: also known as smoothing, averaging or box filter

```
function [ outImg ] = meanFilter( inImg, kernel_size )
```

Mean filtering is a method of smoothing images, *i.e.* reducing the amount of intensity variation between one pixel and the next. It is often used to reduce noise in images. The idea of mean filtering is simply to replace each pixel value in an image with the mean (average) value of its neighbors, including itself. This has the effect of eliminating pixel values that are unrepresentative of their surroundings.
The input argument `kernel_size` will determine the size of the smoothing kernel. For example, if `kernel_size` is 3, the smoothing kernel is of 3 x 3 size like in the picture below.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Note: Pay close attention to the pixels on the edge (first row, last row, first column, last column). How many neighboring pixels do they have? Use selection statements to address these special cases or pad the image with extra rows and columns (details in lecture, remember the NaN value!).

Choosing the *Mean Filter* menu button should result in:
a. Ask the user to input the size of the smoothing kernel (a positive integer)
b. Call the `meanFilter` function, with the current image as input, plus the value entered by the user for the size of the smoothing kernel.
c. Display the original image and the image returned by the function, side by side (use subplots)
d. Save the resulting image. Use a naming convention of your choice.

Task 10 (3 points) Frosty Filter
`function [ outImg ] = frosty( inImg, n, m )`
This method applies a filter to the image similar to the frosted glass effect.

Simply replace each pixel value in the image with a random value from one of its neighbors, including self, in an `n` by `m` window.

Choosing the *Frosty Filter* menu button should result in:
a. Ask the user to input a positive value for `n`, within the bounds of the image size
b. Ask the user to input a positive value for `m`, within the bounds of the image size
b. Call the `frosty` function, with the current image as input, plus the values entered by the user for `n and m`.
c. Display the original image and the image returned by the function, side by side (use subplots)
d. Save the resulting image. Use a naming convention of your choice.

Note: You might want to implement two helper functions useful for tasks 11-13:
`function [ value ] = sampleNearest( x, y)`
This method returns the value of the image, sampled at position (x,y) using nearest-point sampling. https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

`function [ value ] = sampleBilinear( x, y)`
This method returns the value of the image, sampled at position (x,y) using bilinear-weighted sampling. https://en.wikipedia.org/wiki/Bilinear_interpolation

Task 11 (5 points) Scale Nearest
`function [ outImg ] = scaleNearest( inImg, factor )`
This method scales an image using **nearest point sampling** to obtain pixel values and returns the new image.

The value of the input parameter `factor` should be positive and it represents the factor by which the height and width of the image are to be scaled. For example, if the value of `factor` is 2, the width of the new image should be twice the width of the original image; same with the height. If the value of `factor` is less than 1, for

example 0.3, then the new width will be 0.3 * *the_value_of_the_original_width*. If effect, a value of `factor` less than 1 will result in a smaller image than the original.

Choosing the *Scale Nearest* menu button should result in:
a. Ask the user to input a positive value for `factor`
b. Call the `scaleNearest` function, with the current image as input, plus the value entered by the user for `factor`.
c. Display the original image and the image returned by the function, side by side (use subplots)
d. Save the resulting image. Use a naming convention of your choice.

Task 12 (7 points) Scale Bilinear

```
function [ outImg ] = scaleBilinear( inImg, factor )
```

This method scales an image using bilinear-interpolation to obtain pixel values and returns the new image. The value of the input parameter `factor` should be positive and it represents the factor by which the height and width of the image are to be scaled. (see more examples at Task 3)

Choosing the *Scale Bilinear* menu button should result in:
a. Ask the user to input a positive value for `factor`
b. Call the `scaleBilinear` function, with the current image as input, plus the value entered by the user for `factor`.
c. Display the original image and the image returned by the function, side by side (use subplots)
d. Save the resulting image. Use a naming convention of your choice.

Task 13 (8 points) Swirl Filter

```
function [ outImg ] = swirlFilter( inImg, factor, ox, oy)
```

This method applies a *swirl filter* to the current image and returns the new image. The swirl filter is a warp or a distortion. In the distorted image, the pixel from location (r,c) in the original image is rotated an angle *theta* with respect to the origin coordinates `ox & oy`, and it will end up at a new coordinate pair (x,y). Pixels closer to the origin will rotate less, while pixles further from the origin will rotate more. The input parameter `factor` determines the magnitude of the rotation, and the direction. A positive value of `factor` will create a clockwise swirl, while a negative value of `factor` should create a counter-clockwise swirl. You task is to inverse map each (x,y) coordinate pair in the final image to the original (r,c) value. You can use a non-linear mapping of your choice (bilinear, nearest-neighbor).

Choosing the *Fun Filter* menu button should result in:
a. Ask the user to enter values for `factor, ox, oy.` Make sure each of them are within the allowed range of values.

b. Call the `funFilter` function, with the current image and the additional 3 parameters as input.
b. Displaying the original image and the image returned by the function, side by side (use subplots).
c. Save the resulting image. Use a naming convention of your choice.

Task 14 (10 points) *Famous Me*
Create a composite image of yourself and a famous tourist attraction. Example: "crop and paste" yourself on the moon. For this task you will need to use two images: the original image (of yourself, ideally in front of a non-busy background, either very light in color of very dark) and the target image (of the moon, for example). For the "crop" part, you will need to create a binary mask image using the image of yourself. Then, use the binary image and select the location in pixel coordinates where you want to "paste" yourself. If you want to resize the picture of yourself (or the mask) with respect to the target, you can use one of the functions you already developed for scaling.

Choosing the *Famous Me* menu button should result in:
a. Call the `famousMe` function. **Note:** You get to decide how you want to implement this task, how many inputs you want your function to have and if it will you use other already developed functions (binary mask, for example).
b. Display the 3 images side by side: original image, image of you and the resulting image (use subplots)
c. Save the resulting image. Use a naming convention of your choice.



Submitting the assignment:
Make sure each script or function file is well commented and it includes a block comment with your name, course number, assignment number and instructor name. Save one resulting image for each of the buttons/functionality implemented. Zip all the .m files and the image files together and submit the resulting *.zip* file through Canvas under assignment H1, by Tuesday, January 26st, by 9:55pm