CSCI 4830/5722 Computer Vision, Spring 2018
Instructor: Fleming
Homework 1, Due Wednesday, January 31st, by 11:55pm

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 (10 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 2 (15 points) Gaussian Filter: smoothing filter, also known as Gaussian blur
`function [ outImg ] = gaussFilter( inImg, sigma )`
The Gaussian filter works in a similar fashion to the mean filter. The values of the weights in the Gaussian smoothing kernel will be different (as opposed to the mean filter where the weights were the same). They follow the Gaussian distribution:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

, where *x* is the distance from the origin in the horizontal axis, *y* is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. The size of the Gaussian kernel will be computed based on the value of `sigma` as follows:

```
2 * ceil ( 2 * sigma) + 1
```

Note: the origin is always at the (x,y) coordinates of the pixel you are trying to replace with a new, smoothed value. Since the size of the kernel will always be an odd number, for the calculation of the Gaussian weights the pixel in the middle of the kernel will be the (0,0) origin, and therefore have the highest weight.

Choosing the *Gaussian Filter* menu button should result in:
a. Ask the user to input a positive value for `sigma`
b. Call the `gaussFilter` function, with the current image as input, plus the value entered by the user for `sigma`.
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 3 (10 points) Frosty Filter
`function [ outImg ] = frosty( inImg, n, m )`
This method applies a filter to the image similar to the frosted glass effect.

Simply to replace each pixel value in the image with a random value from one of its neighbors, including itself, 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`
b. Ask the user to input a positive value for `m`
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.

Task 4 (15 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 5 (15 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 6 (15 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.

Note: You might want to implement two helper functions useful for tasks 3-5:

```
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 7 (20 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 (also see creating the binary mask below).

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.

Helpful function for Task 7: 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*), which will represent the boundaries of the object of interest in the input image `inImg`. A binary image has only values of 0 and 1, so only black and white pixels, no grey shades (this means `outImg` is a 2-dimensional matrix).

Your algorithm should be able to figure out the threshold value that separates the background from the "object of interest" in the image. It might be easier to turn the image into a gray scale image first. Use the following weighting system:

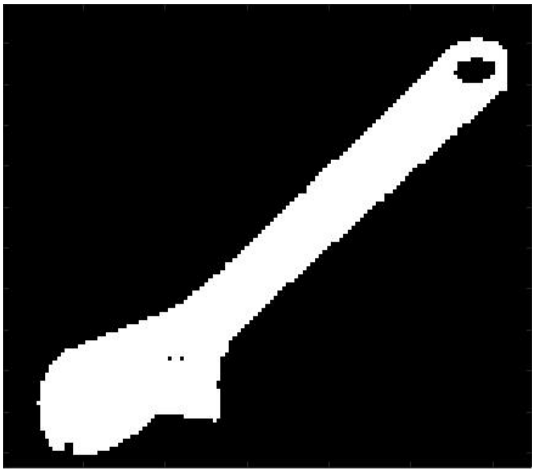
$$I = 0.299R + 0.587G + 0.114B$$

Given this image:

Converting to gray scale using the mean method:

… and using the weighted method:

Please note that the pixels in the binary image will have the value of 0 if they belong to the background, and 1 otherwise.



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 Moodle as Homework 1, by Wednesday, January 31st, by 11:55pm.