Clab-1 Report

ENGN4528

Benjamin Hofmann

u6352049

13/04/2020

ENGN4528 Computer Vision
Computer Lab-1

Task-1: Python Warm-up

1. **Describe the result/function of the following commands**

1) a = np.array([[2,4,5],[5,2,200]])

The function produces the following 2x3 matrix:

$$\begin{bmatrix} 2 & 5 \\ 4 & 2 \\ 5 & 200 \end{bmatrix}$$

Equation 1: output matrix of task 1 part 1

2) b = a[0,:]

The function indexes for the first (0^{th}) column of matrix a. The result of this is the array (or 3x1 matrix): [2,4,5].

3) f = np.random.randn(500,1)

The function produces a 500 column, 1 row matrix where the values follow a pseudo random gaussian distribution with mean 0 and standard deviation 1.

4) g=f[f<0]

All negative values in array f are stored in array g.

5) x = np.zeros(100)+0.35

This command generates an array with 100 0.35-value entries.

6) y=0.6*np.ones([1,len(x)])

This function produces a 1 column, 100 row matrix with values 0.6. The function does this by calculating the number of values in array x to determine the number of rows.

7) z = x-y

The result of z is a 1 column by 100 row matrix of values -0.25. This function takes the difference between each individual value of the x and y arrays.

8) a = np.linspace(1,200)

The function produces an array from 1 to 200 inclusively with each value increasing by a consistent amount. There are 50 entries in the array.

9) b =a[::-1]

The result of this function is the array a being reversed. Thus the last value (200) is the first value of array b.

10) b[b<=50]=0

All values less than (or equal to) 50 are replaced with the value 0.

Task-2: Basic Coding Practise

1. **Load grayscale image and map to its negative image**

Figures 1 & 2: Grayscale image (left) and negative image (right) used in task 2

The negative image was created through subtracting 255 by the grayscale values. Thus, darker regions in figure 1 appear lighter in figure 2 and lighter regions also appear darker.

2. **Flip the image vertically**

Figure 3: Vertically flipped image

The effect of figure 3 was achieved using the technique explained in Part 1.9 of reversing the order of rows. Hence the image appears vertically flipped.

3. **Load colour image and swap red and blue channels of the input of the image**

Figure 4 & 5: Original Colour image and Colour image with red and blue channels switched

The effect achieved in figure 5 was achieved by first reading the colour image in figure 4 and then assigning the red and blue channels to different channels in a corresponding image. The effect maintains the details of the image as the woman is still clearly identifiable, but the image's colour scheme is altered to appear more pink.

4. **Average image with its vertically flipped image**

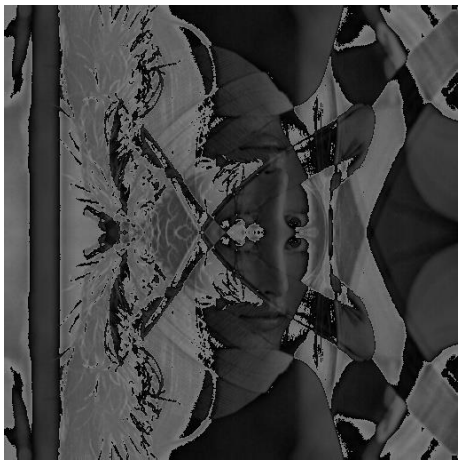

Figure 6: Average between original and vertically flipped images

Figure 6 shows the image where pixel values are the average between the original grayscale and the vertically flipped image (figure 3). This image is mirrored on the x-axis around the centre.

5. **Add random value between 0 and 255 to grayscale image**

Figure 7: Grayscale image with random noise added

Figure 7 shows the grayscale image with pseudo random values being added to each pixel. This was achieved using the inbuilt function np.random.randint to generate a matrix of random values between 0 and 255. This matrix was then added to the originl grayscale image (figure 1). As each random value is greater than or equal to 0, the image appears much lighter as pixel values are increasing. There are more areas of pure white (pixel value 255) as the added noise resulted in a value of 255 or greater.

This effect makes the image slightly more unclear as well as adjusting its apparent brightness.

Task-3:

1. **See images in image folder**
2. **See images in image folder**
3. **Program a short computer code that does the following task**

A) Original image read in and resized

Face #2 was selected as the image was of adequate brightness to clearly see each grayscale image. The image was resized using the inbuilt function cv2.resize().

Figure 8: Original colour image used in task 3

B) Convert image into three grayscale channels and display each separately



Figures 9 & 10 & 11: Visual representation of each input channel; Red, Green and Blue respectively

Figures 9, 10 and 11 show the output of each channel as represented as a grayscale image. Figure 9 appears the darkest suggesting that the red channel typically has lower pixel values than the others. The blue channel (figure 11) is the brightest suggesting greater pixel values overall.

C) Compute the histograms for each channel

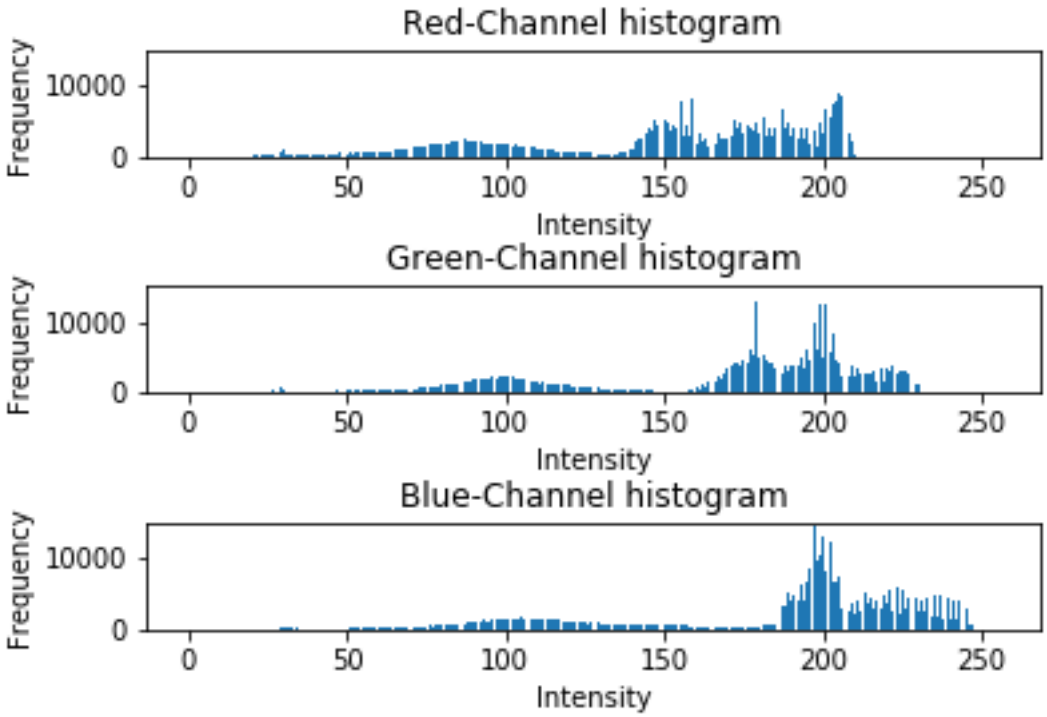

Figure 12: The histogram representation of pixel values and their respective frequencies for each input channel

Figure 12 shows the histogram of each channel of the colour image. The channel histograms share similar behaviours, each has a small peak around 100 and a range of about 50 units with a greater frequency. The red channel has its greatest intensities from about 150-210, the green channel from 160-230 and blue channel from 180-240. Notably the red and green channels have very few values above 220 and 230 respectively.

D) **Apply histogram equalisation and display the resultant histograms**

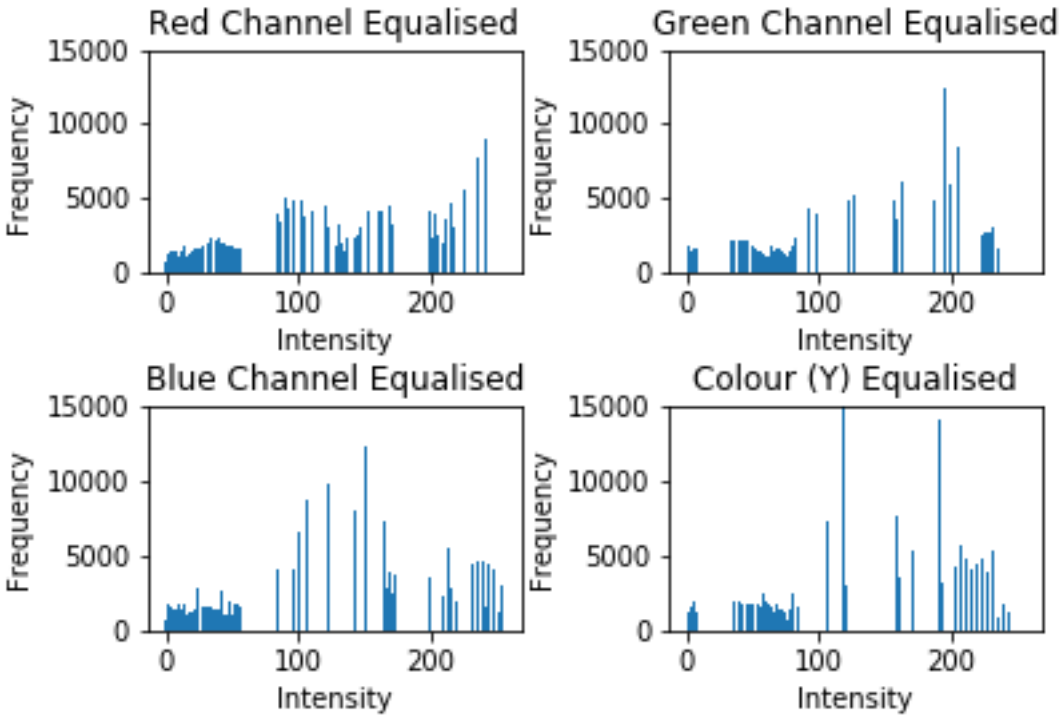

Figure 13: The histogram representation of pixel values and their respective frequencies for the equalised input of each channel and the equalised colour image

Equalisation of the channel inputs was conducted using the inbuilt function cv2.equaliseHist(). It can be seen that equalisation appears to have segmented the histograms in figure 13. Values are spread across a greater range with more values at both the upper and lower end of the intensity range. However, large ‘gaps’ between areas with greater frequency are apparent. These are likely caused from the equalisation process ‘stretching’ the histogram.

The colour image was equalise using a different method to the channel histograms. The image was first converted to YUV representation using cv2.COLOR_BGR2YUV. The Y channel represents the ‘brightness’ of the image [7] and this channel underwent the histogram equalisation process. The image was then converted back to RGB and displayed as figure 14.

Figure 14: Colour image after histogram equalisation

This technique was selected as combining each of the equalised input channels resulted in colour distortion.

Task-4:

1. **Read in face, crop a square image region, resize to 256x256 and save as a grayscale image**

The image was cropped using index notation with the 100^{th} to 356^{th} columns being selected and the 400^{th} to 656^{th} rows also being kept. If the image required resizing then the cv2.resize() function would have been used. The image was converted to grayscale using a custom function color2gray(). This function converted to grayscale by weighting each channel by 0.2989, 0.5870 and 0.114 respectively. These values were selected as they are the ones used in MATLAB's inbuilt rgb2gray() function [4]. This resulted in figure 15 below, which was used as the original image for the remain parts of task 4.

Figure 15: Cropped and gray-scaled image use in task 4

2. **Add gaussian noise to grayscale image. Use noise with 0 means and standard deviation 15**

Gaussian noise was generated using the randn() function demonstrated in Part1.3 and multiplied by 15 to achieve standard deviation of that value. The random noise is then added to the grayscale image with values outside the [0,255] range clipped to produce figure 16.

Figure 16: Original image with gaussian noise added

The noise is figure 16 gives the image a more speckled or grainy appearance. The features of the image are still relative distinct despite the addition of noise.

3. Display histograms of image with and without noise

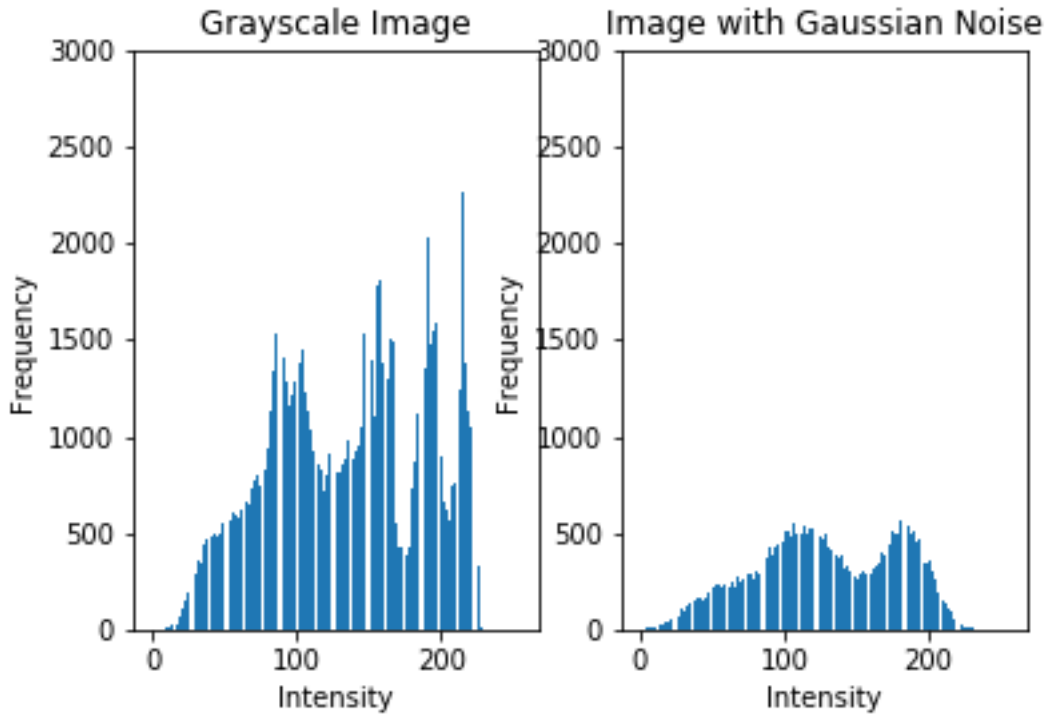

Figure 17: Histogram of cropped grayscale image before (left) and after gaussian noise is added (right).

The comparison of histograms in figure 17 highlights the impact of the addition of noise to an image. The histogram of the original grayscale image shows a number of individual peaks and relatively high disparity in frequency between local intensities. The histogram with noise included however shows a much smoother histogram with relatively low changes in frequencies between local intensities. Rather than display individual peaks there are two noticeable groupings of greater frequencies around 110 and 190. The maximum frequency of any individual intensity is much lower in the image with noise as the random value addition redistributes the values more evenly.

4. Implement your own function that performs a 5x5 gaussian filtering

The function my_Gauss_filter() accepts a grayscale image and a kernel as inputs. The function processes the image and kernel sizes and establishes the dimensions of the output. The function also calculates the padding required and generates this padding around the image. The function uses two imbedded for loops to process each individual pixel by multiplying the kernel by the padded image. The value is divided by the sum of the kernel as the kernel used is not necessarily set to appropriate scale. After the for loops the borders are clamped and output it returned.

The gaussian kernel was calculated using an online resource [3]. A kernel with sigma 1 was selected so to not add too much blur whilst filtering some of the noise.

5. **Apply gaussian filter to noisy image and display output. Compare results with python's inbuilt cv2.gaussianblur() function**

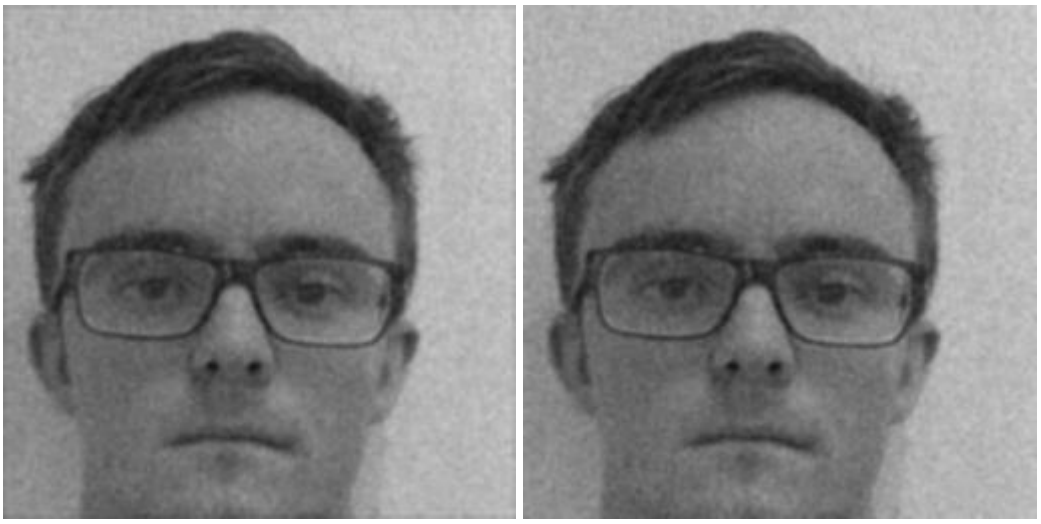

Figure 18 & 19: Filtered image with my_Gauss_filter() (left) and image with python's cv2.Gaussianblur() (right)

The images in figures 18 and 19 are remarkably similar, each still contains some sign of noise but these pixels are more blurred. Evidence of noise could be further reduced by selecting greater values of sigma, however a low sigma was selected for simplicity and to allow for analysis on the blur of the noise itself. The blur appears to be consistent between both images. The image in each is still recognisable and relatively distinct despite the features of each being softened. One subtle difference is that the image using the my_Gauss_filter() appears slightly lighter than that of the inbuilt function. This could be due to error in the kernel introduced by scaling and then rounding values before dividing by the resultant sum.

The default borders of the inbuilt function are reflections, meaning that the outer 2 columns and rows are same as the outer 2 columns and rows of the image that has been processed. For a 5x5 kernel the effect is very similar to clamping.

Task-5:

1. **Implement your own 3x3 Sobel filter**

For this task the function my_Sobel_filter() was developed. This accepts the image as input and produces the result of horizontal, vertical and combined filtering. The kernels for horizontal and vertical filtering are stored within the function. The function is very similar to my_Gauss_filter() described in part 4. A layer of padding is added to the image before two embedded for loops evaluate the result for each pixel. This process is done for both the horizontal and vertical filters in the same set of loops. Clamped borders are then applied to the resultant image and both the horizontally and vertically filtered images are returned.

2. **Test Sobel filter on grayscale image and compare to inbuilt filter functions**

Figures 20, 21 show the comparison in outputs between the horizontal Sobel filter developed for this report and python's inbuilt horizontal Sobel filter. Key features such as the hairline, glasses, nose and mouth are clearly visible in each image. There is some speckling from the jacket in each image, although this is much clearer in the inbuilt filter as it appears much

brighter. This speckling effect is caused by the jumper's colour changes being interpreted as localised edges by the filter. The outline of the body however, is difficult to make out in each. This is due to the fact that it is primarily a vertical difference and thus not picked up by the horizontal filters.

Figure 20 & 21: Image as filtered through the developed horizontal Sobel filter and the inbuilt horizontal Sobel filter respectively.

Figures 20 and 21 show the comparison between the vertical Sobel filter developed for this report and python's inbuilt vertical filter. Similarly to the horizontal filters, each vertical filter clearly identifies the same key features being; the outline of the body, glasses, strings and zip of the jumper. Again, there is some speckling in each image of the jumper and this speckling is again more noticeable in the inbuilt filter due to greater brightness.

Figure 22 & 23: Custom vertical Sobel and inbuilt vertical Sobel filter

The key difference between the images produced by developed filters and the inbuilt Sobel filters is the brightness of each. Close analysis shows that the same features are identified by the developed filter as the inbuilt filter. The inbuilt filter displays each edge more brightly than the developed filter and thus the edges are more distinguishable. This could be corrected through increasing the intensity of brighter pixels to greater highlight edges.

The Sobel filters (equation 2) are used in edge detection as they are capable of identifying significant changes in pixel value of an image. These sudden changes in pixel value usually correspond to an edge or change in image features.

$$horizontal\ filter = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, vertical\ filter = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

equation 2 = Sobel filter kernels for horizontal and vertical filtering [2]

Pixels with neighbouring pixels of greater difference in the relevant direction will return greater value as the difference between these neighbours is compared by the filter. If the neighbouring pixels are of similar value then the output of filtering with a Sobel kernel will have a low value. Thus, areas of greater difference in intensity will appear much more brightly than areas of low variation in intensity, therefore the filter is able to detect potential edges.

Task-6:

1. **Implement your own rotation function and display images**

The results of each rotation by the set values is shown in figure 24.

Figure 24: Top row from left: Original image (24.1), 45-degree rotation (24.2), 90-degree rotation (24.3). Bottom row from left: negative 15-degree rotation (24.4), negative 45-degree rotation (24.5), negative 90-degree rotation (24.6)

The custom function my_rotation() operates through the following process

- Accept inputs of the image and desired rotational angle in degrees

- Take shape of image
- Convert angle to radians
- Calculate appropriate sin and cos values
- Calculate 2 translation kernels and the rotation kernel and combine these
- Apply kernel to each pixel.
- Return rotated image of equal shape to inputted image.

Output pixels not corresponding to a rotational value are left as value 0.

The kernels generated by the function are used to perform a scaled rotation function [2]. The first translation kernel takes the scale of the image, the rotational kernel then performs 2D Euclidean transformation before the final translation process rescales the image to fit into the image size. These kernels are multiplied together to achieve the desired outcome and the inverse is taken to apply backwards mapping technique.

An undesirable feature of this rotation is that a small amount is often sliced off at the corners. This effect is extremely subtle but noticeably if zooming in. This slicing may remove several pixels from the image at each corner. This is possibly due to an offset created in the translation kernels due to rounding. A possible example of this can be best seen in the -90-degree image (figure 24.6) which has a single row of 0 values at the top. Analysing the -45-degree rotation (figure 24.5), the top corner is not cropped but the other three are slightly cropped. This suggests that these images have been shifted at least 1 pixel down.

2. Compare forward and backward mapping

Forward mapping (or forward warping) is where a pixel is copied to its corresponding location in the destination image [2]. There are a number of issues with this approach. Firstly, if a destination location is not well defined then this can lead to several problems. Rounding to integer coordinates can result in aliasing while splatting (distributing the value across nearby pixels) suffers both aliasing and creates some blur. Secondly, forward warping can lead to the appearance of cracks or holes where there are no values corresponding to some pixels in the destination. Fixing this by mapping neighbouring pixels to fill the gaps will then lead to more blur and aliasing.

In contrast, backwards mapping (or inverse warping) is where the destination image's pixel is sampled from the original [2].

The two processes are similar in that the function for backwards mapping can often be the inverse of the function for forward mapping. A significant difference between forwards and backwards mapping is that backwards mapping leads to no holes as the mapping function is specifically defined for all destination pixels. Thus, it is able avoid some of the aliasing and blur introduced by forward mapping. Backwards mapping also allows for the control of aliasing through the use of filters when resampling.

3. Compare different interpolation methods

The interpolation methods would be used were pixel destination values do not directly correspond to an original pixel position.

Nearest neighbour: This interpolation approach takes the value of the nearest pixel to the desired pixel of the original image [1]. The advantage is that this interpolation is the fastest and doesn't introduce any values that are not already present in the image. The downside is that it can reduce fidelity and lead to information being lost.

Linear: This approach takes the weighted average between the closest 2 pixels to the desired pixel [5]. This approach is also relatively fast and is able to maintain some information in scaling however it is still more limited than other techniques. This can potentially introduce values not found in the original image.

Bilinear: This method takes the 2x2 neighbourhood surrounding the desired pixel. The approach performs linear interpolation in each direction to obtain the sample value [6]. This method produces better interpolation results than linear interpolation but requires more calculation time.

Bicubic: Similar to the bilinear interpolation, this method takes the weighted sum of the 4x4 neighbourhood surrounding the desired pixel [6]. This again provides more developed interpolation than bilinear but that the cost of increased computational time.

Windowed Sinc: This method smoothly interpolates values between samples. This uses a normalised sinc function multiplied by a sinc window (a horizontally stretched sinc function) as the kernel [6]. This is the most accurate interpolation method but also the most computationally expensive. This method would be preferred if processing time is not a severe constraint.

Resources:

[1]D. Hooker, *Lenna*. 1972.

[2]R. SZELISKI, COMPUTER VISION: Algorithms and Applications. [S.l.]: SPRINGER NATURE, 2010.

[3]"Gaussian Kernel Calculator | The Devil In The Details", Dev.theomader.com, 2020. [Online]. Available: http://dev.theomader.com/gaussian-kernel-calculator/. [Accessed: 02- Apr- 2020].

[4]"Convert RGB image or colormap to grayscale - MATLAB rgb2gray- MathWorks Australia", Au.mathworks.com, 2020. [Online]. Available: https://au.mathworks.com/help/matlab/ref/rgb2gray.html#buiz8mj-1-RGB. [Accessed: 3- Apr- 2020].

[5]k. atul, "Image Processing – Bilinear Interpolation", TheAILearner, 2020. [Online]. Available: https://theailearner.com/2018/12/29/image-processing-bilinear-interpolation/. [Accessed: 3- Apr- 2020].

[6]"Tutorial: Image Rescaling", Clouard.users.greyc.fr, 2020. [Online]. Available: https://clouard.users.greyc.fr/Pantheon/experiments/rescaling/index-en.html#nearest. [Accessed: 3- Apr- 2020].

[7]O. image, F. Abdo, M. Jazaery and S. Chan, "OpenCV Python equalizeHist colored image", Stack Overflow, 2020. [Online]. Available: https://stackoverflow.com/questions/31998428/opencv-python-equalizehist-colored-image. [Accessed: 3- Apr- 2020].

Other resourced used in informing code:

[8]"Image Gradients — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.io, 2020. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_gradients/py_gradients.html. [Accessed: 02- Apr- 2020].

[9]"matplotlib.pyplot.subplot — Matplotlib 3.2.1 documentation", Matplotlib.org, 2020. [Online]. Available: https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.subplot.html. [Accessed: 3- Apr- 2020].

[10]"Python Random randint() Method", W3schools.com, 2020. [Online]. Available: https://www.w3schools.com/python/ref_random_randint.asp. [Accessed: 3- Apr- 2020].