CENG466 - Fundamentals of Image Processing Report 1

Bengisu Ayan
Computer Engineering
Middle East Technical University
Ankara, Turkey
e223697@metu.edu.tr

Ceren Gürsoy
Computer Engineering
Middle East Technical University
Ankara, Turkey
e2237485@ceng.metu.edu.tr

I. Introduction

In this assignment, we have implemented two of the fundamental spatial domain image enhancement techniques: histogram processing and edge detection. We have used python3.5.2 along with numpy, cv2 and matplotlib libraries.

II. Question 1 - Histogram Processing

In this part of the assignment, we are required to apply histogram matching to two different combinations. First, we need to apply histogram matching to image A1.jpg by taking A2.jpg as reference, then we need to apply histogram matching to A2.jpg by taking A1.jpg as reference.

A. Implementation

We have implemented the following algorithm given in the Week 4 - lecture notes.

1) Equalize the histogram of the original image and obtain the transformation T_1 from the following equation:

$$s_k = T_1(r_k) = round(\frac{L-1}{NM} h_c(r_k)) \tag{1}$$

2) Equalize the histogram of the original image and obtain the transformation T_2 from the following equation:

$$s_k = T_2(r_k) = round(\frac{L-1}{NM} g_d(z_k)) \tag{2}$$

3) Obtain the colors of the desired image from

$$z_k = T^{-1}(T_1(r_k)) \tag{3}$$

for specified histogram.

We have read and saved images using $imread$ and $imwrite$ functions of $cv2$. Opencv reads and writes images as blue, green and red respectively.

As the given images A1.jpg and A2.jpg are 3-channel RGB images, we have decided to process each channel separately, then merge them together to obtain once again a 3-channel RGB image. We have used $numpy$ indexing to split the image into 3 different channels and obtain 3 different arrays for each image in $match\_histograms$ function. (It is said to be more efficient than $split$ function of opencv.)

First, we have implemented a $get\_histogram$ function that takes a flattened array (by the $flatten$ function of $numpy$) as input and returns an array of size 256 that contains the number of pixels for each intensity value in the image. We had initially planned to use the built-in function $histogram$ and $cumsum$ of $numpy$. However, we had trouble manipulating these functions the way we wanted as stated in the algorithm and decided to implement them ourselves. Below is the formula used in the algorithm (taken from Week 4 - lecture notes):

$$h(r_k) = h(r_k) + 1 \tag{4}$$

Secondly, we have obtained the cumulative distribution function from this histogram using the following formula (taken from Week 4 - lecture notes):

$$h_c(r_k) = h_c(r_k - 1) + h_c(r_k) \tag{5}$$

Third, we have implemented (1) and (2) in $transform$ function. We have used $numpy.floor$ function in order to round them to the integer below.

Then, we have used these transforms to create a lookup table that will be used as a transfer function for the pixel values of the original image. The process works as follows, in the $create\_lookup$ function, we iterate over every pixel value of the source and reference image, if the pixel value at the target histogram is greater or equal than the original histogram, we assign the reference pixel to original pixel value in the lookup table. We had difficulty debugging this function, which initially caused noise around different parts of the image. We then realized it yields better result when we map greater or equal pixel values of the reference histogram. Initially, we had just taken the equal values which, looking backwards left a lot of the intensity values to the original image. Finally, we map the values of the reference image to the original image using this lookup table by iterating over every pixel value in the image. In the last step, we merge the R,G,B arrays we have obtained from applying these processes separately by using the $dstack$ function of $numpy$.

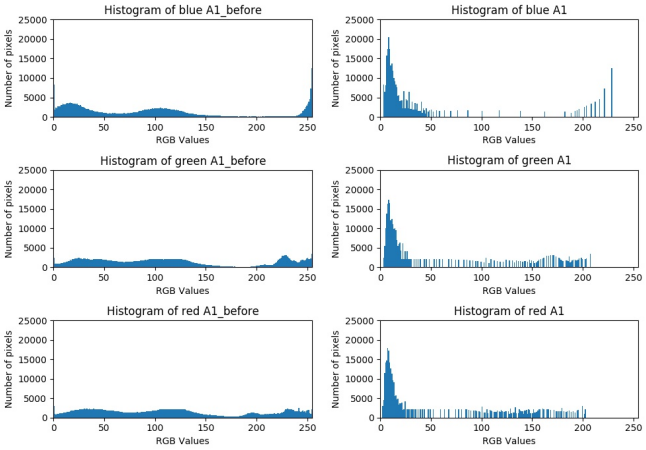
B. Results

We would like to start discussing the results by providing the initial histograms of A1 and A2. As it can be seen from Fig 1, the intensity values of colors in A1 are more balanced and scattered along 0 and 255 in A1. However, the colors are

Fig. 1: Problematic A1 histogram matched with reference image A2

more concentrated along 0 and 50 in A2, making it a darker image.

After applying histogram matching to A1 with reference image A2, we expected A1 to have a histogram with more emphasized darker values, turning the image into a darker image.In contrast, after applying histogram matching to A2 with reference image A1, we expected A2 to now have a more balanced histogram compared to before and the image to have a more balanced histogram and lighter intensity values.



(a) A1 before (b) A1 after

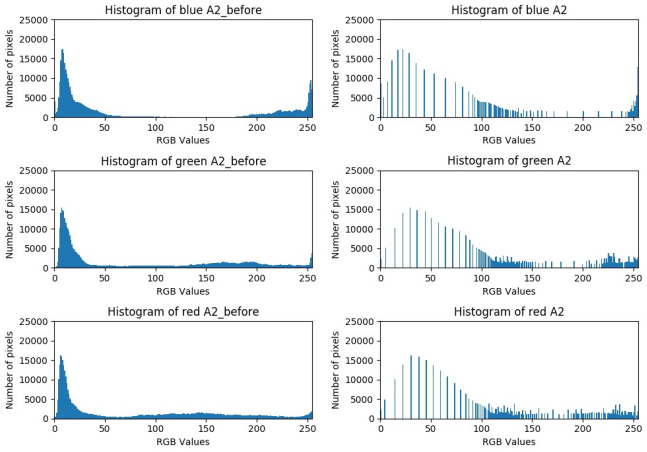
Fig. 2: Histograms of A1 before and after histogram matching

As can be seen from the histograms after the histogram matching process, the results mirrored our initial expectations and the histogram of the first image now resembles the second and vice versa. We have also checked our results by comparing them to built-in function *match_histograms* of *skimage*, which produced identical images. You can see the results of applying histogram matching on A1 with reference image A2 and applying histogram matching on A2 with reference image A1 in Figure 4

III. QUESTION 2 - EDGE DETECTION

In this section, we have implemented Canny edge detection algorithm given in the Week 5 - lecture notes with the following steps:

1) It is asked whether blurring is wanted or not as a parameter. If it is, the image is smoothed by convolution function with gaussian kernel. Both convolution and gaussian kernel are explained in Section A and B respectively.
2) Sobel filter explained in Section C is implemented to the image which is blurred or not (according to Step 1), and edge map and direction map of the image are received by Sobel filtering algorithm.
3) According to edge map and edge direction, the image is suppressed by non-maximal suppression function explained in Section D in order to remove the weak edges.
4) According to low and high thresholds parameters, false edge points in the image are reduced by double thresholding method explained in Section E.
5) These steps are finalized with forming longer edges by controlling the connection of specific edges received from Step 4. This step is also explained in Section E.



(a) A2 before (b) A2 after

Fig. 3: Histograms of A2 before and after histogram matching

(a) A1 after (b) A2 after

Fig. 4: Results after histogram matching

As the given images were 3 channel images, we initially tried to separate the image into 3 channels, process them separately and merge them afterwards. However, it took too much time, making the experimentation stage inefficient, we decided to read images as grayscale images. As our aim is to detect edges, we believe it does not have a significant effect. Separating into 3 channels was an option but since we were aiming to find the edge maps, converting them to grayscale would work the same.

A. Convolution Function

In convolution equation, a filter function is slid over the image. The overlapping pixels of the image and the filter are multiplied and summed in order to receive the value of the pixel placed in the center of the filter.
According to the Week 5 - lecture notes, our convolution implementation happens as the following steps:

1) There is a filter which is in square matrix form and received as parameter.
2) The center of this filter is placed on pixels of the image by starting from the first pixel.
3) For the pixels of the image placed on edges, we have applied zero padding. In other words, a new array is created and filled with zeros with $numpy.zeros$.
4) Moreover, the new image size becomes height of the original imge plus height of kernel filter minus 1, because we have to add new edges on each side as much as half of the kernel size minus 1. For example, when the kernel is 3x3, we must add 1 row on each horizontal edges and 1 column on each vertical edges. For 5x5 kernels, 2 edges are added on each side and so on. (It is rounded to integer if the kernel size is even.)
5) The original image is placed into the padded image in the corresponding pixels. Since the first column of the original image is now in the pixel of the half of height of kernel minus one and the last column of the original image is far as height of original image. The same thing is for rows. Step 4 and 5 can be seen more clearly in Figure 5.

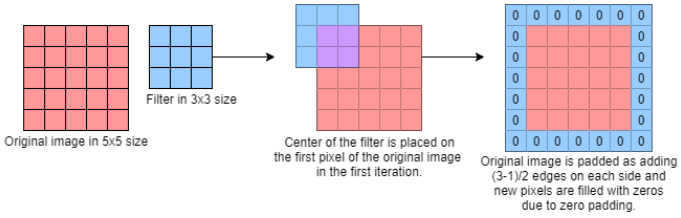


Fig. 5: Steps for creating a padded image

6) Now, filter must be flipped over the padded image pixel by pixel. Therefore, we iterate with two for loops, one of them is for iterating on columns and other one is for iterating on rows. Then, we multiply the filter and corresponding pixels on the padded image element-wise. For example, for the first step, we take pixels from 0 to kernel shape for rows and columns. Then we continue with moving to right and we take pixels from 1 to 1 plus kernel shape. It continues when we reach old pixel height, which means when we come pixels from height of the original image to height plus kernel shape. Then, it continues in the same way for the second row and so on. After the element wise multiplication, we sum all pixels on the multiplication array and assign the result into the pixel on which we placed the center of kernel. Moreover, in order to prevent bugs, we created a new array at the start of the function which has the same size with the original image. Therefore, we put the result of summation into the corresponding pixel on the new array instead of changing the value on the original image.

B. Gaussian Kernel Filter

Gaussian Kernel Filter is widely used for smoothing images. According to the Week 5 - lecture notes, the formula of 2-dimensional Gaussian function is:

$$G(x,y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[\frac{-1}{2\sigma^2}(x^2+y^2)\right] \tag{6}$$

In the code implementation of this function, we received (x, y) values from $numpy.mgrid$ function. This function gives two lists containing number distribution according to ranges that it gets in a list. Since we wanted to produce a filter that can be changed by the size, we arranged these ranges according to size. By the help of the lecture note, we started the list from minus of the half of the size of kernel and ended with the half of the size of kernel. If the size is an odd number, the half of it is rounded to integer. For example if the kernel size is 3, $numpy.mgrid$ returns two lists as:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Then, we use these two matrices in the formula (6) and get the Gaussian kernel. Finally, we normalize the kernel by dividing with the sum of values in the kernel, otherwise the blurred image becomes too bright.
Moreover, we take sigma value as well as size as a parameter. In Figure 6, we can observe the results of canny edge detection for not blurred, blurred with sigma = 2, kernel size=5 and sigma=2, kernel size=15 images. If the image is not blurred, unnecessary edges due to noises are also detected. Similarly, as we increase the kernel size, the image becomes more blurred, which results in fewer details and edges in the resulting images. This was very visible in B1 and B2, however it resulted in few changes visible to the human eye in B3.

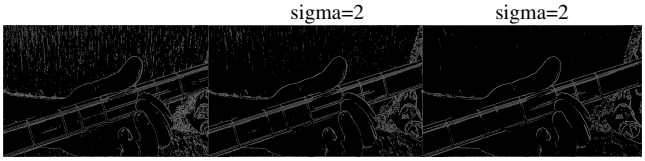
In Figure 7, we can observe the results of canny edge detection for images blurred with different sigma values in Gaussian Filter. As we learned in lectures, low sigma values have less uniform Gaussian filters (the values closer to center of the matrix are bigger), making the filter take the actual pixel value more into account, resulting in less blurred images. As we increase the sigma value, the image gets more blurred, thus the canny edge detection algorithm shows more important edges, instead of very detailed edge maps.

As we can see from the figures, when sigma values and/or the kernel size increases, the image becomes more blurred.

C. Sobel Filter

In this implementation, in order to detect edges, the vertical and horizontal filters are applied to the image, which are:

(a) not blurred (b) kernel size=5 and sigma=2 (c) kernel size=15 and sigma=2



(d) not blurred (e) kernel size=5 and sigma=2 (f) kernel size=15 and sigma=2

Fig. 6: Canny Edge Detection Results of B1 and B2 when the kernel size of Gaussian Filter changes

(a) blurred with sigma = 0.5 kernel size = 9 (b) blurred with sigma = 2 kernel size = 9 (c) blurred with sigma = 5 kernel size = 9

(d) blurred with sigma = 0.5 kernel size = 9 (e) blurred with sigma = 2 kernel size = 9 (f) blurred with sigma = 5 kernel size = 9

Fig. 7: Canny Edge Detection Results of B1 and B2 when the sigma values of Gaussian Filter changes

$$W_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } W_h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

We define them as numpy arrays and convolve the image with each of them via convolution function explained in Section A. From the result of the convolutions for vertical and horizontal filters, we receive edge maps, $g_h(x, y)$ and $g_v(x, y)$ and we calculated the gradient magnitude as taking squares of vertical and horizontal edge maps and square root of their summation according to the following formula from the Week 5 - lecture notes:

$$|g(x, y)| = \sqrt{g_h^2(x, y) + g_v^2(x, y)} \tag{7}$$

However, from this formula, we received an edge map having values more than 255. Therefore, we have to normalize the array according to Formula (8) for range $[a, b]$.

$$normalization = a + (b - a) * \frac{x - min}{max - min} \tag{8}$$

, where min and max are the minimum and maximum values of x respectively.
An image is normalized by shifting the physical brightness values, as follows, according to the Week 3 - lecture notes:
$L_{min} \longrightarrow 0$
$L_{max} \longrightarrow L - 1$
where L is 256.
Therefore, we implemented Formula (8) as a is 0 and b is 255 and normalized the gradient magnitude array.

Finally, we calculated direction map by taking arctan of vertical and horizontal edge maps according to the formula:

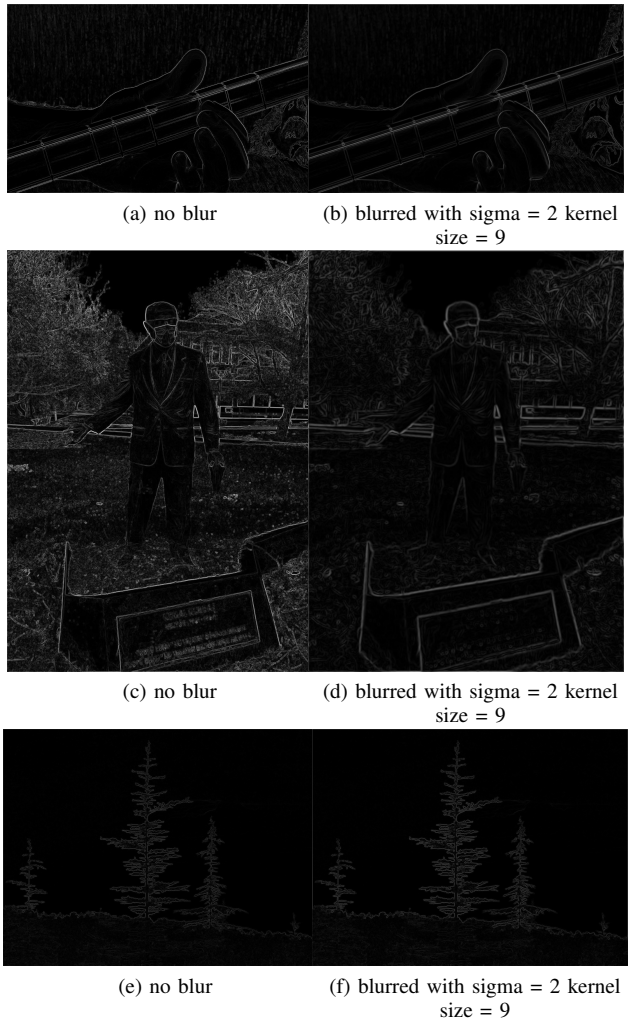
$$\theta(x, y) = \arctan \frac{g_v(x, y)}{g_h(x, y)} \tag{9}$$

Moreover, since we asked whether image would be blurred or not in the main function, if the answer is yes, we apply Sobel filters on smoothed image, otherwise apply on the original image. The results for blurred versus original images applied Sobel filters are shown in Figure 8. As you can see, when blurred, sobel edge detection algorithm conceals some of the details. The amount of edges lost depends on the amount of blurring, in other words, kernel size and sigma of the Gaussian Filter. The edges are sharper in the non blurred images, as we expected. It is normal since the Gaussian filter scatters the contrast a little bit.

D. Non-maximal suppression

In order to remove weak edges, this algorithm is implemented as the following steps according to our lecture book [1]:

1) We copy the edge map to a new edge map to prevent losing values.
2) We iterate over columns and rows from second rows and columns to penultimate rows and columns respectively, because we cannot compare the pixels on edges with their neighbors as they do not exist.
3) In this algorithm, each pixel value is compared with another pixel placed in its positive and negative gradient direction. Therefore, we have to determine its pixels in gradient direction for comparison and we help from edge direction map received by Sobel function explained in Section C. We had initially compared the values obtained directly from $arctan2$, however we discovered it returns radian values instead of degrees. As a result, we applied $numpy.rad2deg$ function to the direction map. Pixels on gradient direction are determined as the following steps according to Figure 9 taken from [2]:
   a) If the direction of edge is;
      i) greater than or equal to -22.5 and smaller than 22.5,



(a) no blur (b) blurred with sigma = 2 kernel size = 9

(c) no blur (d) blurred with sigma = 2 kernel size = 9

(e) no blur (f) blurred with sigma = 2 kernel size = 9

Fig. 8: Edges detected by only Sobel of B1, B2, B3 with blur and no blur

ii) or greater than or equal to 157.5 and smaller than and equal to 180,
iii) or greater than or equal to -180 and smaller than -157.5,

we take the pixel values on horizontal direction which are one pixel up and down of the pixel.

b) If the direction of edge is;
   i) greater than or equal to 22.5 and smaller than 67.5,
   ii) or greater than or equal to -157.5 and smaller than -112.5,

   we take the pixel values on right diagonal direction which are one pixel up and one pixel right to the pixel, and one pixel down and one pixel left to the pixel.
c) If the direction of edge is;
   i) greater than or equal to 67.5 and smaller than 112.5,
   ii) or greater than or equal to -112.5 and smaller than -67.5,

   we take the pixel values on vertical direction which are one pixel left and one pixel right to the pixel.
d) If the direction of edge is;
   i) greater than or equal to 112.5 and smaller than 157.5,
   ii) or greater than or equal to -67.5 and smaller than -22.5,

   we take the pixel values on left diagonal direction which are one pixel down and one pixel right to the pixel, and one pixel up and one pixel left to the pixel.

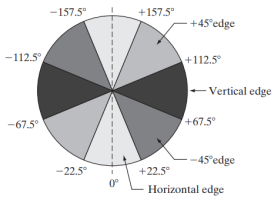


Fig. 9: The angle ranges of the edge normals for the four types of edge directions

4) We compare the magnitudes of the pixel and the other two pixels selected in Step 3, and if the magnitude of the pixel is smaller than at least one of them, we make it zero on edge map created in Step 1.

E. Thresholding

In order to reduce false edge points on the image received from non-maximal suppression algorithm, we eliminate some pixels according to two thresholds received as parameter in the main function and one of them is high and other one is low. While there were many variations of double threshold, we decided to mainly follow the algorithm in the textbook. According to steps in [3], we implemented it as the following steps:

1) We create two arrays in order to hold values got from comparison with low threshold and high threshold, which are gNL and gNH respectively. Both of them are initialized with zero and their sizes are the same as the image taken from non maximal suppression.
2) For each pixel on the suppressed image (we iterate two for loops again), if the pixel value is grater than and equal to high threshold, we changed the corresponding pixel value on gNH, with that pixel value. Similarly, if the pixel value is smaller than high threshold and greater than and equal to low threshold, we assigned it to the corresponding pixel on the gNL. In [3], they changed the pixel value on gNL without comparing with high threshold, and subtracted gNH from gNL, because there were values greater than and equal to high threshold in gNL. However, we simplified it by comparing also with high threshold value.
3) All nonzero pixels in gNH are assigned as “strong”, whereas all nonzero pixels in gNL are assigned as

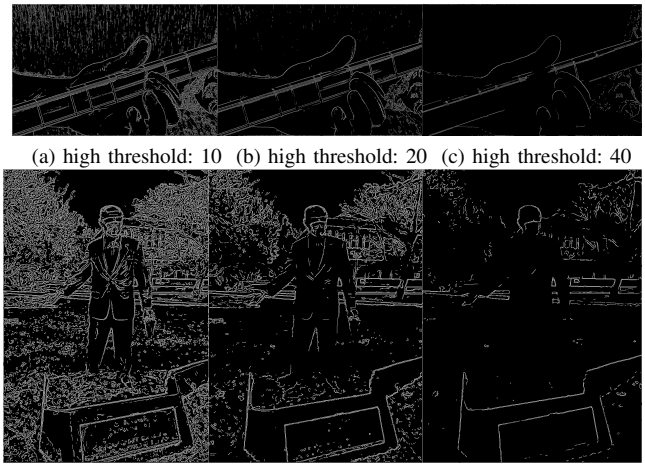
"weak". In order for a value to stay, it must be "valid". While all "strong" edges are thought as "valid", in order for "weak" edges to be "valid", the following rules must be correct: If the "weak" edge is connected to at least one "strong" edge, it is said to be "valid". Therefore, we check this condition as iterating over gNL from second rows and columns to penultimate rows and columns respectively, because we cannot compare the pixels on edges with their neighbors as they do not exist. If the pixel on gNL is nonzero, which means "weak" as it is said before, it is checked that is there any "strong" edges around its neighbors. We control this condition with another function which gets row and column indexes of the pixel, gNH and neighborhood system size (4, 6 or 8) and checks pixels on gNH according to indexes. For example, for 4-neighborhood system it checks one pixel up, down, left and right of indexes, and if at least one of them is nonzero ("strong"), it returns True, otherwise False.

4) If there is a "strong pixel" connected to the "weak" pixel, it is assigned as "valid". In order to do that, we created another zero array, named gNL_valid in the size of gNL, and we change the corresponding pixel value on that array as an arbitrary number, which is 9999, because we cannot assign a string like 'valid' word due to Python rules.
5) We again iterate over gNL and check the corresponding pixel in gNL_valid for whether it is marked as valid (9999) or not. If it is not "valid", we make it zero.
6) For all nonzero ("valid") values on gNL, if the corresponding value in gNH is zero (which means if it is an empty edge), we make it as the value of the "valid" edge.
7) In the end, we make the "strong" and "valid" values 255 and others 0, so that we have brighter and more definite edges visible to the human eye.

As you can see from Figure 10, as we increase the high threshold, some of the strong edges start to disappear. It is expected since when we increase the high threshold, the number of pixels that represent a strong edge decreases. Changes were a lot more visible in images B1 and B2 compared to B3. We believe, it is because there are a lot more details in these two picture compared to B3. Moreover, changes in low threshold were much more subtle than the changes in high threshold, making it almost invisible in this document. We expect, it is due to the fact that strong edges are more eminent in an image.

After all these steps, we finalize the detection of edges.

If we compare our the results of Canny Edge Detection Algorithm to the results of Sobel Edge Detection Algorithm blurred with Gaussian Filters, we see that Canny Edge Detection produces brighter results due to double tresholding. Double tresholding gets rid of most of the false edges in Sobel such as the creases of the sofa in B1. Moreover, the Non-maximum suppression step of canny edge detection makes the edges sharper compared to $only$ Sobel. As can be seen in Figure 11, the results of Canny Edge Detection are clearly better and more visible, however the finetuning of low and high thresholds in addition to kernel size and sigma values takes more effort. Furthermore, as we implement the iteration along images ourselves, it takes a lot longer compared to only Sobel.



(a) high threshold: 10 (b) high threshold: 20 (c) high threshold: 40

(d) high threshold: 10 (e) high threshold: 20 (f) high threshold: 40

(g) high threshold: 10 (h) high threshold: 20 (i) high threshold ratio: 40

Fig. 10: Changes in B1,B2 and B3 when the low threshold is 5 and the high threshold varies

IV. QUESTION 3 - OLD EXAM QUESTION

A.

According to Nearest Neighbor Interpolation, when we expanded 4x4 images to 8x8 images, we filled blank pixels with the brightness values of their nearest pixel. We chose the nearest pixel as the following rule: If there is a value in one pixel up, choose it. Otherwise, choose the value in one pixel left. If the left one is also empty, choose the pixel in left diagonal. As a result, we obtained the images in Figure 12.

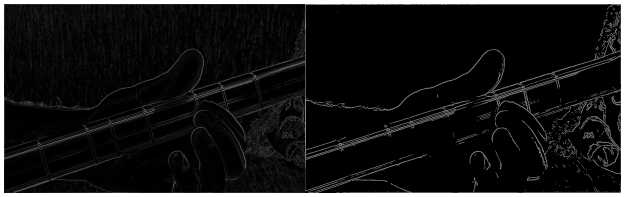
B.

1) In 8 Neighborhood: This means, if pixels either share an edge or a vertex, they are neighbours.

R: 1 connected component
G: 1 connected component
B: 1 connected component

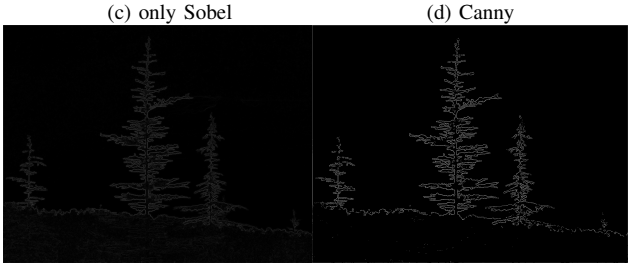
2) In 4 Neighborhood: This means, if pixels share an edge, they are neighbours.

R: 1 connected component
G: 3 connected components
B: 2 connected components



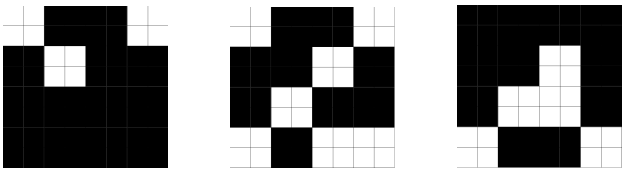
(a) only Sobel (b) Canny

(c) only Sobel (d) Canny



(e) only Sobel (f) Canny

Fig. 11: Results of Canny Edge Detection vs Only Sobel Edge Detection



(a) R (b) G (c) B

Fig. 12: Interpolated Image

3) In Mixed Neighborhood: This means, if pixels are 4 connected, it is preferred but if not and they are 8 connected, they are also neighbours.

R: 1 connected component

G: 1 connected component

B: 1 connected component

REFERENCES

[1] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.719–725.

[2] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.721.

[3] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.722–723.