EE569 - INTRODUCTION TO DIGITAL IMAGE PROCESSING
HW # 3 - 03/028/2018

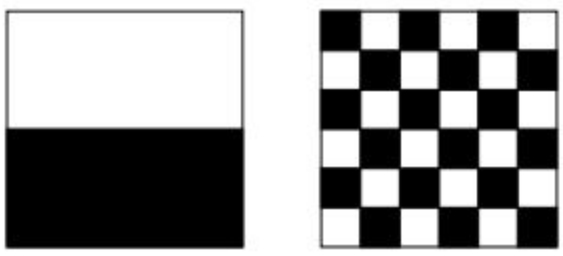
Table of Contents

Manasa Manohara
EE569 - Hw#3
3722587777

PROBLEM 1: TEXTURE ANALYSIS AND SEGMENTATION

1.1 MOTIVATION :

Texture of an image gives us an overview of the arrangement of pixels in the spatial domain. Two images might have the same histogram curve but the texture of the image can be completely dissimilar. For example, consider two images which have 50% foreground and 50% background pixels as shown -



block pattern checkerboard

Source: Shapiro, Computer Vision.

By visual inspection, we see that the above two images are strikingly different. But histogram of both the images are the same. Hence, just the histogram for image analysis does not suffice and so we perform texture analysis.

To define what a texture is, there are two main methodologies -

1. Structural approach - “Texture is a set of primitive *texels* in a repeated manner”
2. Statistical approach - “Texture is a quantitative measure for intensity arrangement in a region”

Source: Shapiro, Computer Vision.

We apply the second method as it is easier to compute for classification and segmentation purposes.

Law’s devised filters to detect features in images[5]. They are given below:

Law’s texture energy measures :

$$\begin{array}{lll} \text{L5} & \text{(Level)} & = [\ 1\ \ 4\ \ 6\ \ 4\ \ 1\ ] \\ \text{E5} & \text{(Edge)} & = [\ -1\ -2\ \ 0\ \ 2\ \ 1\ ] \\ \text{S5} & \text{(Spot)} & = [\ -1\ \ 0\ \ 2\ \ 0\ -1\ ] \\ \text{R5} & \text{(Ripple)} & = [\ 1\ -4\ \ 6\ -4\ \ 1\ ] \end{array}$$

The Law’s filters are as given above.
L5 - gives a local average of the center weighted pixel.
E5 - gives us the edges
S5 - detects spots
R5 - detects ripples

We need to find the 5x5 masks, hence we take the outer product of the individual masks. For example, E5L5 outer product will give us -

$$\begin{bmatrix} -1 \\ -2 \\ 0 \\ 2 \\ 1 \end{bmatrix} \times [\,1\ \ 4\ \ 6\ \ 4\ \ 1\,] = \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

1.2 APPROACH :

1.2.a. Texture Classification

$$E5 = \frac{1}{6}[-1\ \ -2\ \ 0\ \ 2\ \ 1], \qquad S5 = \frac{1}{4}[-1\ \ 0\ \ 2\ \ 0\ -1], \qquad W5 = \frac{1}{6}[-1\ \ 2\ \ 0\ -2\ \ 1],$$

Each of the 12 images were filtered using 9 filters - E5E5, E5S5, E5W5, S5E5, S5S5, S5W5, W5E5, W5S5, W5W5.

1. Pre-processing : We need to first remove the illumination effects. Hence we consider the global mean of the image. This eliminates the DC component of the image. This ensures that we don’t have high energy features which yield less to no information from the textures.

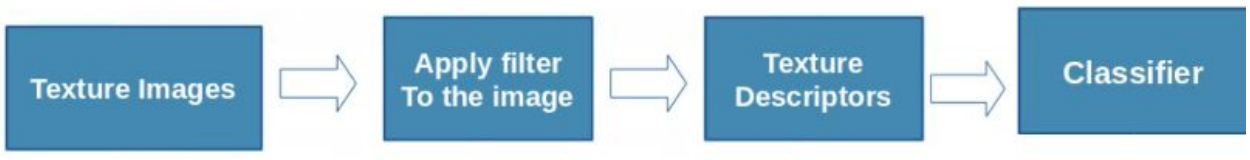
2. Feature Extraction : The 9 Law's filters were applied using the above method to all the textures1 - textures 12 images. The

$$\textbf{Energy} = \frac{1}{N*M} * \sum_{i=0}^{N} \sum_{j=0}^{M} (I(i,j))^2$$

texture energy was calculated using the formula -

This gave me a 12x9 feature matrix.

3. Clustering : I have used k-means algorithm which uses the concept of nearest centroid to a point by calculating the euclidean distance. It randomly initializes k centroids for the first iteration. K-means is an iterative algorithm. In every iteration, the centroids are updated by calculating the average of the points which are classified to a new class in an iteration. The stopping condition used was to iterate to a given number of iterations.



1.2.a.1 ALGORITHM :

1. Load the 12 input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for each input image.
4. Allocate memory for feature array, means array, and energy arrays for every input image.
5. Convert the unsigned char image to double array for easier calculation of pixels.
6. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
7. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
8. Call the 'texture_analysis ' function for every input image.
9. Calculate the tensor product for the all given pairs 5x5 Laws filters. We will obtain 9 such filters.
10. Remove DC component for the input image by calculating a global mean and updating it to the same image.
11. Obtain the nearest neighborhood pixels of the double input image by extension of the image by pixel padding.
12. Obtain the nearest neighbors and multiply it with the 5x5 Law's filter and sum it up. Return the summed value to be the value at the current location.
13. Square the summed value and divide it by the neighborhood area to calculate the feature energy for that image for the corresponding filter.
14. Repeat steps 11 - 13 for all pairs of filters and for all 12 images.
15. In the end, we will have a populated feature array of size 12x9.
16. Set the initial mean vectors to be one of the prototype of every texture image by visual inspection. This makes sure that our algorithm works properly.
17. Apply the k-means clustering algorithm to the obtained feature array with the initialized mean array.
18. Set the number of iterations for the k-means to run to be 20.
19. Run the loops for the length of the feature array and the number of classes chosen. Here the number of classes = 4.
20. Initialize minimum distance for the particular input to be 0.
21. Calculate the Euclidean distance between the current feature vector and the means.
22. Compare the Euclidean distance of the current feature vector with all the means.
23. Obtain the minimum distance and update the minimum distance variable.
24. Get the corresponding class to which the feature vector has minimum distance.
25. Repeat steps 19-13 for all sets of feature vectors and means.
26. Update the means by calculating the means of features which are now classified to the class. Hence at the end, we will have new means.
27. Go to the next iteration.
28. Repeat steps 19-27 for the number of iterations set.
29. In the end, we will have the class labels to which every input image belongs to.

1.3 RESULTS :

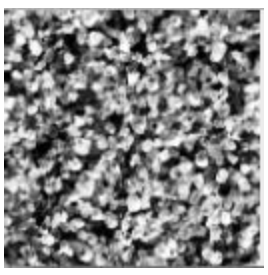
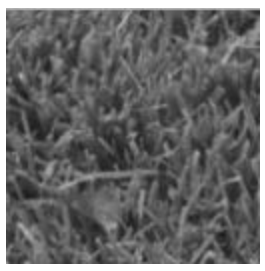
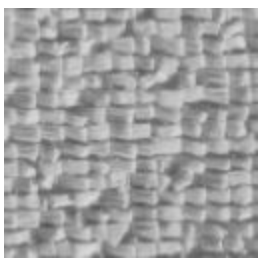
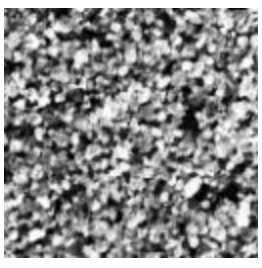
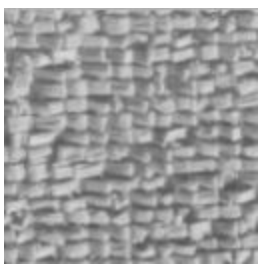
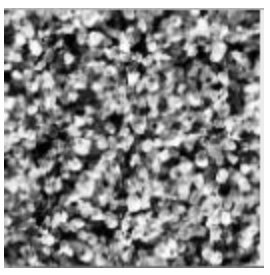
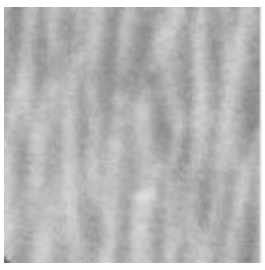
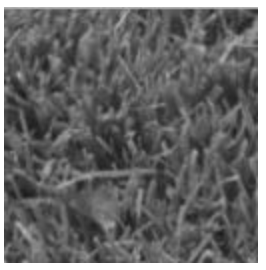
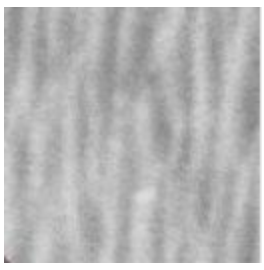
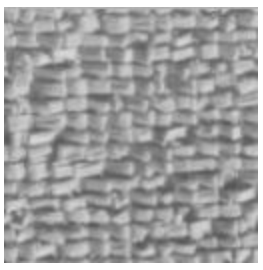
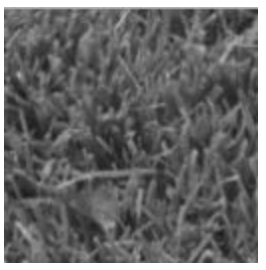
1.3.a. Texture Classification

1.3.a.(i)Legend for classification :

Label	Class
0	Rock
1	Grass

2	Weave
3	Sand

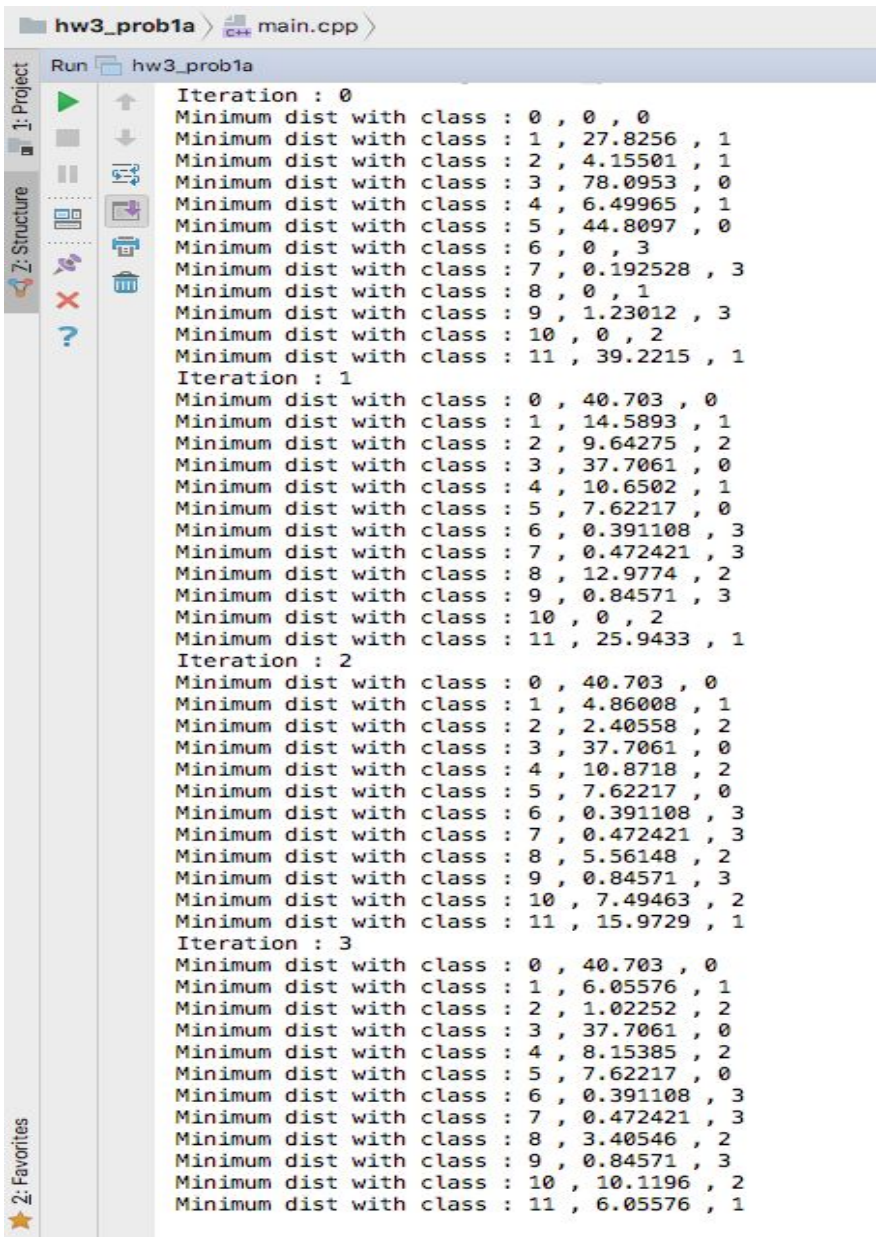
1.3.a.(ii)Input - Texture1 - Texture 12 images :

 Texture 1	 Texture 2	 Texture 3
 Texture 4	 Texture 5	 Texture 6
 Texture 7	 Texture 8	 Texture 9
 Texture 10	 Texture 11	 Texture 12

1.3.a.(iii)Output :

Texture image	Visual inspection	Program prediction(after 4 iterations)
Texture 1	0 [Rock]	0 [Rock]
Texture 2	1[Grass]	1[Grass]
Texture 3	2[Weave]	2[Weave]
Texture 4	0 [Rock]	0 [Rock]
Texture 5	2[Weave]	2[Weave]
Texture 6	0 [Rock]	0 [Rock]
Texture 7	3[Sand]	3[Sand]
Texture 8	3[Sand]	3[Sand]
Texture 9	1[Grass]	2[Weave]
Texture 10	3[Sand]	3[Sand]
Texture 11	2[Weave]	2[Weave]
Texture 12	1[Grass]	1[Grass]

1.3.a.(iv) Output :



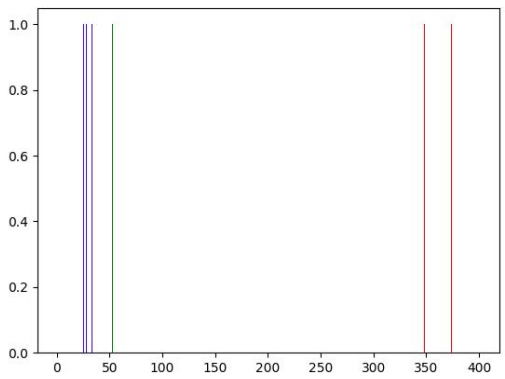

```
Iteration : 0
Minimum dist with class : 0 , 0 , 0
Minimum dist with class : 1 , 27.8256 , 1
Minimum dist with class : 2 , 4.15501 , 1
Minimum dist with class : 3 , 78.0953 , 0
Minimum dist with class : 4 , 6.49965 , 1
Minimum dist with class : 5 , 44.8097 , 0
Minimum dist with class : 6 , 0 , 3
Minimum dist with class : 7 , 0.192528 , 3
Minimum dist with class : 8 , 0 , 1
Minimum dist with class : 9 , 1.23012 , 3
Minimum dist with class : 10 , 0 , 2
Minimum dist with class : 11 , 39.2215 , 1
Iteration : 1
Minimum dist with class : 0 , 40.703 , 0
Minimum dist with class : 1 , 14.5893 , 1
Minimum dist with class : 2 , 9.64275 , 2
Minimum dist with class : 3 , 37.7061 , 0
Minimum dist with class : 4 , 10.6502 , 1
Minimum dist with class : 5 , 7.62217 , 0
Minimum dist with class : 6 , 0.391108 , 3
Minimum dist with class : 7 , 0.472421 , 3
Minimum dist with class : 8 , 12.9774 , 2
Minimum dist with class : 9 , 0.84571 , 3
Minimum dist with class : 10 , 0 , 2
Minimum dist with class : 11 , 25.9433 , 1
Iteration : 2
Minimum dist with class : 0 , 40.703 , 0
Minimum dist with class : 1 , 4.86008 , 1
Minimum dist with class : 2 , 2.40558 , 2
Minimum dist with class : 3 , 37.7061 , 0
Minimum dist with class : 4 , 10.8718 , 2
Minimum dist with class : 5 , 7.62217 , 0
Minimum dist with class : 6 , 0.391108 , 3
Minimum dist with class : 7 , 0.472421 , 3
Minimum dist with class : 8 , 5.56148 , 2
Minimum dist with class : 9 , 0.84571 , 3
Minimum dist with class : 10 , 7.49463 , 2
Minimum dist with class : 11 , 15.9729 , 1
Iteration : 3
Minimum dist with class : 0 , 40.703 , 0
Minimum dist with class : 1 , 6.05576 , 1
Minimum dist with class : 2 , 1.02252 , 2
Minimum dist with class : 3 , 37.7061 , 0
Minimum dist with class : 4 , 8.15385 , 2
Minimum dist with class : 5 , 7.62217 , 0
Minimum dist with class : 6 , 0.391108 , 3
Minimum dist with class : 7 , 0.472421 , 3
Minimum dist with class : 8 , 3.40546 , 2
Minimum dist with class : 9 , 0.84571 , 3
Minimum dist with class : 10 , 10.1196 , 2
Minimum dist with class : 11 , 6.05576 , 1
```

1.4 DISCUSSION :

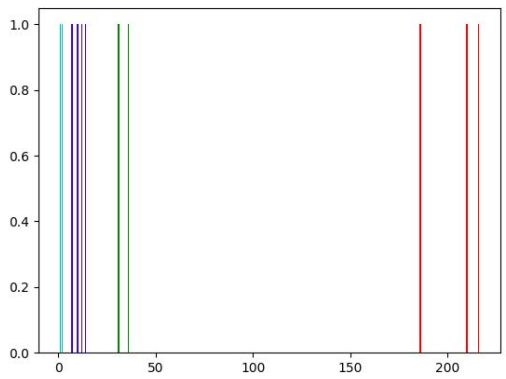
1.4.a. Texture Classification

```
/Users/mansi/CLionProjects/hw3_prob1a/cmake-build-debug/hw3_prob1a /Users/mansi/Documents/imageProcessing/images/EE569_hw3_images/texture1.raw /Users/mansi/Documents/imageProcessing/images/
 Tensor product filter E5E5:
0.0277778 0.0555556 0 -0.0555556 -0.0277778 0.0555556 0.111111 0 -0.111111 -0.0555556 0 0 0 0 0 -0.0555556 -0.111111 0 0.111111 0.0555556 -0.0277778 -0.0555556 0 0.0555556 0.0277778
 Tensor product filter E5S5:
0.0416667  0  -0.0833333  0  0.0416667  0.0833333  0  -0.166667  0  0.0833333  0  0  0  0  0  -0.0833333  0  0.166667  0  -0.0833333  -0.0416667  0  0.0833333  0  -0.0416667
 Tensor product filter E5W5:
0.0277778 -0.0555556 0 0.0555556 -0.0277778 0.0555556 -0.111111 0 0.111111 -0.0555556 0 0 0 0 0 -0.0555556 0.111111 0 -0.111111 0.0555556 -0.0277778 0.0555556 0 -0.0555556 0.0277778
 Tensor product filter S5E5:
0.0416667 0.0833333 0 -0.0833333 -0.0416667 0 0 0 0 0 -0.0833333 -0.166667 0 0.166667 0.0833333 0 0 0 0 0 0.0416667 0.0833333 0 -0.0833333 -0.0416667
 Tensor product filter S5S5:
0.0625 0 -0.125 0 0.0625 0 0 0 0 0 -0.125 0 0.25 0 -0.125 0 0 0 0 0 0.0625 0 -0.125 0 0.0625
 Tensor product filter S5W5:
0.0416667 -0.0833333 0 0.0833333 -0.0416667 0 0 0 0 0 -0.0833333 0.166667 0 -0.166667 0.0833333 0 0 0 0 0 0.0416667 -0.0833333 0 0.0833333 -0.0416667
 Tensor product filter W5E5:
0.0277778 0.0555556 0 -0.0555556 -0.0277778 -0.0555556 -0.111111 0 0.111111 0.0555556 0 0 0 0 0 0.0555556 0.111111 0 -0.111111 -0.0555556 -0.0277778 -0.0555556 0 0.0555556 0.0277778
 Tensor product filter W5S5:
0.0416667 0 -0.0833333 0 0.0416667 -0.0833333 0 0.166667 0 -0.0833333 0 0 0 0 0 0.0833333 0 -0.166667 0 0.0833333 -0.0416667 0 0.0833333 0 -0.0416667
 Tensor product filter W5W5:
0.0277778 -0.0555556 0 0.0555556 -0.0277778 -0.0555556 0.111111 0 -0.111111 0.0555556 0 0 0 0 0 0.0555556 -0.111111 0 0.111111 -0.0555556 -0.0277778 0.0555556 0 -0.0555556 0.0277778
```

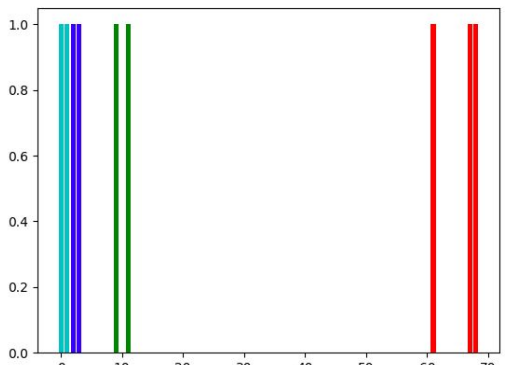
```
Feature vectors for 1 a :
0 , 0 : 348.429  0 , 1 : 186.763  0 , 2 : 61.3237  0 , 3 : 185.493  0 , 4 : 118.002  0 , 5 : 43.5267  0 , 6 : 53.9798  0 , 7 : 38.377  0 , 8 : 15.3386
1 , 0 : 47.5558  1 , 1 : 31.1735  1 , 2 : 9.43407  1 , 3 : 23.504  1 , 4 : 16.9658  1 , 5 : 5.81954  1 , 6 : 5.97848  1 , 7 : 4.7019  1 , 8 : 1.85115
2 , 0 : 25.6384  2 , 1 : 10.6018  2 , 2 : 2.89253  2 , 3 : 13.0281  2 , 4 : 5.70674  2 , 5 : 1.65945  2 , 6 : 4.12969  2 , 7 : 1.85916  2 , 8 : 0.554682
3 , 0 : 399.888  3 , 1 : 216.666  3 , 2 : 67.5494  3 , 3 : 231.678  3 , 4 : 139.945  3 , 5 : 47.3686  3 , 6 : 67.6661  3 , 7 : 45.7684  3 , 8 : 16.7502
4 , 0 : 33.8796  4 , 1 : 12.6823  4 , 2 : 2.89708  4 , 3 : 16.9947  4 , 4 : 6.64002  4 , 5 : 1.57867  4 , 6 : 4.9563  4 , 7 : 2.03836  4 , 8 : 0.487559
5 , 0 : 374.351  5 , 1 : 210.394  5 , 2 : 68.9585  5 , 3 : 207.306  5 , 4 : 133.021  5 , 5 : 49.0818  5 , 6 : 62.6901  5 , 7 : 45.0478  5 , 8 : 17.9895
6 , 0 : 2.65409  6 , 1 : 2.32582  6 , 2 : 1.20472  6 , 3 : 2.38193  6 , 4 : 2.27593  6 , 5 : 1.22917  6 , 6 : 1.22167  6 , 7 : 1.24336  6 , 8 : 0.697996
7 , 0 : 2.88115  7 , 1 : 2.35499  7 , 2 : 1.24216  7 , 3 : 2.39445  7 , 4 : 2.31956  7 , 5 : 1.29019  7 , 6 : 1.19406  7 , 7 : 1.25079  7 , 8 : 0.722932
8 , 0 : 28.3581  8 , 1 : 14.1254  8 , 2 : 3.2764  8 , 3 : 13.2958  8 , 4 : 7.47142  8 , 5 : 1.92072  8 , 6 : 3.65611  8 , 7 : 2.26212  8 , 8 : 0.636587
9 , 0 : 2.42936  9 , 1 : 1.54457  9 , 2 : 0.484627  9 , 3 : 1.97534  9 , 4 : 1.42324  9 , 5 : 0.475001  9 , 6 : 0.920923  9 , 7 : 0.706593  9 , 8 : 0.251835
10 , 0 : 17.7414  10 , 1 : 7.58545  10 , 2 : 2.23867  10 , 3 : 7.62397  10 , 4 : 3.24359  10 , 5 : 1.01368  10 , 6 : 1.71569  10 , 7 : 0.750639  10 , 8 : 0.245106
11 , 0 : 53.2846  11 , 1 : 36.6017  11 , 2 : 11.9383  11 , 3 : 28.484  11 , 4 : 23.5744  11 , 5 : 9.01584  11 , 6 : 8.81258  11 , 7 : 8.14492  11 , 8 : 3.61774
```



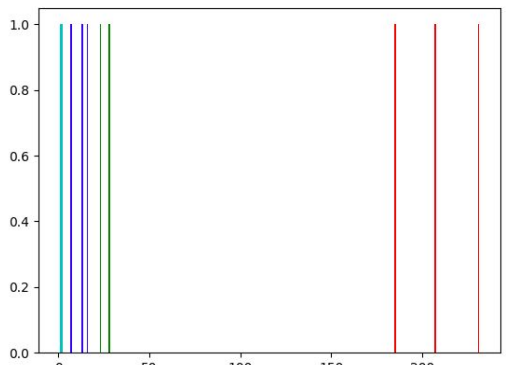
Filter 1 - E5E5



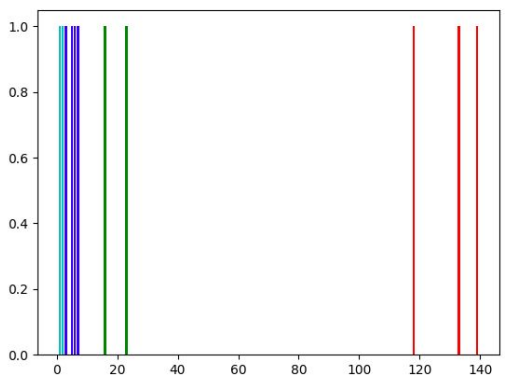
Filter 2 - E5S5



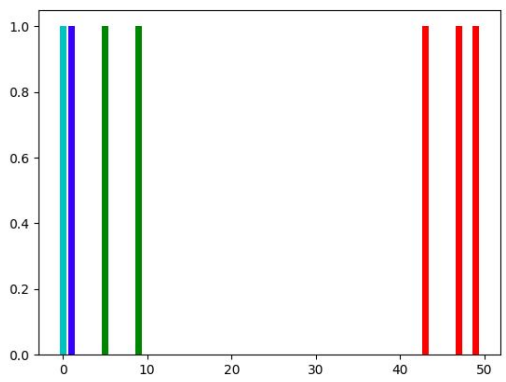
Filter 3 - E5W5



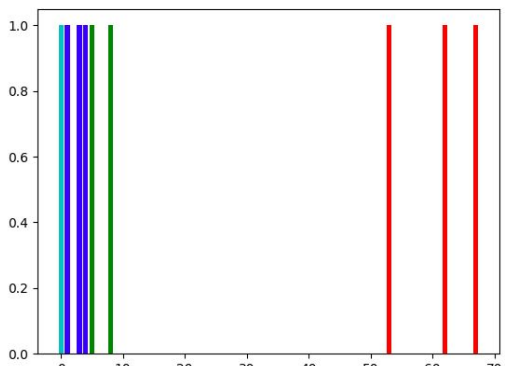
Filter 4 - S5E5



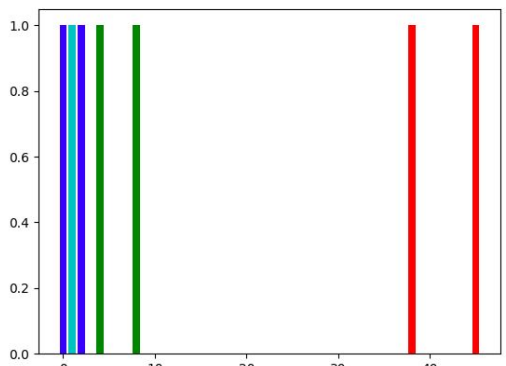
Filter 5 - S5S5



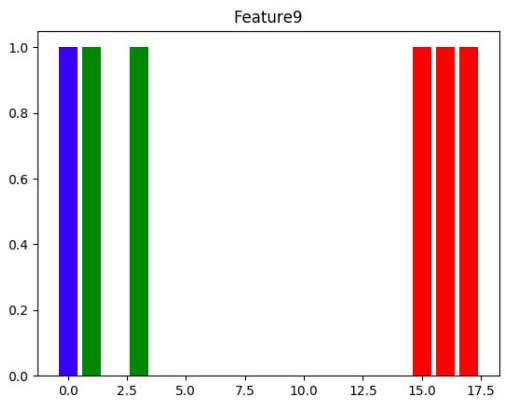
Filter 6 - S5W5



Filter 7 - W5E5



Filter 8 - W5S5



Filter 9 - W5W5

The dimension with the strongest discriminant power was dimension 2 which corresponds to - E5S5. The dimension with the weakest discriminant power is 7 which corresponds to W5S5.

The strongest dimension clusters all of the images with no overlapping between clusters. The strongest can also classify all textures. This gives us good discriminant distance and makes sure there are no ambiguous clusters. But we can also see that two different clusters in the strongest that I have chosen are close to each other, so technically a combination of filters should be used to classify textures and not stick to one strong discriminant dimension.

Texture image	Visual inspection	Program prediction(after 4 iterations)
Texture 1	0 [Rock]	0 [Rock]
Texture 2	1[Grass]	1[Grass]
Texture 3	2[Weave]	2[Weave]
Texture 4	0 [Rock]	0 [Rock]
Texture 5	2[Weave]	2[Weave]
Texture 6	0 [Rock]	0 [Rock]
Texture 7	3[Sand]	3[Sand]
Texture 8	3[Sand]	3[Sand]
Texture 9	1[Grass]	2[Weave]
Texture 10	3[Sand]	3[Sand]
Texture 11	2[Weave]	2[Weave]
Texture 12	1[Grass]	1[Grass]

From the above figure, we can see that the texture_9 is getting misclassified. We can see from the 1.3.a.(iv) Output that texture_9 gets misclassified in the iteration : 1. This is because of the centroid updation after the iteration. It can also be seen from the same image that the minimum distance is now with respect to the class Weave. This is one misclassified point from the k-means algorithm.

```
/Users/mansi/CLionProjects/hw3_prob1a/cmake-build-debug/hw3_prob1a /l
348.429  , 28.3581  , 33.8796
186.763  , 14.1254  , 12.6823
61.3237  , 3.2764  , 2.89708
185.493  , 13.2958  , 16.9947
118.002  , 7.47142  , 6.64002
43.5267  , 1.92072  , 1.57867
53.9798  , 3.65611  , 4.9563
38.377  , 2.26212  , 2.03836
15.3386  , 0.636587  , 0.487559
```

I compared the energy values of 3 textures to check if texture 9 was initially closer to class Weave. The first column is an energy array of class Grass, second column is the texture 9 energy value, and the third column is the feature values of class Weave. We can mentally calculate the difference and see that the values in the second and the third column are similar. Hence, initially texture 9 was closer to class Weave.

In iteration :1, after the updated calculation of centroids, the texture_9 was misclassified.

1.2 APPROACH :

1.2.b. Texture Segmentation

L3 = [⅛, ⅓ ,⅛]

E3 = [-½,0,½]

S3 = [-½,1,-½]

Comb.raw was filtered using 9 filters - E3E3, E3S3, E3W3, S3E3, S3S3, S3W3, W3E3, W3S3, W3W3.

1. Pre - processing : We need to first remove the illumination effects. Hence we consider a small window of its neighboring pixels and obtain the local mean and then update the same input image with the new obtained illumination removed component. I have considered a window size of 21 to calculate the local mean.
2. Laws feature extraction : All the 9 3x3 Law's filters was applied to the image and the 9 grayscale outputs were displayed.
3. Energy computation : Here, we need to calculate the energy per pixel. Hence, a window of size() was considered. The energy feature value for each pixel is given by -

$$Energy = \frac{1}{(WindowSize * WindowSize)} \times \sum_{i=0,j=0}^{i=Height,j=Width} (Img(i,j) * Img(i,j))$$

4. Energy feature normalization : $L3L3^T$ does not have a zero mean, hence its energy can be used to normalize all other pixels. Both the methods were tried - with normalizing and without normalizing.

5. Segmentation : K-means algorithm was used to segment the image. There are 6 classes and the centroids for each class was initialized by visual inspection.

Class	Centroid	Class color
0	21095	0
1	24895	51
2	55093	102
3	132891	153
4	222680	204
5	218883	255

1.2.a.2 ALGORITHM :

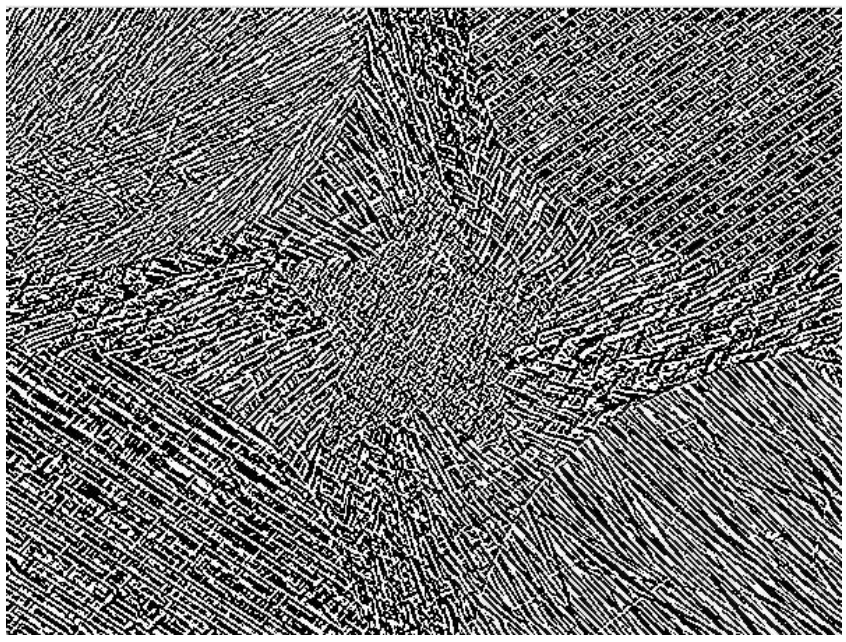
1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Allocate memory for feature array, means array, and energy arrays for the input image.
5. Convert the unsigned char image to double array for easier calculation of pixels.
6. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
7. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
8. Call the 'texture_analysis ' function for the input image.
9. Find the current pixel location by (row*width) +col.
10. Calculate the tensor product for the all given pairs 5x5 Laws filters. We will obtain 9 such filters.
11. Remove DC component for the input image by calculating a local mean of window size 21x21 and updating it to the same image.
12. Obtain the nearest neighborhood pixels of the double input image by extension of the image by pixel padding.
13. Obtain the nearest neighbors and multiply it with the 5x5 Law's filter and sum it up. Return the summed value to be the value at the current location of the image.
14. Square the summed value and divide it by the neighborhood area to calculate the feature energy for that current pixel location for the corresponding filter.
15. Repeat steps 11 - 13 for through the height and width of the image.
16. In the end, we will have a populated feature array of size widthxheightx9.
17. Set the initial mean vectors to be one of the prototype of every texture image by visual inspection. This makes sure that our algorithm works properly.
18. Apply the k-means clustering algorithm to the obtained feature array with the initialized mean array.
19. Set the number of iterations for the k-means to run to be 20.
20. Run the loops for the length of the feature array and the number of classes chosen. Here the number of classes = 6. Every class is depicted by its color. The colors chosen here are - {0,51,102,153,204,255}.
21. Initialize minimum distance for the particular input to be 0.
22. Calculate the Euclidean distance between the current feature vector and the means.
23. Compare the Euclidean distance of the current feature vector with all the means.
24. Obtain the minimum distance and update the minimum distance variable.
25. Get the corresponding class to which the feature vector has minimum distance.
26. Repeat steps 19-13 through the length of feature vectors and means.
27. Update the means by calculating the means of features which are now classified to the class. Hence at the end, we will have new means.
28. Go to the next iteration.
29. Repeat steps 19-27 for the number of iterations set.
30. In the end, we will have the class labels to which every input current pixel location belongs to.

1.3 RESULTS :

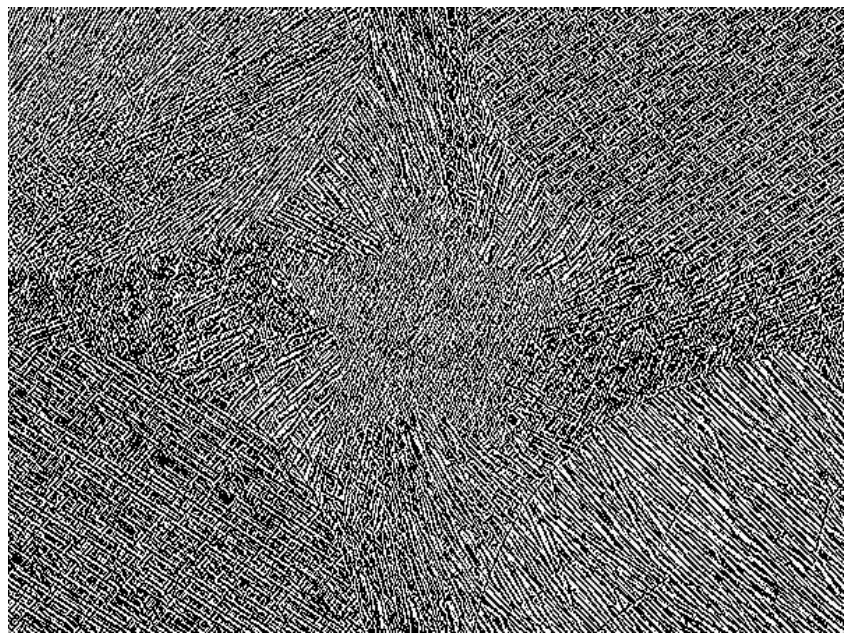
1.2.b. Texture Segmentation

1.3.b.(i)9 filtered GrayScale outputs :

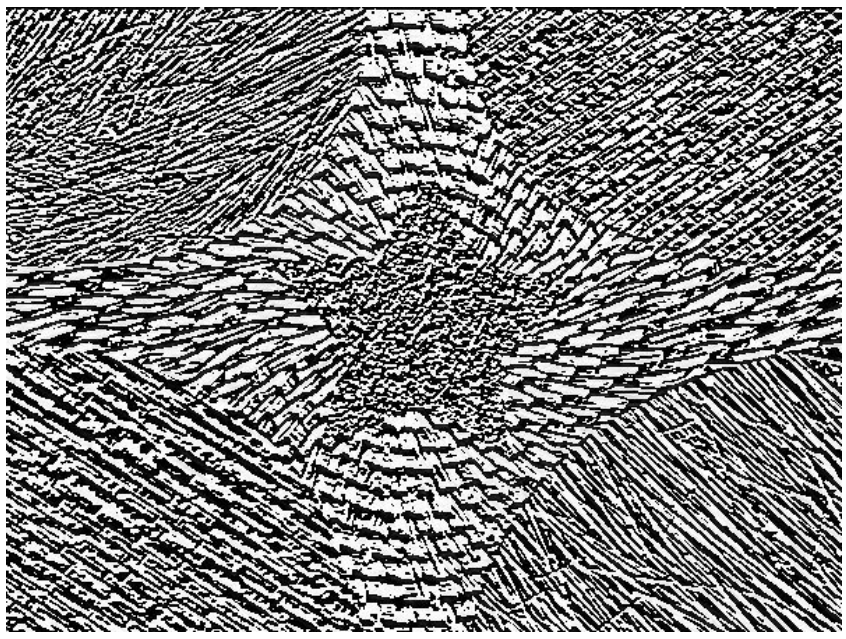
Manasa Manohara
EE569 - Hw#3
3722587777



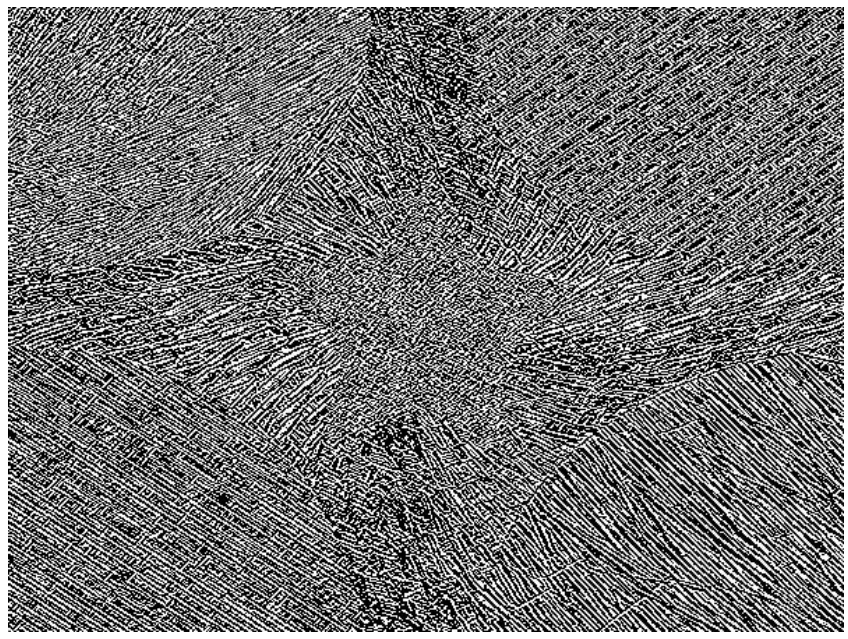
Filter 1 - E3E3



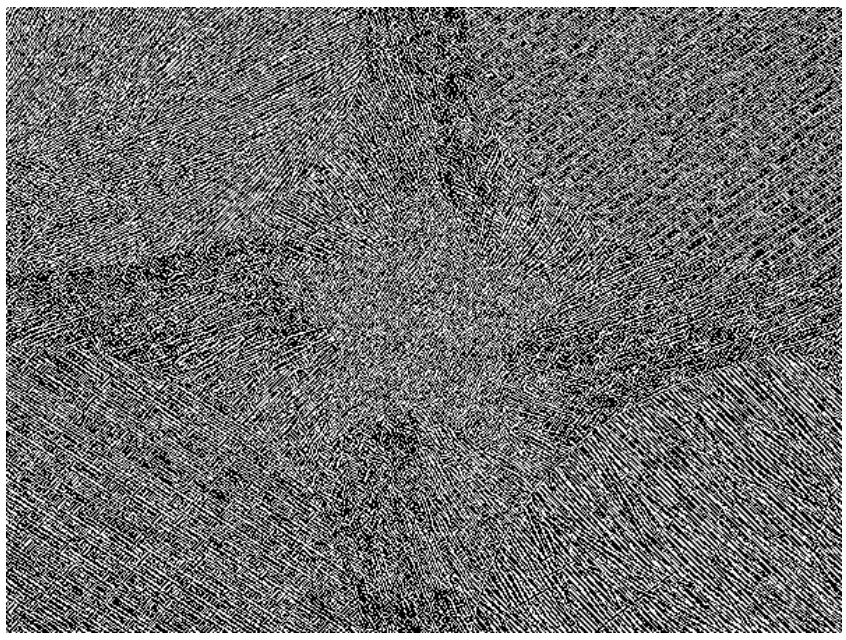
Filter 2 - E3S3



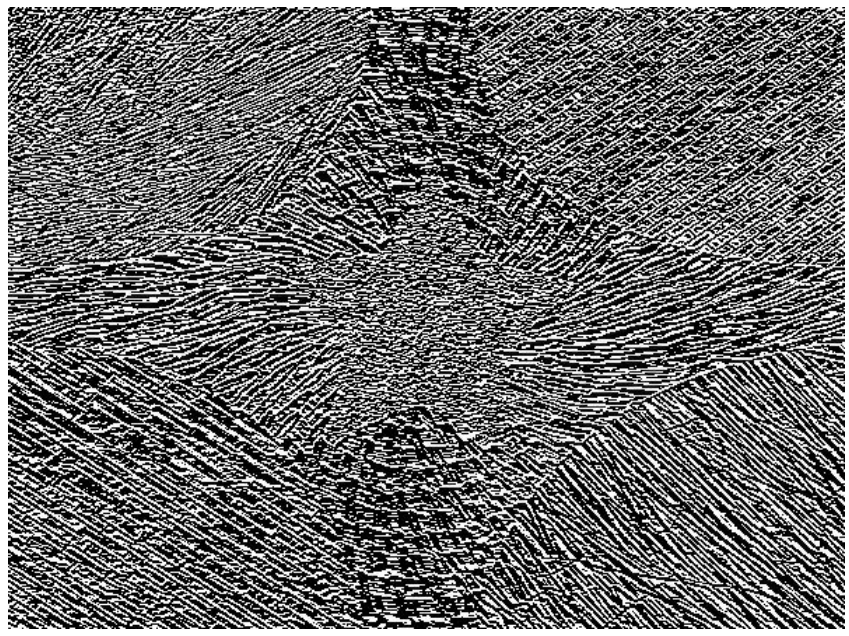
Filter 3 - E3L3



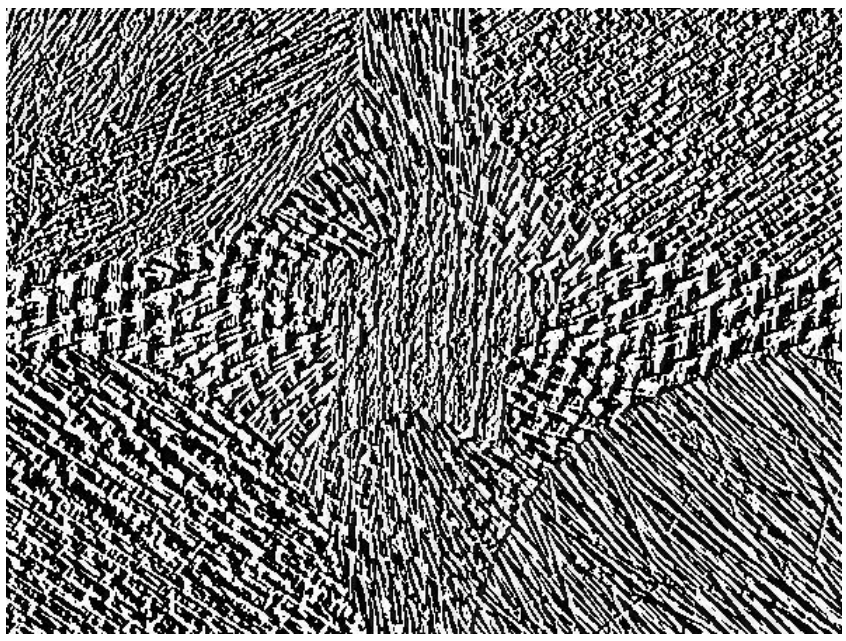
Filter 4 - S3E3



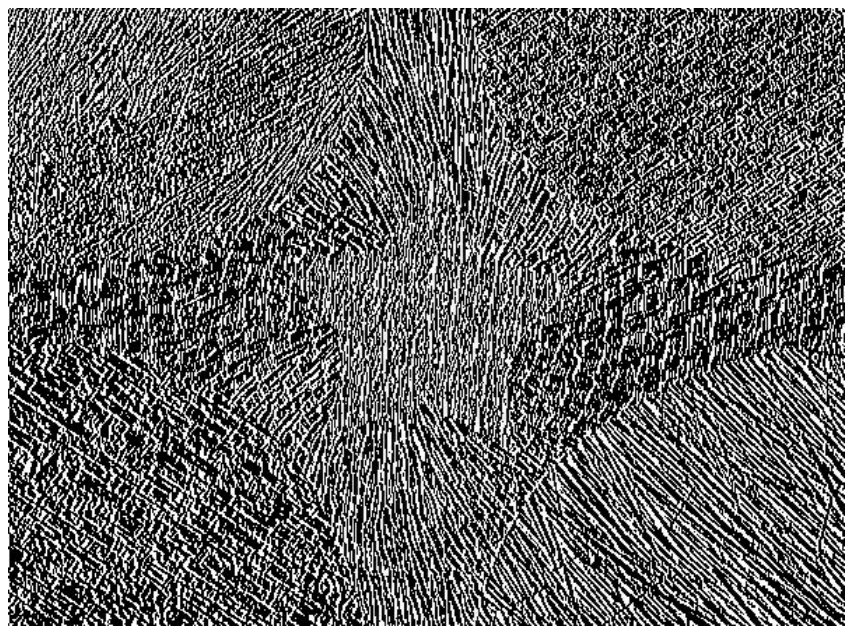
Filter 5 - S3S3



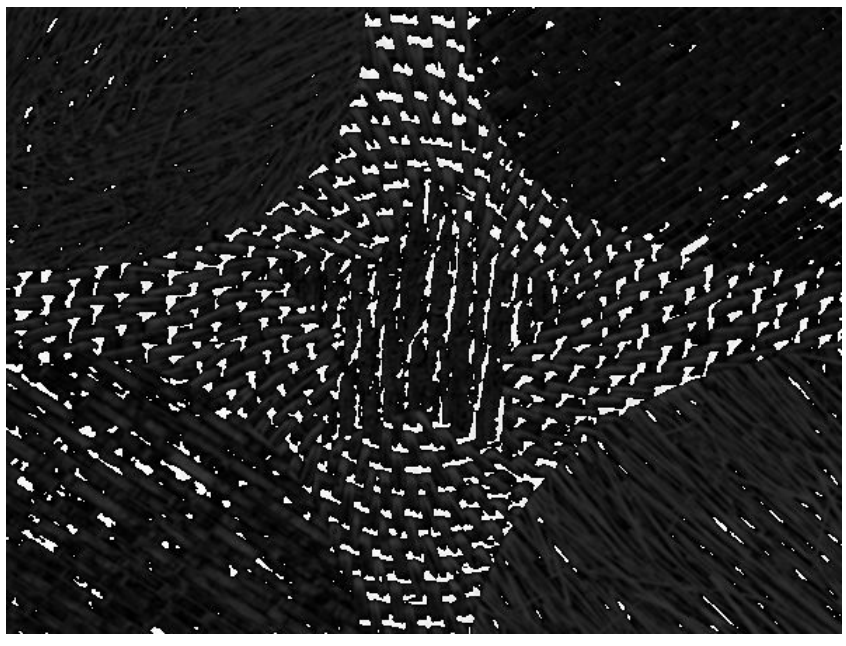
Filter 6 - S3L3



Filter 7 - L3E3

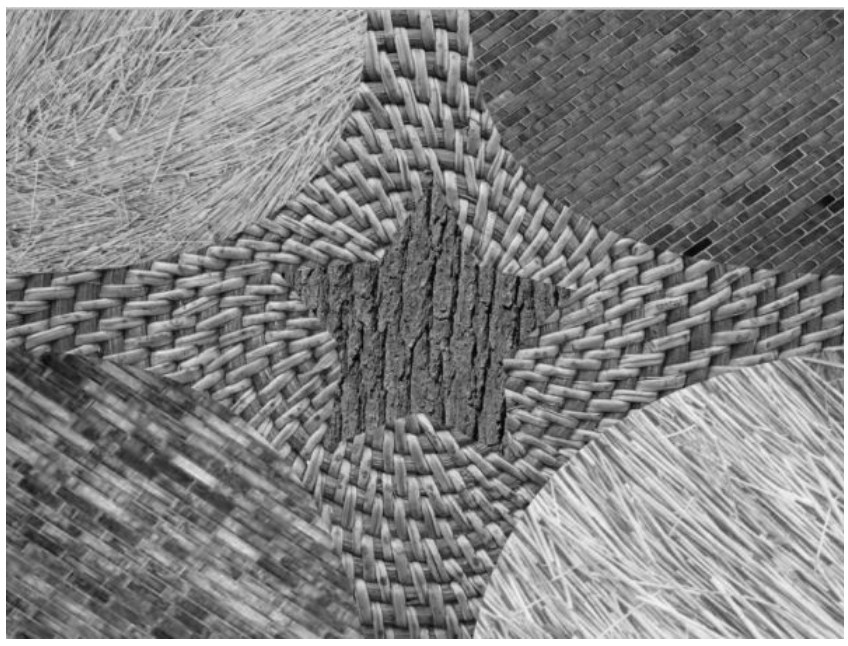


Filter 8 - L3S3

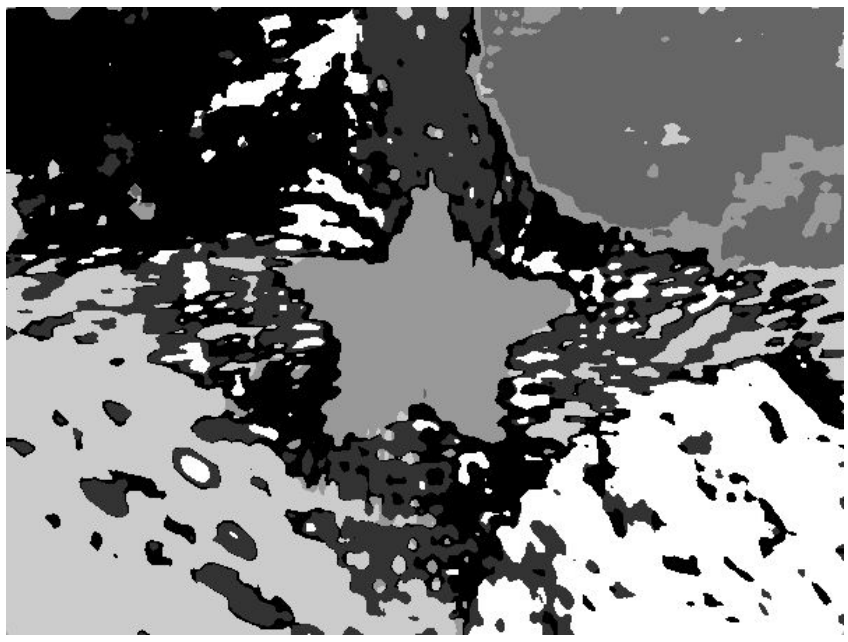


Filter 9 - L3L3

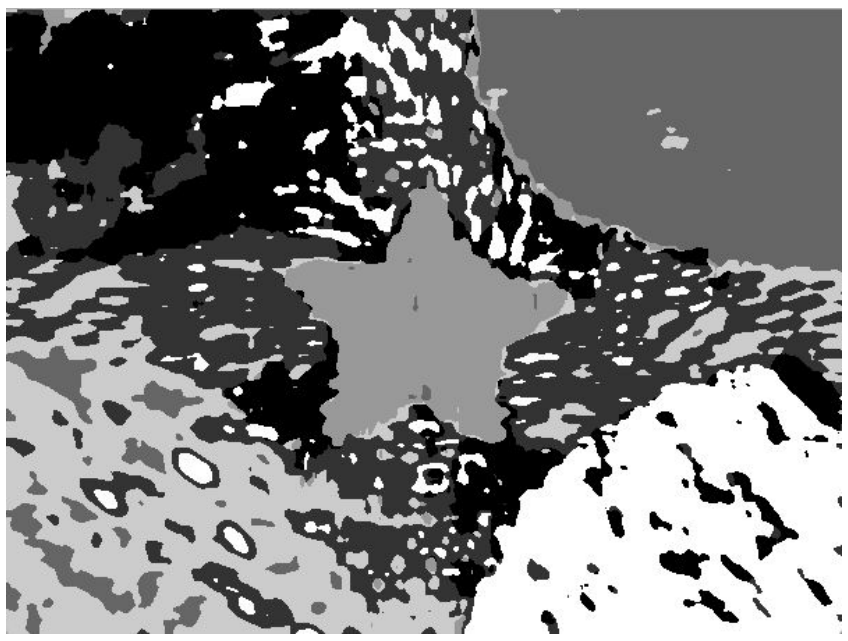
1.3.b.(ii)Clustering Outputs :
N = 19 output



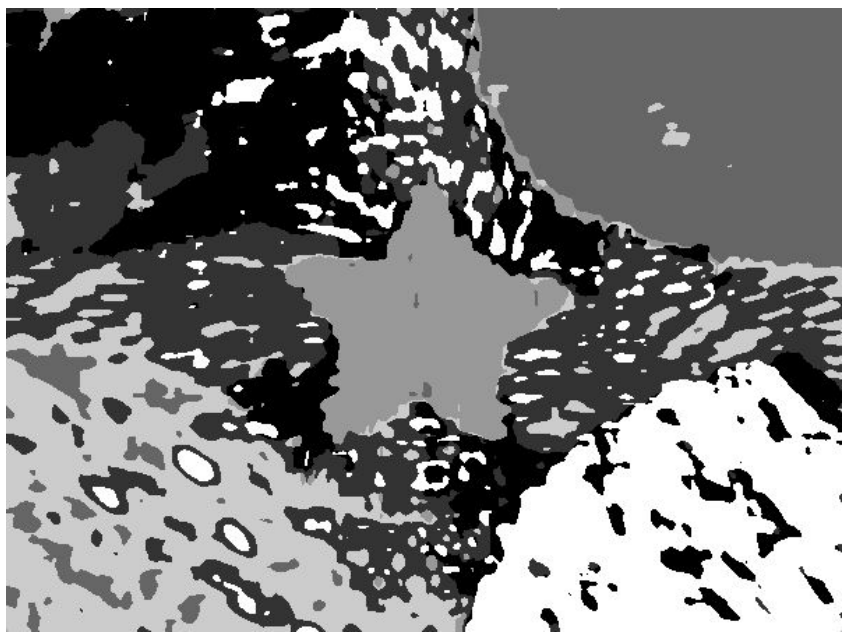
Original input = comb.raw



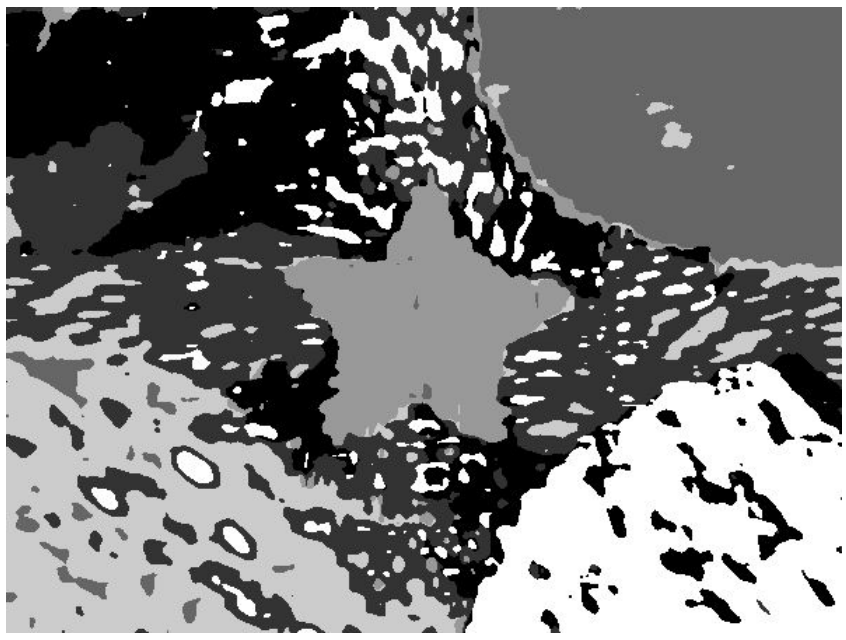
N = 19, iteration = 1



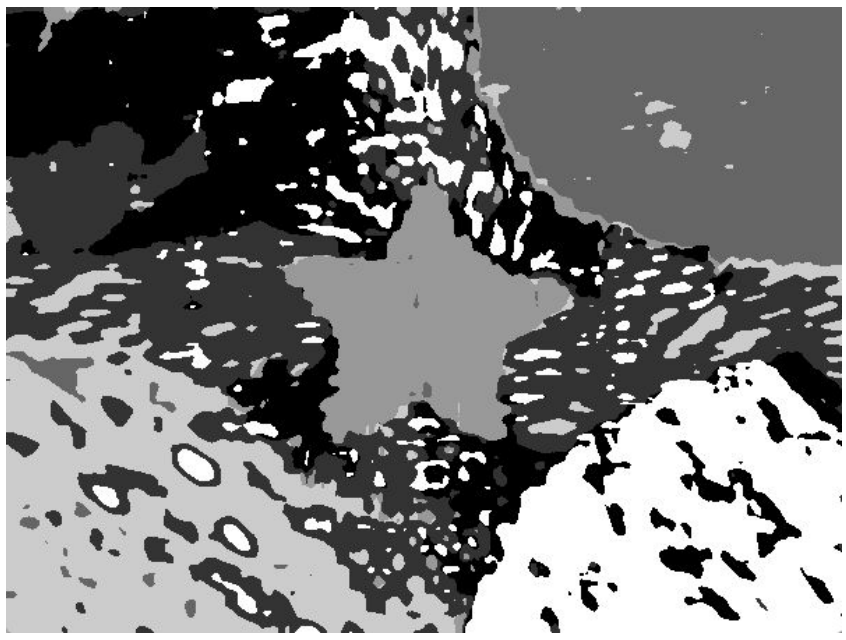
N = 19, iteration = 5



N = 19, iteration = 10

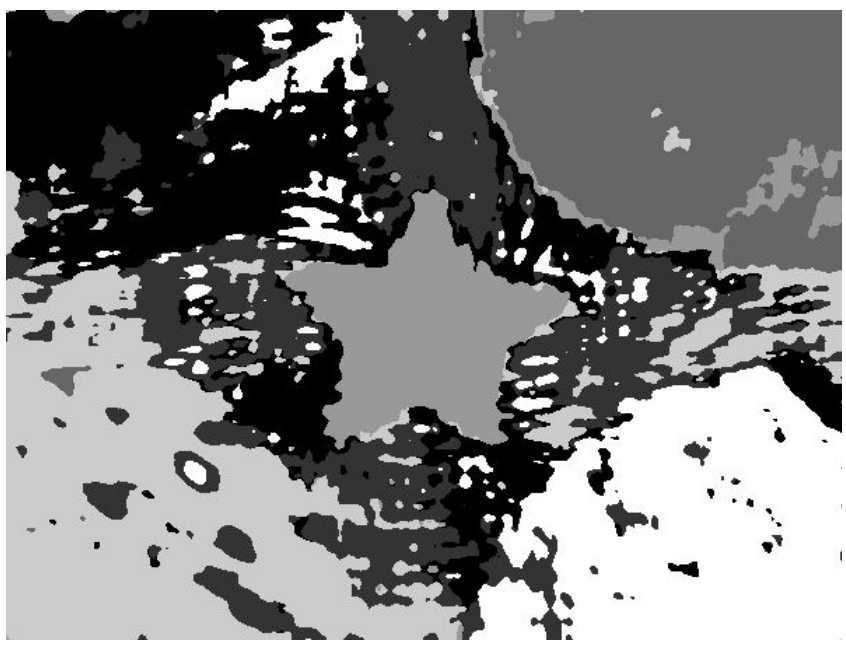


N = 19, iteration = 15



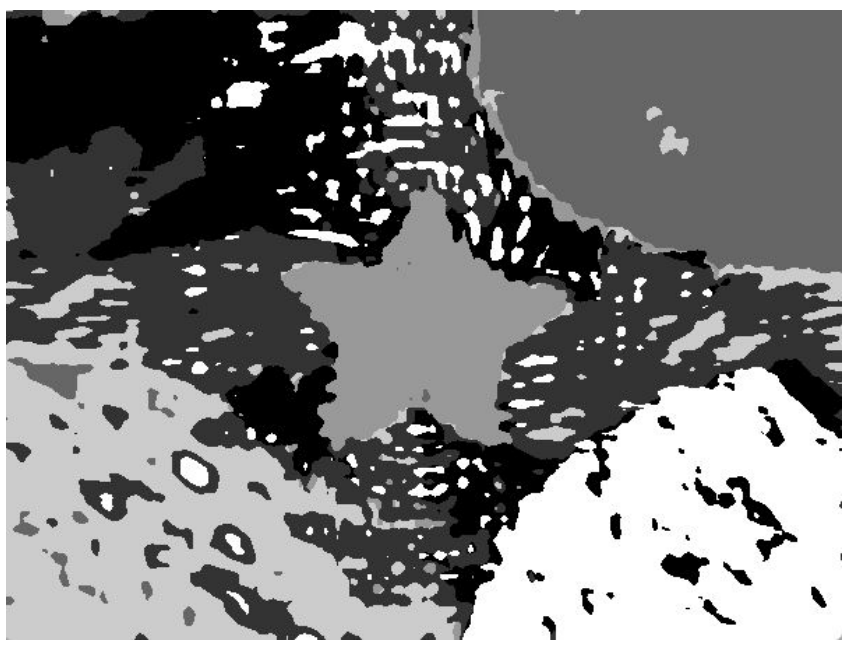
N = 19, iteration = 20

N = 21 - BEST OUTPUT

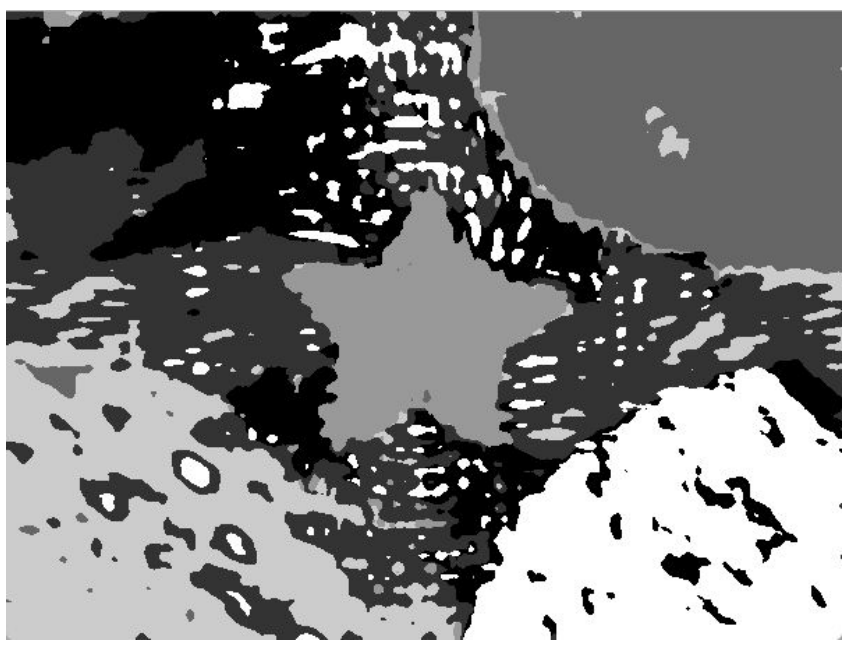


N = 21, iteration = 1

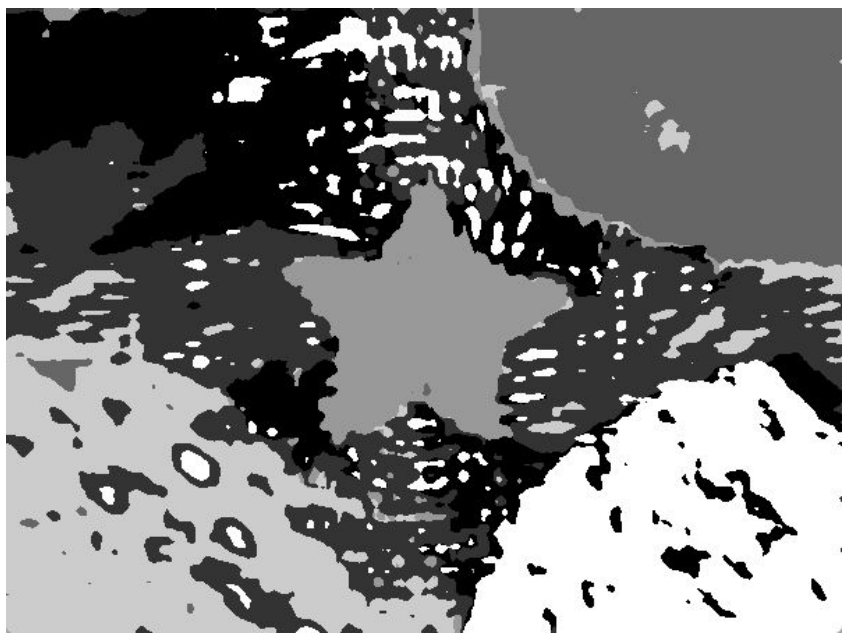
N = 21, iteration = 5



N = 21, iteration = 10

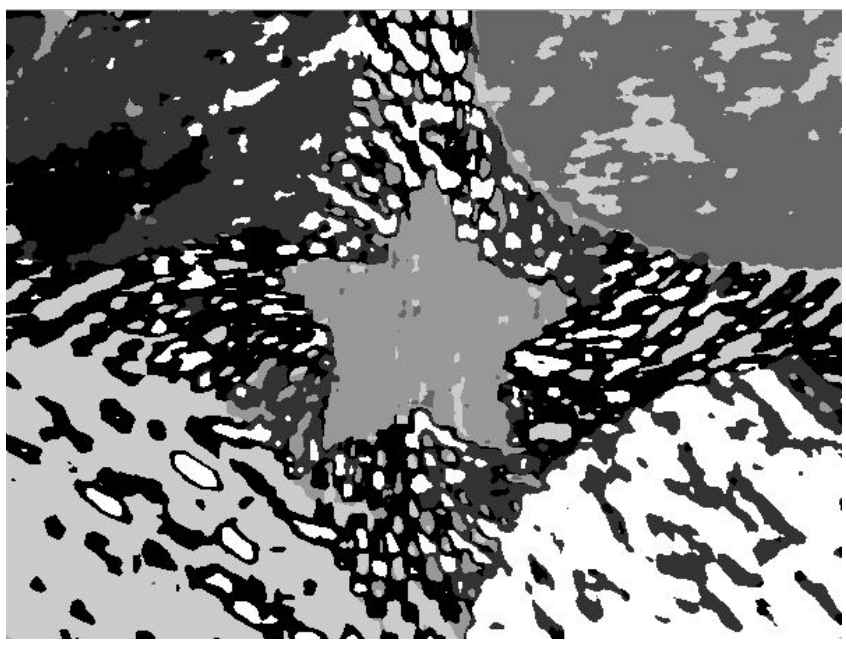


N = 21, iteration = 15

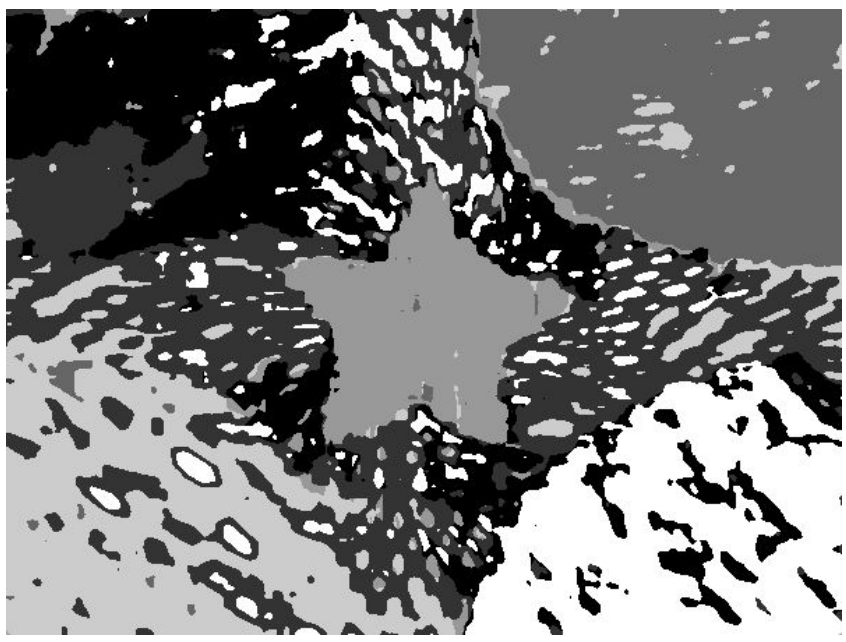


N = 21, iteration = 20

Other window sizes = 15, 17

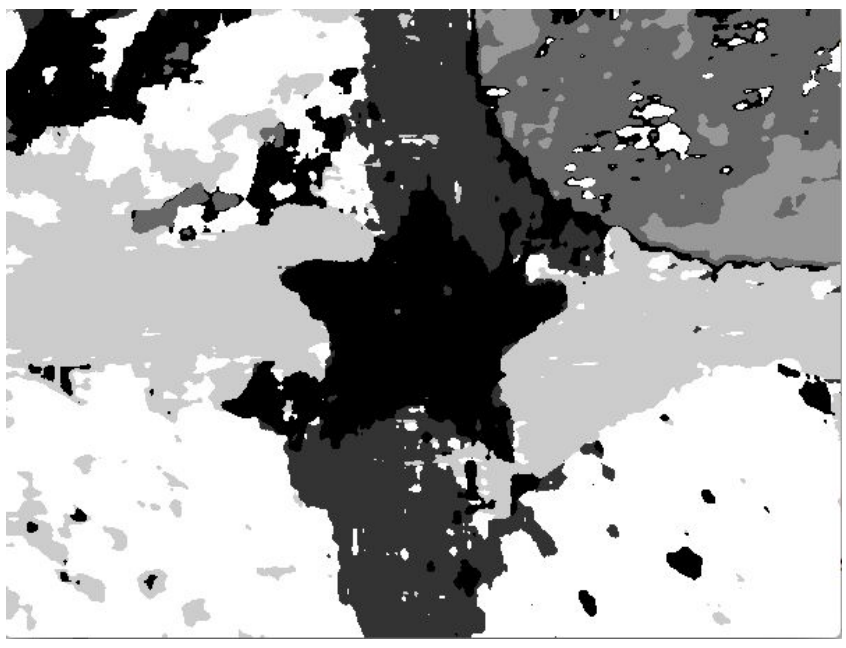


N = 15, iteration = 20

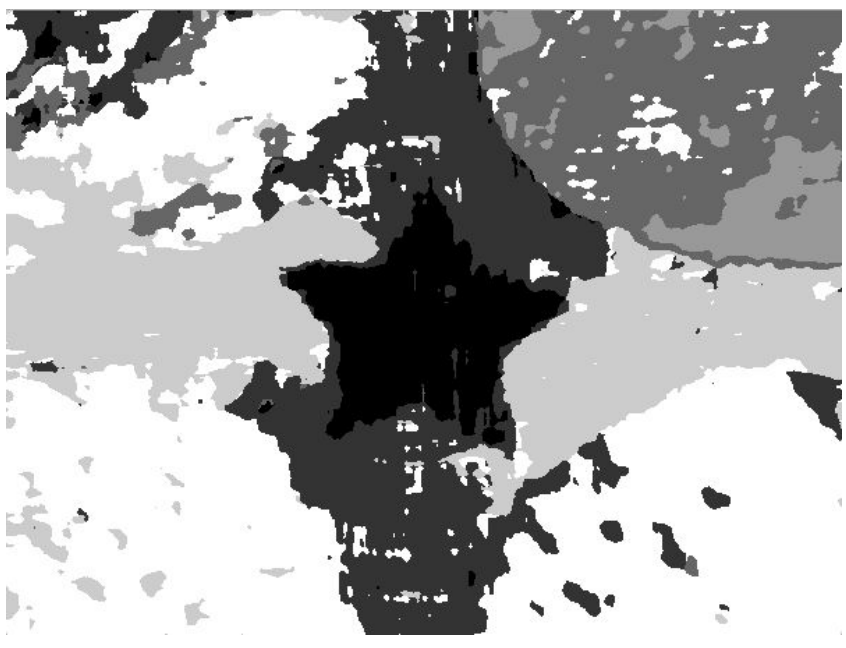


N = 17, iteration = 20

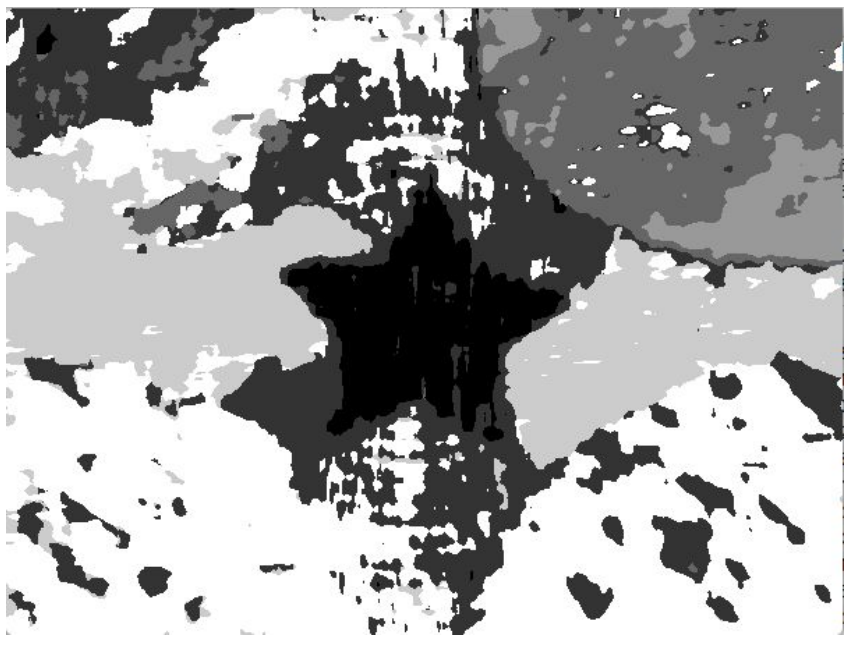
With L3L3^{T} normalization :



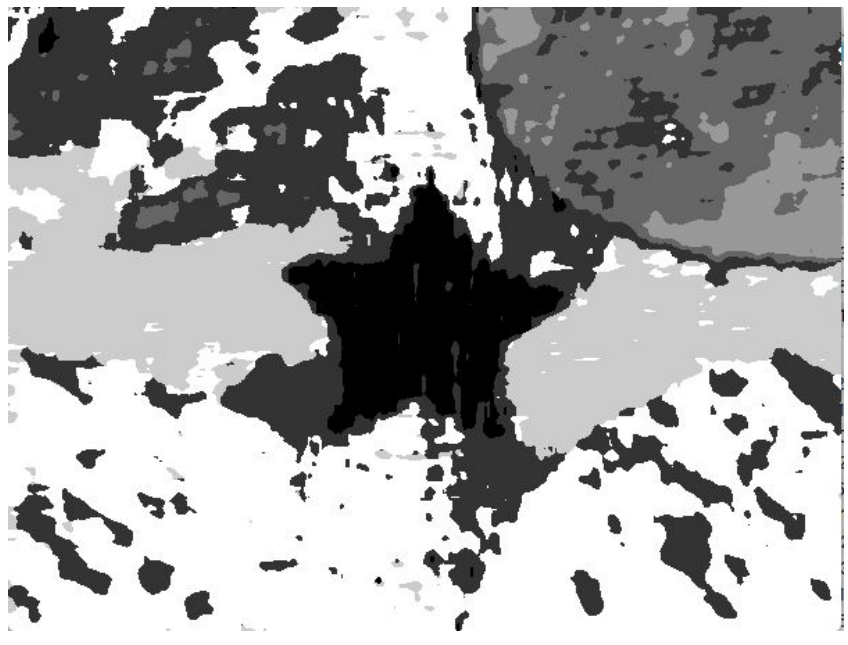
N = 19, iteration = 5



N = 19, iteration = 10

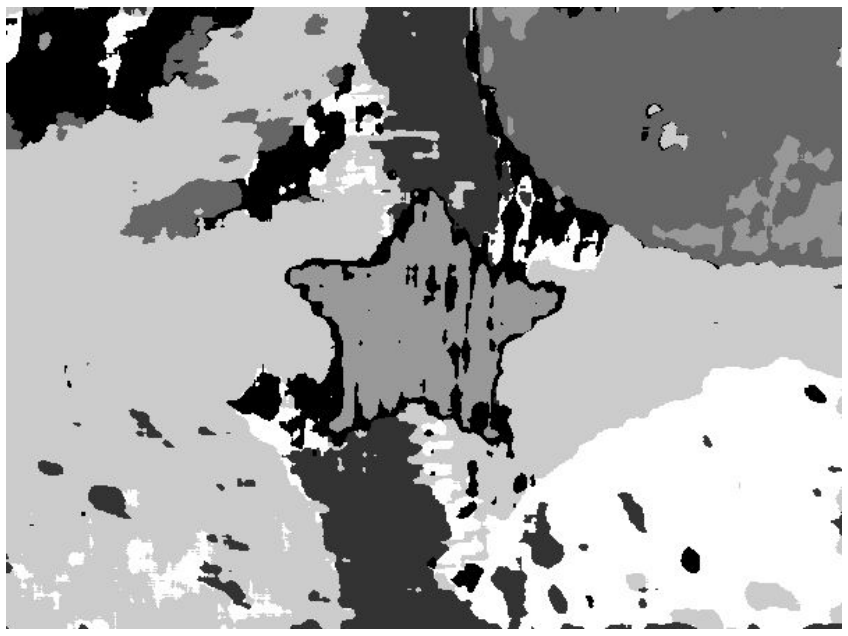


N = 19, iteration = 15

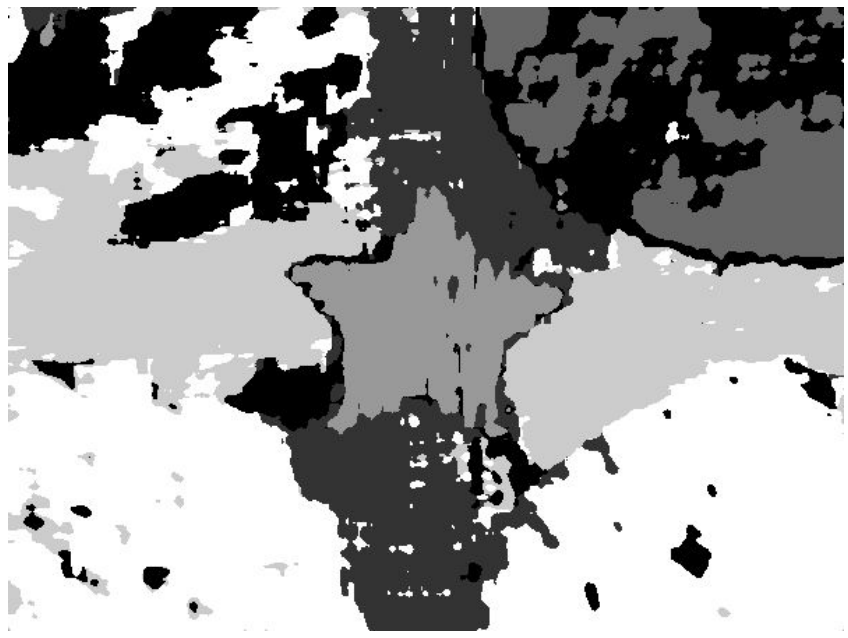


N = 19, iteration = 20

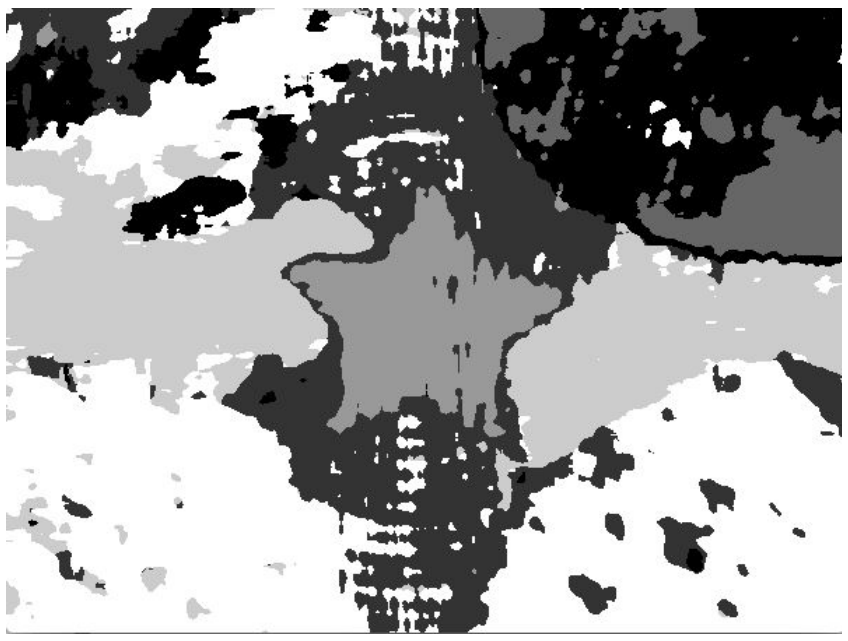
With $L3L3^T$ normalization : N = 21



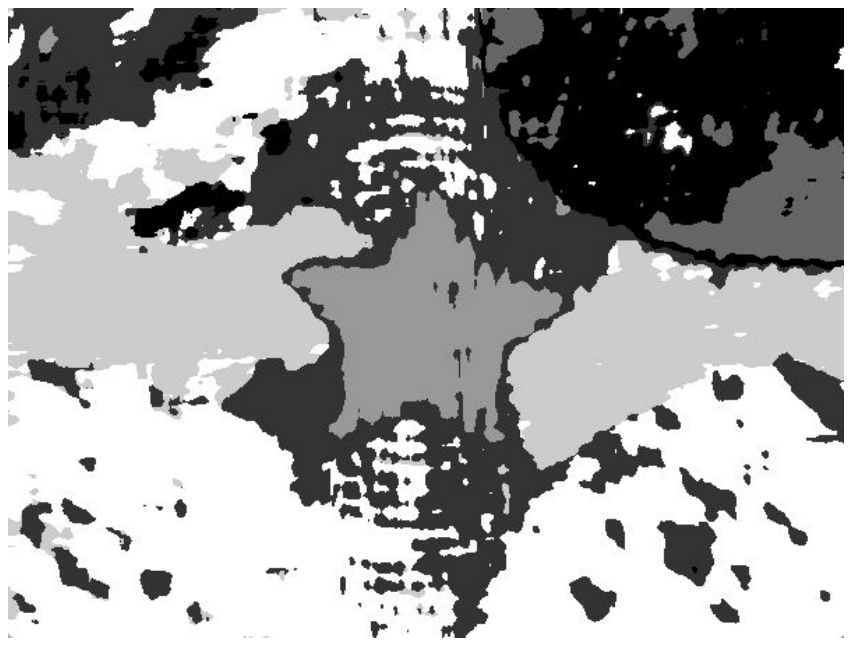
N = 21, iteration = 1



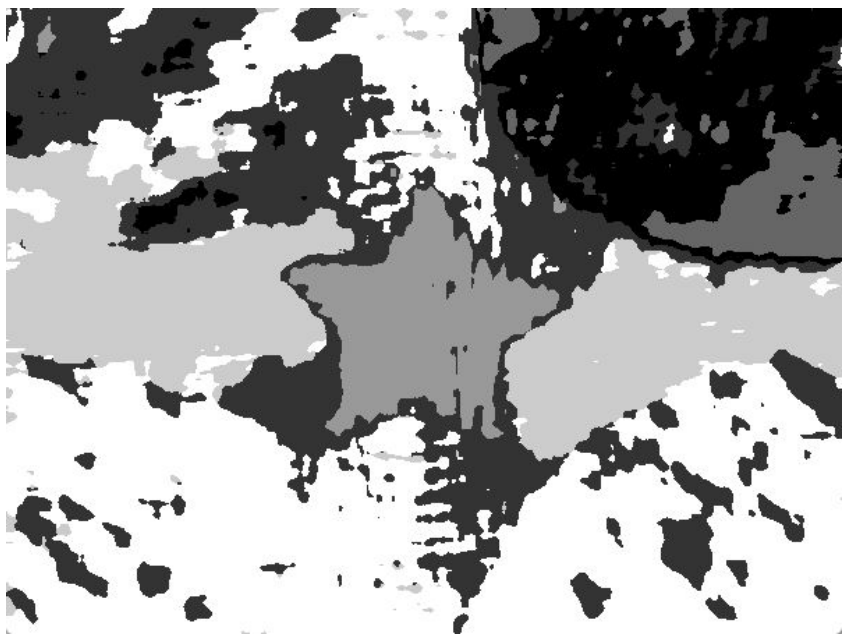
N = 21, iteration =5



N = 21, iteration = 10

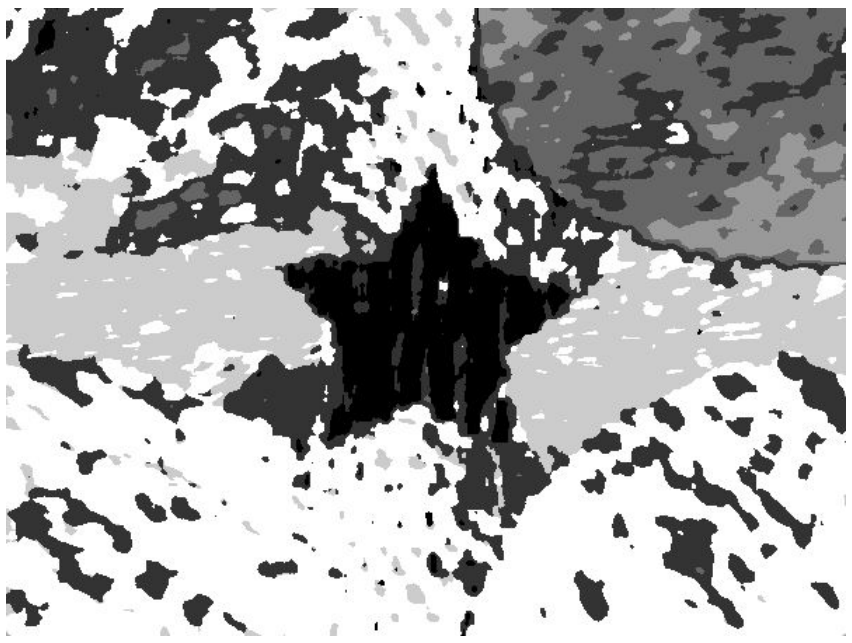


N = 21, iteration = 15

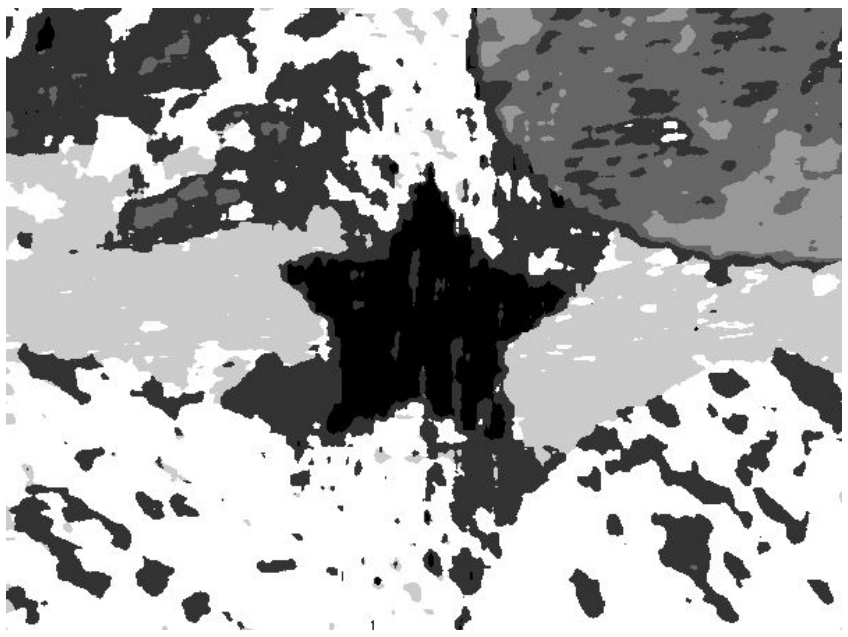


N = 21, iteration = 20

L3L3^T Normalized output : For different window sizes : N= 15, N = 17



N = 15, iteration = 20



N = 17, iteration = 20

1.4 DISCUSSION :

1.2.b. Texture Segmentation

After considering multiple combinations of window sizes and normalized and unnormalized energy vectors, I have come to conclude that the window size of 21x21 and unnormalized energy vector yields the best output as we can see from 1.3.b.(ii). As I changed the window sizes, I observed that for bigger window sizes the output was better. This is in coherence with the window taking in all of its neighbors to compute the energy array. Stark difference was noticed of window sizes in the range 15 - 21.

We can see not so distinct groups at the bottom of the star in the unnormalized outputs. This may be due to the similar values of energy vectors and the k-means algorithm clustering all of them into one group. This can be noticed in the normalized outputs too. The rightmost cluster and the star looks are segmented clearly in both normalized and unnormalized outputs.
I also observed that L3L3^T introduced a lot of outliers. The unnormalized output looks better than the normalized output.

1.2. APPROACH :

1.2.b. Texture Segmentation using PCA

Dimensionality reduction : To reduce my computation time of my program, PCA was used to reduce the dimensions. PCA gives us a set of features in the direction of maximum variance. PCA calculates eigen vectors and eigen values in the higher dimension space and reduces it to a lower dimension. I have considered the reduced dimension space to be 3. I have used MATLAB's PCA function to calculate the principal components.

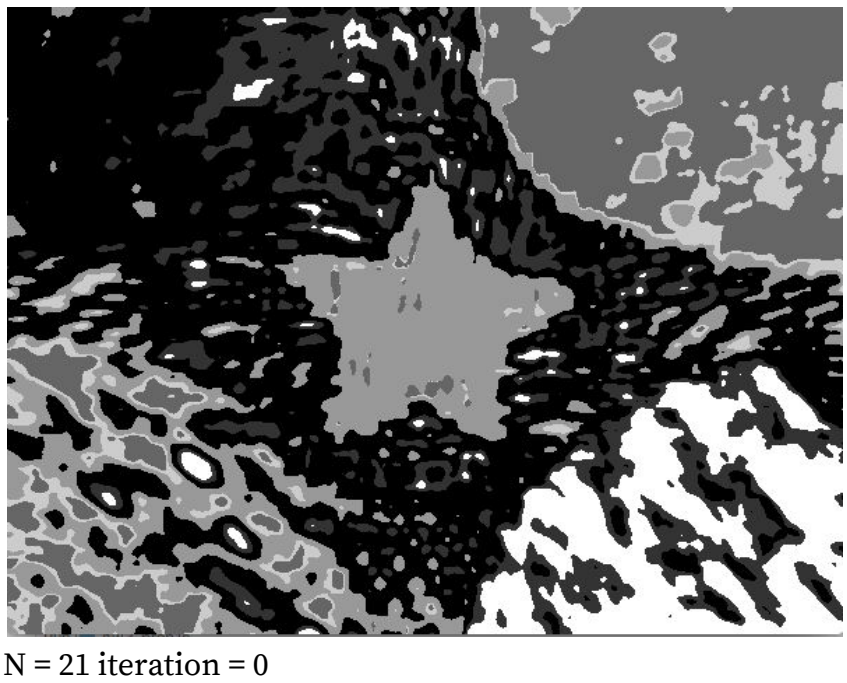
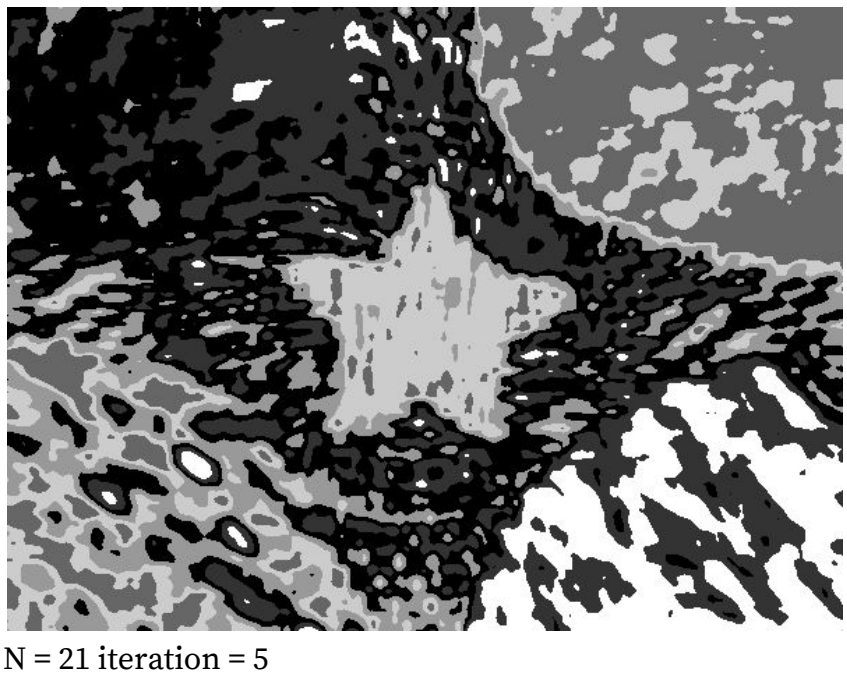
All the 25 5x5 Law's filters were applied on the comb.raw to improve the segmentation result.
PCA gives the principal components along which the data varies with maximum variance. Hence it gives us reduced dimensions.
The number of reduced dimensions obtained was from 25 -> 3.

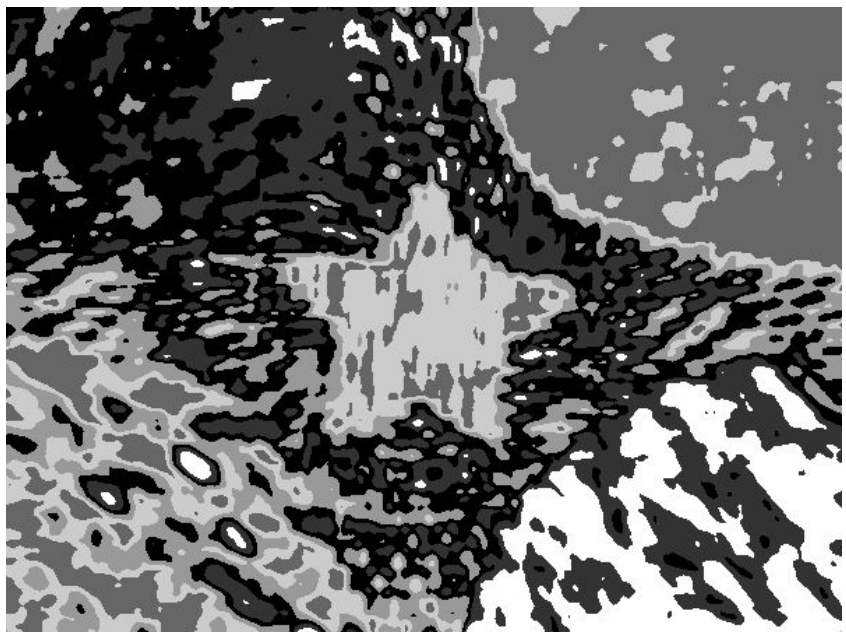
1.2.a.2 ALGORITHM :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Allocate memory for feature array, means array, and energy arrays for the input image.
5. Convert the unsigned char image to double array for easier calculation of pixels.
6. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
7. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
8. Call the 'texture_analysis ' function for the input image.
9. Find the current pixel location by (row*width) +col.
10. Calculate the tensor product for the all given pairs 5x5 Laws filters. We will obtain 25 such filters.
11. Remove DC component for the input image by calculating a local mean of window size 21x21 and updating it to the same image
12. Obtain the nearest neighborhood pixels of the double input image by extension of the image by pixel padding.
13. Obtain the nearest neighbors and multiply it with the 5x5 Law's filter and sum it up. Return the summed value to be the value at the current location of the image.
14. Square the summed value and divide it by the neighborhood area to calculate the feature energy for that current pixel location for the corresponding filter.
15. Repeat steps 11 - 13 for through the height and width of the image.
16. In the end, we will have a populated feature array of size widthxheightx25.
17. Write the obtained feature vectors to a file.
18. In MATLAB, pick up the file and perform PCA, by choosing of reduced dimensions to be 5. We finally obtain a feature matrix of size widthxheightx5 from the initial feature matrix of size widthxheightx25. Write the obtained feature matrix to a file.
19. In C++, pick up the PCA reduced dimensions file, and perform K-means algorithm by randomly initializing centroids.
20. Apply the k-means clustering algorithm to the obtained feature array with the initialized mean array.
21. Set the number of iterations for the k-means to run to be 20.
22. Run the loops for the length of the feature array and the number of classes chosen. Here the number of classes = 6. Every class is depicted by its color. The colors chosen here are - {0,51,102,153,204,255}.
23. Initialize minimum distance for the particular input to be 0.
24. Calculate the Euclidean distance between the current feature vector and the means.
25. Compare the Euclidean distance of the current feature vector with all the means.
26. Obtain the minimum distance and update the minimum distance variable.
27. Get the corresponding class to which the feature vector has minimum distance.
28. Repeat steps 19-13 through the length of feature vectors and means.
29. Update the means by calculating the means of features which are now classified to the class. Hence at the end, we will have new means.
30. Go to the next iteration.
31. Repeat steps 19-27 for the number of iterations set.
32. In the end, we will have the class labels to which every input current pixel location belongs to.

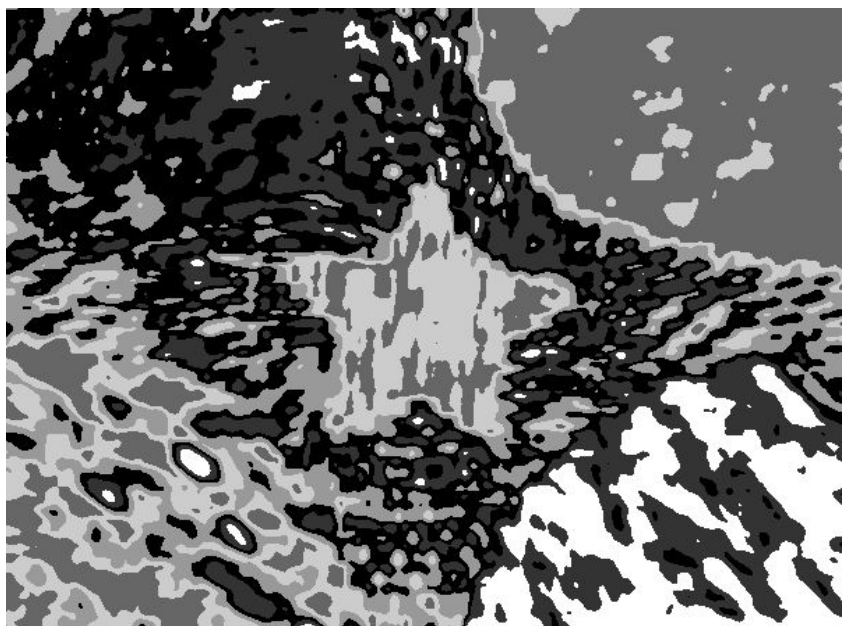
1.3 RESULTS :

1.3.c. Texture Segmentation using PCA

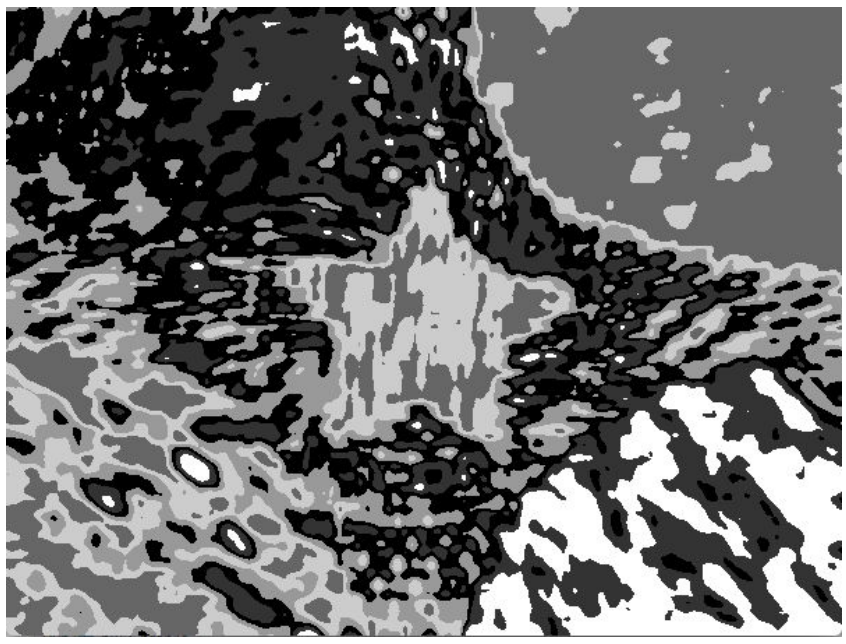
<table>
<tr>
<td> 
N = 21 iteration = 0</td>
<td> 
N = 21 iteration = 5</td>
</tr>
</table>



N = 21 iteration = 10



N = 21 iteration = 15



N = 21 iteration = 20

1.4 DISCUSSION :

1.4.c. Texture Segmentation using PCA

As the above methods of texture segmentation produced unsatisfactory outputs, I tried the dimensionality reduction algorithm - PCA. It reduced the computation time but did not improve the quality of the output significantly.We can still observe that the clusters at the bottom of the star are not classified distinctly. I presume that in PCA by arranging pixels as a 1D vector of its height and width, we are losing out on relations with the neighboring row pixels.
A small change in any calculation of the energy vector can lead to a drastic change in the eigen representation of the image, and thus PCA is also affected.

I have considered a 3 dimension to apply k means. The algorithm computed faster compared to 25 dimensions.

Other post processing techniques which were considered were graph cuts which solves the problem of image segmentation. It considers the flow of the energy, to maximize flow in a graph. The disadvantages of graph-cuts was that it introduced unwanted artifacts such as “blockiness”. It is also biased towards producing a small contour. The usage of memory also increases drastically s and when the image size increases.

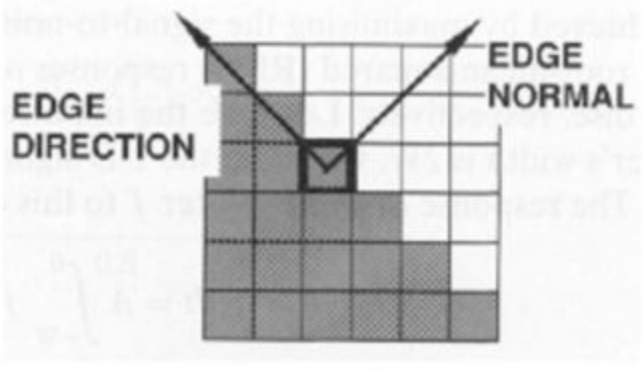
Another technique considered was belief propagation, which produced an optimized graphical probability model and produces exact number of tree structured graphs. It has technical issues such as storage needs to be larger as the image size increases and needs an increased dimensionality of variables.

PROBLEM 2: EDGE DETECTION :

2.1 MOTIVATION :

2.1.a. Basic Edge Detector

Edge detection is one of the most common techniques in image processing to extract important features in an image. Edges are characterized by strong intensity variation in a region. It is usually picked up by a jump in intensity from a pixel to its neighbor. It usually appears in the demarcation between an object and its background, or two different regions in an image.
An edge can be described using its components. They are -



1. Edge normal - a unit vector which points in the direction of maximum intensity change.
2. Edge direction - unit vector perpendicular to edge normal
3. Edge center - the pixel at which we are trying to check for an edge
4. Edge strength - local image intensity variation along the normal

As edges are described as a sudden variation in the intensity change along the normal, we can use the derivatives to check for an edge. Edges can be detected by -

1. Using the first derivative and checking for local maxima or minima
2. Using second derivative and checking for zero crossings.

The features extracted by applying various edge detection algorithms can be used to perform other computer vision tasks such as object recognition.

2.2 APPROACH :

2.2.a. Basic Edge Detector

1) **SOBEL EDGE DETECTOR**

As we consider that edges occur at the points of intensity variation, we can take the first derivative of the intensity function and find the gradient. If we find the maximum of the derivative function, we can locate an edge. Sobel Operator uses the gradient change as the basis of the edge detection rule.

$$\Delta x = \frac{f(x+dx,y)-f(x,y)}{dx}$$

$$\Delta y = \frac{f(x,y+dy)-f(x,y)}{dy}$$

where Δx *and* Δy are along the x and y directions respectively.

As image is a discrete quantity, we can consider Δx *and* Δy in terms of pixel distance between two points. To find intensity variation between the adjacent pixels, then

$$\Delta x = f(i+1,j) - f(i,j)$$
$$\Delta y = f(i,j+1) - f(i,j)$$

The Sobel Operator masks are :

$$\Delta x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \Delta y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Δx gives us the change in the horizontal direction and Δy gives us the change in the vertical direction.

Consider Gx and Gy as the original image's horizontal and vertical derivative components, then it is calculated by :

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} * A \quad \text{and} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * A$$

Where A is the original source image.

The magnitude measure helps us to determine if there is a sudden change in the gradient measure. The magnitude is given by :

$$G = \sqrt{G_x^2 + G_y^2}$$

and the direction of gradient change is given by :

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

1.2.a.2 ALGORITHM FOR SOBEL EDGE DETECTION :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Convert the RGB image into GrayScale for edge detection using the formula : I(x,y) = 0.21R + 0.72G + 0.07B
5. Run two for loops through the height and width of the image. Calculate the current location using the formula - (row*width)+col
6. Get the 8 connectivity nearest neighbors of the current pixel location (so we end up with a 1D 9 elements array).Pixel padding was done according to the size of the filter by replication.
7. Use the gradient x and gradient y Sobel operators given above. Multiply with the 9 1D elements and cumulatively sum them.
8. Use the formula given above to find the magnitude.
9. The maximum and minimum values were obtained from the loops and they were used to normalize the pixel intensity at the location using the above formula
10. Repeat steps 6 - 9 through the height and width of the image.
11. Run two for loops through the height and width of the image.
12. Obtain the histogram and the cdf of the output image. Create a sorted array to have the intensity and the current pixel location.
13. I have chosen my threshold to be 90% of the transfer function (cdf). As I have arranged my pixels in the increasing order of intensity, I set my threshold to be the intensity value in the location 90th % of my sorted address array. If the intensity value at the pixel location is less than the threshold copy a black pixel value else copy a white pixel value.
14. Recalculate the intensities in the current pixel locations from the sorted Intensity and address array.
15. Write the image to a new file and check output.

2) LAPLACIAN OF GAUSSIAN (LOG)

Laplacian makes use of the 2nd spatial derivative of an image. It highlights sudden intensity change and hence can be used to detect edges. As second derivative components are sensitive to noise, the image is pre-smoothened using a Gaussian filter.
This two step procedure gives us LoG filter.

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Kernels which can be used to approximate a Laplacian are -

0	−1	0
−1	4	−1
0	−1	0

−1	−1	−1
−1	8	−1
−1	−1	−1

To make the computation faster, as convolution is associative, Gaussian kernel is first convolved with Laplacian kernel as they are usually much smaller than the image and then applied.
LoG kernel is precalculated so only one convolution is performed at run-time.

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right]e^{-\frac{x^2+y^2}{2\sigma^2}}$$

An LoG kernel with Gaussian $\sigma = 1.4$ is given below :

0	1	1	2	2	2	1	1	0
1	2	4	5	5	5	4	2	1
1	4	5	3	0	3	5	4	1
2	5	3	-12	-24	-12	3	5	2
2	5	0	-24	-40	-24	0	5	2
2	5	3	-12	-24	-12	3	5	2
1	4	5	3	0	3	5	4	1
1	2	4	5	5	5	4	2	1
0	1	1	2	2	2	1	1	0

As LoG calculates second derivative of an image, in regions where there is constant intensity, the LoG response is zero. When there is a sudden change of intensity, the LoG response is positive on the darker side and negative on the lighter side. Hence, concept of zero-crossing is used here.

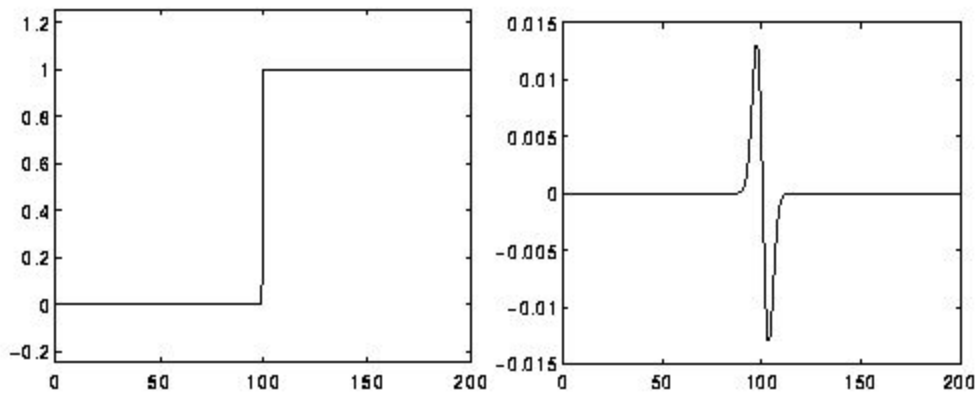


Figure 4 Response of 1-D LoG filter to a step edge. The left hand graph shows a 1-D image, 200 pixels long, containing a step edge. The right hand graph shows the response of a 1-D LoG filter with Gaussian $\sigma = 3$ pixels.

Source : https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

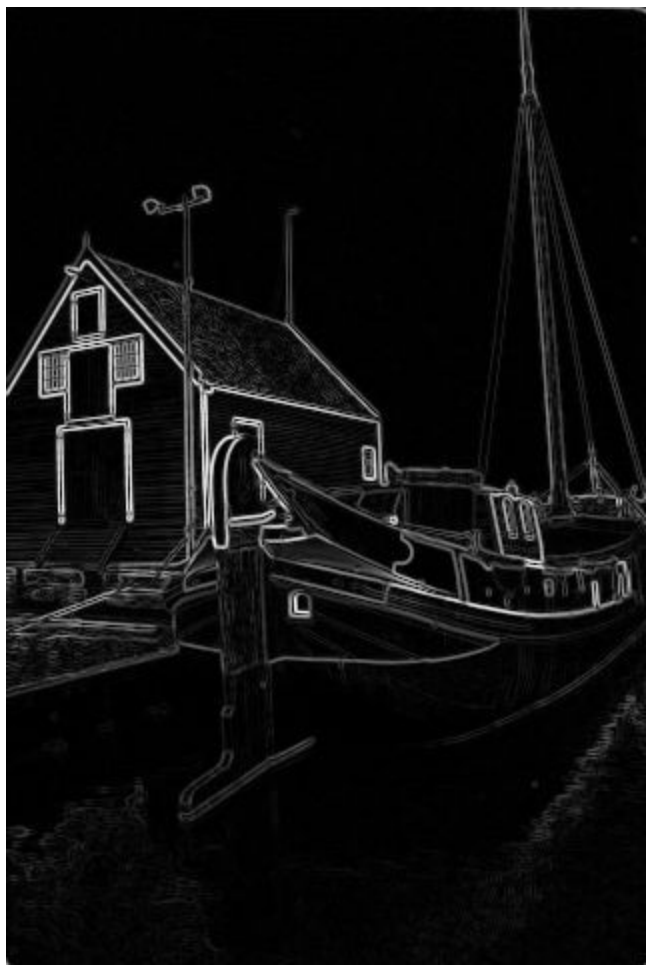
1.2.a.2 ALGORITHM FOR LOG EDGE DETECTION :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Allocate memory for magnitude array of image size and datatype double.
5. Convert the RGB image into GrayScale for edge detection using the formula : I(x,y) = 0.21R + 0.72G + 0.07B
6. Run two for loops through the height and width of the image. Calculate the current location using the formula - (row*width)+col
7. As the LOG filter is of size 9x9, we get the 9x9 neighbors of the current pixel location. Pixel padding was done according to the size of the filter by replication.
8. Apply the 9x9 LOG filter to the obtained neighborhood pixels. Multiply the filter value with the neighborhood values and sum them cumulatively. Save them in the magnitude at the current location.
9. Allocate memory for Normalized array to store the normalized values of the magnitude of size image size and datatype int.
10. Calculate the minimum and maximum occuring in the magnitude array and use that to normalize the magnitude array.
11. Obtain the frequency of intensities from the normalized array and plot it. Find the knee points in the plot and place the thresholds accordingly.
12. Allocate memory for threshold image of type unsigned char and threshold array for zero crossings.
13. The chosen thresholds are 122 and 156.
14. If the normalized value at a pixel location is less than 122, set the intensity value as 64 and corresponding value for zero crossing as -1. If the pixel location value is between 122 and 156, set the intensity value as 128 and corresponding value for zero crossing as 0. If the normalized value at a pixel location is greater than 156, set the intensity value as 192 and corresponding value for zero crossing as 1.
15. Now to detect if it is an edge, obtain the 3x3 nearest neighbors from the threshold from zero crossing array and check for zero crossing.
16. If the value at the pixel location is 0 and if there is a 1 or -1 in its 8-connectivity, then it is an edge and copy a black pixel value to the location else copy a white pixel value to the location in the output image.
17. If the value at the pixel location is not 0, copy a white pixel location to the output image.
18. Write the image to a new file and check output.

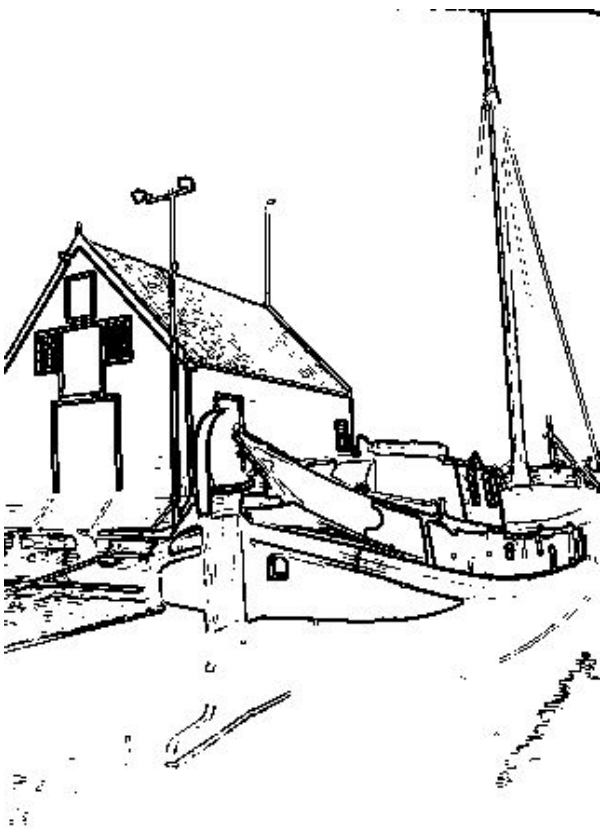
2.3 RESULTS :

2.3.a. Basic Edge Detector

Input - boat.raw



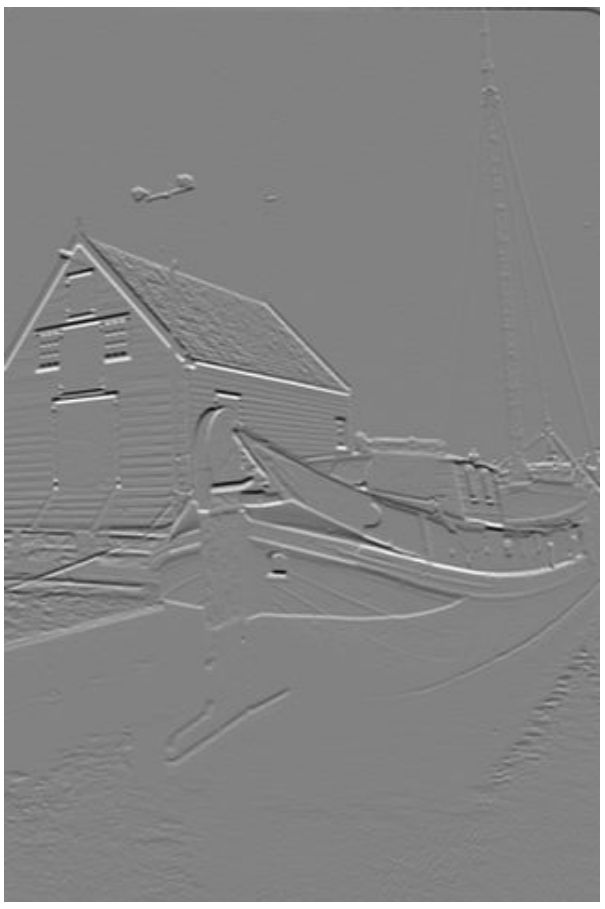
Sobel output before applying threshold - boat.raw



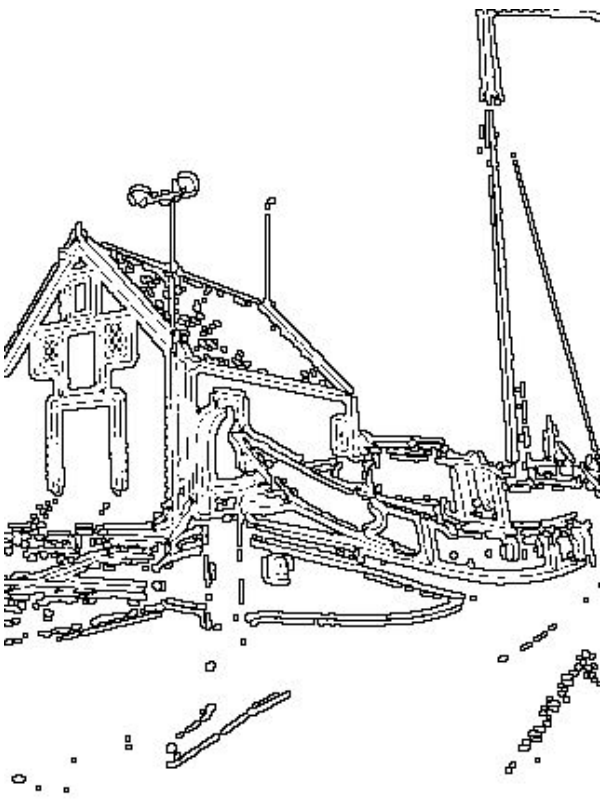
Sobel output after applying threshold - boat.raw

LoG grayscale output - boat.raw

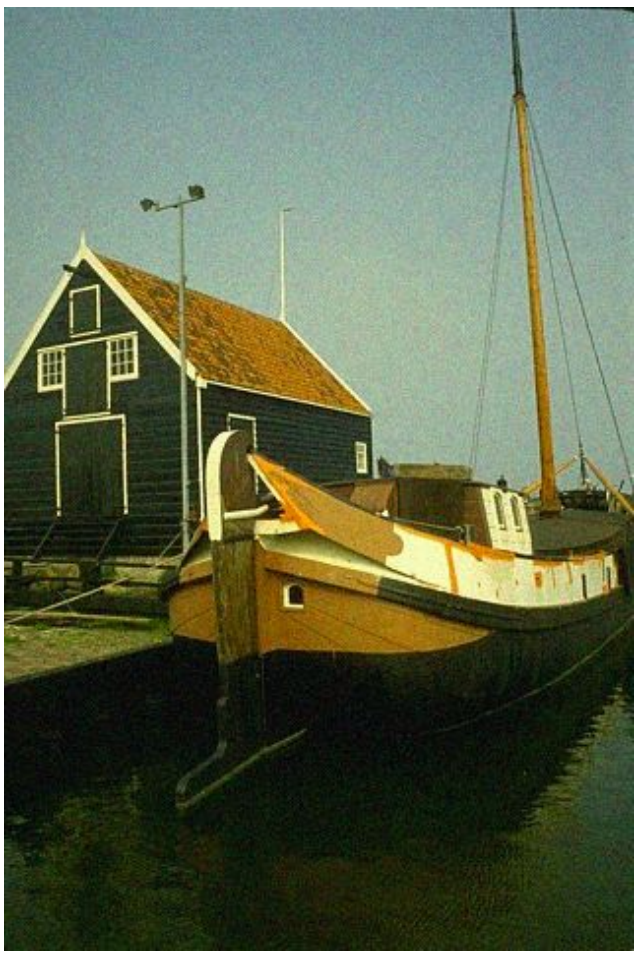
LoG x_gradient output - boat.raw



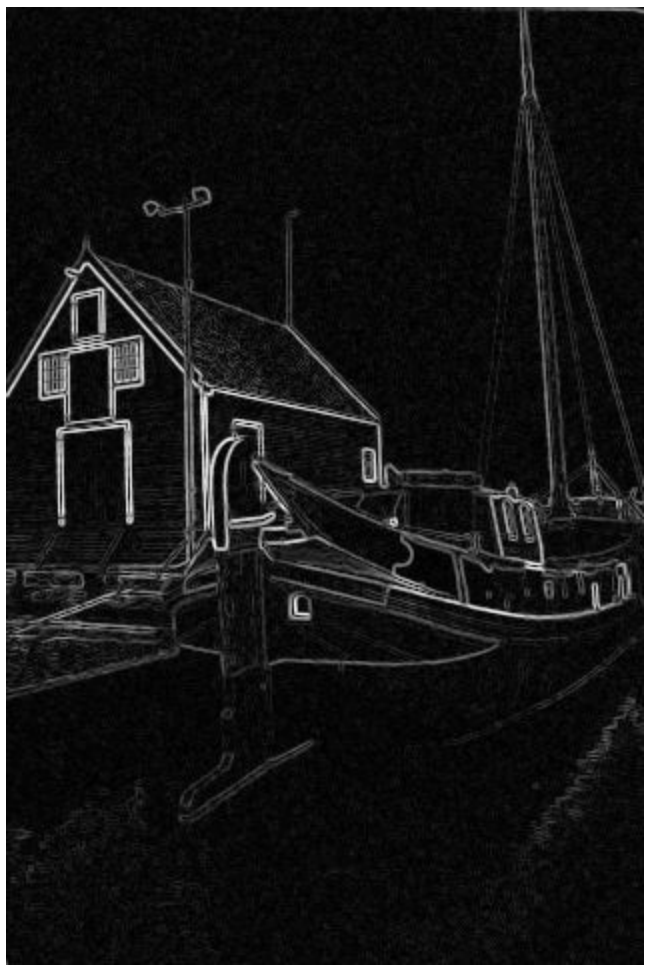
LoG y_gradient output - boat.raw



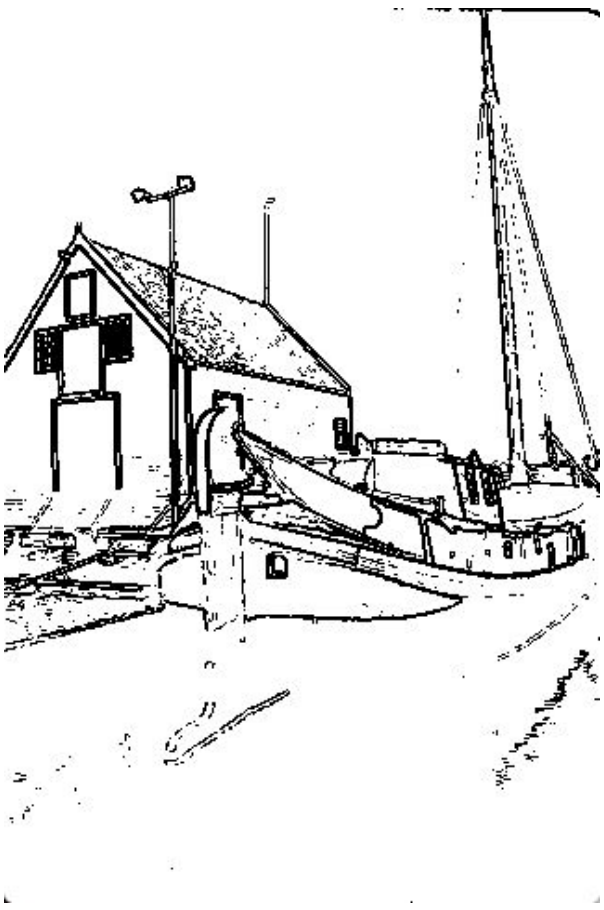
LoG output - boat.raw



Input - boat_noisy.raw

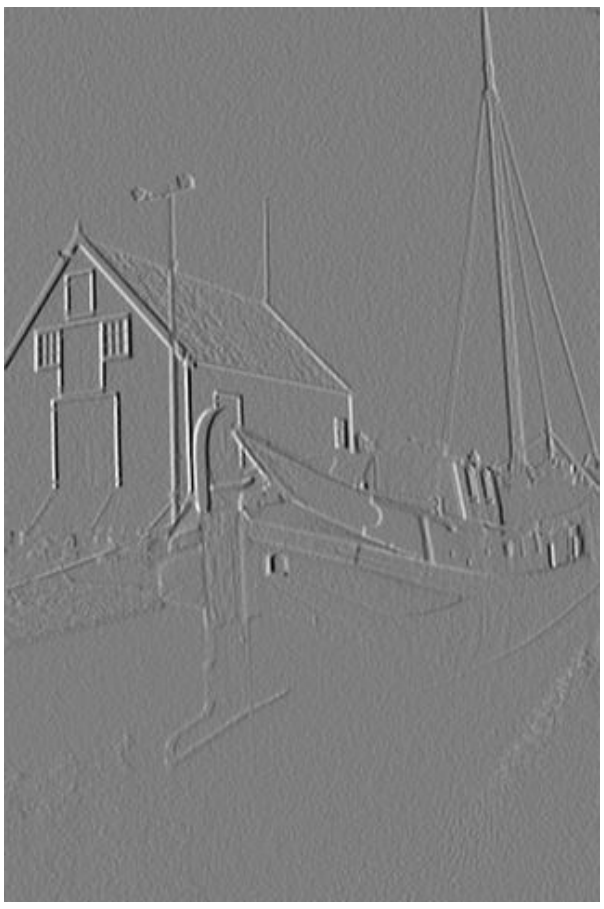


Sobel output before applying threshold - boat_noisy.raw

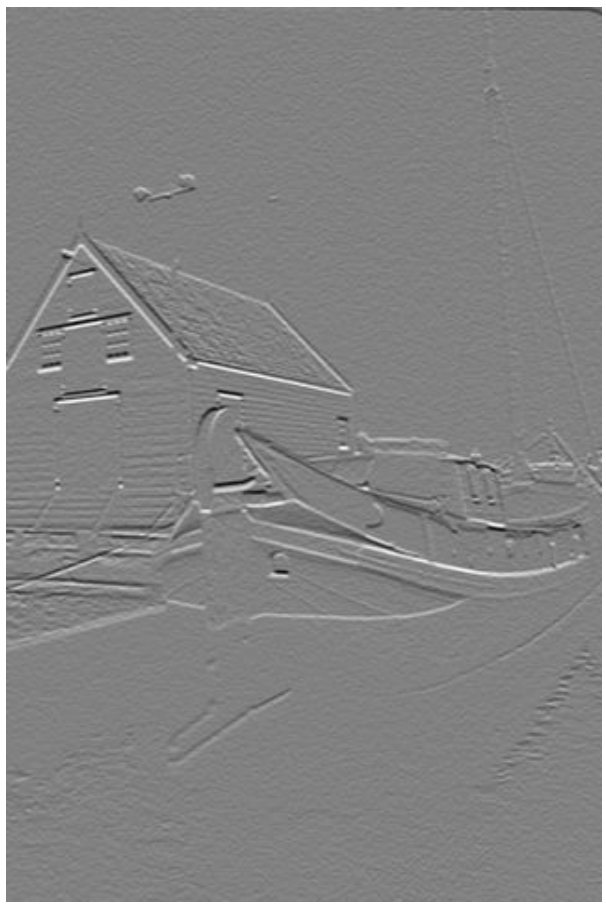


Sobel output after applying threshold - boat_noisy.raw

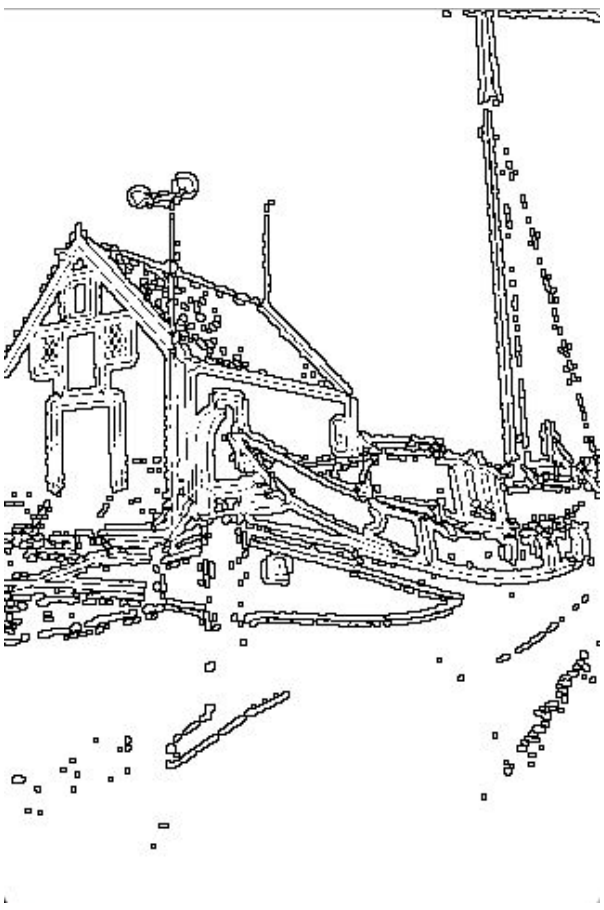
LoG grayscale output - boat_noisy.raw



LoG x_gradient output - boat_noisy.raw



LoG y_gradient output - boat_noisy.raw



LoG output - boat_noisy.raw

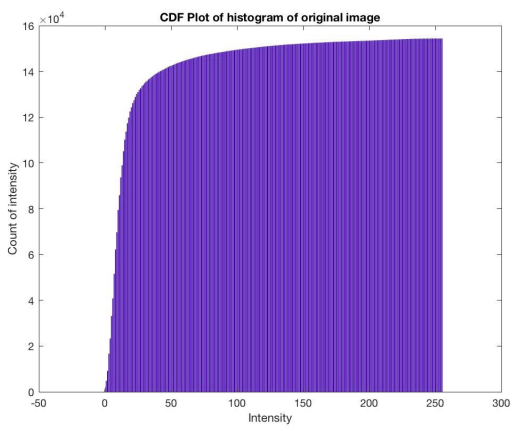
2.4 DISCUSSION :

2.4.a. Basic Edge Detector

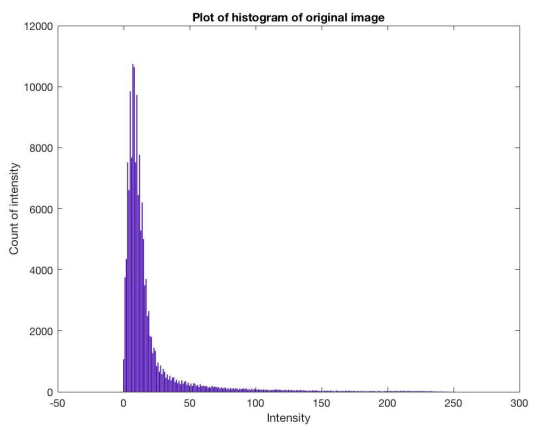
After experimenting with both the filters, the following observations were made :

- Sobel detector sensitive to noise in pictures, hence highlights them as edges.
- NMS is used after Sobel to find gradient and extreme edges.
- In Sobel, it is difficult to find maxima points - Hence LoG was introduced.
- LoG is also desensitized to noise by application of Gaussian smoothing filter.
- LoG is rotation invariant

SOBEL :



Cdf plot

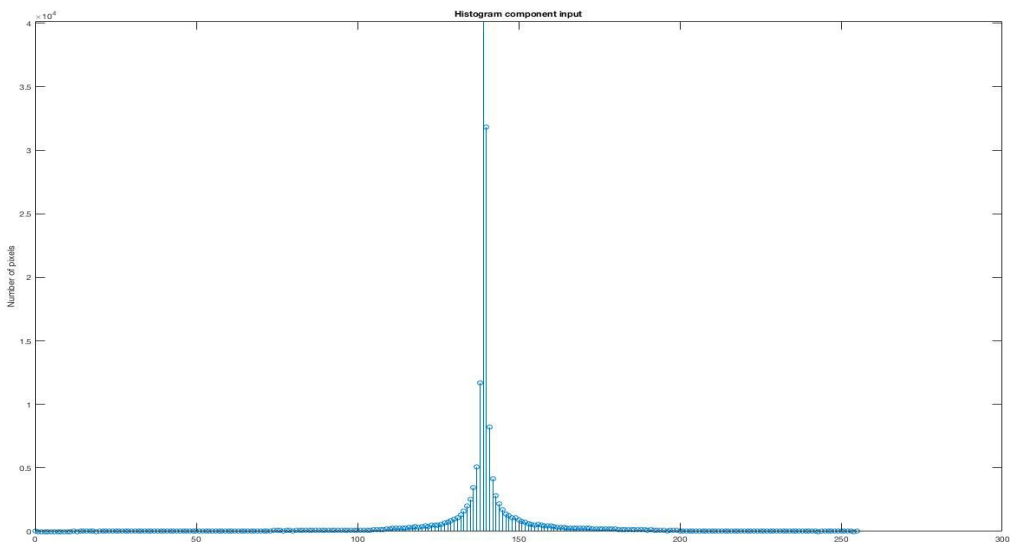


Histogram plot

After the normalization of the x and y gradient, to improve the representation and contrast and to binarze the image, thresholding needs to be set. This binarization shows us the edges distinctly. I chose two thresholds of 80% and then 90% and checked for the better one. The 80% had a lot of unwanted and redundant edges which would not yield any new information. To set the threshold, from the cdf plot, I decided to keep the threshold to 90% of the maximum value reached by the cdf. From the histogram plot, it also gave me an idea to set the threshold. The threshold set was about 37.

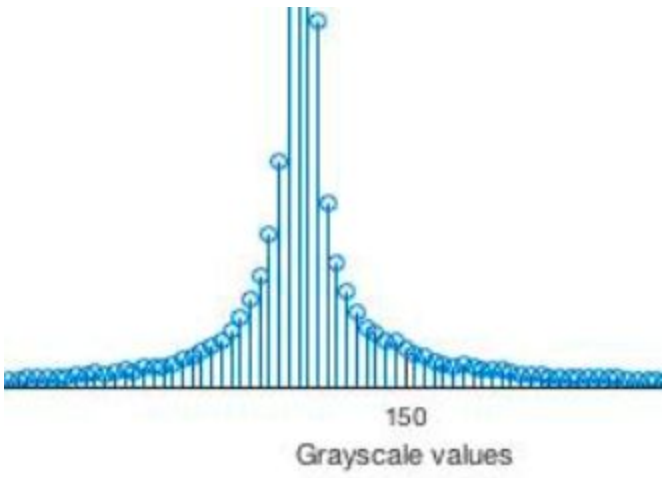
LoG:

To set the thresholds using the knee points for the next step of zero-crossing, I checked the histogram of the normalized values obtained after the application of the LoG filter. The choosing of knee points is a very crucial step as it decides which ones need to be identified as edges. I observed that there was a sudden change at two points in the histogram given below and decided to set them as my knee points.



Histogram for LoG after normalization is shown above.

Algorithm to select knee point:

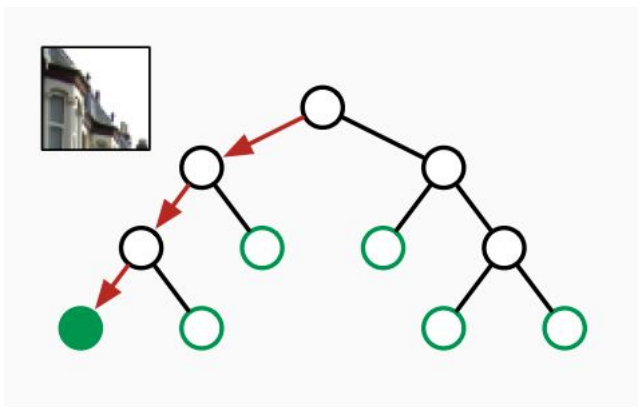


I zoomed into the the histogram of the LoG and from the center of the the peak, I considered the two knee points values to be equidistant from the peak. The Knee points were calculated by determining the rapid increase and decrease in the histogram. This gave us our threshold under which zero crossing would be based and hence determination of a potential edge. This is one of the key steps in the algorithm. The knee points I have considered for LoG was 122 and 156. For zero-crossing - Everything below 122 was set to -1, everything between 122 and 156 was set to 0 and everything greater than 156 was set to 1.

2.2 APPROACH :

2.3.b. Structured Edge

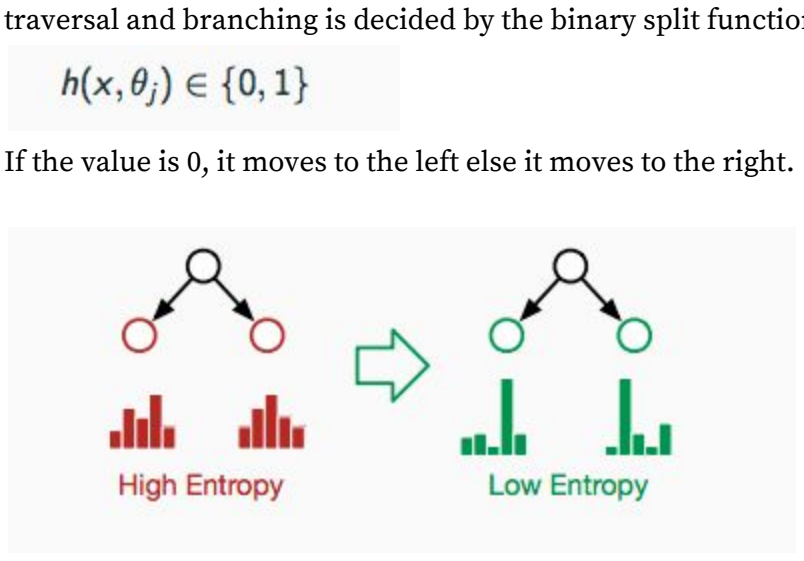
Structured forest is an advanced data driven algorithm to detect edges in an image. This state of the art algorithm set new bars for accuracy and speed for classification and edge detection. This algorithm was curated on the basis that edges have structures. It was devised by Piotr and Zitnik. The algorithm uses Random forests approach for classification and decision trees during the training phase.
Decision tree algorithm is shown below:



A decision tree classifies an input based on recursive calls. It keeps branching down either left or right until it hits a leaf node. The traversal and branching is decided by the binary split function -

$$h(x, \theta_j) \in \{0, 1\}$$

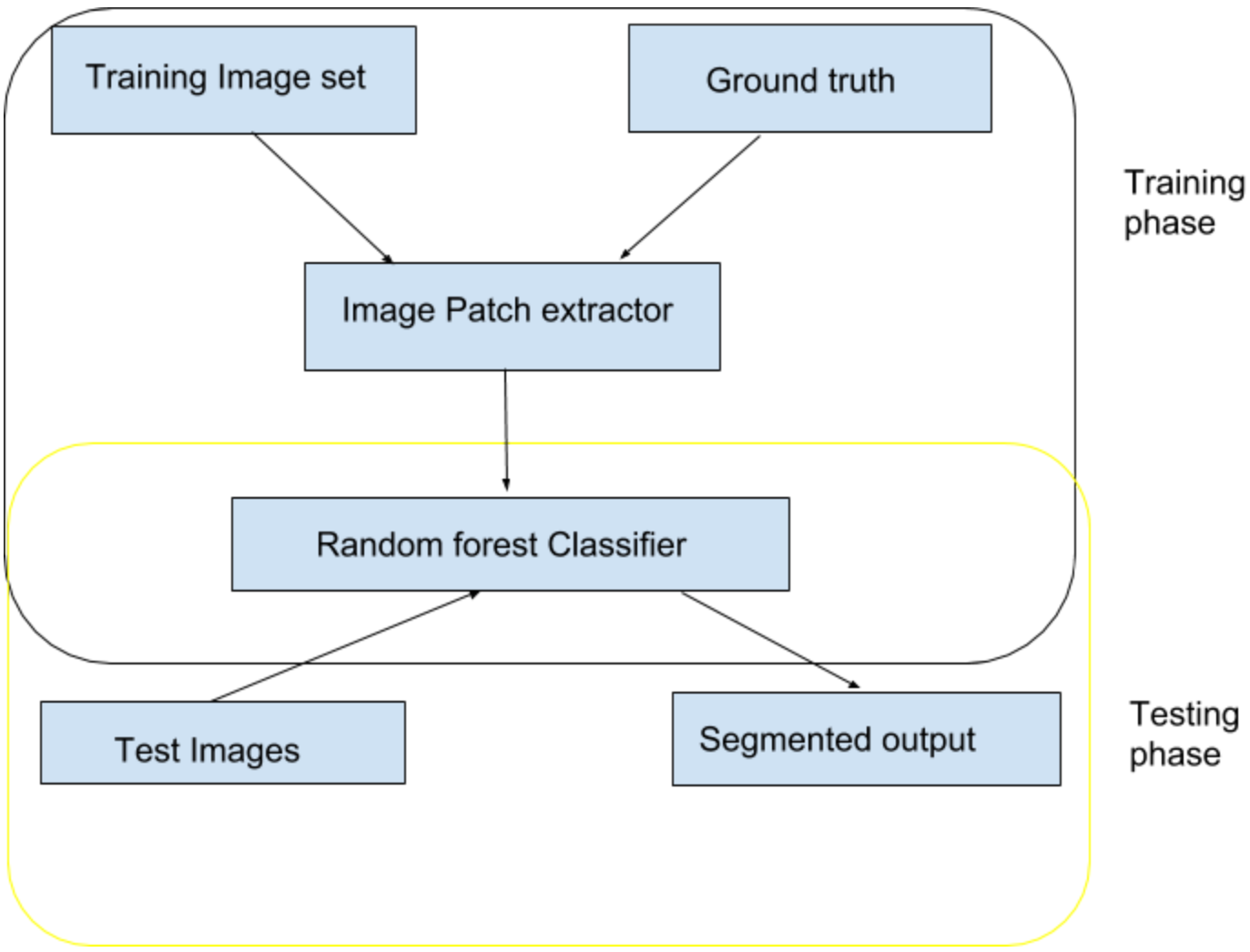
If the value is 0, it moves to the left else it moves to the right.



Each tree is trained simultaneously and without influence of the other to maximize the accuracy and the information gain whereas simultaneously minimizing entropy.

The structured edge detection mechanism considers a patch around the current pixel and determines likelihood of it being an edge or not. The input and the output spaces here are devised by *x* which is the image space and *y* which encodes local image annotation. These sketch tokens which were generated represented a variety of local edge structures like T junctions, parallel lines, straight lines etc.
The primary idea of the algorithm is to map all labels at a given tree node to a discrete set of classes such that similarly structured labels are assigned to the same class.
The learning of random forests is achieved by a divide and conquer mechanism. It consists of multiple random forests and for the image patch to be classified into the tree, it should satisfy a certain threshold. At each stage, the decision threshold is rechecked, if it satisfies, it can iterate through the length of the tree until it hits the leaf node; if not it should move forward. It works like a standard tree structure : where if the computed value is less than the threshold, it moves to the left; lse it moves right. As now the

random forest is trained, it can be tested. The test image is now introduced and because of the accuracy and computation speed, we get a segmented output.
This uses an ensemble approach where multiple trees are trained independently. This results in entropy split at every node and information gain. Thus the output is more coherent and faster.



In the training phase, a structured 16x16 segmentation mask is taken from a 32x32 image patch.
The input feature is os 7228 dimension is computed. The mapping function from space *y to z* is transformed. We reduce dimension of z to 256 and number of clusters to two clusters.
As we consider multiple random forests, the final output is computed using an ensemble approach. It is obtained by averaging of the predictions.
I have used the available online source codes to implement this[4].

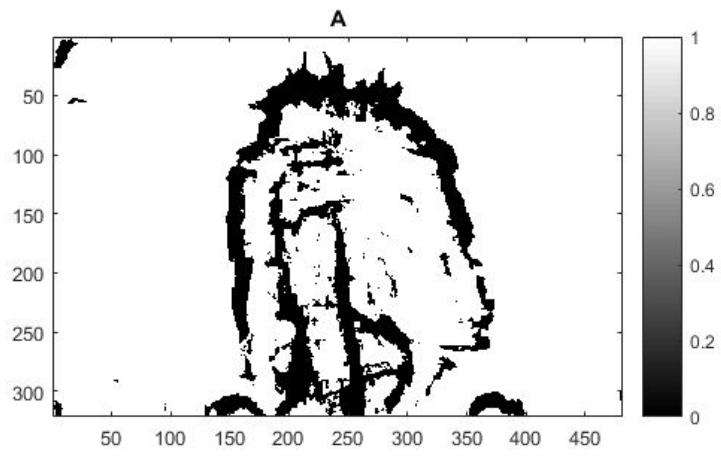
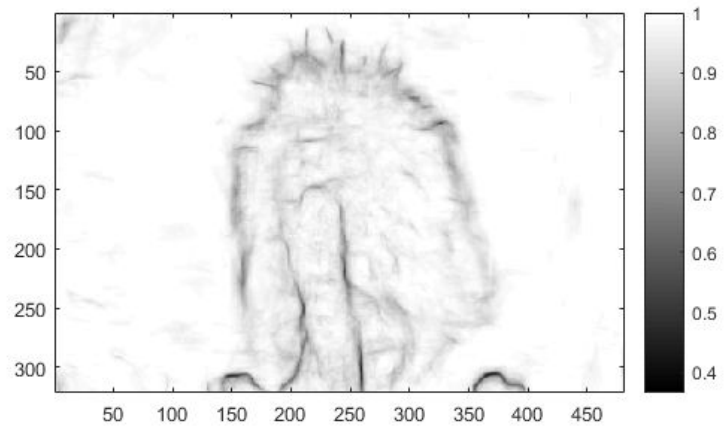
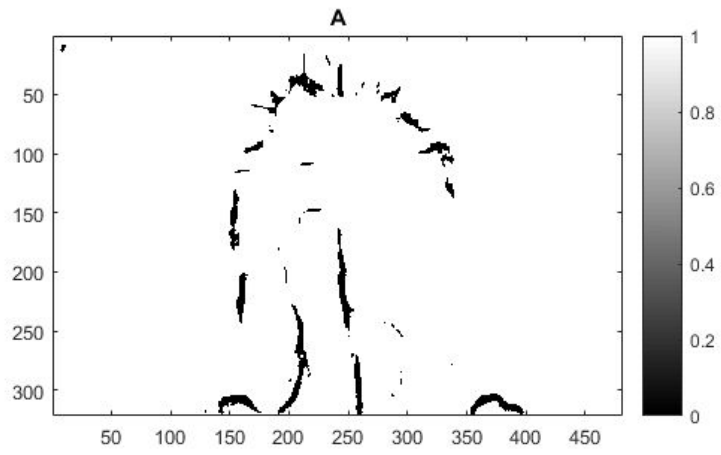
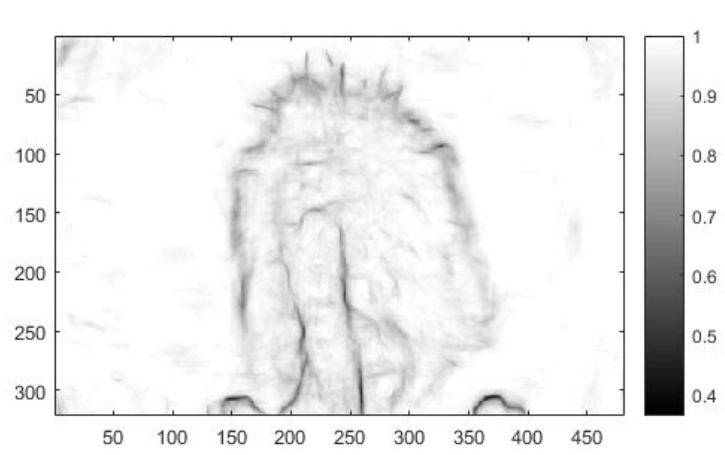
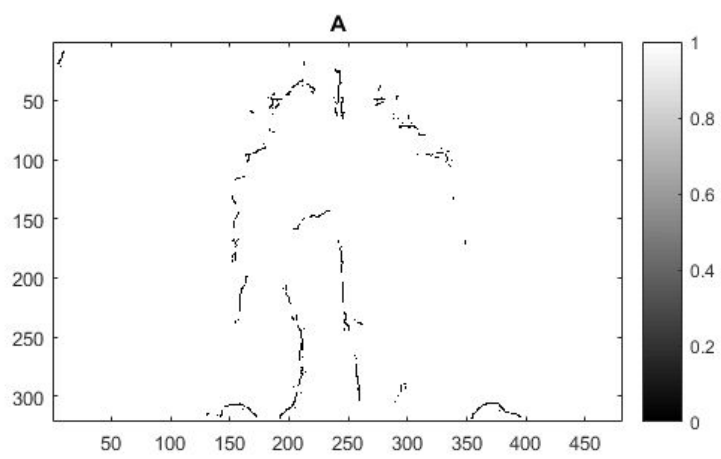
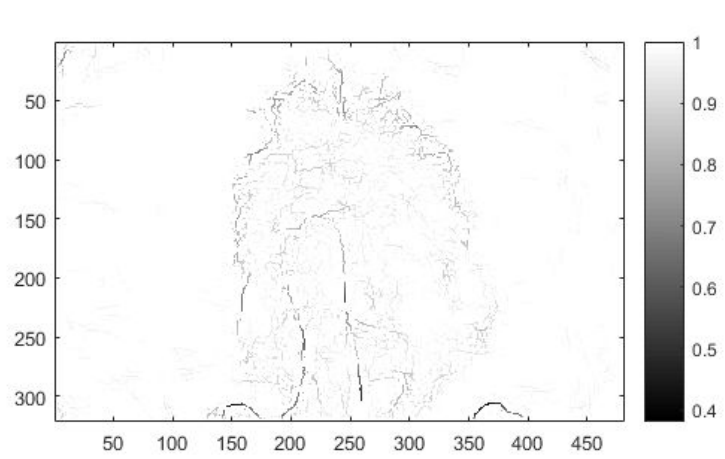
2.3 RESULTS :

2.3.b. Structured Edge

Input:

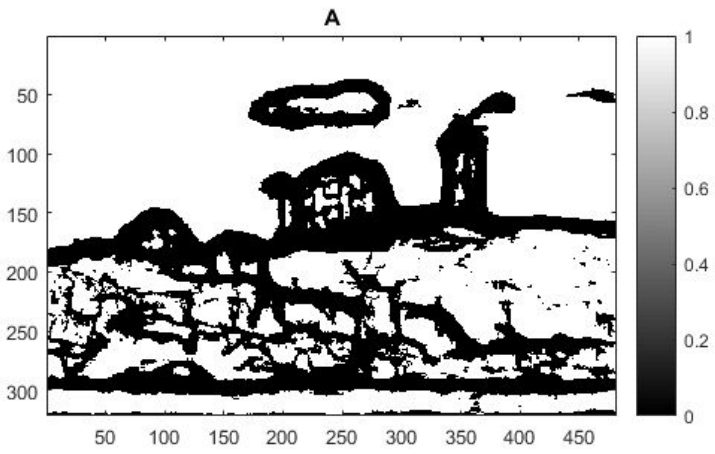
	
Animal.jpg	House.jpg

Binary edge map	Probability edge map

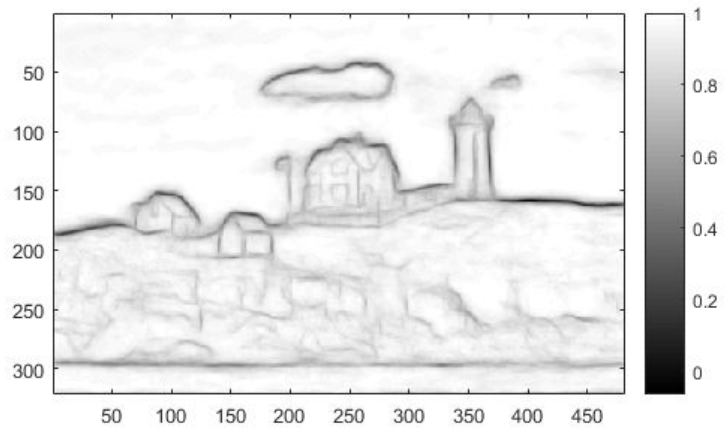
 Threshold = 0.07, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0	 Threshold = 0.07, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0
 Threshold = 0.17, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	 Threshold = 0.17, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1
 Threshold = 0.17, multiscale = 0, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	 Threshold = 0.17, multiscale = 0, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

House:

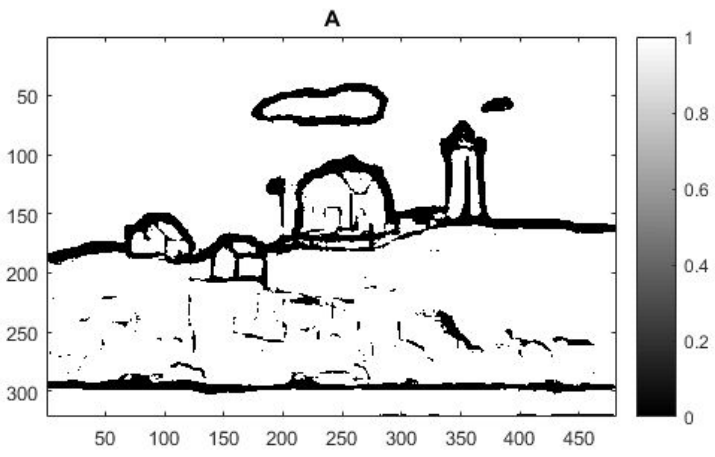
Binary edge map	Probability edge map



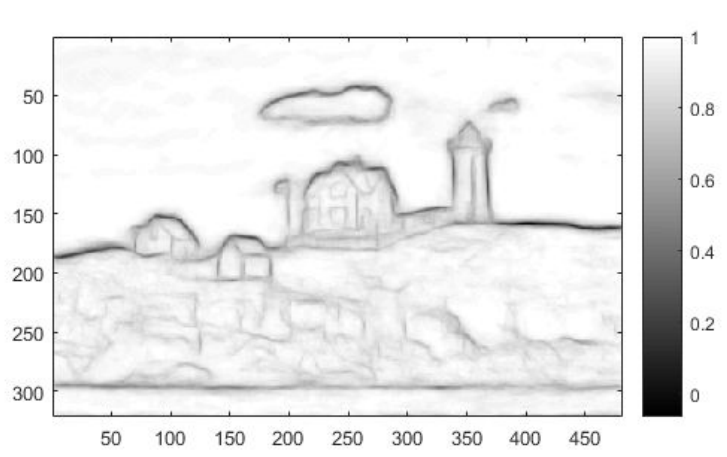
Threshold = 0.07, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0



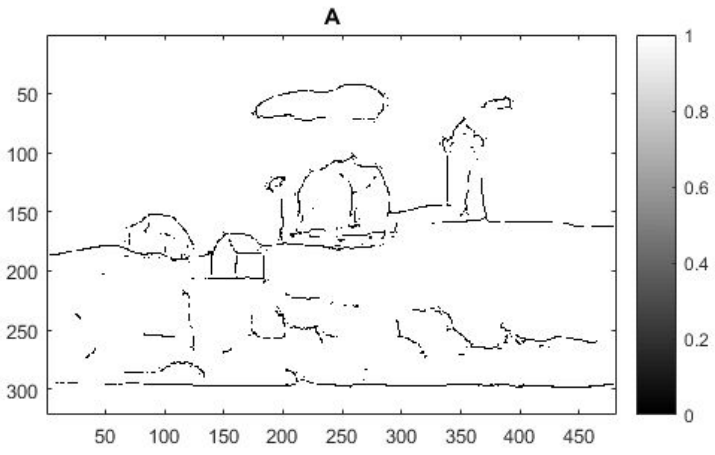
Threshold = 0.07, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0



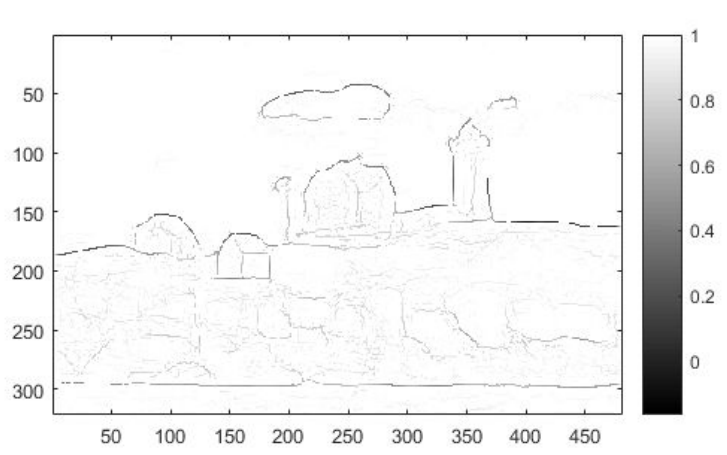
Threshold = 0.17, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0



Threshold = 0.17, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 0

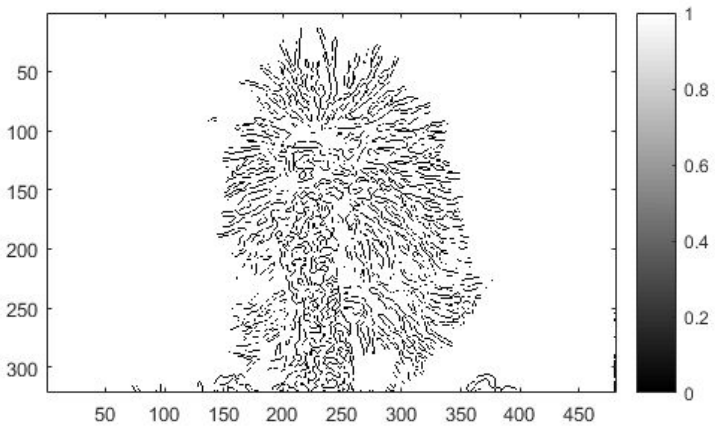
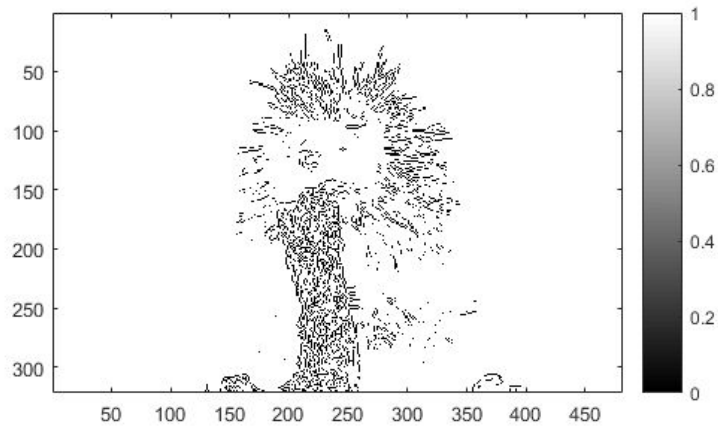
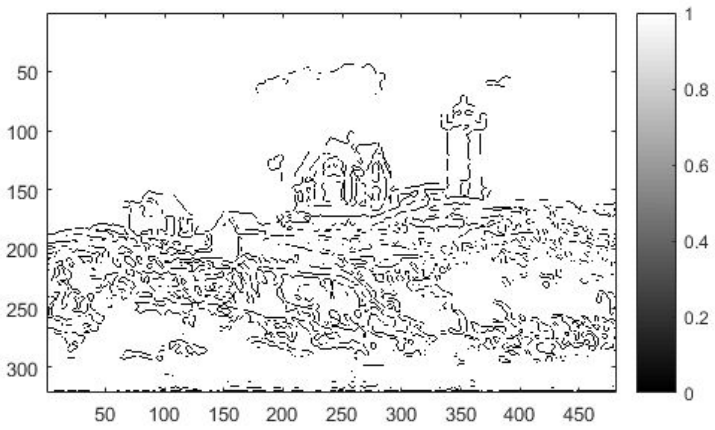
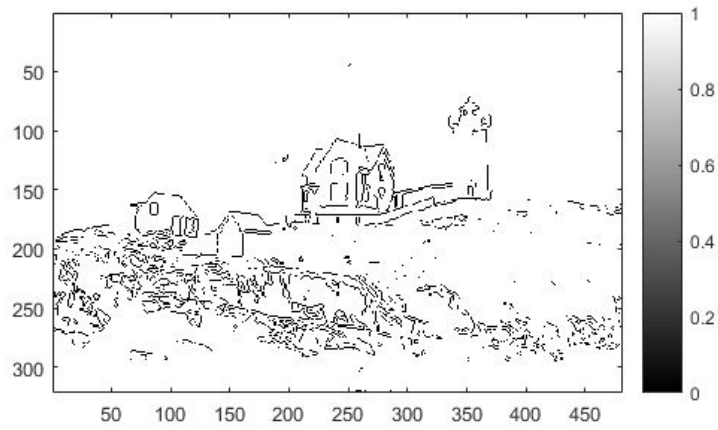


Threshold = 0.17, multiscale = 0, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

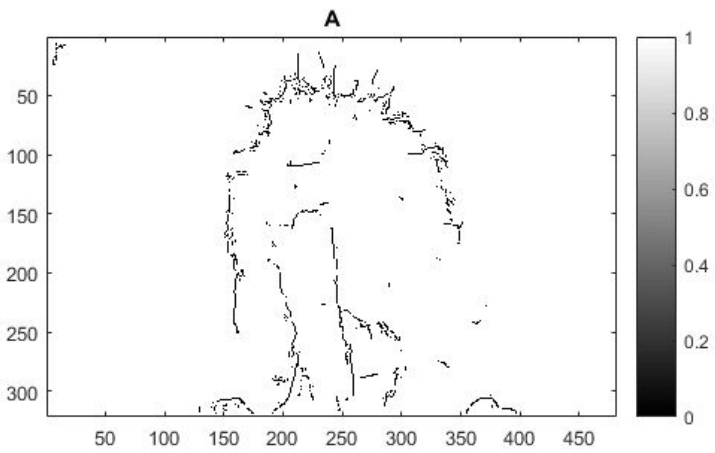
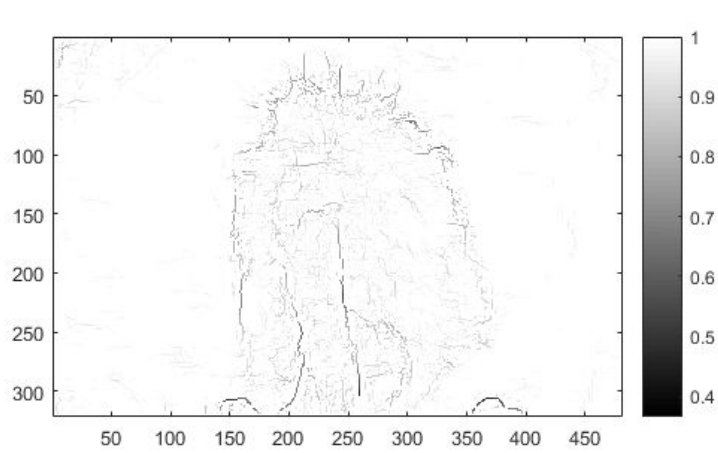


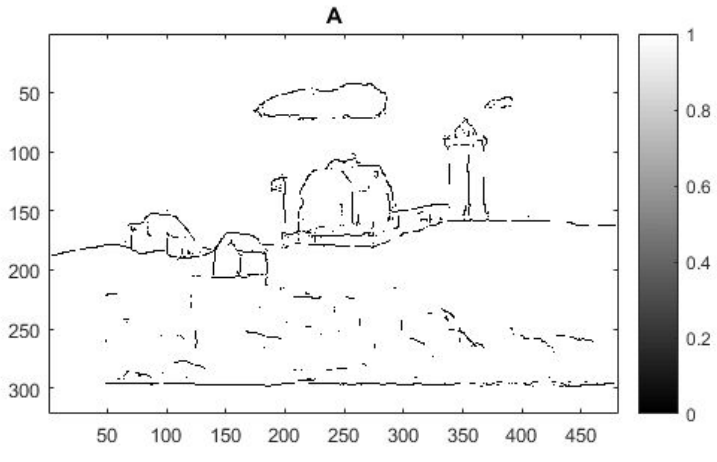
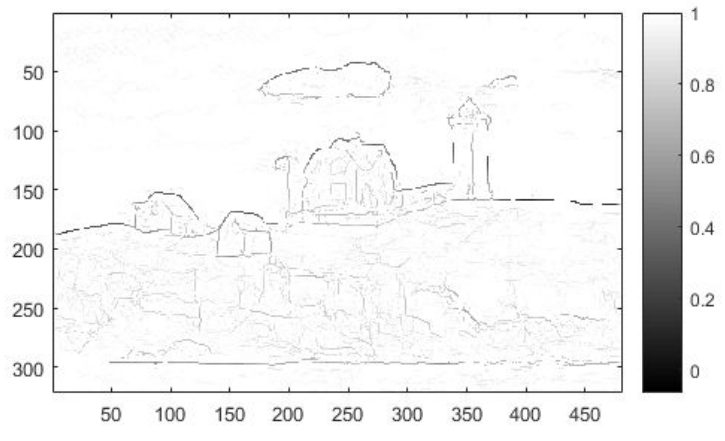
Threshold = 0.17, multiscale = 0, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

Manasa Manohara
EE569 - Hw#3
3722587777

Animal_log	Animal_sobel
	
House_log	**House_sobel**
	

Best outputs:

Binary edge map	Probability edge
	
Threshold = 0.12, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	Threshold = 0.12, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

 Threshold = 0.2, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	 Threshold = 0.2, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

2.4 DISCUSSION :

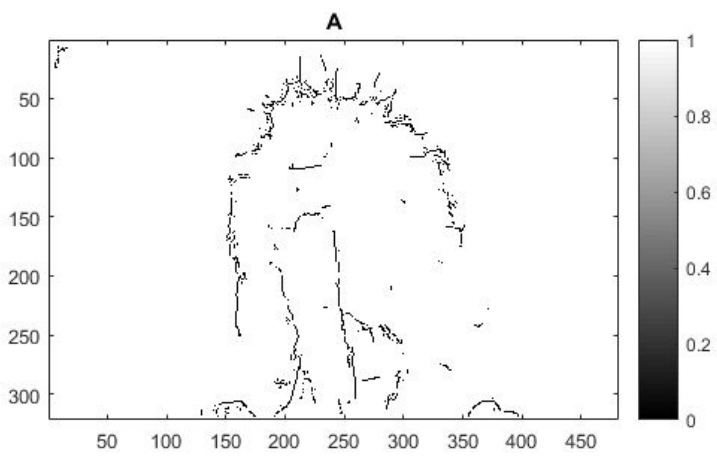
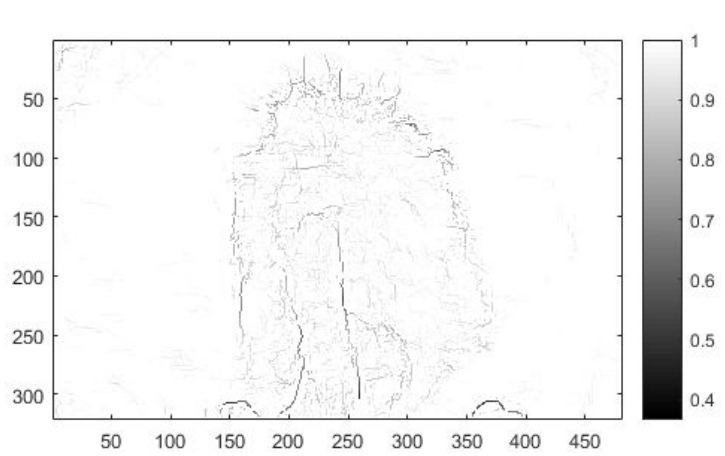
2.4.b. Structured Edge

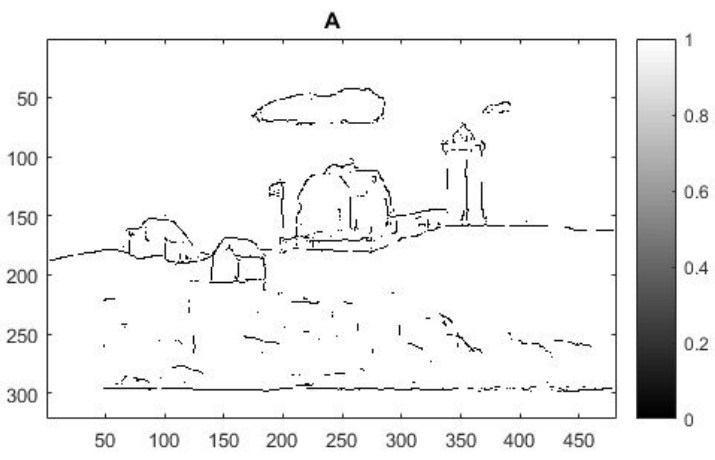
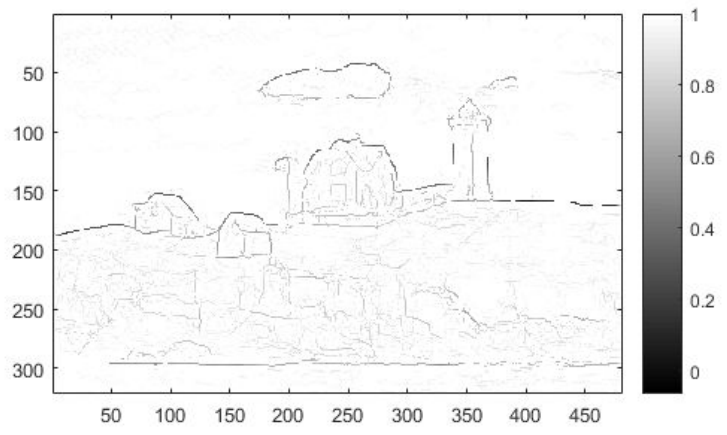
The outputs are with the chosen parameters are shown in the results section of the question.[]

```
%% set detection parameters (can set after training)
model.opts.multiscale=0;          % for top accuracy set multiscale=1
model.opts.sharpen=2;             % for top speed set sharpen=0
model.opts.nTreesEval=4;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number threads for evaluation
model.opts.nms=0;                 % set to true to enable nms
```

- The above mentioned parameters were changed. The threshold parameters were set as either 0.17 or 0.07. The optimum output was observed for the parameters of threshold for 0.2 for house.jpg and 0.12 for animal.jpg.
- Multiscale variable value was used to obtain accuracy of the generated edges. If the value was 0, the generated output had a lot of noisy edges and the output was substandard. When the value was changed to 1, the output generated has clearer and relevant edges.
- Nms value was changed either to 0 or 1. For a value of 0, thicker and darker edges were obtained and for value of 1, thinner and clear edges were obtained. Hence, for running multiple trials of the code, the value of nms was set to 1.
- Sharpen, nTreesEval, nThreads are useful to observe the speed of computation of the program. The change in values for these parameters did not influence the quality of the output.

The optimal output generated was -

Binary edge map	Probability edge
 Threshold = 0.12, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	 Threshold = 0.12, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

	
Threshold = 0.2, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1	Threshold = 0.2, multiscale = 1, sharpen = 0, nTreesEval = 1;nThreads = 4; nms = 1

From the results section of the question, I could see that Structured edge generated better outputs compared to Sobel and LoG detector. Sobel and LoG are sensitive to noise, hence the outputs were grainy and unclear. Structured edge has it basis on sketch tokens and has solid truths to compare for multiple edges, hence redundant and noisy edges were eliminated and it produced a clear and distinct output.

2.2 APPROACH :

2.3.b. Performance Evaluation

To help machines learn which edge to preserve and which to discard which were important for image analysis by scientists and programmers, performance quantitative measures were calculated. To handle the difference in opinions, mean precision and mean recall values were calculated.

To evaluate performance parameters, a pixel in an edge map is said to belong to one of the four classes:

- True positive: Edge pixels of in the edge map match completely with the edge pixels in the ground truth.These pixels are identified by the algorithm perfectly.
- True negative: Non-edge pixels in the edge map coinciding with non-edge pixels in the ground truth. This is again termed as a success.
- False positive: Edge pixels in the edge map being matched to non-edge pixels in the ground truth. This is an error. These pixels are identified wrongly.
- False negative: Non-edge pixels in the edge map matched to true edge pixels in the ground truth. These are edge pixels which algorithm misses.

The performance evaluation parameters are shown below:

$$\text{Precision}: P = \frac{\#\text{True Positive}}{\#\text{True Positive} + \#\text{False Positive}}$$

$$\text{Recall}: R = \frac{\#\text{True Positive}}{\#\text{True Positive} + \#\text{False Negative}}$$

$$F = 2 \times \frac{P \times R}{P + R}$$

A higher F measure implies a better edge detector and to obtain a higher F measure, we need to take account of both precision and recall parameters jointly to achieve a better F measure.

The drawbacks of considering only f-measure as a tool of analysis :

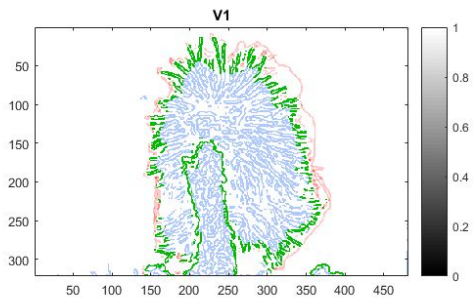
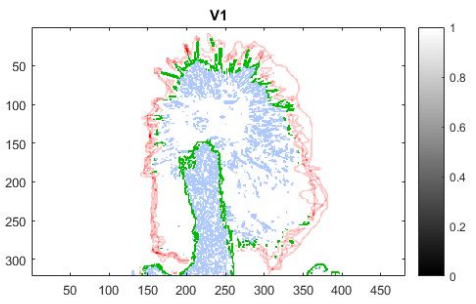
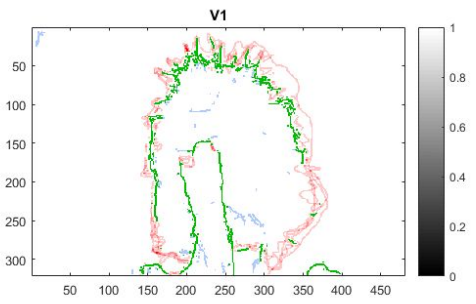
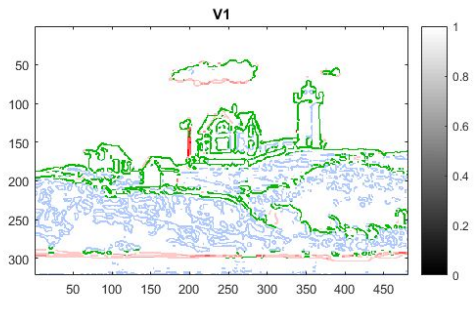
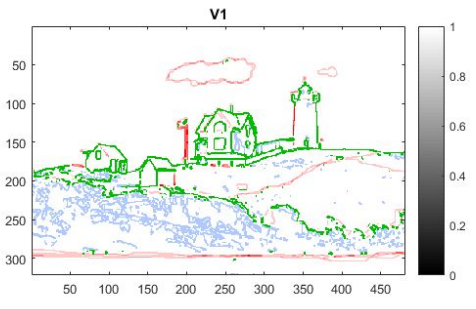
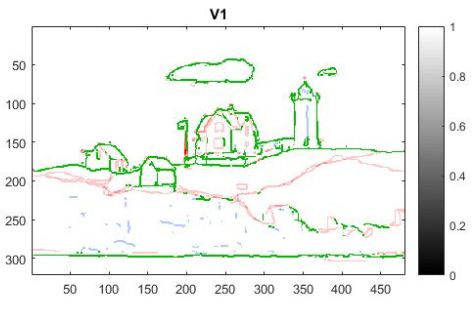
- F-measure as a probability assumes that recall and prediction distributions are identical.
- It is easily biased

2.3 RESULTS :

2.3.b. Performance Evaluation

Performance evaluation :

green=true positive, blue=false positive, red=false negative

 Animal_log	 Animal_Sobel	 Animal_SE
 House_log	 House_Sobel	 House_SE

PERFORMANCE EVALUATION PARAMETERS:

(i)Structured Edge - House.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.26	0.7369	0.8757	0.8003
Ground truth 2	0.23	0.7779	0.6758	0.7233
Ground truth 3	0.25	0.8045	0.7101	0.7544
Ground truth 4	0.22	0.6989	0.7454	0.7214
Ground truth 5	0.19	0.5683	0.7534	0.6479
Mean :	**0.2**	**0.8879**	**0.7878**	**0.8349**

(ii)Structured Edge - Animal.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.13	0.6421	0.6276	0.6348
Ground truth 2	0.12	0.6337	0.6075	0.6203
Ground truth 3	0.16	0.5023	0.3971	0.4436
Ground truth 4	0.13	0.7395	0.6473	0.6903
Ground truth 5	0.13	0.6099	0.5551	0.5812
Mean :	**0.12**	**0.7656**	**0.6069**	**0.6770**

(iii)Sobel Edge detection - House.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.01	0.1626	0.6153	0.2572
Ground truth 2	0.01	0.2356	0.5486	0.3297
Ground truth 3	0.01	0.2037	0.5425	0.2962
Ground truth 4	0.02	0.2528	0.6885	0.3699
Ground truth 5	0.01	0.3030	0.8616	0.4484
Mean :	**0.68**	**0.8879**	**0.6475**	**0.4961**

(iv)Sobel Edge detection - Animal.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.01	0.1467	0.5449	0.2312
Ground truth 2	0.01	0.1690	0.5377	0.2571
Ground truth 3	0.01	0.09	0.4094	0.1491
Ground truth 4	0.01	0.1890	0.6287	0.2907
Ground truth 5	0.01	0.1512	0.5230	0.2346
Mean :	**0.01**	**0.2428**	**0.5352**	**0.3340**

(v)LoG Edge detection - House.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.02	0.1880	0.5892	0.2851
Ground truth 2	0.01	0.2592	0.4996	0.3414

Ground truth 3	0.01	0.2436	0.5372	0.3352
Ground truth 4	0.01	0.2817	0.6350	0.3902
Ground truth 5	0.01	0.3417	0.8045	0.4797
Mean :	**0.005**	**0.4400**	**0.6023**	**0.5107**

(vi)LoG Edge detection - Animal.jpg

Ground truth	**Threshold**	**Mean precision**	**Recall**	**F-score**
Ground truth 1	0.01	0.1614	0.9007	0.2738
Ground truth 2	0.01	0.1785	0.8532	0.2952
Ground truth 3	0.01	0.0920	0.6209	0.1603
Ground truth 4	0.01	0.1910	0.9542	0.3182
Ground truth 5	0.01	0.1594	0.8283	0.2673
Mean :	**0.54**	**0.2674**	**0.8418**	**0.4059**

2.3 DISCUSSION :

2.3.b. Performance Evaluation

- The tables for the performance evaluations is shown above. The mean precision, mean recall and the corresponding threshold are also shown in the last row of the table. From the tables (iii), (iv),(v),(vi) from the above results section, we can see that LoG performs better than the Sobel detector. While we know that the computation of LoG takes longer than Sobel, we can see that the outputs are far more superior than that produced by Sobel. This is also backed by the f- measure values. Sobel is said to be a first degree approximation of LoG and thus it is susceptible to more noisy edges.
- It can be summarized as follows :

Filters	Pro	Con
Sobel	Detects edges and their orientations	Susceptible to noise and produces an inferior output
LoG	As it is computed using the second derivative and smoothed gaussian filters, it can recognize contours and edges at multiple scales and orientations	Susceptible to noise but not as much as Sobel. Detection of curves is sometimes inaccurate
Structured Edge	This uses the approach of trained classifiers and a training model. This is a data driven approach and rigorously classifies all image patches until convergence.	

From the f-score values obtained above, we clearly see that structured edge detector performs better than SObel and LoG.

- From the tables posted above, we can see that the F-measure is image dependant. F-measure of the house image is greater than that of the animal image. We can visually observe that the house image is easier to segment than the animal image. In animal image, the important features which will be necessary for image analysis is occluded by a lot of redundant edges.

- F-measure is the quantitative value of accuracy. As F-measure is a harmonic mean of precision and recall, both of the precision and recall should be varied dependant of each other simultaneously to obtain an optimum output. Precision is inversely proportional to recall, and hence we need to find an expression which manages the trade off between precision and recall not compromising f-measure.

  To find this trade-off, consider P - precision and R- recall;
  P + R = C (a constant)
  → P = R-C
  We know that, F - f-measure is given by;
  F = 2 * (R-C)* (R/C)
  To maximize F, we take the derivative of F w.r.t R, therefore equating it to 0;
  → (2 * 2 * R-2 * C) / C = 0
  → 4*R = 2*C
  → R = C/2

→ P = C/2
→ Precision = Recall
Therefore, F-measure achieves its maximum when Precision = Recall.

PROBLEM 3: SALIENT POINT DESCRIPTORS AND IMAGE MATCHING :

3.1 MOTIVATION :

Feature extraction is one of the most important steps in any computer vision problem. An image contains a lot of information which can help in fields of forensics, object identification, human face recognition etc. The field of computer vision is booming with ImageNet competitions and competitors coming up with faster and more accurate algorithms to solve image processing problems.
Salient point descriptors and image matching can give us a lot of information regarding similarity and dissimilarity between to images or videos. This is one of the first steps in the image classification pipeline.

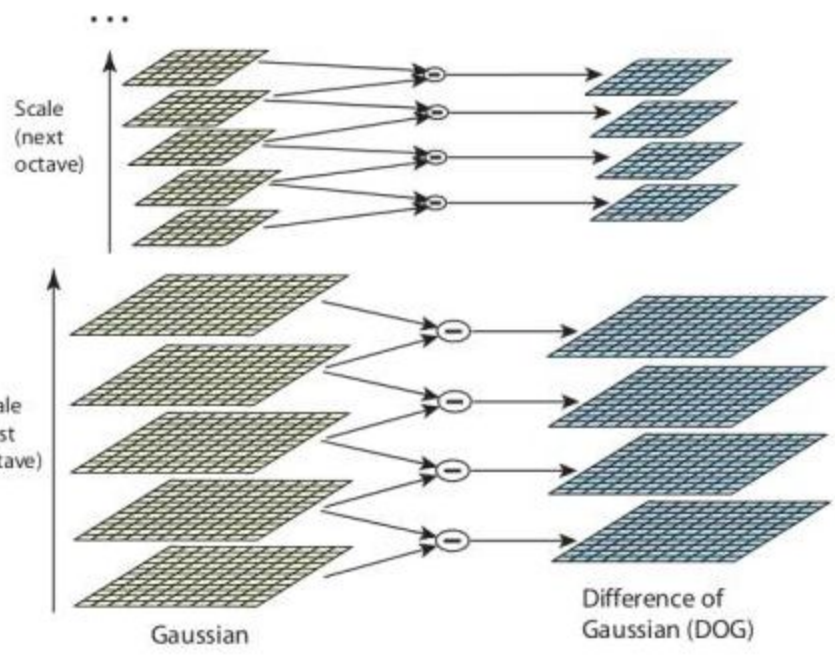
3.2 APPROACH :

For human beings, it is very easy to identify an object in a scene. But for a computer to perform the same task, it needs to extract points which gives maximum information. Such points are referred to as features.
Features essentially gives information of an edge, contour, corner, and blobs. Determining such salient features is known as feature detection. As, it is not sufficient to only get keypoints, the computer also needs its surrounding to make a better judgement of its importance. This is called as feature description.

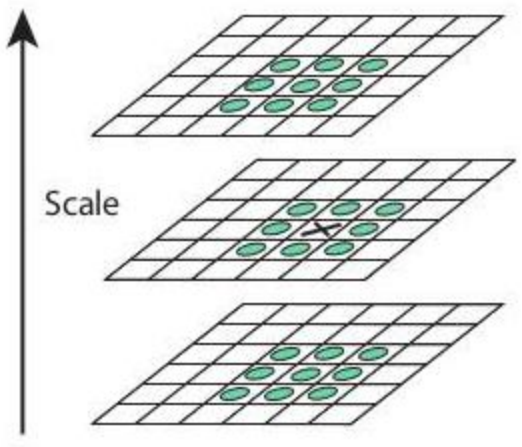
So, D. Lowe introduced an algorithm in his paper[1] which provides a robust algorithm to detect keypoints and compute its descriptors. The Scale- Invariant feature transform is explained below -

SCALE-INVARIANT FEATURE TRANSFORM(SIFT) :
The SIFT algorithm is mainly comprised of 4 steps -

1. Scale-space Extrema Detection : One of the highlights of this algorithm is that it is scale-invariant. As we cannot compare keypoints of differently scaled images, SIFT uses the concept of scale-space filtering. We know that LOG detectors gives us the blobs detected in an image. Therefore, LOG with different σ values are used to compensate for the different scales of the images. σ is a scaling parameter, where if the value of σ is small, it gives us a high value for a small corner and large value of σ is suited for a larger corner. Hence, a local maxima across the scale and space gives us a potential keypoint.
   Gaussian blurring highlights the edges and contours in an image processing and hence is used in Scale invariant transforms. The Gaussian blurring algorithm is applied to the image with different values of σ . AN octave in SIFT is defined as an image where gaussian blurring is applied at every step. It was found that it is optimum to have 3 images in every octave. After the first octave, each octave size is downsampled by a factor of 2. Also, σ in every octave is a factor of $\sqrt{2}$ .As LOG is computationally expensive, SIFT makes use of the Difference of Gaussian Algorithm. DoG gives us an edge map.



Then it is compared along the scaling dimension to detect the keypoints considering the neighborhood which lies in the local extremum.



2. Key point Localization : After finding the key points, we need to eliminate the keypoints which were considered because of noise and other artifacts. Hence, Taylor series expansion of DoG is applied to remove points which are below a threshold of 0.03. To remove repetitive edges along the same direction, we apply Hessian matrix which gives us the principal curvature. We compute the eigen values. If the ratio is above 10, we discard it. This is defined by the Harris corner detector which tells us that if the difference in eigen values is large, then it is an edge. This step eliminates redundant edge keypoints and low-contrast keypoints leaving us with strong salient points.

3. Orientation Assignment : SIFT is rotationally invariant, it assigns an orientation value to each key point. This is done by taking its neighborhood. A histogram of 36 bins is created to comprise all the 360 degrees of rotation with 10 degrees in every bin. The orientation of each pixel is weighed by its gradient magnitude which gives us the direction and a Gaussian weighted circular kernel with σ 1.5x scaling factor. The maximum value of the histogram is selected and all orientations within 80% of the maximum is considered for the orientation.

4. Key point descriptors : We need to look at the neighborhood of the selected keypoints to give us an overview of the keypoint. We consider a 16x16 neighborhood. It is divided into 4x4 blocks from which histogram of the pixels in every block is calculated. As the number of bins is 8, it returns a 128 histogram bin from its neighborhood. This gives us the feature vector. OpenCV's detectandCompute function performs this operation.

SPEEDED-UP ROBUST FEATURES (SURF)[2] :

As SIFT took a lot of computation time, SURF was introduced. Bay, Tuytelaars and Gool published "SURF : Speeded Up Robust Features".

SIFT used the concept of DoG to find LoG, whereas SURF conceptualized using Box Filter. This was computationally faster and it was integral convolution. It can also be done in parallel at different scales. SURF also considers the determinant value of Hessian Matrix for both location and scale.
SURF uses wavelet responses for orientation assignment. These responses are calculated in both horizontal and vertical directions for a neighborhood of 6s whereas for feature description it uses a neighborhood for 20x20s. Dominant response is computed by calculating sum of all responses with a sliding window at an orientation of 60 degrees.
Additionally, SURF offers functionality Upright-SURF. This improves speed and robustness. If upright is 0, orientation is calculated, else it is not calculated.
The 20x20s neighborhood for feature description, is divided into 4x4 regions. This gives a total of 64 dimensions. To provide more robustness, SURF provides a 128 default dimension.

For blob detection,SURF uses sign of Laplacian to determine the underlying interest point. This allows for faster matching, without reducing performance.

I have used the available online source codes to implement this[7].

IMAGE MATCHING :

From the above algorithm, we extracted important SIFT and SURF features. Now to compare images which have been captured in different illuminations, backdrop, different viewing angles - keypoint matching needs to be performed.

In Keypoint matching, SIFT features from two images are compared and nearest neighbor algorithm is applied to determine matching keypoints. If two neighbors are close to a keypoint, then the ratio of the first best and second best is computed. If the ratio is above 0.8, it is rejected. This problem arises due to noise and hence cannot be differentiated easily.

I have used Brute Force Matcher and Flann matcher from OpenCV to obtain matching points.

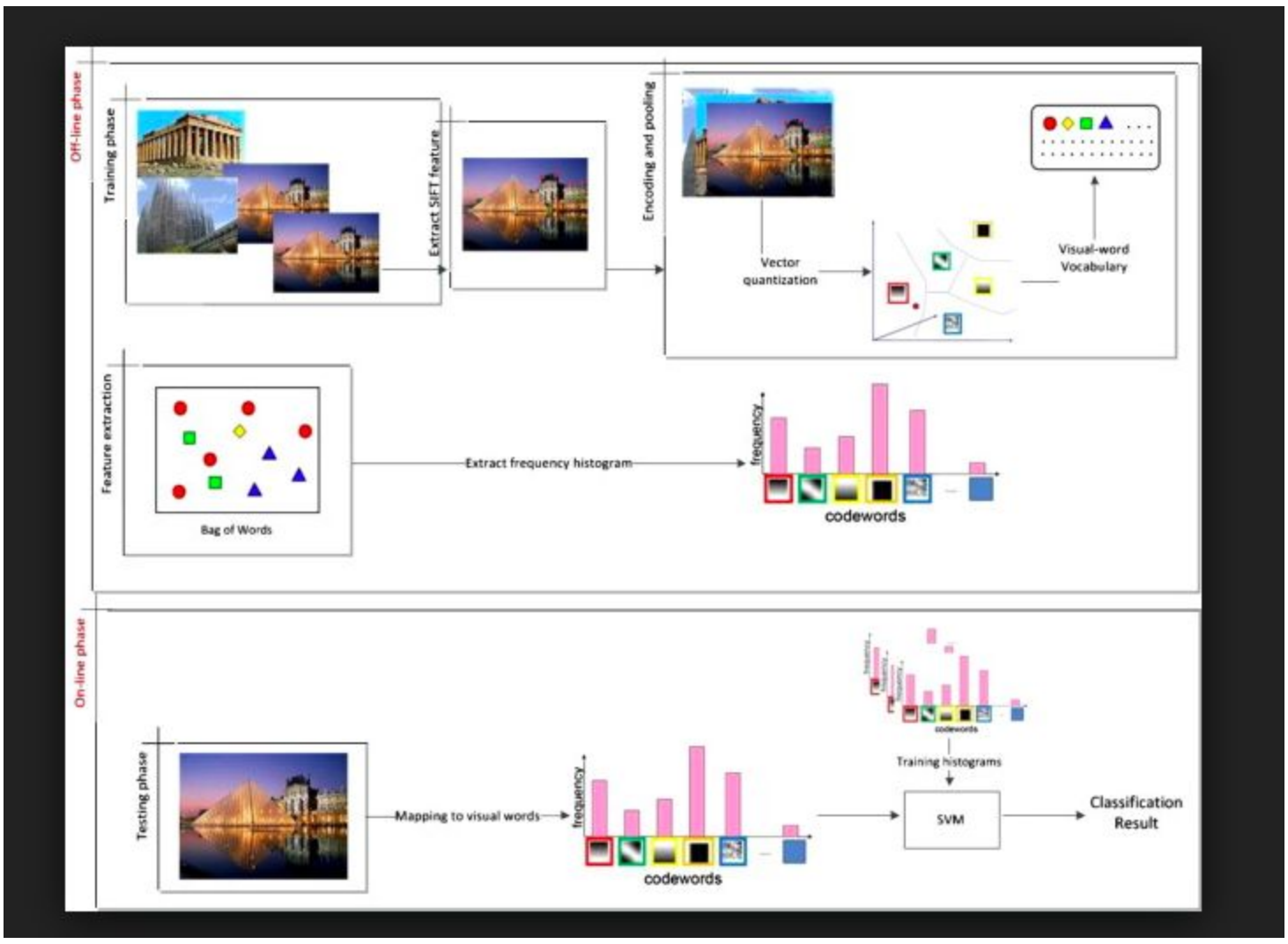
BFMatcher tries all possibilities and exhaust all combinations and hence will find the best matches. For each descriptor in the first set, this matcher finds the closest descriptor in the second set by trying each one.
FLANN is faster but will approximate nearest neighbors.

BAG OF WORDS ALGORITHM :

Bag of Words for computer vision is an algorithm where images are treated a documents. It is followed by three steps - feature detection, feature description, and codebook generation.

Each images is characterized by a set of visual codewords. These visual codewords are quantitatively represented by feature descriptors. Here we have used SIFT descriptor and we end up with a 128 dimension vector.

For codebook generation, a vector of size number of clusters in obtained after performing k-means algorithm.



Source : https://www.sciencedirect.com/science/article/pii/S0957417413002856

ALGORITHM FOR BAG OF WORDS :

1. Images are read as Mat using the OpenCv library.
2. SIFT algorithm is applied to obtain the keypoints and the descriptors .
3. A Bag of Words object is initialised and the descriptors are added to it.
4. A cluster size of 8 is declared and k-means was performed. This gives us a vocabulary of visual words that can be used to define an image. This is the training part of the algorithm.
5. We find the euclidean distance between the descriptors and the generated vocabulary of code words. We then calculate the histogram which will give us the frequency of hits for every code word of every image.
6. Now, we have to compare ferrari_2 to all other images and determine which one it is closest to. So we take the absolute difference between the codewords generated for every image to that of ferrari_2. Then we calculate error which calculates the cumulative sum and then divides by the number of clusters.

2.3 RESULTS :
2.3.b. Structured Edge

(i) Output 3.a)

Manasa Manohara
EE569 - Hw#3
3722587777

SIFT - minHessian = 400

SURF - minHessian value = 400

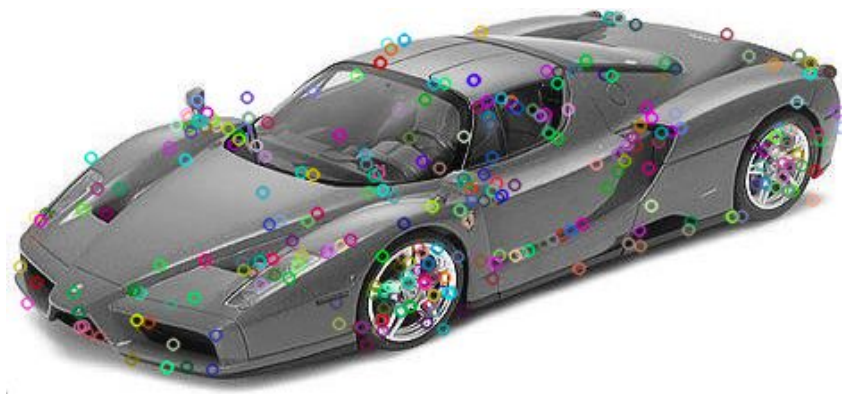
SIFT - minHessian = 400

SURF - Hessian value = 400

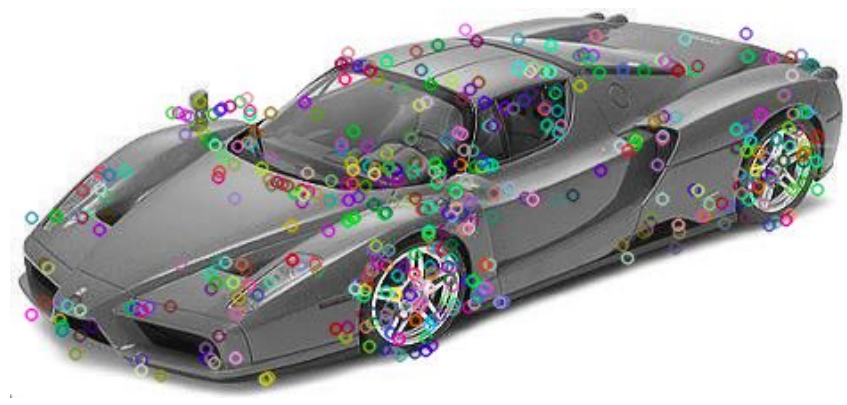
(ii) Output 3.b)

Ferrari_ 1 - SIFT minHessian = 400

Ferrari_ 1 - SURF minHessian= 400



Ferrari_ 2 - SIFT minHessian = 400



Ferrari_ 2 - SURF minHessian = 400

Manasa Manohara
EE569 - Hw#3
3722587777

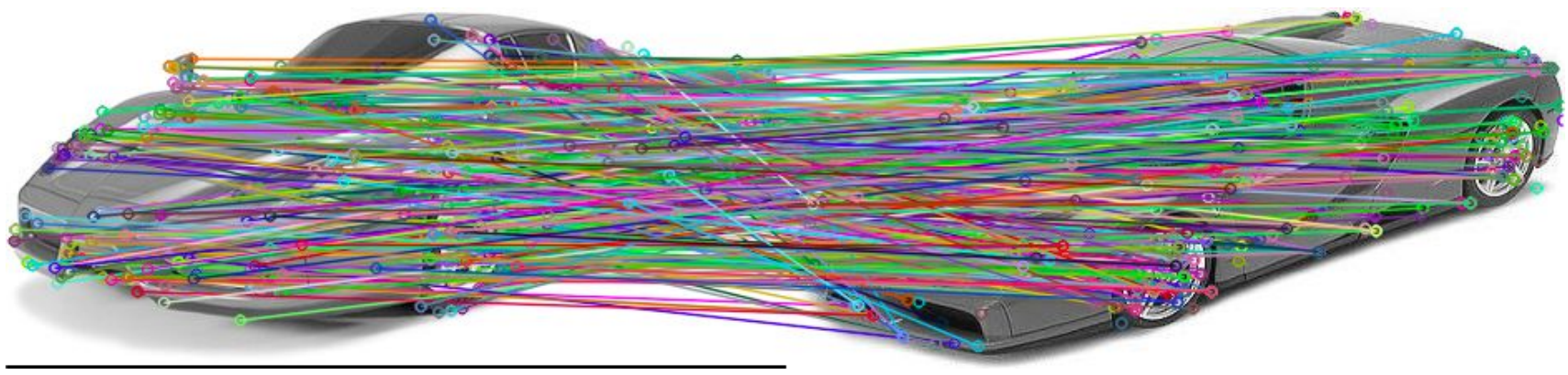
Optimus_prime - SIFT minHessian = 400

Optimus_prime - SURF minHessian = 400

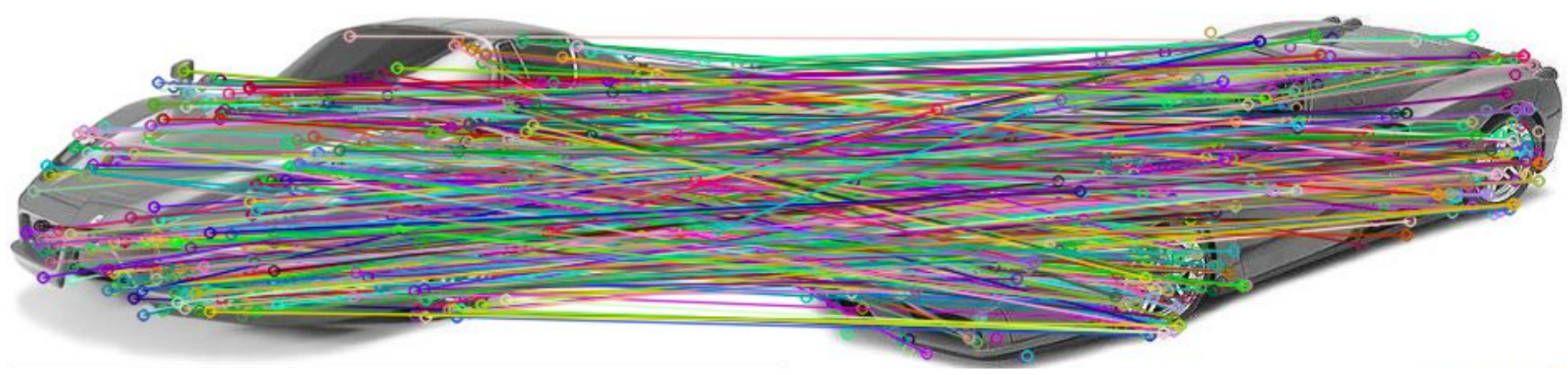
bumblebee - SIFT minHessian = 400

bumblebee - SURF minHessian = 400

a) **Using BF Matcher**

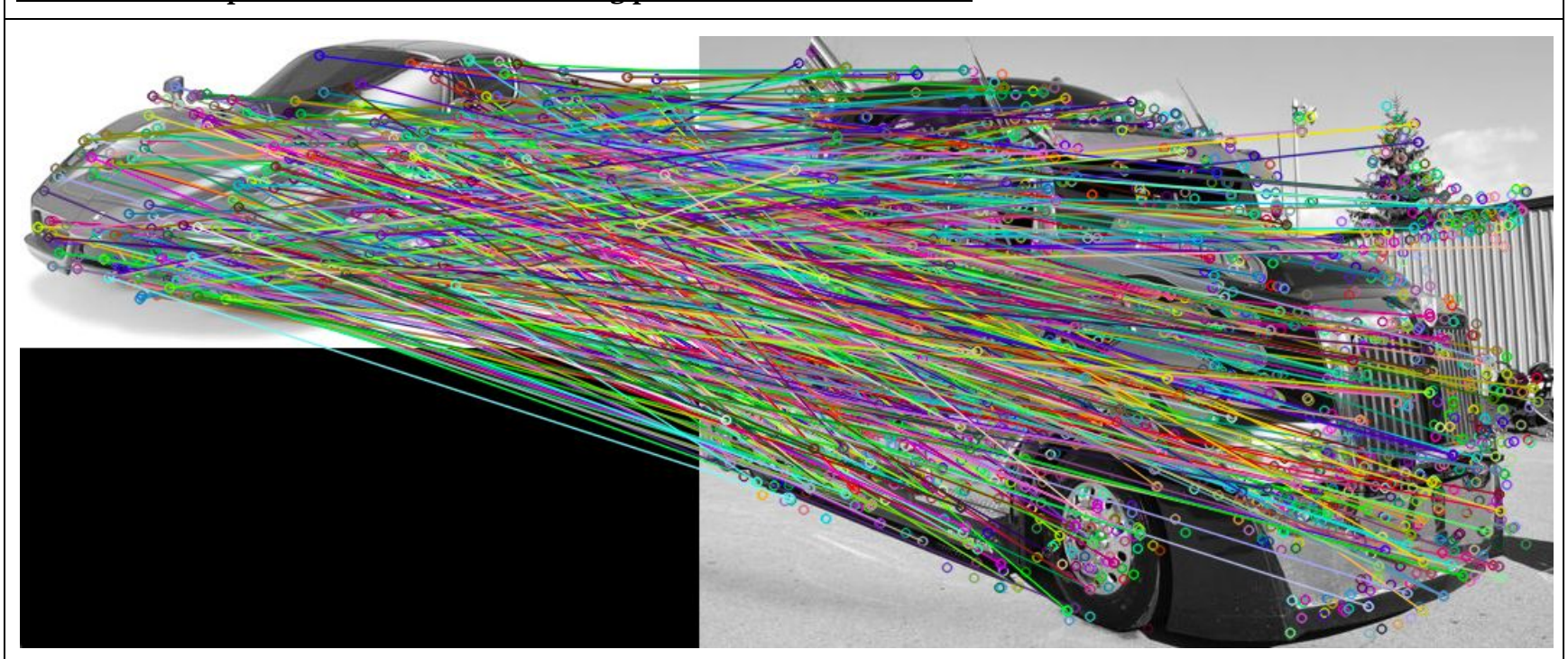


SIFT matching keypoints - minHessian = 400

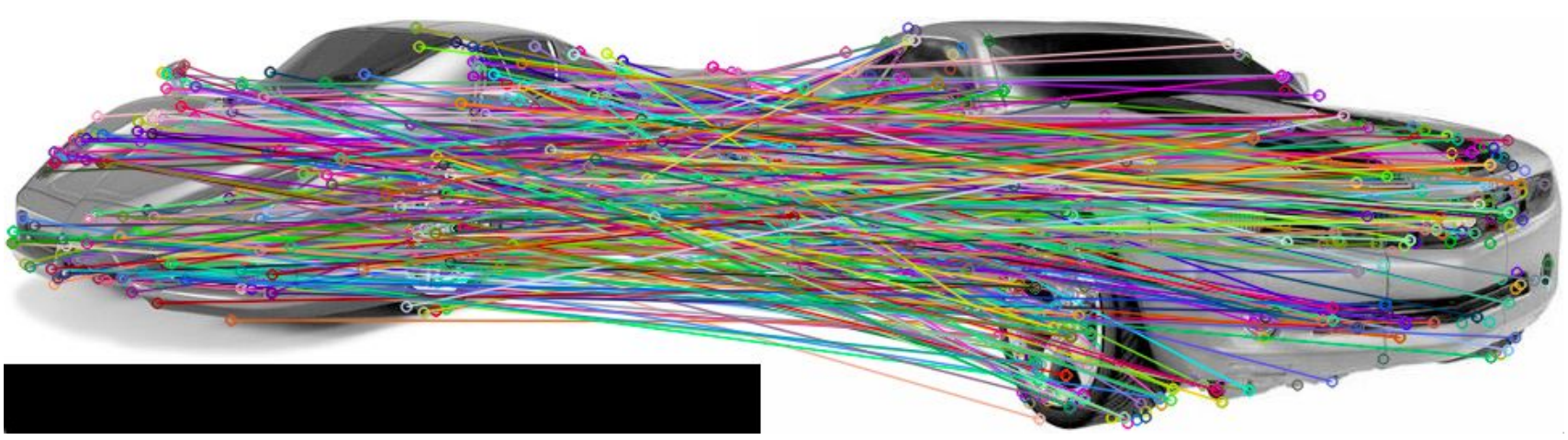


SURF matching keypoints - minHessian = 400

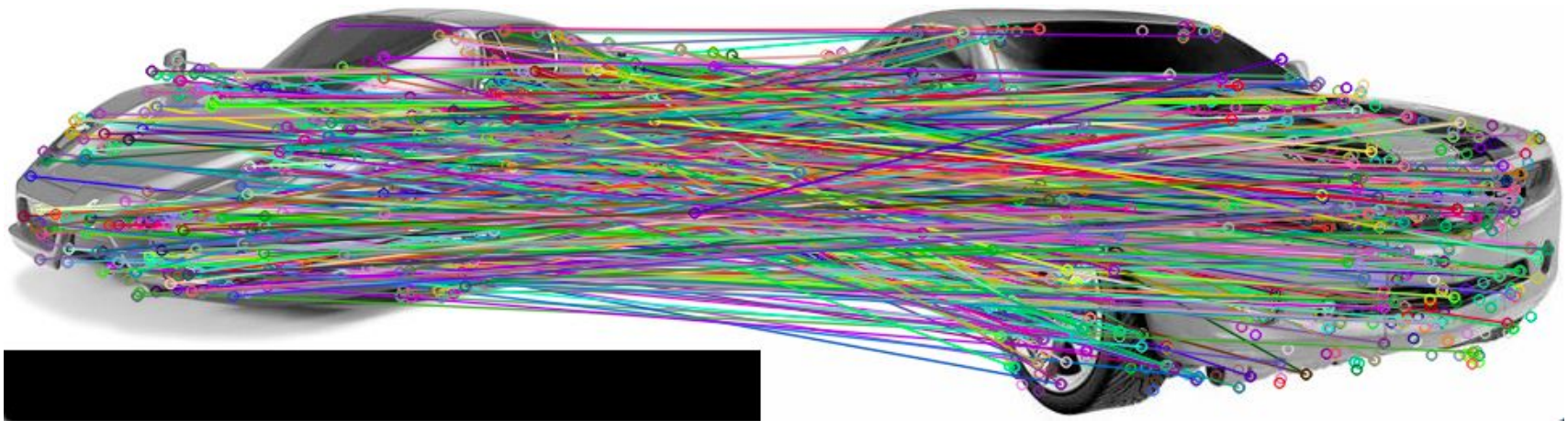
Ferrari_1 and Optimus Prime - SIFT matching points - minHessian = 400



Ferrari_1 and Optimus Prime - SURF matching points - minHessian = 400



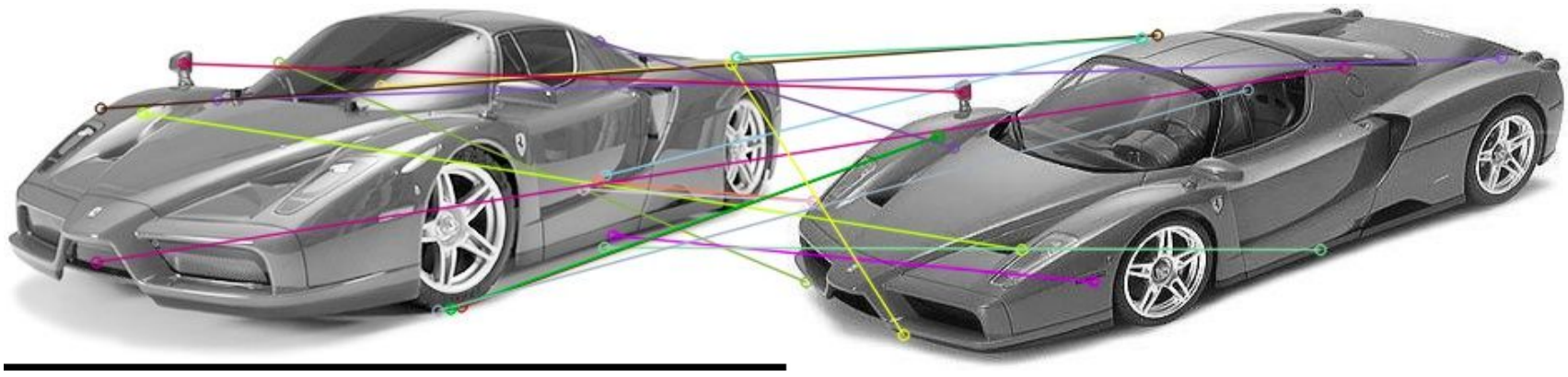
Ferrari_1 and bumblebee - SIFT matching points - minHessian = 400



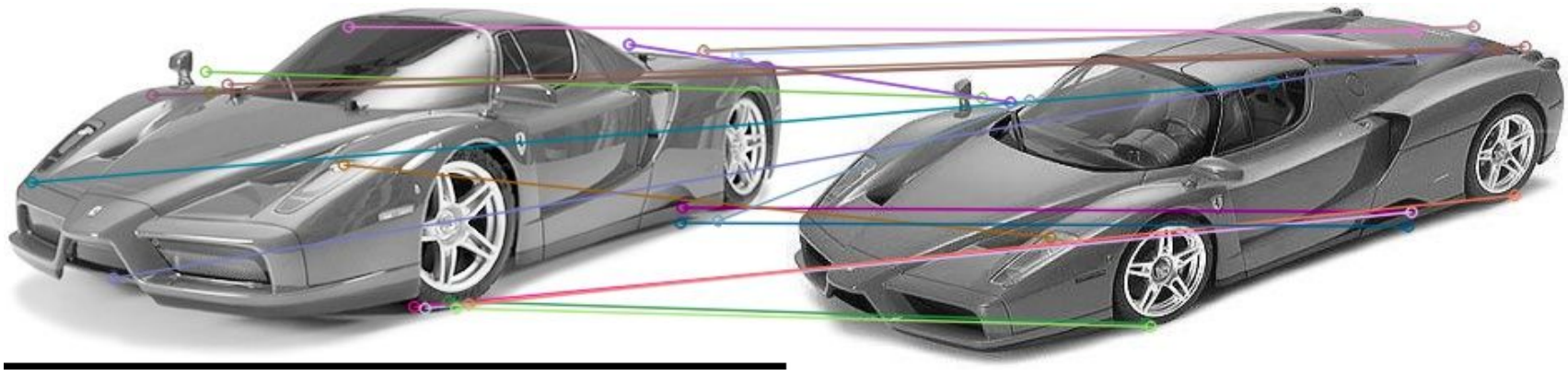
Ferrari_1 and bumblebee - SURF matching points - minHessian = 400

b) Using FLAN Matcher

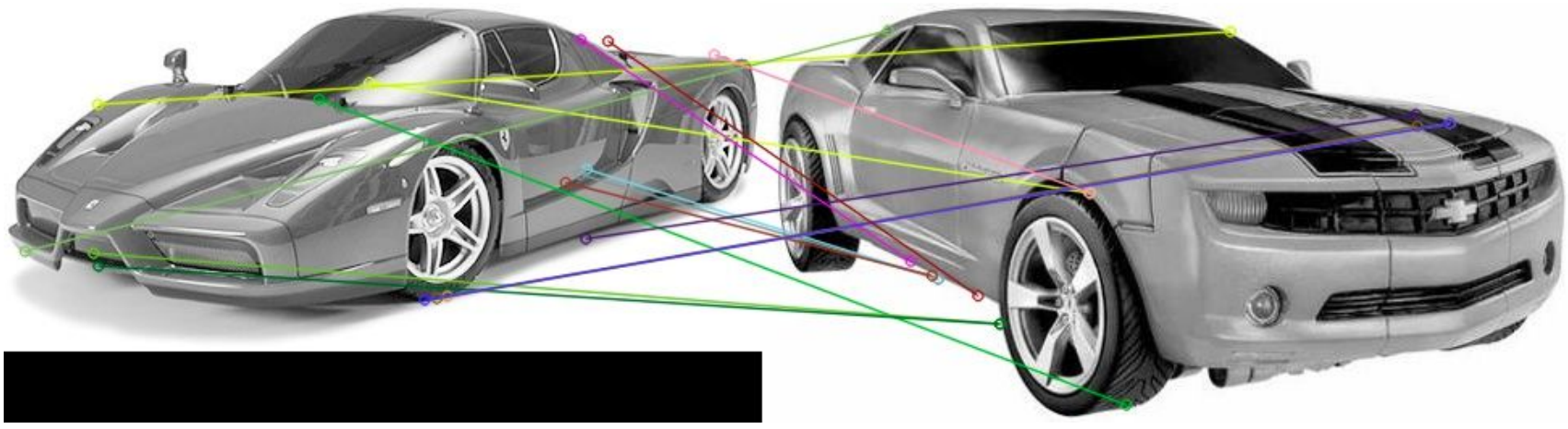
Manasa Manohara
EE569 - Hw#3
3722587777



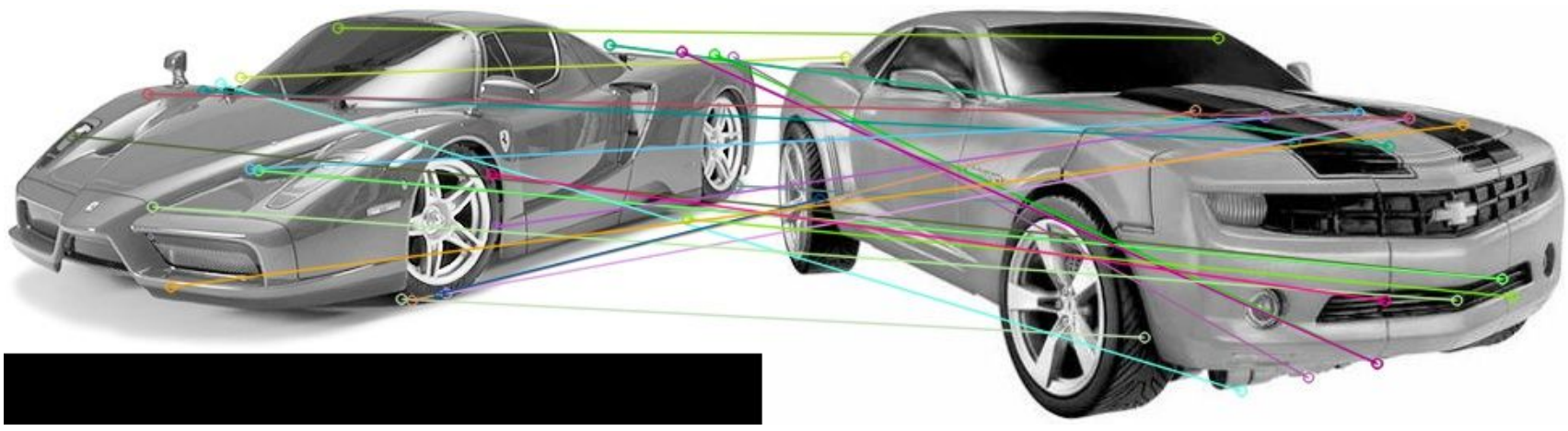
Ferrari_1 and Ferrari_2 - SURF - minHessian = 400



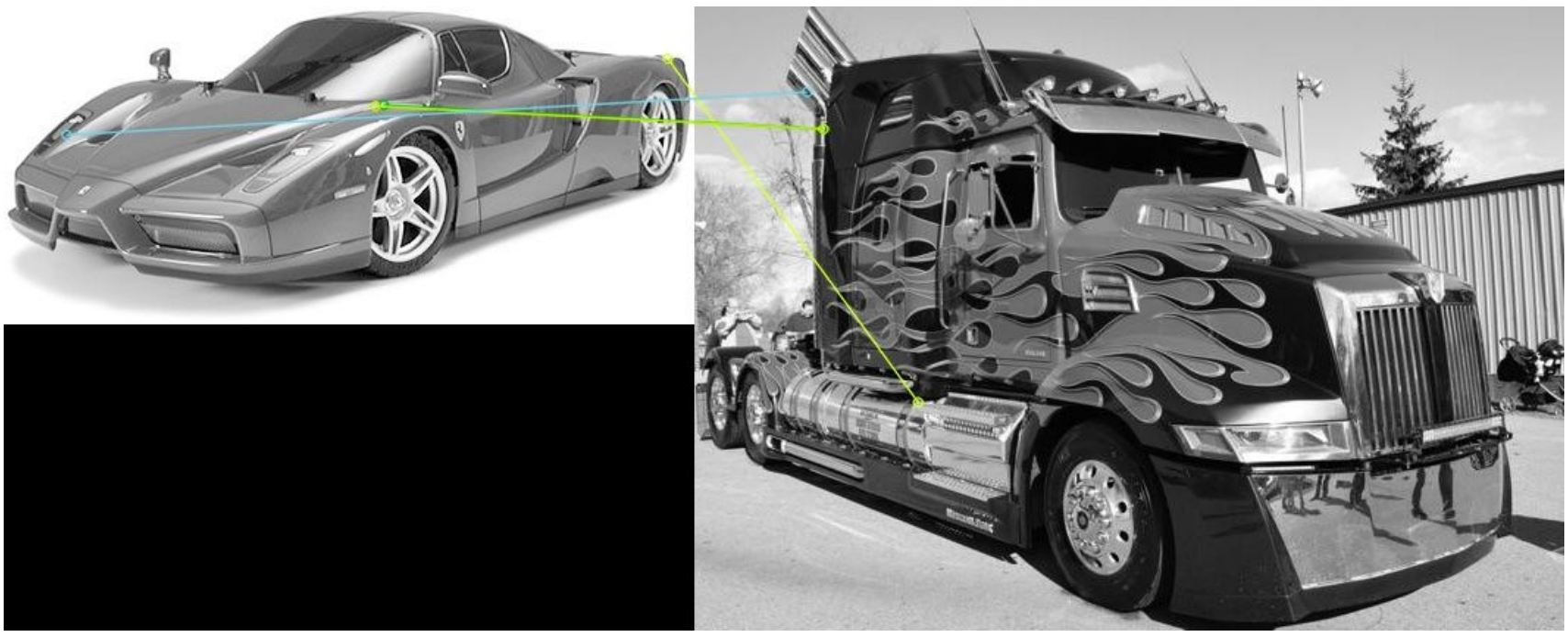
Ferrari_1 and Ferrari_2 - SURF - minHessian = 400



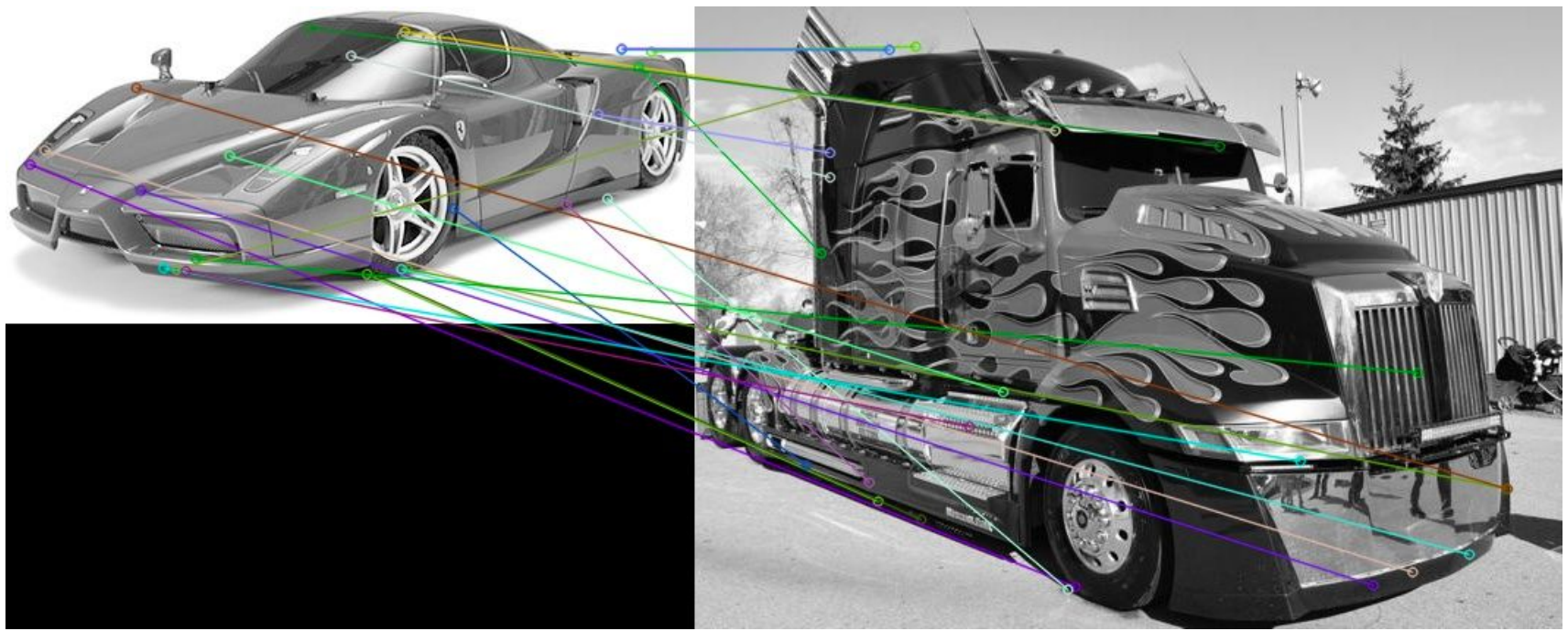
Ferrari_1 and bumblebee - SIFT - minHessian = 400



Ferrari_1 and bumblebee- SURF - minHessian = 400

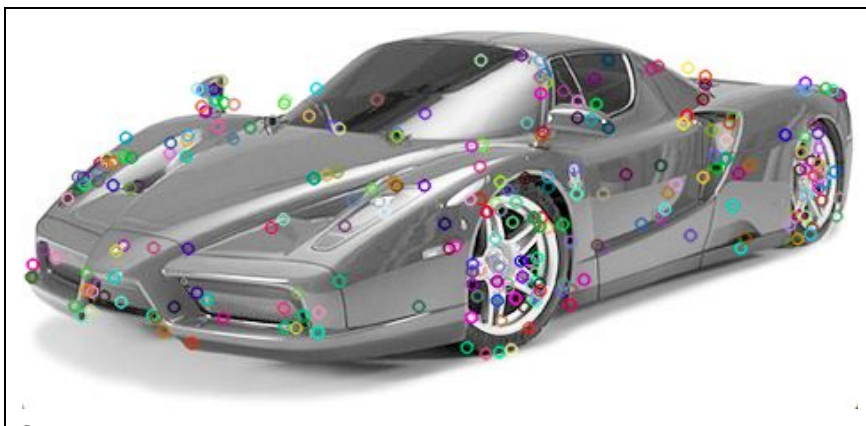


Ferrari_1 and optimus_prime- SIFT - minHessian = 400

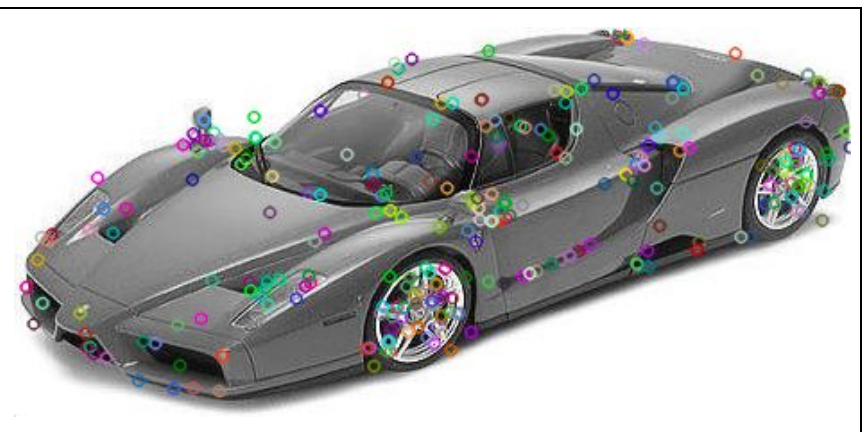


Ferrari_1 and optimus_prime- SURF- minHessian = 400

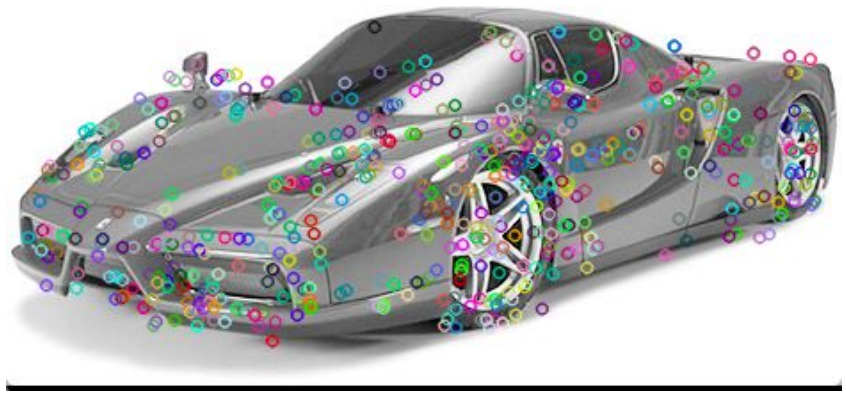
c) Ferrari_1 and Ferrari_2 with minHessian = 300



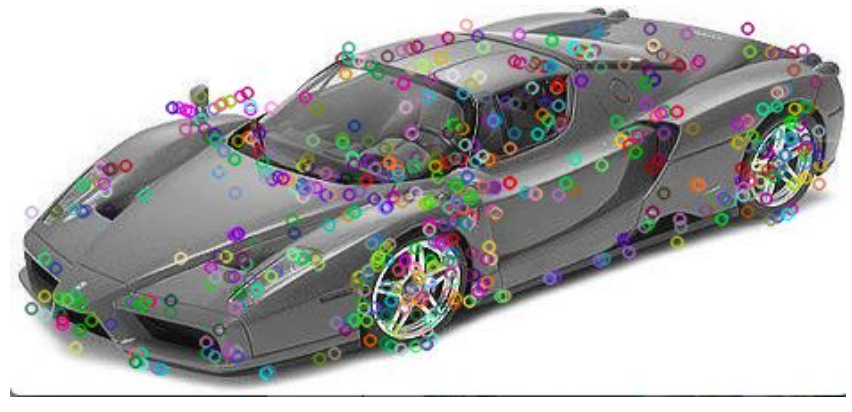
SIFT



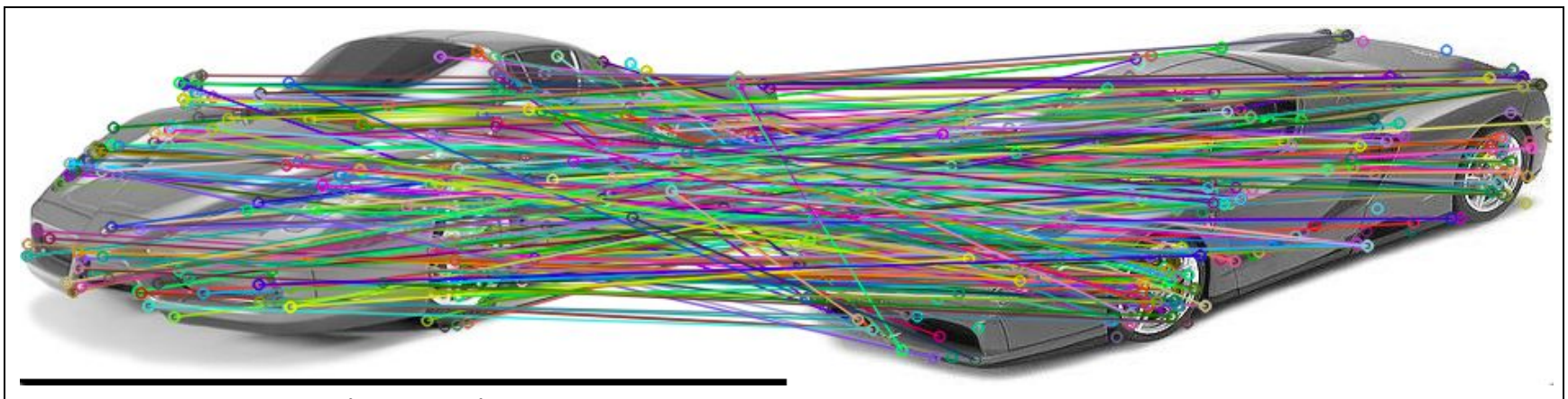
SIFT



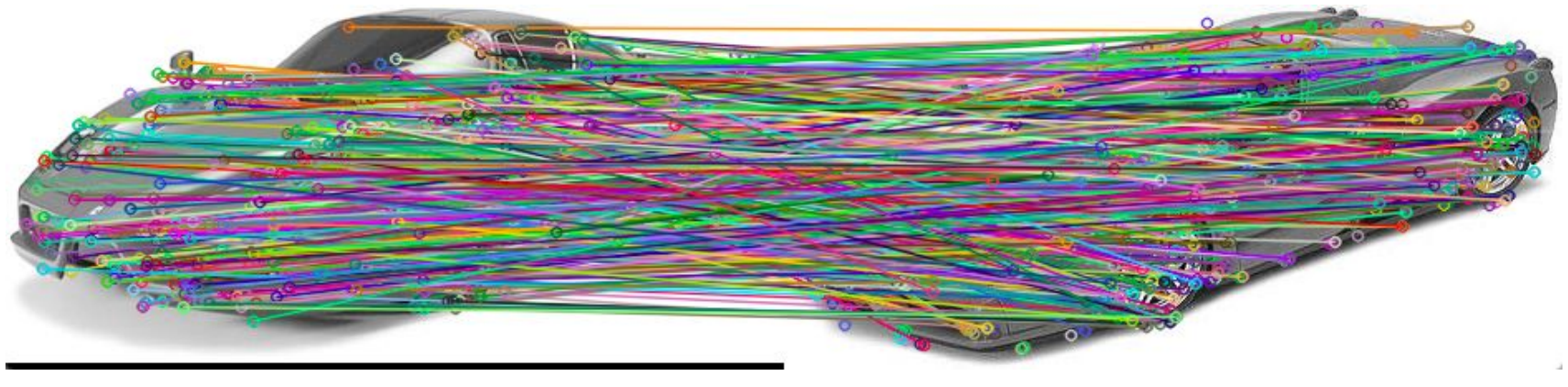
SURF



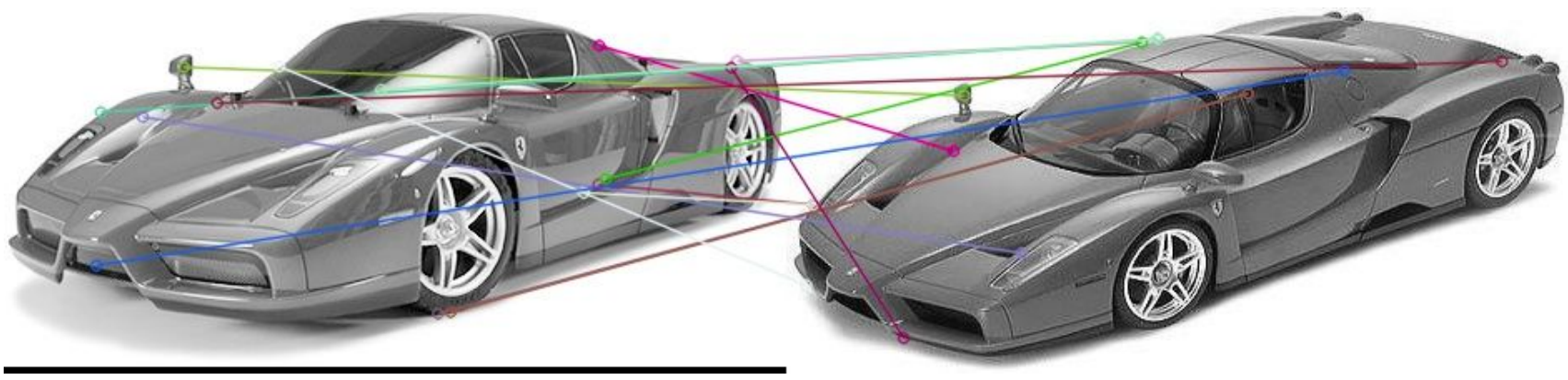
SURF



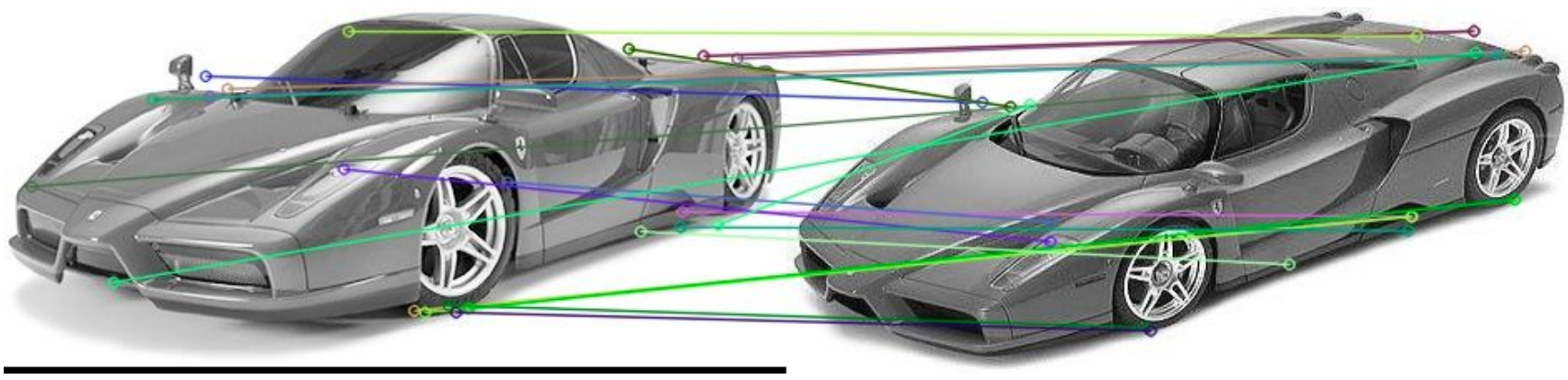
BF Matcher - SIFT matching keypoints



BF Matcher - SURF matching keypoints



FLAN matcher - SURF - minHessian = 300



FLAN matcher - SURF - minHessian = 300

3.4 DISCUSSION :

a.) Extraction and Description of Salient points

We can see from the results above that SURF gives us more keypoints than SIFT for the default Hessian value of 400. The differences between SIFT and SURF are -

- In the scale space, SIFT uses DoG convolved with Gaussian filtering to compute LoG whereas SURF uses box filters convolved with integral image.
- SIFT uses weighted orientations of surrounding pixels to calculate orientation whereas SURF uses a window size of pi/3 to detect the dominant orientation. SURF also offers upright-SURF.
- Key point descriptors of SIFT by taking histogram of its 16x16 neighborhood whereas SURF considers wavelet responses in the 4x4 square sub region.
- SURF is also 3 times faster than SIFT.
- SURF does not perform well at illumination change and view point changes in images.
- SIFT handles images with blurring and rotation better.

I have also experimented with different minHessian values. The outputs can be seen in section 3. In the openCV documentation, it has been mentioned that the minHessian value needs to be between 300-500 to get optimum number of keypoints. I have presented outputs for minHessian = 400 and minHessian = 300 in the section. Changing the parameter of minHessian affects the number of keypoints selected. Determinant of minHessian is calculated for every image patch and then sets the threshold .
Hence, in SIFT I could not identify any large difference between the number of key points selected for minHessian = 300 and 400.
In SURF, as I decreased the minHessian value, I got more keypoints. This is induced because as the threshold is set low, it picks up a lot of features with less repeatability. Assigning a higher value to minHessian in SURF eliminated a lot of noisy key points as it picked up points with more repeatability. In general, I got more keypoints using SURF than SIFT for default minHessian threshold.

b.) Image matching

After extracting key points using the SIFT and SURF algorithms, we are asked to perform image matching. I have tried matching the images using two matchers available in OpenCv. They are - Brute Force matcher and FLANN matcher.

BFMatcher computes all combinations of the selected key points and exhaust all of them to provide the best matches. FLANN computes the nearest neighbor when selecting matching key points between two images and then provides us with the matched image. From the above results, we can observe that FLANN outperforms BFMatcher. FLANN was also computationally faster than BFMatcher.

The following observations were made -
For the same Hessian value of 400 -

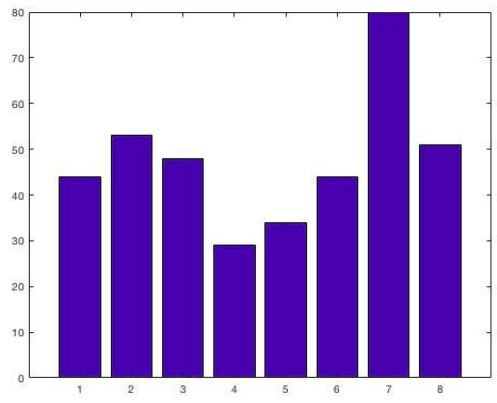
- In BFMatcher, the front of Ferrari_2 was getting matched to the rear of Ferrari_1. From visual inspection, we can be sure that front cannot be mapped to the rear. BFMatcher produced a lot of noisy key points.
- BFMatcher matched the entire of the Ferrari_1 body to the bumblebee body. We know that both of them are different cars,but they still have similar features. But mapping of entire ferrari_1 to bumblebee is due to a lot of noisy keypoints.
- FLANN matcher on the other hand, gave a better output as it compares the nearest neighbor values before matching the keypoints. For the same value of minHessian of 400, matching points were clearly visible in FLANN than it was in BFMatcher. The image matching was not sensitive to a lot of noisy keypoints in FLANN.

Hence, the following conclusion was arrived at -

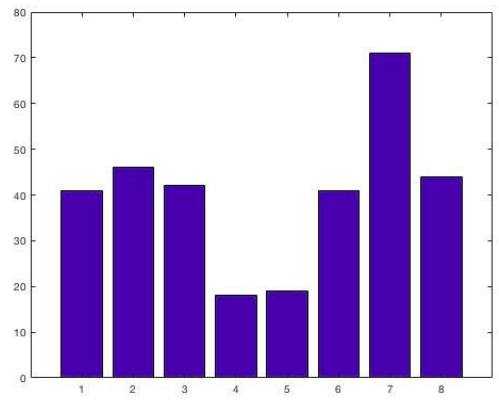
- FLANN matcher works better than BFMatcher for the given images.
- SURF gives us better keypoints than SIFT algorithm. This is because it has the U-SURF feature additionally which is not available in SIFT.
- But we can still observe some wrongly matched keypoints in FLANN with Ferrari_1 and Ferrari_2 where the front of the car is getting mapped to the rear of ferrari. This is one of the noisy keypoints. The selection of keypoints depends on the large viewing angle here. BFMatcher produced a lot of noisy keypoints thus producing an inferior output.
- Even though, Ferrari_1 and bumblebee are still cars and cars have similar features, BFMatcher produced a substandard output as it matched the entire body of Ferrari_1 to bumblebee.
- Matching between Ferrari_1 and optimus_prime was done better by FLANN matcher than BFMatcher. BFMatcher matched most of the Ferrari_1 to optimus_prime body, producing a lot of noisy keypoints. FLANN performed better as we can see that only two keypoints were matched.

b.) Bag of Words.

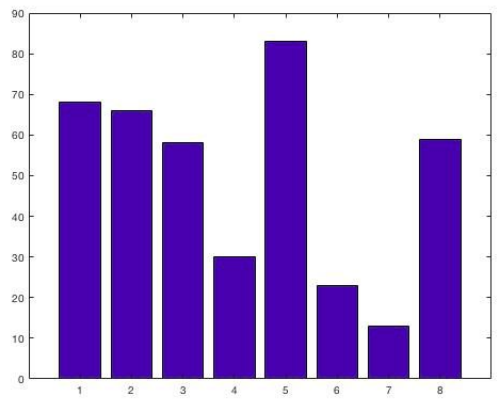
This is one of the most applied computer vision classification algorithm. The algorithm clusters a group of visual words obtained from the training image set for comparison with a testing image. The training images considered here are - Ferrari_1, bumblebee, and Optimus prime. The testing image is Ferrari_2. The descriptors of the training images were trained using BOWKMeans Trainer which uses the KMeans approach in OpenCV. The number of clusters chosen was 8. A vocabulary was formed using the descriptors. The vocabulary formed is a codebook. Then euclidean distance was calculated by computing the distance between every descriptor and vocabulary. For every hit in a cluster for every image, frequency was incremented. The histogram of all the images were obtained and is shown below -



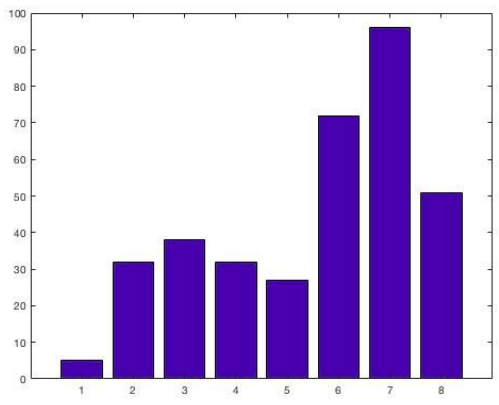
Histogram for Ferrari 1



Histogram for Ferrari 2



Histogram for Optimus Prime



Histogram for bumblebee

We can see visually that the histogram of Ferrari_1 and Ferrari_2 are similar. The task was to match Ferrari_2's codeword with others.A codeword refers to the histogram of a image. So to compare codewords, the cumulative absolute difference was considered and the output was displayed.

```
/Users/mansi/CLionProjects/hw3_prob3c/cmake-build-debug/hw3_prob3c /Users/mansi/Documents/imageProcessing/images/EE569_h
Comparing Ferrari2 with Ferrari1: The error is: 7.625
Comparing Ferrari2 with OptimusPrime: The error is: 28.75
Comparing Ferrari2 with Bumblebee: The error is: 14.25
 Ferrari 2 is closest to Ferrari 1

Process finished with exit code 0
```

The error rate was calculated as the(cumulative absolute difference / number of clusters). The output obtained is in coherence with visual inspection as we can observe that Ferrari_2 would be closest to Ferrari_1. This is a very sophisticated technique to find the percentage of mapping between two images by using a training set of images to compute clusters and form a codebook and then comparing the matching between the codewords of the images with the codebook.

REFERENCES:

[1]David G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, 60(2), 91-110, 2004
[2]Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, "SURF: Speeded Up Robust Features", *Computer Vision and Image Understanding (CVIU)*, Vol. 110, No. 3, pp. 346--359, 2008
[3] Canny, John. "A computational approach to edge detection." Pattern Analysis and Machine Intelligence, IEEE Transactions on 6 (1986): 679-698.
[4] Dollár, Piotr, and C. Lawrence Zitnick. "Structured forests for fast edge detection." Computer Vision (ICCV), 2013 IEEE International Conference on. IEEE, 2013.
[5] www.wikipedia.com
[6]www.stackoverflow.com
[7]https://opencv.org/